Journal of Scheduling
https://doi.org/10.1007/s10951-022-00751-9

Algorithms for rescheduling jobs with a LIFO buffer to minimize the weighted number of late jobs

Ulrich Pferschy[1] · Julia Resch[1] · Giovanni Righini[2]

Accepted: 1 August 2022

Abstract
Rescheduling can help to improve the quality of a schedule with respect to an initially given sequence. In this paper, we consider the possibility of rescheduling jobs arriving for processing at a single machine under the following limitations: (a) jobs can only be moved toward the end of the schedule and not toward the front, and (b) when a job is taken out of the sequence, it is put on a buffer of limited capacity before being reinserted in its new position closer to the end of the sequence. The buffer is organized as a stack with a last-in/first-out policy. As an objective function, we consider the minimization of the weighted number of late jobs. For this NP-hard problem, we first provide two different integer linear programming (ILP) formulations. Furthermore, we develop a branch-and-bound algorithm with a branching rule based on the movement of jobs. Then a new pseudo-polynomial dynamic programming algorithm is presented which utilizes dominance criteria and an efficient handling of states. Our computational experiments with up to 100 jobs show that this algorithm performs remarkably well and can be seen as the current method of choice.

Keywords Scheduling · Rescheduling · Integer linear programming · Dynamic programming · Branch-and-bound

1 Introduction

In current production environments, planners are facing pressure to ensure the best possible utilization of costly machinery, while at the same time the imperative for high flexibility and last-minute arrivals of orders implies that production data are becoming more volatile. Product customization up to unit lot size requires a continuous reconsideration of the production sequence instead of a once-for-all production plan for the traditional manufacturing of identical products with large lot sizes. Usually, jobs go through several work stages which should be coordinated (cf. Agnetis et al. 2006). If the job sequence is optimized for one critical machine (or simply given by the arrival of parts), it would be highly desirable to reorganize the job sequence for the subsequent machine which may have different characteristics and thus would be less efficiently utilized by the original sequence. For example, in automobile production, cars have different colors, which asks for a grouping of similarly colored cars in the paint shop to reduce setup times. However, subsequent production stages, where for example different sets of options have to be installed requiring widely differing production times, will require quite different sequences to optimally utilize the production capacity and to fulfill deadlines at the end of the line (see e.g. Drexl et al. 2006). Thus, different production stages would imply different optimal sequences. Clearly, considering space and handling effort, it is out of the question to fully reorder the sequence of jobs between any two production stages. However, a limited reordering operation where product units are taken out from the line and put into a waiting area of limited capacity before being reinserted is well within the technical capabilities of many production environments. Also last-minute changes of the expected or estimated processing times or the late disclosure of job data cause a need for changes in the sequence of jobs as they arrive for processing.

✉ Ulrich Pferschy
ulrich.pferschy@uni-graz.at

Julia Resch
julia.resch@uni-graz.at

Giovanni Righini
giovanni.righini@unimi.it

[1] Department of Operations and Information Systems, University of Graz, Universitaetsstrasse 15, 8010 Graz, Austria

[2] Department of Computer Science, University of Milan, Via Celoria 18, 20133 Milan, Italy

Published online: 07 October 2022

In this paper, we consider the following setting of a single-machine problem which was introduced in Nicosia et al. (2019) and treated in more detail in Nicosia et al. (2021).

There is a sequence of jobs given to be processed sequentially on a single machine. Preceding production stages only play a role insofar as the given sequence of jobs originated from these stages. We can rearrange the jobs before they enter the considered machine to improve the value of a certain objective function depending on the job sequence. However, the rearrangement is restricted by a physical handling device. In our setting, we envision jobs to arrive on a (slowly) moving conveyor belt which transports them toward the considered single machine for processing. A job (which is equivalent to a physical part) can be picked, e.g., by a robot, and taken out of the sequence to be placed in a buffer of limited capacity. Then additional jobs may be taken out of the sequence, but at some point the job is reinserted into the sequence at a later position on the conveyor belt. Thus, it is only possible to "move a job backwards" in the sequence, but never "forward" to an earlier position. Moreover, we assume that the buffer is organized as a stack, i.e., the most recently removed job must be reinserted first which amounts to a last-in/first-out (LIFO) organization of the buffer.

The main contributions of the preceding paper by Nicosia et al. (2021) for the rescheduling problem were strictly polynomial algorithms based on dynamic programming for the minimization of the following classical objective functions:

- total weighted completion time,
- maximum lateness (and maximum of regular scheduling functions),
- number of late jobs.

Furthermore, the authors considered the minimization of the *weighted number of late jobs*. For this objective, the rescheduling problem was shown to be NP-hard. In Nicosia et al. (2021), only a pseudo-polynomial dynamic programming algorithm was given, as an extension of the respective algorithm for the unweighted case. However, the development and evaluation of actually applicable solution strategies was left open in this previous work.

In this paper, we will study more extensively this relevant NP-hard case which reflects the case of due dates for products at the end of the line imposed, for example, by tight outbound logistic plans. Any violation of the due date of an item causes inconvenience as well as handling and penalty costs, weighted by the importance of the customer. Note that besides being the most natural NP-hard version of the rescheduling problem, the weighted number of late jobs can be of particular importance for on-demand production scenarios. If orders arrive for possible processing with a tight time frame, but not all of them can ultimately be accepted for production, it is crucial to keep those with the highest profit or, equivalently, those which would incur the highest loss if they were not produced. For orders with low profit, i.e., low weights, it is less disturbing if they cannot meet the deadline. However, while in the standard scheduling situation late jobs are usually moved to the end of the sequence or completely canceled, this will often be impossible in rescheduling problems due to the rearrangement restrictions. Therefore, there will be orders being processed although they will miss their deadline. This outcome poses an interesting challenge and leaves room for additional negotiations between the producer and the ordering customer about the monetary compensation for orders fulfilled after their deadline.

Other operational restrictions on possible rearrangements for such a conveyor-based scenario were considered in Alfieri et al. (2018b). In the literature, various other situations of rescheduling were considered. One research branch going back to the seminal work of Hall and Potts (2004) considers the arrival of new jobs to be integrated into the sequence of old jobs (see also Hall et al. 2007 and Rener et al. 2022). Another direction moving into the area of robust optimization deals with changes in the original data of the jobs. The corresponding rescheduling issues were discussed, e.g., in Abumaizar and Svestka (1997); Detti et al. (2019); Niu et al. (2019) and Ballestín et al. (2019). An interesting aspect in this direction is that of recoverable robustness (see e.g. Liebchen et al. (2009)). There, one looks for a solution that can be made feasible by a simple recovery step (see Akker et al. 2018 for a scheduling contribution). In this context, rescheduling can also be seen as a recovery step. Also, disruptions can give rise to the need for rescheduling a previously planned schedule. This was treated, e.g., by Hall and Potts (2010) for the case of delayed jobs, Nouiri et al. (2018) for random disruption events, and Li and Li (2020) for accidents. The recently published monograph Wang et al. (2020) provides a comprehensive overview. Two survey papers treating rescheduling topics are presented by Ouelhadj and Petrovic (2009) and Vieira et al. (2003). Additional pointers to the related literature can be found in Nicosia et al. (2021).

In this paper, we perform a comprehensive algorithmic treatment of the minimization of the weighted number of late jobs. Our contributions can be summarized as follows: After giving the definition of the problem of *rescheduling jobs with a LIFO buffer in order to minimize the weighted number of late jobs* in Sect. 2, we propose ILP formulations in Sect. 3. One is based on decision variables for every move of a job, while the other utilizes assignment variables for placing jobs at their new position. In the former model, the number of constraints is asymptotically larger than in the latter model by one or two orders of magnitude. However, it turns out that computationally, the latter, more compact model is hardly tractable and is clearly outstripped by the more capacious move-based model.

In Sect. 4, we develop a new realization of a dynamic programming algorithm. While the previous dynamic program from Nicosia et al. (2021) turns out to be hardly applicable even for moderate-sized test instances, our new algorithm employs dominance criteria and an efficient handling of lists of states, resulting in a competitive solution method.

While dynamic programming can be seen as a bottom-up strategy, we also develop a branch-and-bound algorithm following a top-down approach. As described in Sect. 5, we use a branching rule based on moves of jobs. Note that the structure of the rescheduling problem requires particular care to avoid the multiple treatment of identical subproblems. The lower bound is derived from identifying jobs that can never be on time in the considered subproblem.

Section 6 reports on experimental results with all the presented algorithms. It turns out that the new dynamic programming algorithm and the branch-and-bound algorithm are both relevant solution strategies, while the ILP models and the previous dynamic program are not. Among the two remaining approaches, branch-and-bound dominates for smaller stack sizes while dynamic programming takes the lead for stacks with larger capacities. Finally, concluding remarks and suggestions for future research are given in Sect. 7.

2 Problem definition

In this section, we give a formal problem definition of our rescheduling problem and introduce terminology and notation. We mostly follow the description of Nicosia et al. (2021) where the problem was originally introduced. A deterministic single-machine environment is considered where a given set J of n jobs has to be scheduled on a single machine in order to minimize the weighted number of late jobs. In the following, the standard notation for scheduling problems is adopted: For each job $j \in J$, its processing time, its due date, and its weight are denoted by p_j, d_j, and w_j, respectively. It is assumed that all these quantities are given as positive integers. Furthermore, for each job $j \in J$, the completion time of job j in a schedule σ is denoted by $C_j(\sigma)$, and its lateness in σ by $L_j(\sigma) = C_j(\sigma) - d_j$. To formulate the objective function, a binary variable $U_j(\sigma)$ is introduced which takes value 1 if $C_j(\sigma) > d_j$, i.e., job j is late in σ, and 0 otherwise. Additionally, for a schedule $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ with $\sigma_k \in J, k = 1, \ldots, n$, if $i \leq j$, the ordered set of jobs $\langle \sigma_i, \sigma_{i+1}, \ldots, \sigma_j \rangle$ is referred to as *subsequence* $\sigma(i, j)$.

Additionally, it is assumed that an initial sequence σ_0 of the n jobs of set J is given in which the jobs are numbered from 1 to n, i.e., $\sigma_0 = \langle 1, 2, \ldots, n \rangle$. This simply reveals that *job j is placed in the jth position of σ_0*. Hence, it holds for any $i, j \in J$ with $i \leq j$ that $\sigma_0(i, j) = \langle i, i+1, \ldots, j \rangle$. For any of these subsequences $\sigma_0(i, j)$, its total processing time and its total weight are indicated below as $P(i, j) = \sum_{k=i}^{j} p_k$ and $W(i, j) = \sum_{k=i}^{j} w_k$, respectively.

In the problem under consideration, a new job sequence σ is to be determined such that on the one hand, $\sum_{j \in J} w_j U_j(\sigma)$ is minimum, and on the other hand, σ can be derived from σ_0 by applying a number of so-called *feasible moves*.

In the scheduling environment considered here, any move is performed by a physical device, for example, a robot arm. This operates on the ordered sequence of jobs arriving at the machine, e.g., on a moving conveyor, where the initial sequence corresponds to σ_0. The rescheduling process then takes place as follows. The device picks up a job j and puts it on a *stack* with bounded capacity S. Next, it possibly picks up other jobs and places them on the stack too. The device can also pick jobs from the stack, according to a LIFO policy, and place them back at a proper position *later* in the sequence. In doing so, it is assumed that there is always enough space between the jobs even to deposit all jobs in the stack between any two jobs on the conveyor. It should be emphasized at this point that in this environment, a job can only be reinserted in a *later* position of the sequence, since the conveyor belt keeps moving forward while the robot arm operates at a fixed position before the machine. At the end of the rescheduling process, the stack must be empty and all jobs be put back in the sequence again.

Introducing further terminology, a *move* $i \rightarrow j, i < j$ consists of deleting job i from a given sequence and reinserting it immediately after all jobs of the subsequence $\sigma_0(i+1, j)$. Due to the stated LIFO policy, two moves $i_1 \rightarrow j_1$ and $i_2 \rightarrow j_2$ with $i_1 < i_2$ are feasible only if either $i_1 < j_1 < i_2 < j_2$ or $i_1 < i_2 < j_2 \leq j_1$. In the latter case, we define $i_2 \rightarrow j_2$ to be *nested in* move $i_1 \rightarrow j_1$. Additionally, since the problem setting stipulates that there must not be more than S moves nested inside each other, each move is performed on a specific *level* in the following sense: a move that does not contain nested moves is said to be at level 1. Recursively, a move is at level $\ell > 1$ if ℓ is the smallest value such that it contains feasible nested moves at level up to $\ell - 1$. To avoid ambiguity regarding the level of a move, a move $i \rightarrow j$ at level ℓ is referred to as (i, j, ℓ). For notational convenience, we also allow a move $i \rightarrow i$ for all levels ℓ, denoted as (i, i, ℓ), meaning that job i is not moved at all (in this case the level of this non-move is meaningless). Finally, a *feasible schedule* σ for the rescheduling problem with stack capacity S is a schedule that results from a set of feasible moves at levels up to a maximum of S starting from σ_0.

As pointed out in the introduction, Nicosia et al. (2021) considered the four possibly most common objective functions for this rescheduling problem. In this paper, we concentrate on finding a feasible schedule that *minimizes the weighted number of late jobs*. Following the standard

three-field classification scheme of Graham et al. (1979), the problem is succinctly encoded as $1|resch\text{-}LIFO|\sum w_j U_j$.

3 Mathematical model

In this section, we present two integer linear programming (ILP) formulations for our problem $1|resch\text{-}LIFO|\sum w_j U_j$. The first model introduces binary variables for possible moves, while the second one employs binary variables for the assignment of jobs to positions and auxiliary binary variables for the relative order of jobs. Note that both formulations have $O(n^2)$ variables (specifically, $\frac{n}{2}(n+1)$ and $2n^2$, respectively), but while in the first model $O(n^4)$ constraints are needed (we will also introduce a variant with $O(n^3)$ constraints), the number of constraints in the second one is only $O(n^2)$.

3.1 Move formulation

In the first ILP formulation, hereinafter referred to as *ILPmove1*, we associate with each move $i \to j, i, j \in J$ with $i < j$ a binary variable x_{ij} such that each variable x_{ij} takes value 1 if and only if move $i \to j$ is performed. Moreover, for each $j \in J$, binary variables U_j indicate whether or not job j is late in the final sequence σ. With these variables, the problem can be formulated as follows:

$$\text{minimze} \quad \sum_{j\in J} w_j U_j \tag{1}$$

$$\text{subject to} \quad \sum_{k=j+1}^{n} x_{jk} \le 1 \qquad \forall j \in J \tag{2}$$

$$x_{ij} + x_{kh} \le 1 \qquad \begin{array}{l}\forall i, j, k, h \in J,\\ i < k \le j < h\end{array} \tag{3}$$

$$\sum_{i=1}^{j-1}\sum_{k=j}^{n} x_{ik} \le S \qquad \forall j \in J \tag{4}$$

$$C_j(\sigma_0) - \sum_{i=1}^{j-1} p_i \sum_{k=j}^{n} x_{ik} + \sum_{k=j+1}^{n} \times P(j+1,k)\, x_{jk} - d_j \le M \cdot U_j \forall j \in J \tag{5}$$

$$x_{ij}, U_j \in \{0,1\} \qquad \forall i, j \in J,\ i < j \tag{6}$$

In (1), we minimize the weighted number of late jobs. While constraints (2) ensure that each job is moved at most once, (3) guarantees the LIFO policy. Constraints (4) limit to S the total number of jobs in the stack at any time and (5) ensure U_j to be 1 if job j is late in the final sequence σ (for a large enough constant M). More precisely, the left-hand side (LHS) of (5) reflects the lateness of job j in the final sequence: while the first term indicates the completion time of job j in σ_0, the second term expresses the decrease of the completion time resulting from all moves of jobs from $1, \ldots, j-1$ in which j is nested in the final sequence; the third term indicates the increase of the completion time of job j as a consequence of its own move, and the fourth term is simply the due date of job j.

One might see the $O(n^4)$ constraints (3) as a major burden on the computational performance of solving *ILPmove1*. Indeed, one may aggregate these constraints and replace (3) by constraints

$$\sum_{h=j}^{k} x_{ih} + \sum_{h=k+1}^{n} x_{jh} \le 1 \qquad \forall i, j, k \in J,\ i < j \le k. \tag{3'}$$

This yields a second version of the move-based ILP, which is referred to as *ILPmove2* in the following. At first glance, this replacement may seem advisable, as it reduces the number of constraints to $O(n^3)$. We investigated this presumption empirically (see Sect. 6) and found that *ILPmove2* outperforms *ILPmove1* for stack sizes larger than or equal to 4, 6, or 8 (depending on the particular test set), while for smaller stack sizes *ILPmove1* retains the lead. This is somewhat surprising, as the respective constraints are not directly related to the stack size.

3.2 Assignment formulation

The second ILP formulation is based on assignment variables matching the jobs of J, which are numbered according to the initial sequence σ_0, to new positions from $I = \{1, \ldots, n\}$. In detail, the model employs assignment variables $x_{jh} \in \{0, 1\}$ for $j \in J$, and $h \in I$ where $x_{jh} = 1$ indicates that job j is placed in position h in σ. We also use an additional set of variables: for all $k, h \in I$ with $k < h$, $y_{kh} \in \{0, 1\}$ indicates whether the job in position k has a larger index number than the job in position h. In other words, $y_{kh} = 1$ if and only if the job in position k was initially placed after the job in position h (while satisfying the LIFO constraint). Beyond that, the variables $z_{ij}, i, j \in J$ with $i < j$ indicate whether job i is placed at a larger position number than job j in the final sequence, i.e. $z_{ij} = 1$ if and only if job i is placed after job j in σ. As in the move formulation, variable U_j, $j \in J$ also indicates here whether or not job j is late in the final sequence σ.

The job in position h in the final sequence σ can be expressed as $\gamma(h) = \sum_{j\in J} j\, x_{jh}$, while the position of job j can be derived as $\pi(j) = \sum_{h\in I} h\, x_{jh}$. With these quantities, we get the following ILP formulation:

$$\text{minimze} \quad \sum_{j\in J} w_j U_j \tag{7}$$

$$\text{subject to} \quad \sum_{h\in I} x_{jh} = 1 \qquad \forall\, j \in J \tag{8}$$

$$\sum_{j\in J} x_{jh} = 1 \qquad \forall\, h \in I \tag{9}$$

$$z_{ij} \geq \frac{1}{n}\left(\pi(i) - \pi(j)\right) \quad \forall\, i, j \in J,\ i < j \tag{10}$$

$$z_{ij} - 1 \leq \frac{1}{n}\left(\pi(i) - \pi(j)\right) \quad \forall\, i, j \in J,\ i < j \tag{11}$$

$$y_{kh} \geq \frac{1}{n}\left(\gamma(k) - \gamma(h)\right) \quad \forall\, k, h \in I,\ k < h \tag{12}$$

$$y_{kh} - 1 \leq \frac{1}{n}\left(\gamma(k) - \gamma(h)\right) \quad \forall\, k, h \in I,\ k < h \tag{13}$$

$$y_{kh} \geq \frac{1}{n}\left(k - \gamma(h) + \sum_{l=h+1}^{n} y_{hl} + 1\right) \quad \forall\, k, h \in I,\ k < h \tag{14}$$

$$\sum_{h=k+1}^{n} y_{kh} \leq S \qquad \forall\, k \in I \tag{15}$$

$$\sum_{i=1}^{j-1} p_i(1 - z_{ij}) + p_j + \sum_{l=j+1}^{n} p_l z_{jl} - d_j \leq M \cdot U_j \qquad \forall\, j \in J \tag{16}$$

$$x_{jh},\ y_{kh},\ z_{ij},\ U_j \in \{0, 1\} \times \begin{array}{l}\forall\, i, j \in J,\ \forall\, k, h \in I,\\ i < j,\ k < h\end{array} \tag{17}$$

In (7), we minimize the weighted number of late jobs. Equations (8) and (9) are the usual assignment constraints. Constraints (10) and (11) guarantee that z_{ij} is 1 if $\pi(i) > \pi(j)$. In other words, variable z_{ij}, $i < j$, assumes value 1 if and only if job i is placed after job j in σ. (12) and (13) ensure y_{kh} to be 1 if $\gamma(k) > \gamma(h)$, meaning that variable y_{kh}, $k < h$, assumes value 1 if and only if the job in position k was initially placed after the job in position h. Constraints (14) ensure the LIFO policy. In fact, (14) is equivalent to the following implication:

$$\forall\, k, h \in I,\ k < h:\ \gamma(h) - \sum_{l=h+1}^{n} y_{hl} \leq k \implies y_{kh} = 1.$$

Because of the LIFO constraint, when moving job $\gamma(h)$ from its initial to its final position, there have to be jobs with a larger index number than $\gamma(h)$ in all positions nested in this move. However, since $\gamma(h)$ is in turn nested in $\sum_{l=h+1}^{n} y_{hl}$ moves, the positions of the entire subsequence are moved forward by this number. Therefore, we have to consider all positions k in the final sequence σ for $k \in \left\{\gamma(h) - \sum_{l=h+1}^{n} y_{hl}, \ldots, h-1\right\}$ and ensure that the index of the job placed in such a position k is larger than $\gamma(h)$, i.e., that all jobs in these positions were initially placed after the job in position h, which is expressed by $y_{kh} = 1$. Constraints (15) limit the total number of jobs in the stack at any given time to S. Finally, the LHS of inequality (16) reflects the lateness of job j in the final sequence where the completion time of j is composed of the first three expressions, namely, the sum of the processing times of all jobs that are in the initial sequence as well as in the final sequence located before job j, the processing time of job j, and the sum of the processing times of those jobs that succeed job j in σ_0 but precede it in the final sequence σ. Thus, (16) ensures U_j to be 1 if job j is late in the final sequence σ (for a large enough constant M).

4 Dynamic programming

A straightforward dynamic programming algorithm for $1|\textit{resch-LIFO}|\sum w_j U_j$ with five nested *for*-loops was given in Nicosia et al. (2021). In that paper, a more complex dynamic program was also developed and presented in detail for the unweighted problem of minimizing the number of late jobs. Moreover, an adaptation of this approach to the weighted case was briefly mentioned in Nicosia et al. (2021). For the sake of self-containment, we will first lay out the resulting dynamic programming algorithm *DynProStatic* for $1|\textit{resch-LIFO}|\sum w_j U_j$ in detail. This approach does not yield a practically usable algorithm (see the experiments in Sect. 6.2), but it serves as a starting point for a number of substantial algorithmic improvements leading to our considerably faster dynamic program *DynProImpr*.

4.1 Static dynamic program

This section provides a detailed description of the pseudo-polynomial dynamic programming algorithm for $1|\textit{resch-LIFO}|\sum w_j U_j$ which is briefly sketched in Nicosia et al. (2021). It is an extension of the algorithm presented in that paper for the unweighted case. For this purpose, let us first introduce the term *ℓ-rearrangement* to denote a feasible rearrangement of a subsequence that can be obtained by moves up to level ℓ. Moreover, all ℓ-rearrangements of a subsequence which lead to m for the weighted number of late jobs in it when the starting time is reduced by $\widetilde{s}'$ are *dominated* by the ℓ-rearrangements that lead to the same weighted number of late jobs m with a time shift of $\widetilde{s}'' < \widetilde{s}'$.

In our dynamic program, the *state* is defined as a quadruple (i, j, m, ℓ), where i and j identify a subsequence, m is the weighted number of late jobs in the subsequence, and ℓ is the level. With each state (i, j, m, ℓ), we associate the

minimum required reduction of the starting time of subsequence $\sigma_0(i, j)$ to reach the given weighted number of late jobs m with moves up to level ℓ. This piece of information is formally defined as follows:

Definition 1 For each $i, j \in J$ with $i \leq j$, $m = 0, \ldots, W(i, j) - 1$ and $\ell = 0, \ldots, S$, let $\widetilde{s}(i, j, m, \ell)$ be the *minimum decrease of the starting time* of the subsequence $\sigma_0(i, j)$ in order to ensure a value of at most m for the weighted number of late jobs in it when the subsequence can be rearranged with moves up to level ℓ.

At this point, we want to emphasize that $\widetilde{s}(i, j, m, \ell)$ may be both positive and negative where negative values, i.e., an increase of the starting time, occur if the weighted number of late jobs in the (rearranged) subsequence is less than m.

At level $\ell = 0$, the state values $\widetilde{s}(i, j, m, 0)$ are evaluated on the initial sequence using the lateness values of the jobs in it. First observe that although for each subsequence $\sigma_0(i, j)$ the states $\widetilde{s}(i, j, m, 0)$ are computed for $m = 0, \ldots, W(i, j) - 1$, i.e., for $W(i, j)$ different values of m, we only have $j - i + 1$ different values for $\widetilde{s}(i, j, \cdot, 0)$, namely, the lateness values $L_k(\sigma_0)$, $k \in \sigma_0(i, j)$.

This follows from the fact that any late job k in $\sigma_0(i, j)$ only becomes on time if the subsequence is moved forward by $L_k(\sigma_0)$. Through this starting time reduction, there is (at least) one late job less (i.e. job k), and the corresponding number m for the weighted number of late jobs reduces by the weight w_k. However, as the number m, i.e., the largest allowed weighted number of late jobs in the subsequence, changes, the minimum reduction of the starting time does not necessarily have to change too. Hence, w_k different states are assigned the same value $L_k(\sigma_0)$. Consequently, there exists (at most) one of these w_k states that has exactly the given number of weighted late jobs with a starting time reduction of $\widetilde{s}$.

In order to provide a formal initialization, we use the notation $\max_{[q]}(\mathcal{L})$ to indicate the qth largest value in the multi-set $\mathcal{L}$ which corresponds to the qth entry in the non-increasingly sorted multi-set. Additionally, we use $\uplus$ to denote a union of multi-sets where the multiplicity of each element is the sum of its multiplicities in each single multi-set. Formally, the initialization for all $i, j \in J, i \leq j$ and for all $m = 0, \ldots, W(i, j) - 1$ is then as follows:

$$\widetilde{s}(i, j, m, 0) = \max_{[m+1]} \left(\biguplus_{k \in \sigma_0(i,j)} \biguplus_{u=0}^{w_k - 1} \{L_k(\sigma_0)\} \right).$$

For each level $\ell \geq 1$, the recursive extension rule is given as follows. For every given subsequence $\sigma_0(i, j)$, level ℓ, and number m, the state value $\widetilde{s}(i, j, m, \ell)$ is obtained by taking the minimum reduction of the starting time over all ℓ-rearrangements of $\sigma_0(i, j)$ that yield at most m for the weighted number of late jobs in it. For this reason, we compute for each possible move (i, k, ℓ'), and $\ell' \leq \ell$, the multi-set $\widetilde{\mathcal{S}}(i, j, k, \ell)$ that contains the minimum reduction of the starting time for all weighted numbers of late jobs, that is, $m = 0, \ldots, W(i, j) - 1$:

$$\widetilde{\mathcal{S}}(i, j, k, \ell) = \biguplus_{u=0}^{W(i+1,k)-1} \{\widetilde{s}(i + 1, k, u, \ell - 1) - p_i\} \uplus \biguplus_{u=0}^{W(k+1,j)-1} \{\widetilde{s}(k + 1, j, u, \ell)\} \uplus \biguplus_{u=0}^{w_i - 1} \{C_k(\sigma_0) - d_i\}. \quad (18)$$

As can be seen from (18), the multi-set $\widetilde{\mathcal{S}}(i, j, k, \ell)$ is composed of three multi-sets representing the following three job subsets: the jobs nested in the move (i, k, ℓ'), the jobs from $\sigma_0(k + 1, j)$, and the moved job k. The values from the first multi-set are computed from the non-dominated $(\ell - 1)$-rearrangements of the subsequence $\sigma_0(i + 1, k)$ where the starting time of all jobs is reduced by p_i time units due to move $i \to k$. The second multi-set contains the $\widetilde{s}(\cdot)$ values of $\sigma_0(k + 1, j)$ where the starting time of the corresponding subsequence is not affected by the move. Finally, the third term corresponds to the multi-set that contains w_i times the lateness of job i after move $i \to k$ is performed.

Recall that the multi-set $\widetilde{\mathcal{S}}(i, j, k, \ell)$ contains the minimum starting time reductions for all $m = 0, \ldots, W(i, j) - 1$. Accordingly, the $(m + 1)$-largest element of the multi-set represents the necessary time units by which $\sigma_0(i, j)$ needs to be moved forward after a move (i, k, ℓ') with $\ell' \leq \ell$, to have a value of m for the weighted number of late jobs. Hence, the value of state $\widetilde{s}(i, j, m, \ell)$ corresponds to that move $i \to k$ for which the corresponding decrease of the starting time is minimal. Consequently, the new state value can be computed as follows:

$$\widetilde{s}(i, j, m, \ell) = \min_{k \in \sigma_0(i,j)} \left\{ \max_{[m+1]} \left(\widetilde{\mathcal{S}}(i, j, k, \ell) \right) \right\} \quad (19)$$

Finally, the optimal solution value, $\sum_{j \in J} w_j U_j(\sigma^*)$, is given by $\min\{m : \widetilde{s}(1, n, m, S) \leq 0\}$.

Hereinafter, we present a straightforward implementation of the algorithm sketched above which is illustrated in Algorithm 1. In the pseudocode, we can see that the states are generated from level $\ell = 0$ up to S, since the recursion requires that the effects of moves nested in any move of level ℓ have already been evaluated beforehand. Additionally, for each level $\ell = 1, \ldots, S$, the subsequences need to be computed in a specific order to guarantee that when move (i, k, ℓ) is examined, the evaluation of the effects of all moves (i', k', ℓ) with $k < i' \leq k'$ has already taken place. Therefore, in an outer loop (Steps 9–21), the index j runs from 1 up to n, while in an inner loop the index i runs from j down to 1 (Steps 10–21). For any given subsequence $\sigma_0(i, j)$ and

any level ℓ, we compute for each $k = i, \ldots, j$ the multi-set $\widetilde{\mathcal{S}}(i, j, k, \ell)$. The union of the three multi-sets from which $\mathcal{S}(i, j, k, \ell)$ is composed (see Eq. (18)) is temporally stored in the list $\widetilde{\mathcal{S}}_k$ (Steps 12–18). Note that the procedure *Append* used in these steps is used for simply adding an element to the end of a list.

Algorithm 1 Dynamic Program: *DynProStatic*

```
1: procedure INITIALIZATION (for ℓ = 0)
2:   for i = 1, ..., n do
3:     for j = i, ..., n do
4:       Let S̃ be a list with values L_k(σ_0), k = i, ..., j, where each L_k(σ_0) occurs w_k times
5:       for m = 0, ..., W(i, j) − 1 do
6:         s̃(i, j, m, 0) ← (m + 1)-largest value of the list S̃
7: procedure RECURSION (for ℓ ≥ 1)
8:   for ℓ = 1, ..., S do
9:     for j = 1, ..., n do
10:      for i = j, ..., 1 do
11:        for k = i, ..., j do
12:          Let S̃_k be an empty list
13:          for u = 0, ..., W(i + 1, k) − 1 do
14:            Append s̃(i + 1, k, u, ℓ − 1) − p_i to the list S̃_k
15:          for u = 0, ..., W(k + 1, j) − 1 do
16:            Append s̃(k + 1, j, u, ℓ) to the list S̃_k
17:          for u = 0, ..., w_i − 1 do
18:            Append C_k(σ_0) − d_i to the list S̃_k
19:            for m = 0, ..., W(i, j) − 1 do
20:              Let S̃ be a list with the (m + 1)-largest values of S̃_i, ..., S̃_j
21:              s̃(i, j, m, ℓ) ← min(S̃)
22:  return min{m : s̃(1, n, m, S) ≤ 0}
```

Obviously, Algorithm 1 has a pseudo-polynomial running time of $O(n^4 W)$ where $W = W(1, n) = \sum_{j \in J} w_j$ is the sum of the weights of all jobs.

4.2 Improved dynamic program

In order to present an improved implementation of the dynamic programming algorithm, we take a closer look at the properties of the states $\widetilde{s}(i, j, m, \ell)$ computed in *DynProStatic*. From these, we can derive the following observations which will then lead to the improved version.

Observation 2 *For decreasing m, the values of $\widetilde{s}(i, j, m, \ell)$ are non-decreasing.*

In other words, to achieve a smaller weighted number of late jobs, it will be necessary to move the subsequence forward by a larger (or at least not smaller) amount of time.

Observation 3 *Let m' be the minimum weighted number of late jobs that can be achieved by an optimal ℓ-rearrangement of the subsequence $\sigma_0(i, j)$ with fixed starting time $C_{i-1}(\sigma_0)$. Then it holds that $\widetilde{s}(i, j, m, \ell) \leq 0$ for all $m \geq m'$.*

The statement that $\widetilde{s}(i, j, m', \ell) \leq 0$ follows from the definition of a state. To obtain more late jobs than necessary, one would have to delay the subsequence which implies non-positive $\widetilde{s}(i, j, m, \ell)$ values for $m > m'$ (cf. Observation 2).

Note that after performing the move (i, j, ℓ), the resulting (already rearranged) subsequence cannot be split by any move at level $\ell' > \ell$ due to the LIFO constraint. However, if it is nested in a higher-level move, it will be moved forward as a whole. Therefore, all non-positive $\widetilde{s}(i, j, m, \ell)$ values, except for the one with the smallest m argument among them, are of no interest since they entail delaying the corresponding subsequence, and their computation can be skipped in order to save time. Consequently, for each $\sigma_0(i, j)$ and ℓ, we only record non-negative state values and the first non-positive one, but we disregard states with larger m corresponding to non-positive state values.

However, we will not consider all positive states either, as there is a natural limit how far a subsequence can be moved forward. In particular, after having performed moves up to level ℓ, for a job at position i at most $S - \ell$ preceding jobs can be moved behind i in the remaining levels. This yields the following bound.

Observation 4 *The rearrangement of a subsequence $\sigma_0(i, j)$ produced by moves up to level ℓ cannot be moved forward by more than the processing time of $S - \ell$ jobs preceding it (or even fewer if $i \leq S - \ell$). Therefore, the possible reduction of its starting time is limited by the sum of processing times of jobs $1, \ldots, i - 1$ and by the sum of the $S - \ell$ largest processing times. Formally, the reduction is bounded by*

$$a(i, \ell) = \sum_{q=1}^{\min\{S-\ell, i-1\}} \max_{[q]} (\{p_k : k \in \sigma_0(1, i-1)\}) .$$

It follows immediately from Observation 4 that if there is some state with $\widetilde{s}(i, j, m, \ell) > a(i, \ell)$, then none of the states $\widetilde{s}(i, j, m', \ell)$ with $m' \leq m$ can be reached by any set of feasible moves. Consequently, these states do not have to be considered in the algorithm.

Hence, in order to reduce the number of state values that have to be calculated, it is useful to precompute the threshold values $a(i, \ell)$ for each job $i \in J$ and each level $\ell = 0, \ldots, S$. This is done by Algorithm 2. It builds a decreasingly sorted list A of the S largest processing times before each job i. If $i - 1 < S$, the missing elements are set to zero. Every time i is incremented by 1, p_{i-1} is inserted in A, and the smallest element in A is discarded (while $i \leq S$, this element will be 0). For each fixed i, $a(i, S)$ is always equal to zero, and the threshold values for each level $\ell < S$ can be easily obtained from the values at level $\ell + 1$ by adding the next largest element from A. The complexity of the whole procedure is $O(nS)$.

Algorithm 2 Computation of the thresholds $a(i, \ell)$

```
1: procedure THRESHOLD
2:    Let A be a list of length S with value 0 in each position
3:    for ℓ = 0, ..., S do
4:       a(1, ℓ) ← 0
5:    for i = 2, ..., n do
6:       Insert p_{i−1} into list A such that A is sorted in non-increasing order
7:       Remove the smallest element of A
8:       a(i, S) ← 0
9:       for ℓ = S − 1, ..., 0 do
10:         a(i, ℓ) ← a(i, ℓ + 1) + A(S − ℓ)
```

A further reduction of nonrelevant states can be obtained by the following dominance principle.

Definition 5 A state $\widetilde{s}(i, j, m', \ell)$ is m-dominated by $\widetilde{s}(i, j, m'', \ell)$ if both state values are equal, i.e., $\widetilde{s}(i, j, m', \ell) = \widetilde{s}(i, j, m'', \ell)$, but $m'' < m'$.

In our improved dynamic programming algorithm, we only record non-m-dominated states. This dominance relation between states enables us to significantly reduce the number of states to be considered which results in a crucial improvement in terms of running time.

Another observation which will give rise to an improved implementation concerns the number of moves that have to be examined when calculating $\widetilde{s}(i, j, m, \ell)$. Recall that for every given number m, the state value $\widetilde{s}(i, j, m, \ell)$ is obtained by taking the minimum reduction of the starting time over all possible moves (i, k, ℓ') for $k = i, \ldots, j$ at some level $\ell' \leq \ell$ that produce (at most) m late jobs in $\sigma_0(i, j)$. However, the following observation shows that it is not necessary to consider *all* moves (i, k, ℓ'), $k = i, \ldots, j$.

Observation 6 *In order to optimally rearrange the subsequence $\sigma_0(i, j)$ with moves up to level ℓ, a move (i, k, ℓ') at some level $\ell' \leq \ell$ with $i < k$ is only worthy of examination if job k is late.*

Proof Suppose job k is on time with respect to the given starting time $C_{i-1}(\sigma_0)$ of $\sigma_0(i, j)$. For the weighted lateness of job k, move (i, k, ℓ') does not lead to any improvement compared to move $(i, k-1, \ell')$, since job k is not late in either case. The weighted lateness of job i is obviously smaller if move $(i, k-1, \ell')$ is performed than if move (i, k, ℓ') is executed. While for the weighted lateness of the jobs in $\sigma_0(i+1, k-1)$ the effects of both moves are the same, the weighted lateness for jobs in $\sigma_0(k+1, j)$ is smaller (or at least not greater) for move $(i, k-1, \ell')$ compared to move (i, k, ℓ'). Hence, a move (i, k, ℓ') is worth examining only if job k is late.

In the following, we will present the algorithm *DynProImpr*, an improved implementation of dynamic programming that takes all observations stated above into account. For this purpose, we define four operations for handling sorted lists $\mathcal{L}$ of tuples $v = (v_1, v_2)$: *Append* v to $\mathcal{L}$ adds the tuple v at the end of list $\mathcal{L}$; *Head*$(\mathcal{L})$ returns the first tuple of $\mathcal{L}$; *Tail*$(\mathcal{L})$ returns the last tuple of $\mathcal{L}$; and *DeleteHead*$(\mathcal{L})$ deletes the first element of $\mathcal{L}$. Furthermore, we will *Merge* two or more sorted lists into a single sorted list (which can be done in linear time). Last but not least, the function *LessOrEqual*$((v_1', v_2'), (v_1'', v_2''))$ returns *true* if and only if $v_1' \leq v_1''$ and $v_2' \leq v_2''$.

Taking into account Observation 2, for a fixed subsequence $\sigma_0(i, j)$ and a fixed level ℓ, the values $\widetilde{s}(i, j, m, \ell)$ form a (not strictly) decreasing sequence in m. Naturally, the sequence can be restricted to $m \leq W(i, j) - 1$. The first part of this sequence consists (possibly) of values greater $a(i, \ell)$ which are irrelevant by Observation 4. State values $\widetilde{s}(i, j, m, \ell)$ smaller or equal to $a(i, \ell)$ and a $m \geq \overline{m} = \min\{m : \widetilde{s}(i, j, m, \ell) \leq 0\}$ are non-dominated and therefore relevant and will be stored in a list $\theta(i, j, \ell)$. Finally, non-positive state values except for the first one, that is $\widetilde{s}(i, j, \overline{m}, \ell)$, will be simply ignored (cf. Observation 3).

As already mentioned, for each subsequence $\sigma_0(i, j)$ and for each level ℓ, we record an ordered list of non-m-dominated states denoted by $\theta(i, j, \ell)$. Each element of such a list is described by a tuple (m, s) and represents an ℓ-rearrangement with a total weighted number of late jobs of m that can be reached by a minimum advance s for the subsequence $\sigma_0(i, j)$, considering rearrangements up to level ℓ. The list $\theta(i, j, \ell)$ is sorted in decreasing order of the cost value s, and the elements (or tuples) in it represent a Pareto frontier of possible outcomes for the subsequence $\sigma_0(i, j)$. The endpoints of each such list $\theta(i, j, \ell)$ are given by $(0, \max\{\max_{k \in \sigma_0(i,j)}\{L_k(\sigma_0)\}, 0\})$ and $(\sum_{k \in \sigma_0(i,j): L_k(\sigma_0) > 0} w_k, 0)$.

Algorithm 3 Dynamic program: *DynProImpr*

```
1: procedure INITIALIZATION (for ℓ = 0)
2:    Let L be a list of all late jobs, sorted in non-increasing order of its lateness values in σ_0
3:    for i = 1, ..., n do
4:       for j = i, ..., n do
5:          M ← 0
6:          for all k ∈ L with i ≤ k ≤ j do
7:             if L_k(σ_0) ≤ a(i, 0) then
8:                if θ(i, j, 0) empty or [θ(i, j, 0) not empty and
                        LessOrEqual((M, L_k(σ_0)), Tail(θ(i, j, 0)))] then
9:                   Append the tuple (M, L_k(σ_0)) to the list θ(i, j, 0)
10:            M ← M + w_k
11:         Append (M, 0) to the list θ(i, j, 0)
```

Procedure INITIALIZATION of Algorithm 3 settles the case $\ell = 0$. At this level, the values of $\theta(i, j, 0)$ are evaluated on the initial subsequence $\sigma_0(i, j)$ using the lateness values of the jobs in it. Accordingly, we first create a list $\mathcal{L}$ of all

late jobs in σ_0, sorted in non-increasing order of the corresponding lateness values (Step 2). Subsequently, to span all subsequences $\sigma_0(i, j)$, two loops over all indices i and j are performed. For each subsequence $\sigma_0(i, j)$, the list is scanned from high to low lateness values in Steps 6–10: each late job k between i and j that is non-m-dominated (Step 8) originates a record that is inserted in $\theta(i, j, 0)$ (Step 9). The record has a time value m equal to the lateness of job k and a cost value s equal to the accumulated weights of all jobs with lateness not smaller than $L_k(\sigma_0)$. Note that the last entry in $\theta(i, j, 0)$ indicates the cost, i.e., the total weighted number of late jobs, of the subsequence with no time shift (Step 11).

For each subsequence $\sigma_0(i, j)$ and for each level $\ell = 1, \ldots, S$, the list $\theta(i, j, \ell)$ of non-m-dominated states corresponding to $\sigma_0(i, j)$ is generated as follows. All relevant insertion positions $k \in \sigma_0(i, j)$ (see Observation 6) for job i are considered. For each move (i, k, ℓ), its effects on the subsequence $\sigma_0(i + 1, k)$ are computed from all non-m-dominated states in $\theta(i + 1, k, \ell - 1)$, knowing that the whole subsequence $\sigma_0(i + 1, k)$ is advanced by p_i owing to the move. No effects occur on subsequence $\sigma_0(k + 1, j)$ whose non-m-dominated states are stored in $\theta(k + 1, j, \ell)$. The effect on job i can be computed directly: its new lateness value is $C_k(\sigma_0) - d_i$. These three lists of states are composed to produce the list of states that can be reached in $\sigma_0(i, j)$ with move (i, k, ℓ). The *compose* procedure is described in Loop 1 of Algorithm 3. After computing the composed lists $\mathcal{R}_k$ for all relevant $k \in \sigma_0(i, j)$, these lists $\mathcal{R}_k$ are merged, and only non-m-dominated states are kept (Observation 5), which leads to the lists $\theta(i, j, \ell)$. The *merge* procedure is outlined in Loop 2 of Algorithm 3.

In the main procedure RECURSION of Algorithm 3, we compute for each level $\ell = 1$ up to S and every subsequence $\sigma_0(i, j)$ the list $\theta(i, j, \ell)$ in two loops. In Loop 1, we compute lists $\mathcal{R}_k$ for $k = i$ and every late job $k \in \sigma_0(i + 1, j)$ (cf. Observation 6). For this purpose, we first set up three temporary sorted lists $\mathcal{R}_{k,1}$, $\mathcal{R}_{k,2}$, and $\mathcal{R}_{k,3}$ in Steps 18–31, resembling the three components in (18). In detail, $\mathcal{R}_{k,1}$ consists of all those pairs of $\theta(i + 1, k, \ell - 1)$ where the s-entry is reduced by p_i and thereby remains positive. Additionally, $\mathcal{R}_{k,1}$ contains the first pair with a non-positive reduced s-value, where the s-value is replaced by 0 for the sake of simplicity (Steps 20–25). $\mathcal{R}_{k,2}$ equals $\theta(k+1, j, \ell)$ (Step 27), and the components of list $\mathcal{R}_{k,3}$ depend on the new lateness value of the moved job k: if $C_k(\sigma_0) - d_i > 0$, then $\mathcal{R}_{k,3}$ consists of the two states $(0, C_k(\sigma_0) - d_i)$ and $(w_i, 0)$; otherwise the list contains only the tuple $(0, 0)$ (Steps 28–31). In Steps 34–40, the lists $\mathcal{R}_{k,1}$, $\mathcal{R}_{k,2}$, and $\mathcal{R}_{k,3}$ are merged in order to get the sorted list $\mathcal{R}_k$. Note that if the s argument is larger than $a(i, \ell)$ (Step 37), the state related to the tuple is not included in $\mathcal{R}_k$, since it cannot be reached in any case. Finally, in Steps 41–48, the list $\theta(i, j, \ell)$ is created by keeping all non-m-dominated states from the merged lists $\mathcal{R}_k$, $k \in \mathcal{K}$ and removing all m-dominated states: after identifying a non-m-dominated state (M, ζ) (Steps 42–44), it is added to $\theta(i, j, \ell)$ (Step 45). Afterwards, all states from all lists $\mathcal{R}_k$, $k \in \mathcal{K}$ that are m-dominated by (M, ζ) are discarded (Steps 46–48). Whether the states are m-dominated or not is indicated by a call to the function *LessOrEqual*.

Algorithm 3 Dynamic program: *DynProImpr*

```
12: procedure RECURSION (for ℓ ≥ 1)
13:   for ℓ = 1, ..., S do
14:     for j = 1, ..., n do
15:       for i = j, ..., 1 do
16:         Let K be a set containing job i and all late jobs from {i + 1, ..., j}
17:         for all k ∈ K do                                   ▷ Loop 1
18:           Let R_{k,1}, R_{k,2} and R_{k,3} be empty lists
19:           if k ≠ i then                                    ▷ Fill R_{k,1}
20:             for all (m, s) ∈ θ(i + 1, k, ℓ − 1) do
21:               if s − p_i > 0 then
22:                 Append (m, s − p_i) to the list R_{k,1}
23:               else
24:                 Append (m, 0) to the list R_{k,1}
25:                 break
26:           if k ≠ j then                                    ▷ Fill R_{k,2}
27:             Append each tuple (m, s) of θ(k + 1, j, ℓ) to the list R_{k,2}
28:           if C_k(σ_0) − d_i > 0 then                       ▷ Fill R_{k,3}
29:             Append (0, C_k(σ_0) − d_i) and (w_i, 0) to the list R_{k,3}
30:           else
31:             Append (0, 0) to the list R_{k,3}
32:           Let R_k be an empty list    ▷ Merge lists R_{k,1}, R_{k,2} and R_{k,3}
33:           Let P ⊆ {1, 2, 3} such that R_{k,p} is not empty for all p ∈ P
34:           while |R_{k,1}| + |R_{k,2}| + |R_{k,3}| ≥ |P| do
35:             p* ← arg max_{p∈P} {Head(R_{k,p}).s}
36:             ζ ← Head(R_{k,p*}).s
37:             if ζ ≤ a(i, ℓ) then
38:               M ← Σ_{p∈P} Head(R_{k,p}).m
39:               Append (M, ζ) to the list R_k
40:             DeleteHead(R_{k,p*})
41:         while (R_k ≠ ∅ holds for all k ∈ K) do             ▷ Loop 2
42:           M ← min_{k∈K} {Head(R_k).m}
43:           K̄ ← {k ∈ K : Head(R_k).m = M}
44:           ζ ← min_{k∈K̄} {Head(R_k).s}
45:           Append (M, ζ) to the list θ(i, j, ℓ)
46:           for all k ∈ K do
47:             while R_k ≠ ∅ and LessOrEqual ((M, ζ), Head(R_k)) do
48:               DeleteHead(R_k)
49:   return Head(θ(1, n, S)).m
```

As Algorithm 3 is far from trivial, we give a detailed numerical example in Appendix 1 to illustrate the steps of the improved dynamic programming recursion.

It remains to analyze the computational complexity of Algorithm 3. It is easy to determine that the worst-case computational complexity of procedure INITIALIZATION is $O(n^3)$. Since this is not the bottleneck of the complex-

ity of the algorithm, we did not study any specialized data structure to possibly improve its complexity. In RECURSION, Loop 1 (Steps 17–40) and Loop 2 (Steps 41–48) are executed $O(n^2 S)$ times, namely, over all pairs (i, j) and for all levels ℓ. Each loop is executed $O(n)$ times. Since for the tuples in $\theta(\cdot)$, the m-argument is bounded by $W(i, j)$, and the s-argument is bounded by the sum of the S largest processing times, each iteration in Loop 1 and Loop 2 takes $O(B)$ time where $B = \min\left\{\sum_{j\in J} w_j, \sum_{q=1}^{S} \max_{[q]}(\{p_j : j \in J\})\right\}$. Thus, the overall worst-case time complexity of Algorithm 3 is $O(n^4 B)$.

The above discussion can be summed up in the following theorem, which improves upon the complexity of Algorithm 1 for large weight coefficients.

Theorem 7 *Problem* $1|resch\text{-}LIFO|\sum w_j U_j$ *is pseudo-polynomial solvable within* $O(n^4 B)$ *where* $B = \min\left\{\sum_{j\in J} w_j, \sum_{q=1}^{S} \max_{[q]}(\{p_j : j \in J\})\right\}$.

5 Branch-and-bound

As an alternative to the dynamic programming approaches which might become impractical for larger data coefficients, in particular for large weights, we devised a combinatorial branch-and-bound algorithm *CombB&B* for $1|resch\text{-}LIFO|\sum w_j U_j$. A preceding variant of the algorithm is described in Polimeno (2019). Hereafter, we discuss the branching strategy and bounding procedures as well as the search technique we employed.

5.1 Branching strategy

While dynamic programming works bottom-up since states at lower levels are computed first, branch-and-bound works top-down so that moves at higher levels correspond to earlier branching decisions.

This processing strategy, however, does not allow the use of the definition of a move at level ℓ (recall that ℓ is the smallest value such that the move contains feasible nested moves at levels up to $\ell-1$) since we do not know in advance the moves nested in it (due to the top-down approach) and therefore also not level ℓ. Thus, further terminology is necessary.

Definition 8 A move that does not contain nested moves is said to be at branching-level (B-level) 1. Recursively, a move is at B-level $\ell > 1$ if it can contain feasible nested moves up to a B-level of $\ell - 1$.

Definition 9 A move $i \rightarrow j$ at B-level ℓ is denoted as branching-move (B-move) (i, j, ℓ). For notational convenience, we also define B-moves (i, i, ℓ) for every B-level ℓ, meaning that job i is not moved at all.

Observe that

- a move at B-level ℓ can but does not necessarily have to contain $\ell - 1$ moves nested inside one another, whereas for a move at level ℓ it must contain $\ell - 1$ moves nested inside each other;
- a B-move (i, j, ℓ) is itself nested in $S - \ell$ B-moves;
- for a move at B-level ℓ, there always exists an $\ell' \leq \ell$ such that it is a move at level ℓ'.

The observation presented next is a direct result of the LIFO constraint and serves as a basis for the branching policy.

Observation 10 *When a* B*-move* (h, k, ℓ) *is performed within a subsequence* $\sigma_0(i, j)$, $i \leq h \leq k \leq j$, *the subsequence* $\sigma_0(i, j)$ *is divided into at most three parts (less only if* $h = i$ *and/or* $k = j$*), namely,* $\sigma_0(i, h-1)$, $\sigma_0(h, k)$, *and* $\sigma_0(k+1, j)$, *such that for all feasible* B*-moves* (h', k', ℓ') *within* $\sigma_0(i, j)$ *with* $\ell' \leq \ell$ *jobs* h' *and* k' *must belong to the same part. Furthermore, the rearrangement of any of the (at most) three resulting subsequences does not affect the starting time of the others.*

When a subsequence $\sigma_0(i, j)$ is examined, all moves in which it is nested have already been decided due to the top-down approach, and therefore, its starting time is known and does not change subsequently. Hence, we can define a subproblem in the branch-and-bound tree as follows:

Definition 11 A *subproblem* in the branch-and-bound tree corresponds to a quadruple (i, j, ℓ, μ) where the subsequence $\sigma_0(i, j)$ is to be optimally rearranged with moves up to a B-level of ℓ with a starting time $C_{i-1}(\sigma_0) - \mu$; i.e., $\mu \geq 0$ is the reduction in the starting time of the subsequence compared to its initial starting time.

When examining a subproblem (i, j, ℓ, μ), the starting time and the residual number of levels ℓ are fixed, and therefore there is a natural limit on how far a job $k \in \sigma_0(i, j)$ can be moved forward by rearranging the subsequence $\sigma_0(i, j)$. In detail, at most $k - i$ preceding jobs can be moved behind k in the remaining B-levels, so that the following threshold arises.

Observation 12 *A job* k *of the subproblem* (i, j, ℓ, μ) *cannot be moved forward by more than the processing time of* $k - i$ *jobs preceding it (or even fewer if* $\ell \leq k - i$*). Therefore, the possible reduction of its starting time is limited by the quantity*

$$\alpha(i, k, \ell) = \sum_{q=1}^{\min\{\ell, k-i\}} \max_{[q]}\left(\{p_h : h \in \sigma_0(i, k-1)\}\right).$$

Note that $\alpha(\cdot)$ and the threshold $a(\cdot)$ (defined in Sect. 4.2) are closely related to each other, since $\alpha(1, k, \ell) = a(k, S - \ell)$. Consequently, because the computation of $\alpha(\cdot)$ can be done in a similar way as the one for $a(\cdot)$ (see Algorithm 2), we do not provide a detailed description for computing the threshold values at this point.

Whether or not a late job k in a considered subproblem (i, j, ℓ, μ) can be made on time with a set of feasible moves, depends on the value $\alpha(i, k, \ell)$. The corresponding definition is given below.

Definition 13 Consider a subproblem (i, j, ℓ, μ). A job $k \in \sigma_0(i, j)$ is called *repairable* if it fulfills the following two properties:

(i) without rearranging the subsequence $\sigma_0(i, j)$ job k is late, i.e., $C_k(\sigma_0) - \mu > d_k$;
(ii) the maximal possible reduction of the starting time $\alpha(i, k-1, \ell)$ of job k in $\sigma_0(i, j)$ is larger than or equal to the current lateness of k, i.e., $\alpha(i, k-1, \ell) \geq C_k(\sigma_0) - \mu - d_k$.

Note that if part (ii) of Definition 13 does not hold, job k is late regardless of how the subsequence $\sigma_0(i, j)$ is subsequently rearranged.

With the definition above, we are now able to state an observation that is of fundamental importance to limit the number of branches that need to be examined in the branch-and-bound algorithm.

Observation 14 *There exists an optimal solution of the subproblem (i, j, ℓ, μ) where all performed B-moves (h, k, ℓ), $i \leq h < k \leq j$ satisfy that job k is repairable.*

Proof Suppose job k is not late with respect to the given starting time $C_{i-1}(\sigma_0) - \mu$ of the subsequence $\sigma_0(i, j)$.

For the lateness of job k, any B-move (h, k, ℓ) with $h < k$ does not yield an improvement compared to the B-move $(h, k-1, \ell)$, since job k is not late in either case. For the lateness of job h, B-move $(h, k-1, \ell)$ is obviously preferable to B-move (h, k, ℓ), while for the lateness of the jobs in $\sigma_0(i, h-1)$, $\sigma_0(h+1, k-1)$, and $\sigma_0(k+1, j)$, the effects of the two B-moves are the same. Thus, a B-move $(h, k-1, \ell)$ should be preferred over a B-move (h, k, ℓ) for any $h < k$ if k is on-time.

Contrarily, suppose that job k is late but not repairable. For the lateness of job k, any B-move (h, k, ℓ) with $h < k$ cannot yield any improvement compared to B-move $(h, k-1, \ell)$, since job k is late in any case. For the other jobs, the same arguments apply as above. Hence, a B-move $(h, k-1, \ell)$ should be preferred over a B-move (h, k, ℓ) for any $h < k$ if job k is late but not repairable.

Consequently, there exists an optimal solution of the subproblem (i, j, ℓ, μ) that does not contain any B-moves (h, k, ℓ) for non-repairable jobs k.

After these preparations, we can define the branching rule of our algorithm. From any given subproblem (i, j, ℓ, μ), several subproblems are generated as follows: on the basis of Observation 14, we choose job k so that among all repairable jobs, k is the one with the largest index number, meaning that among all repairable jobs it is placed furthest behind. After fixing job k, all possible jobs $h \in \sigma_0(i, k)$ that can be moved after job k are considered. Thus, a branching decision in a subproblem (i, j, ℓ, μ) of the branch-and-bound tree is imposed by a B-move (h, k, ℓ) with $i \leq h \leq k \leq j$ such that each subproblem is divided into $k - i + 1$ subproblems.

As a result of the choice of k, the weighted lateness of $\sigma_0(k+1, j)$ is known without further branching decisions, and only $\sigma_0(i, h-1)$ and $\sigma_0(h+1, k)$ have to be examined further. Thus, we can see that Observation 14 is essential for limiting the number of branches that need to be examined in the branch-and-bound algorithm.

Each branching decision imposed by a B-move (h, k, ℓ) induces four different cost terms: the optimal costs $z^{(1)}$ of subsequence $\sigma_0(i, h-1)$ obtained by a feasible rearrangement with moves up to B-level ℓ, the optimal costs $z^{(2)}$ of subsequence $\sigma_0(h+1, k)$ obtained by a feasible rearrangement with moves up to B-level $\ell - 1$, the costs $z^{(3)}$ of the moved job h in its new position, and the optimal costs $z^{(4)}$ of subsequence $\sigma_0(k+1, j)$ (without any rearrangement). When all four values are known, they are summed up to get the optimal solution value z of the subproblem (i, j, ℓ, μ).

The recursive procedure $Branch$, outlined in Algorithm 4, takes the quadruple (i, j, ℓ, μ) as input and returns the optimal solution value z of the corresponding subproblem. When a subproblem (i, j, ℓ, μ) is examined, a loop runs from j down to i, until a repairable job k is found (Steps 3–6). All jobs succeeding k that are non-repairable are skipped (see Observation 14). If they are late, their weight is summed up in order to obtain $z^{(4)}$ (Step 5). Afterwards, for each job h in $\sigma_0(i, k)$, the corresponding branching decision is considered (Steps 8–13). Because of this approach, the number of successors in the branch-and-bound tree is not fixed a priori. For each choice of h, a B-move (h, k, ℓ) is performed, and the corresponding costs are evaluated (Steps 9–13). Costs $z^{(1)}$ and $z^{(2)}$ are evaluated by recursive calls (Steps 10 and 11): $z^{(1)}$ is the optimal solution value of the subproblem $(i, h-1, \ell, \mu)$; that is, the minimum cost of subsequence $\sigma_0(i, h-1)$ at level ℓ with time shift equal to μ; $z^{(2)}$ is the optimal solution value of the subproblem $(h+1, k, \ell-1, \mu+p_h)$, which is the minimum cost of subsequence $\sigma_0(h+1, k)$ at level $\ell - 1$ with time shift $\mu + p_h$. Empty subsequences that occur when either $h = i$ or $h = k$ do not incur any costs and therefore do not imply any recursive calls; they form the recursion base. The value $z^{(3)}$ can be computed directly (Step 12): it corresponds to the weight of job h if $C_k(\sigma_0) - \mu > d_h$ and zero otherwise.

The procedure in Algorithm 4 is initially invoked with $i = 1$, $j = n$, $\ell = S$, and $\mu = 0$, which leads to the initial call $Branch(1, n, S, 0)$.

Algorithm 4 Branching strategy

```
1: procedure Branch(i, j, ℓ, μ)
2:     z ← ∞, z^(4) ← 0
3:     k ← j
4:     while (k > i and (ℓ = 0 or C_k(σ_0) − μ ≤ d_k or C_k(σ_0) − μ −
       α(i, k − 1, ℓ) > d_k)) or k = i do
5:         if C_k(σ_0) − μ > d_k then z^(4) ← z^(4) + w_k
6:         k ← k − 1
7:     if k > i then
8:         for h = i, ..., k do
9:             z^(1) ← 0, z^(2) ← 0, z^(3) ← 0
10:            if h > i then z^(1) ← Branch(i, h − 1, ℓ, μ)
11:            if h < k then z^(2) ← Branch(h + 1, k, ℓ − 1, μ + p_h)
12:            if C_k(σ_0) − μ > d_h then z^(3) ← w_h
13:            if Σ_{u=1}^{4} z^(u) < z then z ← Σ_{u=1}^{4} z^(u)
14:    else
15:        z ← z^(4)
16:    return z
```

5.1.1 Duplicate avoidance

The branching scheme described above has a major drawback, namely, that the same subproblems have to be optimized several times since they occur in several distinct branches of the branch-and-bound tree. For instance, the optimization of the subproblem $(1, h, \ell, 0)$ will be required in all nodes of the branch-and-bound tree of the form $(1, h', \ell, 0)$ with $h' > h$, when $h + 1$ is the job selected to be moved. To avoid solving the same subproblems multiple times, a suitable data structure is defined to store the results of the subproblems once they have been solved. At the expense of increased consumption of memory space, this technique allows us to substantially reduce the computing time.

5.2 Bounding

The definition of a repairable job is not only the basis for the branching policy, but it is also useful for the purpose of a lower bound. In detail, for each subproblem (i, j, ℓ, μ), the sum of the weights of late but non-repairable jobs in it is a valid lower bound of its cost. For each successor corresponding to a move $h \rightarrow k$, lower bounds $\zeta^{(1)}$ and $\zeta^{(2)}$ for $z^{(1)}$ and $z^{(2)}$, respectively, are evaluated. If $\zeta^{(1)}+\zeta^{(2)}+z^{(3)}+z^{(4)} < z$, then the successor node in the branch-and-bound tree is optimized; otherwise it is fathomed.

In addition to the lower bound, an upper bound is computed for each subproblem. It initially corresponds to the cost of the subsequence with no rearrangements and is updated every time a better rearrangement is found in one of the successors of the subproblem.

Table 1 Distribution of the processing times and the weights for the different test sets

set 1	$p_j \sim U(1, 100)$	$w_j \sim U(1, 100)$
set 2	$p_j \sim U(1, 1000)$	$w_j \sim U(1, 100)$
set 3	$p_j \sim U(1, 100)$	$w_j \sim U(1, 1000)$
set 4	$p_j \sim U(1, 1000)$	$w_j \sim U(1, 1000)$

5.3 Search strategy

A subproblem represented by the quadruple (i, j, ℓ, μ) can have up to $j - i + 1$ successors. They are examined in an order that depends on their associated lower bound, i.e., $\zeta^{(1)} + \zeta^{(2)} + z^{(3)} + z^{(4)}$. More precisely, the successors are solved in non-decreasing order of their associated lower bound. Some preliminary computational tests showed that this strategy enables early detection of the optimal solution and thus makes it possible to substantially reduce the number of nodes in the branch-and-bound tree.

Apart from this local ordering of the successors, the branch-and-bound tree is visited depth-first, since this allows for recursive implementation.

6 Computational experiments

This section provides computational experiments based on five data sets aiming for comparisons between the different solution approaches. All experiments were performed on an Intel Core i5 2.10 GHz computer (in 64-bit mode) with a memory of 8 GB. Each of the algorithms was coded in Pascal language, and for the integer linear programs, we used the commercial ILP solver Gurobi v9.1.1. Hereinafter, first, the procedure for generating the various instances is described, followed by a discussion of the results obtained.

6.1 Generation of test instances

We generated test instances with 50 and 100 jobs according to the data generation scheme proposed by Potts and Wassenhove (1988). For each instance of the test sets 1 to 4, we chose integer processing times p_j and integer weights w_j, independent and identically, uniformly distributed, respectively, with distribution parameters as given in Table 1.

Moreover, we chose independent and identically, uniformly distributed integer due dates $d_j \sim U(P \cdot d^{\ell},\ P \cdot d^{u})$ for various lower and upper bounds d^{ℓ} and d^{u}, where $P = \sum_{j \in J} p_j$ is the sum of the processing times for all jobs. The choices for the two parameters d^{ℓ} and d^{u} are

$d^\ell \in \{0.2, 0.4, 0.6, 0.8\}$ and $d^u \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$, with $d^\ell \leq d^u$. Accordingly, this results in 14 pairs of parameter values. For each of the 14 resulting classes of instances, 10 instances were randomly generated, ultimately yielding a total of 140 instances for each value of n. Note that a similar test environment was used in Nicosia et al. (2021) to illustrate the influence of the stack capacity S on the solution structure for the three objective functions where strictly polynomial algorithms exist, but not for $1|resch\text{-}LIFO|\sum w_j U_j$.

Test set 5 was generated in a completely different way by the following procedure. For each instance of data set 1, the optimal solution for minimizing $\sum w_j U_j$ (completely reordering the given jobs) was computed. Subsequently, randomly selected jobs were moved forward in compliance with the LIFO policy and a stack capacity of 12. Clearly, when solving our stated problem with $S = 12$, we again obtain the optimal solution value. For smaller values of S, we expected to obtain hard-to-solve instances.

6.2 Performance comparison

Table 2 and Table 3 report on the experimental comparison between the different solution procedures for $1|resch\text{-}LIFO|\sum w_j U_j$ presented in Sects. 3 to 5.

Although the number of constraints of the assignment-based ILP (*ILPassign*) is asymptotically smaller by one or two orders of magnitude than the number of constraints in the move-based ILPs (*ILPmove1* and *ILPmove2*) (cf. Sect. 3), the running times for *ILPassign* were disproportionately long compared to all other solution approaches. Thus, we have not solved the majority of the instances with this solution method, and so we do not present any results for *ILPassign*. In this context, we would like to point out that for *ILPassign*, the relative gap between the optimal value of the LP relaxation at the root node and the optimal objective value is on average about 96% for all values of S, while for the move-based ILPs, it is often considerably smaller (see Table 4). Furthermore, *DynProStatic* could not be executed for test sets 3 and 4 due to excessively high memory usage caused by the high number of states required for solving the rescheduling problem. Recall that for these two sets, the weights are random variables with uniform distribution in [1, 1000], and that for each triple (i, j, ℓ), the number of states to be computed is $W(i, j)$.

In Tables 2 and 3, we report for each of the considered algorithms the average computation time (over 140 instances) needed to compute the optimal solution value. Since in some cases obtaining an optimal solution may take too long, we set a time limit of 900 000 milliseconds (15 minutes) for each computational experiment. In cases where the algorithm did not terminate within this time limit or where it terminated prematurely due to insufficient memory, we used a computation time limit of 900 000 milliseconds for taking the average values. For all the solution methods, additional information is provided in the respective second row. The values in these second rows have different meanings for the various algorithms, which is indicated by the different brackets: the values in parentheses (*DynProStatic* and *DynProImpr*) indicate the average number of states computed until the program terminates; the values in square brackets (*CombB&B*) indicate the average number of solved subproblems; for the values in curly brackets, the first number (*CombB&B*, *ILPmove1*, and *ILPmove2*) denotes the number of instances (out of 140) that cannot be solved within the predefined time limit or due to lack of memory; and the second number (*ILPmove1* and *ILPmove2*) indicates the number of instances for which an optimal solution was found, but the optimality guarantee was not provided because the time out expired before the algorithm terminated. Clearly, the unsolved instances are also not taken into account when calculating the average number of solved subproblems.

As can be seen in Table 2, the dynamic programming algorithm *DynProStatic* presented in Nicosia et al. (2021) cannot compete at all with the other solution procedures, even with the move-based ILP formulation. Even for very small values of S, it consumes enormous computation times. For the ILPs, it must be said that they do not always terminate within the predefined time limit, but for almost all instances the optimal solution value has been found at that point, but without proving its optimality. In only 12 cases for *ILPmove1* and 3 cases for *ILPmove2* (out of 6300), the ILP solver stopped with a suboptimal solution after being interrupted by the time limit.

As can be expected, for smaller stack sizes, the lower computation times for *ILPmove1* coincide with smaller relative LP gaps. However, it is interesting to note that for larger stack sizes, the average computation time for *ILPmove2* is smaller than the time for *ILPmove1*, but the relative LP gap for *ILPmove2* is on average still larger compared to *ILPmove1* for all values of S. For $S \geq 10$, the gaps of *ILPmove2* even get closer to the gaps of *ILPassign*, although the running times are incomparable. This illustrates that the value of the LP gap at the root node has only limited explanation power for the performance of the ILP. The results for the gaps are presented in Table 4, where the relative LP gap for each instance is calculated by taking the difference between the optimal objective value of the ILP and the optimal LP relaxation objective value at the root node and dividing it by the former.

Nonetheless, compared to the ILP models and *DynProStatic*, both *DynProImpr* and *CombB&B* dominate by several orders of magnitude and require comparatively really short average running times. For this reason, we only evaluated these two procedures for instances with 100 jobs, as shown in Table 3.

As a main outcome of our computational analysis, it turns out that for smaller stack capacities, i.e., less than or equal

Table 2 Computational results for all instances with $n = 50$ jobs for the different algorithms.

set	algorithm	stack capacity S								
		1	2	3	4	5	6	8	10	12
1	*DynProStatic*	48 604	95 923	143 590	191 493	239 456	287 465	383 604	479 792	575 994
		(2 211 952)	(3 317 929)	(4 423 905)	(5 529 881)	(6 635 857)	(7 741 834)	(9 953 786)	(12 165 739)	(14 377 691)
	DynProImpr	3	8	12	17	26	33	57	87	128
		(3 420)	(7 020)	(12 329)	(19 536)	(28 700)	(39 486)	(67 981)	(104 657)	(149 537)
	CombB&B	0	0	0	1	3	15	146	618	1 718
		[3]	[11]	[59]	[574]	[4 322]	[21 721]	[154 209]	[458 921]	[1 049 708]
	ILPmove1	2 860	7 917	14 841	33 530	64 239	108 914†	234 330†	428 617†	532 145†
		—	—	—	—	—	{3:3}	{16:16}	{31:30}	{57:52}
	ILPmove2	2 925	12 253	24 864	48 724	78 360	106 353	188 309†	300 028†	376 844†
		—	—	—	—	—	—	{2:2}	{12:11}	{22:20}
2	*DynProStatic*	49 760	98 231	147 068	196 029	245 136	294 258	392 634	491 079	589 663
		(2 233 175)	(3 349 762)	(4 466 349)	(5 582 936)	(6 699 524)	(7 816 111)	(10 049 286)	(12 282 460)	(14 515 635)
	DynProImpr	3	6	10	18	25	35	60	91	130
		(3 507)	(7 170)	(12 790)	(20 085)	(29 450)	(41 016)	(71 812)	(110 256)	(156 611)
	CombB&B	0	0	0	1	4	81	4 893	34 375	111 434†
		[3]	[10]	[49]	[465]	[4 338]	[34 485]	[476 325]	[1 996 347]	[3 941 074]:{5}
	ILPmove1	2 755	7 498	14 380	36 640	68 124†	124 753†	293 043†	457 957†	532 068†
		—	—	—	—	{1:1}	{3:3}	{17:17}	{48:48}	{62:61}
	ILPmove2	2 831	10 983	24 377	51 376	86 810	125 715	212 146†	340 435†	393 396†
		—	—	—	—	—	—	{2:2}	{18:18}	{20:20}
3	*DynProImpr*	3	6	12	18	26	35	60	92	133
		(3 414)	(6 945)	(12 109)	(19 307)	(29 020)	(40 738)	(70 958)	(110 002)	(155 697)
	CombB&B	0	0	1	0	2	13	127	582	1644
		[3]	[12]	[75]	[478]	[3 435]	[18 011]	[137 637]	[457 486]	[1 066 035]
	ILPmove1	2 723	7 519	16 266	42 791†	67 292	119 729†	273 282†	448 218†	553 379†
		—	—	—	{1:1}	—	{2:2}	{14:14}	{44:43}	{63:59}
	ILPmove2	2 893	11 676	27 430	56 685	83 538	119 797	212 223†	320 751†	392 742†
		—	—	—	—	—	—	{2:2}	{16:16}	{26:26}

Table 2 continued

set	algorithm	stack capacity S								
		1	2	3	4	5	6	8	10	12
4	*DynProImpr*	3	6	10	19	26	36	60	98	149
		(3 441)	(7 061)	(12 462)	(19 620)	(29 323)	(40 892)	(69 907)	(107 806)	(153 875)
	CombB&B	0	1	0	1	11	199	6773	39521	124591†
		[3]	[10]	[66]	[745]	[8 293]	[55 021]	[578 007]	[2 159 811]	[4 654 112]:{4}
	ILPmove1	2 706	7 549	15 117	31 929	67 868†	108 050†	261 412†	389 513†	520 780†
		—	—	—	—	{1:1}	{3:3}	{16:16}	{34:34}	{52:52}
	ILPmove2	2 826	11 196	28 151	50 748	86 333	113 439	205 577†	292 698†	367 620†
		—	—	—	—	—	—	{4:4}	{9:9}	{16:16}
5	*DynProStatic*	52 943	104 841	157 091	209 522	262 051	314 642	419 930	525 313	630 814
		(2 281 783)	(3 422 675)	(4 563 566)	(5 704 458)	(6 845 350)	(7 986 241)	(10 268 025)	(12 549 808)	(14 831 591)
	DynProImpr	2	5	9	14	19	26	40	52	69
		(3 807)	(7 679)	(12 205)	(17 408)	(23 201)	(29 763)	(44 577)	(61 878)	(81 668)
	CombB&B	0	0	1	1	4	15	101	396	1 163
		[4]	[20]	[164]	[1 210]	[6 494]	[22 968]	[120 553]	[343 255]	[765 987]
	ILPmove1	3 227	11 820	33 132	92 193†	153 039†	230 209†	345 942†	401 814†	449 718†
		—	—	—	{6:6}	{10:10}	{21:21}	{41:41}	{50:50}	{52:52}
	ILPmove2	3 499	16 914	37 148	64 994	98 957	140 410	233 540†	339 881†	382 143†
		—	—	—	—	—	—	{12:12}	{27:27}	{37:37}

For each solution approach, the values in the first row reflect the average computation time (or a lower bound†) in milliseconds. The values in the second row have different meanings for the various algorithms, as indicated by the different brackets: the values in parentheses (*DynProStatic* and *DynProImpr*) give the average number of states computed until the program terminates; the values in square brackets (*CombB&B*) indicate the average number of solved subproblems; for the values in curly brackets, the first number (*CombB&B*, *ILPmove1*, and *ILPmove2*) denotes the number of instances that cannot be solved within the predefined time limit or due to lack of memory, the second number (*ILPmove1* and *ILPmove2*) indicates the number of instances that have been solved to optimality despite a premature termination †In cases where the algorithm terminated prematurely due to a reached time limit of 900 000 milliseconds or due to lack of memory, the computation time was set to 900 000 milliseconds when computing the average computation time

Table 3 Computational results for all instances with $n = 100$ jobs for the algorithms *DynProImpr* and *CombB&B*.

set	algorithm	stack capacity S								
		1	2	3	4	5	6	8	10	12
1	*DynProImpr*	21	47	86	137	208	303	597	993	1 531
		(14 174)	(28 079)	(49 078)	(79 339)	(119 266)	(168 256)	(308 950)	(495 818)	(728 877)
	CombB&B	2	1	2	4	11	44	600	4 084	15 786
		[3]	[11]	[57]	[708]	[7 428]	[38 836]	[421 113]	[2 030 680]	[5 854 559]
2	*DynProImpr*	20	49	87	140	217	320	604	1 082	1 958
		(13 987)	(29 175)	(50 674)	(81 444)	(123 061)	(176 385)	(315 362)	(513 246)	(767 406)
	CombB&B	1	1	2	4	41	1 024	49 469†	190 270†	347 756†
		[3]	[12]	[83]	[1 167]	[19 139]	[170 499]	[2 527 454]:{1}	[3 580 168]:{20}	[7 090 305]:{38}
3	*DynProImpr*	20	49	87	141	216	316	616	1 304	1 848
		(13 989)	(29 178)	(51 232)	(81 732)	(123 338)	(174 722)	(319 997)	(520 891)	(770 501)
	CombB&B	2	2	2	3	9	46	605	3 671	14 267
		[3]	[10]	[69]	[680]	[6 612]	[41 303]	[415 817]	[1 901 478]	[5 591 667]
4	*DynProImpr*	21	49	89	141	221	317	605	1 055	1 942
		(13 957)	(27 906)	(49 031)	(78 576)	(122 397)	(176 448)	(319 949)	(518 167)	(780 037)
	CombB&B	2	2	2	3	23	533	35 348	173 531†	353 984†
		[3]	[9]	[57]	[733]	[10 240]	[96 420]	[2 007 289]	[4 091 218]:{15}	[7 861 762]:{35}
5	*DynProImpr*	16	38	70	109	160	214	345	524	766
		(15 222)	(31 634)	(53 954)	(79 921)	(109 707)	(141 360)	(212 147)	(295 875)	(394 869)
	CombB&B	2	2	3	11	56	216	1 635	6 897	18 815
		[4]	[38]	[544]	[7 420]	[48 073]	[175 307]	[929 458]	[2 831 909]	[6 373 561]

For each solution approach, the values in the first row reflect the average computation time (or a lower bound†) in milliseconds. The values in the second row have different meanings for the various algorithms which is indicated by the different brackets: the values in parentheses (*DynProImpr*) give the average number of states computed until the program terminates; the values in square brackets (*CombB&B*) indicate the average number of solved subproblems; the values in curly brackets denote the number of instances that cannot be solved within the predefined time limit or due to lack of memory †In cases where the algorithm terminated prematurely due to a reached time limit of 900 000 milliseconds or due to lack of memory, the computation time was set to 900 000 milliseconds when computing the average computation time

Table 4 Average relative gaps between the LP relaxation objective value at the root node and the optimal objective value for the different ILP formulations

set	ILP model	stack capacity S								
		1	2	3	4	5	6	8	10	12
1	*ILPmove1*	0.01	0.24	0.25	0.31	0.35	0.42	0.52	0.64	0.72
	ILPmove2	0.01	0.58	0.62	0.66	0.70	0.73	0.77	0.81	0.83
	ILPassign	0.97	0.97	0.97	0.97	0.96	0.96	0.96	0.96	0.96
2	*ILPmove1*	0.02	0.23	0.25	0.30	0.37	0.44	0.53	0.63	0.72
	ILPmove2	0.02	0.59	0.62	0.67	0.70	0.73	0.77	0.81	0.83
	ILPassign	0.97	0.97	0.97	0.96	0.96	0.96	0.96	0.96	0.96
3	*ILPmove1*	0.01	0.26	0.28	0.33	0.37	0.45	0.53	0.65	0.74
	ILPmove2	0.02	0.59	0.63	0.67	0.71	0.73	0.78	0.81	0.83
	ILPassign	0.97	0.97	0.96	0.96	0.96	0.96	0.96	0.96	0.95
4	*ILPmove1*	0.01	0.23	0.27	0.32	0.36	0.43	0.50	0.62	0.72
	ILPmove2	0.02	0.57	0.62	0.66	0.70	0.73	0.77	0.81	0.83
	ILPassign	0.97	0.97	0.97	0.97	0.96	0.96	0.96	0.96	0.96
5	*ILPmove1*	0.15	0.44	0.51	0.54	0.57	0.60	0.67	0.75	0.80
	ILPmove2	0.16	0.69	0.71	0.73	0.75	0.77	0.81	0.84	0.85
	ILPassign	0.95	0.94	0.94	0.93	0.93	0.93	0.93	0.93	0.93

to 5 or 6 (depending on the particular test set), *CombB&B* performs better than *DynProImpr*, while for larger values of S, the improved dynamic program delivers outstanding results by comparison.

Another interesting finding from our results is the following observation, which holds for both values of n: when comparing the results of the different solution approaches between sets 1 and 2 (recall that in both test sets, the weights are generated from the uniform distribution on the interval $[1, 100]$, whereas the processing times are generated from $U(1, 100)$ and $U(1, 1000)$, respectively), the average computation times (and also the average number of computed states/subproblems) do not differ significantly for any of the dynamic programming algorithms and any of the considered stack capacities. This is rather surprising as it might have been expected that dynamic programming is more sensitive to the size of the input coefficients. In contrast, the branch-and-bound procedure only produces similar results for a stack capacity smaller or equal to 4, while the running times are noticeably different for larger values of S. The same results can be observed when comparing test sets 3 and 4.

Recall that test sets 1 and 5 contain the same sets of jobs, but the initial ordering of set 5 was designed to obtain more difficult instances. For $n = 100$, this could indeed be observed for *CombB&B*, but not for *DynProImpr*. For $n = 50$, the comparison of algorithms does not show a significant trend. It seems that the effect of perturbing an optimal schedule by moving jobs with a stack capacity 12 (roughly 25%) yields instances which do not differ too much from randomly sorted sequences.

7 Conclusions

In this paper, we considered a rescheduling problem where jobs arrive in a given sequence which can be modified by extracting jobs from the sequence and putting them on a stack of limited capacity. Jobs can be reinserted from the stack into the sequence at a position further to the end (no forward moves) whereby the stack must be accessed according to a LIFO policy. While other objective functions were considered in a preceding paper, our goal in this paper is the minimization of the weighted number of late jobs in the modified sequence.

We study the exact solution of this NP-hard problem from several perspectives. We introduce two ILP models, an advanced algorithm based on dynamic programming and a branch-and-bound approach.

Summarizing our computational experiments, the ILP models soon reach their limits of practicability. However, it turns out that our advanced dynamic programming algorithm *DynProImpr* solves all our classes of test instances in less than 2 seconds (on average), while all other approaches exhibit a dramatic increase in their running time consumption. In particular, we want to emphasize the importance of the technical and implementational improvements applied to the previous dynamic program. Although the underlying algorithmic principle remains the same, the application of dominance relations and state reductions improves the performance by several orders of magnitude. Careful algorithm engineering turns a pseudo-polynomial algorithm of only theoretical value into a viable solution approach.

The branch-and-bound algorithm *CombB&B* performs even better than *DynProImpr* for smaller stack sizes, but

becomes less efficient for larger stack sizes and larger weight coefficients. We pose as an open problem the improvement of its performance by developing tighter and efficiently computable lower bounds.

As a summary for a decision maker, we can conclude that even with today's amazing capabilities of commercial ILP solvers, the combinatorial difficulty of a rescheduling problem with a limited buffer can be much better handled by tailor-made solution algorithms. Looking for an effective one-size-fits-all solution, the dynamic programming approach can be recommended as the most effective and versatile algorithm. Its running time gains considerably from advanced algorithmic features.

For future research, it would be interesting to consider other policies for managing the stack, in particular, a queue environment (first-in/first-out, FIFO) or a random access buffer. Considering, for example, a chain of three sequential production stages would open up the question of two rescheduling phases in between the successive stages, but with an interdependent objective. Of course, generalizations to multistage production systems could follow.

Author Contributions Not applicable

Funding Open access funding provided by University of Graz. This study was partially funded by the University of Graz under the Field of Excellence "COLIBRI" and by the Regione Lombardia, grant agreement no. E97F17000000009, project AD-COM.

Data Availability Not applicable

Code Availability Not applicable

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

A Numerical example for Algorithm 3

In this appendix, we illustrate Algorithm 3 by a numerical example. For ease of comparison, we continue the example given in Nicosia et al. (2021), but with additional weights as required for our purpose.

Example 1 Consider a schedule $\sigma_0 = \langle 1, 2, 3, 4\rangle$ with 4 jobs. The respective processing times, due dates, and weights are given in Table 5. Since the intention behind the example is to demonstrate how the lists $\theta(i, j, \ell)$ of non-m-dominated states are calculated, we confine our computation to the case $S = 1$. Note that it can be proceeded analogously for larger values of S.

To determine the minimum weighted number of late jobs in the sequence σ_0 with moves up to level 1, all ordered lists $\theta(i, j, \ell)$ of non-m-dominated states with $i, j \in \{1, 2, 3, 4\}$, $i \leq j$, and $\ell \in \{0, 1\}$ need to be calculated.

First, however, the threshold values $a(i, \ell)$ have to be determined for each job $i \in \{1, 2, 3, 4\}$ and each level $\ell \in \{0, 1\}$ according to Algorithm 2: $a(1, \ell) = 0$ for $\ell \in \{0, 1\}$ (Steps 3–4), $a(i, 0) = 25$ and $a(i, 1) = 0$ for $i \in \{2, 3, 4\}$ (Steps 5–10).

At level $\ell = 0$, the initialization of the improved dynamic programming algorithm is performed (Steps 1–11 of Algorithm 3). Recall that the values of $\theta(i, j, 0)$ are evaluated on the initial sequence $\sigma_0(i, j)$. Hence, with the lateness values already presented in Table 5 and the previously computed threshold values, $a(i, 0), \theta(i, j, 0)$ can easily be determined.

For example, when computing $\theta(2, 4, 0)$, Steps 5–11 need to be performed. In so doing, we first initialize $M = 0$ (Step 5). Subsequently, we have to consider all late jobs in $\sigma_0(2, 4)$ in non-increasing order of their lateness values, i.e., in the order 3, 2, and 4, as also indicated by the columns of Table 6. For $k = 3$, we have $L_3(\sigma_0) = 30 > a(2, 0) = 25$. Consequently, Steps 8–9 do not need to be performed. However, Step 10 demands M to be increased by $w_3 = 2$ such that $M = 2$. In contrast, for $k = 2$, the if-statement of Step 7 is satisfied ($L_3(\sigma_0) = 20 \leq a(2, 0) = 25$). Since $\theta(2, 4, 0)$ is (still) empty, the tuple $(2, 20)$ is added to $\theta(2, 4, 0)$ (Step 9) and M is set to 5 (Step 10). For $k = 4$, we have $L_4(\sigma_0) = 20 \leq a(2, 0) = 25$, i.e., the if-condition of Step 7 is fulfilled, but when comparing $(5, 20)$ with the last element of $\theta(2, 4, 0)$ which is $(2, 20)$, we observe that

Table 5 Processing times, due dates, weights, completion times, and lateness values of an initial schedule $\sigma_0 = \langle 1, 2, 3, 4\rangle$ with a weighted number of late jobs of 8

job	1	2	3	4
p_i	25	10	5	10
d_i	45	15	10	30
w_i	1	3	2	3
$C_i(\sigma_0)$	25	35	40	50
$L_i(\sigma_0)$	−20	20	30	20

LessOrEqual ((5, 20), (2, 20)) does not hold (Step 8). Therefore, M is increased by $w_4 = 3$ without previously adding an element to $\theta(2, 4, 0)$. Last but not least, the tuple $(8, 0)$ is appended to $\theta(2, 4, 0)$ (Step 11) such that the list $\theta(2, 4, 0)$ finally consists of the elements $(2, 20)$ and $(8, 0)$. All lists of non-m-dominated states at level 0 are shown in Table 6.

At level $\ell = 1$, the recursion of the improved dynamic programming algorithm is performed (Steps 12–49 of Algorithm 3). To compute the list $\theta(i, j, 1)$ of non-m-dominated states, we first have to compute $\mathcal{R}_k$ for all $k \in \mathcal{K}$ according to Steps 16–40 where $\mathcal{K}$ contains all late jobs in $\sigma_0(i, j)$ in addition to job i. As $\mathcal{R}_k$ is composed of the sets $\mathcal{R}_{k,1}$, $\mathcal{R}_{k,2}$ and $\mathcal{R}_{k,3}$, the middle part of Table 7 states each of the three sets separately.

When, for example, $\mathcal{R}_4$ in $\sigma_0(2, 4)$ is to be computed, we first have to determine $\mathcal{R}_{4,1}$, $\mathcal{R}_{4,2}$, and $\mathcal{R}_{4,3}$ (Steps 18–31). The set $\mathcal{R}_{4,1}$ contains, on the one hand, the tuple $(2, 10)$, which is the first element of $\theta(3, 4, 0)$, but with an s-value reduced by $p_2 = 10$, and on the other hand, $(5, 0)$, which is the second element of $\theta(3, 4, 0)$, but with an s-value equal to 0, since a reduction of the s-value would lead to a negative entry. While set $\mathcal{R}_{4,2}$ is empty since k and j both equal 4, $\mathcal{R}_{4,3}$ consists of the two elements $(0, 35)$ and $(3, 0)$ since $C_4(\sigma_0) - d_2 = 45 - 10 = 35 > 0$. For the purpose of computing $\mathcal{R}_4$ in $\sigma_0(2, 4)$, we first observe that $\mathcal{P} = \{1, 3\}$ because only $\mathcal{R}_{4,1}$ and $\mathcal{R}_{4,3}$ are non-empty. In Step 35 , we find that $p^* = 3$, since from the two tuples $(2, 10)$ and $(0, 35)$, the second one, contained in $\mathcal{R}_{4,3}$, has the larger s-value. Consequently, ζ equals 35 (Step 36). As it is larger than $a(2, 0) = 25$, Steps 38–39 are disregarded. After deleting $(0, 35)$ from $\mathcal{R}_{4,3}$ (Step 40), we find that $p^* = 1$ and thus $\zeta = 10$. Hence, the if-statement is again not satisfied, so we just delete $(2, 10)$ from $\mathcal{R}_{4,1}$. Next, we have $p^* = 1$. Note that we could also choose $p^* = 3$ which leads to the same value of $\zeta = 8$ as it is the sum of the m-values of the first elements of $\mathcal{R}_{4,1}$ and $\mathcal{R}_{4,3}$. After appending $(8, 0)$ to $\mathcal{R}_4$ (Step 39) and deleting the first element from $\mathcal{R}_{4,1}$ (Step 40), the termination condition is reached. It follows that $\mathcal{R}_4$ in $\sigma_0(2, 4)$ contains only the element $(8, 0)$, as also shown in the last column of Table 7.

After computing $\mathcal{R}_k$ for all $k \in \mathcal{K}$, the list $\theta(i, j, 1)$ can be obtained by performing Steps 41–48. In order to do this, a set $\overline{K}$ is formed by selecting all tuples with the smallest m-value, i.e., the smallest first entry, among all the first elements of the lists $\mathcal{R}_k$, $k \in \mathcal{K}$ (Step 43). That tuple of set $\overline{K}$ with the smallest s-value, i.e., the smallest second entry (Step 44) is then added to $\theta(i, j, 1)$ (Step 45). Before the next element to be added to $\theta(i, j, 1)$ can be determined, the lists $\mathcal{R}_k$ must be reduced by all tuples where both entries are larger or equal to the one just added to θ (Steps 46–48). For example, $\theta(2, 4, 1)$ is obtained as follows: Among all first elements of the lists $\mathcal{R}_k$, $k \in \{2, 3, 4\}$, we identify $(8, 0)$ as that element which is the first to be appended to $\theta(2, 4, 1)$ (note that in our case, all the first elements are equal to $(8, 0)$). After decimating the three lists according to Steps 46–48, all three of these lists are empty, and we terminate. Hence, $\theta(2, 4, 1)$ is an (ordered) list containing only the tuple $(8, 0)$. This result is also shown in Table 7 together with all other θ values.

As shown in the last row of Table 7, the optimal solution for stack capacity $S = 1$ has a weighted number of late jobs of 3. In this example, there is only one optimal schedule, namely, $\sigma = \langle 2, 3, 4, 1 \rangle$. It can be obtained by the single move $1 \rightarrow 4$ that causes jobs 1 and 3 to be late.

Table 6 Lists $\theta(i, j, 0)$ of non-m-dominated states for all $i, j \in \{1, 2, 3, 4\}$ with $i \leq j$

		$k = 3$	$k = 2$	$k = 4$	Step 11	$\theta(i, j, 0)$
$j = 1$	$i = 1$				$(0, 0)$	$(0, 0)$
$j = 2$	$i = 2$		$(0, 20)$		$(3, 0)$	$(0, 20), (3, 0)$
	$i = 1$				$(3, 0)$	$(3, 0)$
$j = 3$	$i = 3$				$(2, 0)$	$(2, 0)$
	$i = 2$		$(2, 20)$		$(5, 0)$	$(2, 20), (5, 0)$
	$i = 1$				$(5, 0)$	$(5, 0)$
$j = 4$	$i = 4$			$(0, 20)$	$(3, 0)$	$(0, 20), (3, 0)$
	$i = 3$			$(2, 20)$	$(5, 0)$	$(2, 20), (5, 0)$
	$i = 2$		$(2, 20)$		$(8, 0)$	$(2, 20), (8, 0)$
	$i = 1$				$(8, 0)$	$(8, 0)$

Table 7 Lists $\theta(i, j, 1)$ of non-m-dominated states for all $i, j \in \{1, 2, 3, 4\}$ with $i \leq j$

		move $i \rightarrow k$	$\mathcal{R}_{k,1}$	$\mathcal{R}_{k,2}$	$\mathcal{R}_{k,3}$	$\mathcal{R}_k$
$j = 1$	$i = 1$	$k = 1$			(0, 0)	(0, 0)
		$\boldsymbol{\theta(1, 1, 1)}$				**(0, 0)**
$j = 2$	$i = 2$	$k = 2$			(0, 20), (3, 0)	(3, 0)
		$\boldsymbol{\theta(2, 2, 1)}$				**(3, 0)**
	$i = 1$	$k = 1$		(3, 0)	(0, 0)	(3, 0)
		$k = 2$	(0, 0)		(0, 0)	(0, 0)
		$\boldsymbol{\theta(1, 2, 1)}$				**(0, 0)**
$j = 3$	$i = 3$	$k = 3$			(0, 30), (2, 0)	(2, 0)
		$\boldsymbol{\theta(3, 3, 1)}$				**(2, 0)**
	$i = 2$	$k = 2$		(2, 0)	(0, 20), (3, 0)	(5, 0)
		$k = 3$	(2, 0)		(0, 25), (3, 0)	(5, 0)
		$\boldsymbol{\theta(2, 3, 1)}$				**(5, 0)**
	$i = 1$	$k = 1$		(5, 0)	(0, 0)	(5, 0)
		$k = 2$	(0, 0)	(2, 0)	(0, 0)	(2, 0)
		$k = 3$	(2, 0)		(0, 0)	(2, 0)
		$\boldsymbol{\theta(1, 3, 1)}$				**(2, 0)**
$j = 4$	$i = 4$	$k = 4$			(0, 20), (3, 0)	(3, 0)
		$\boldsymbol{\theta(4, 4, 1)}$				**(3, 0)**
	$i = 3$	$k = 3$		(3, 0)	(0, 30), (2, 0)	(5, 0)
		$k = 4$	(0, 15), (3, 0)		(0, 40), (2, 0)	(5, 0)
		$\boldsymbol{\theta(3, 4, 1)}$				**(5, 0)**
	$i = 2$	$k = 2$		(5, 0)	(0, 20), (3, 0)	(8, 0)
		$k = 3$	(2, 0)	(3, 0)	(0, 25), (3, 0)	(8, 0)
		$k = 4$	(2, 10), (5, 0)		(0, 35), (3, 0)	(8, 0)
		$\boldsymbol{\theta(2, 4, 1)}$				**(8, 0)**
	$i = 1$	$k = 1$		(8, 0)	(0, 0)	(8, 0)
		$k = 2$	(0, 0)	(5, 0)	(0, 0)	(5, 0)
		$k = 3$	(2, 0)	(3, 0)	(0, 0)	(5, 0)
		$k = 4$	(2, 0)		(0, 5), (1, 0)	(3, 0)
		$\boldsymbol{\theta(1, 4, 1)}$				**(3, 0)**

Bold entries represent the outcome of each iteration

References

Abumaizar, R., & Svestka, J. (1997). Rescheduling job shops under random disruptions. *International Journal of Production Research, 35*(7), 2065–2082.

Agnetis, A., Hall, N. G., & Pacciarelli, D. (2006). Supply chain scheduling: Sequence coordination. *Discrete Applied Mathematics, 154*(15), 2044–2063.

Alfieri, A., Nicosia, G., Pacifici, & Pferschy, U. (2018a). Single machine scheduling with bounded job rearrangements. In: Proceedings of 16th cologne-Twente workshop on graphs and combinatorial optimization, 124–127.

Alfieri, A., Nicosia, G., Pacifici, A., & Pferschy, U. (2018b). Constrained Job Rearrangements on a Single Machine. In P. Daniele & L. Scrimali (Eds.), *New Trends in Emerging Complex Real Life Problems, AIRO Springer Series* (Vol. 1, pp. 33–41). Springer.

Ballestín, F., Pérez, A., & Quintanilla, S. (2019). Scheduling and rescheduling elective patients in operating rooms to minimise the percentage of tardy patients. *Journal of Scheduling, 22*(1), 107–118.

Detti, P., Nicosia, G., Pacifici, A., Manrique, Zabalo, & de Lara, G. (2019). Robust single machine scheduling with a flexible maintenance activity. *Computers & Operations Research, 107,* 19–31.

Drexl, A., Kimms, A., & Matthießen, L. (2006). Algorithms for the car sequencing and the level scheduling problem. *Journal of Scheduling, 9,* 153–176.

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Rinnooy Kan, A. H. G. (1979). Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Mathematics, 5,* 287–326.

Hall, N. G., Liu, Z., & Potts, C. N. (2007). Rescheduling for Multiple New Orders. *INFORMS Journal on Computing, 19*(4), 633–645.

Hall, N., & Potts, C. N. (2004). Rescheduling for new orders. *Operations Research, 52*(3), 440–453.

Hall, N. G., & Potts, C. N. (2010). Rescheduling for Job Unavailability. *Operations Research, 58*(3), 746–755.

Liebchen, C., Lübbecke, M., Möhring, & Stiller, S. (2009). The Concept of Recoverable Robustness, Linear Programming Recovery, and Railway Applications. In: Robust and online Large-Scale optimization: models and techniques for transportation systems, vol 5868, Springer, 1–27.

Li, C. L., & Li, F. (2020). Rescheduling production and outbound deliveries when transportation service is disrupted. *European Journal of Operational Research, 286*(1), 138–148.

Nicosia, G., Pacifici, A., Pferschy, U., Polimeno, E., & Righini, G. (2019) Optimally rescheduling jobs under LIFO constraints. In: Proceedings of the 17th cologne-twente workshop on graphs and combinatorial optimization, pp 107–110

Nicosia, G., Pacifici, A., Pferschy, P., Resch, J., & Righini, G. (2021). Optimally rescheduling jobs with a LIFO buffer. *Journal of Scheduling, 24,* 663–680.

Niu, S., Song, S., Ding, J. Y., Zhang, Y., & Chiong, R. (2019). Distributionally robust single machine scheduling with the total tardiness criterion. *Computers & Operations Research, 101,* 13–28.

Nouiri, M., Bekrar, A., Jemai, A., Ammari, A. C., & Niar, S. (2018). A New Rescheduling Heuristic for Flexible Job Shop Problem with Machine Disruption. *Studies in Computational Intelligence, 762,* 461–476.

Ouelhadj, D., & Petrovic, S. (2009). A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling, 12*(4), 417–431.

Polimeno, E. (2019). Optimal rescheduling of jobs under LIFO constraints. Master thesis, University of Milan, Department of Computer Science.

Potts, C. N., & Van Wassenhove, L. N. (1988). Algorithms for Scheduling a Single Machine to Minimize the Weighted Number of Late Jobs. *Management Science, 34*(7), 843–858.

Rener, E., Salassa, F., & T'kindt, V,. (2022). Single machine rescheduling for new orders with maximum lateness minimization. *Computers & Operations Research,144*, 105815.

van den Akker, M., Hoogeveen, H., & Stoef, J. (2018). Combining two-stage stochastic programming and recoverable robustness to minimize the number of late jobs in the case of uncertain processing times. *Journal of Scheduling, 21*(6), 607–617.

Vieira, G. E., Herrmann, J. W., & Lin, E. (2003). Rescheduling Manufacturing Systems: A Framework of Strategies, Policies, and Methods. *Journal of Scheduling, 6*(1), 39–62.

Wang, D., Yin, Y., & Jin, Y. (2020). *Rescheduling Under Disruptions in Manufacturing Systems: Models and Algorithms*. Uncertainty and Operations Research: Springer.