Journal of Scheduling
https://doi.org/10.1007/s10951-021-00707-5

Optimally rescheduling jobs with a Last-In-First-Out buffer

Gaia Nicosia[1] · Andrea Pacifici[2] · Ulrich Pferschy[3] · Julia Resch[3] · Giovanni Righini[4]

Accepted: 13 August 2021

Abstract
This paper considers single-machine scheduling problems in which a given solution, i.e., an ordered set of jobs, has to be improved as much as possible by re-sequencing the jobs. The need for rescheduling may arise in different contexts, e.g., due to changes in the job data or because of the local objective in a stage of a supply chain that is not aligned with the given sequence. A common production setting entails the movement of jobs (or parts) on a conveyor. This is reflected in our model by facilitating the re-sequencing of jobs via a buffer of limited capacity accessible by a LIFO policy. We consider the classical objective functions of total weighted completion time, maximum lateness and (weighted) number of late jobs and study their complexity. For three of these problems, we present strictly polynomial-time dynamic programming algorithms, while for the case of minimizing the weighted number of late jobs NP-hardness is proven and a pseudo-polynomial algorithm is given.

Keywords Scheduling · Rescheduling · Sequence coordination · Supply chain sustainability · Dynamic programming algorithms · Complexity

1 Introduction

Classical single-machine scheduling problems aim at finding an optimal sequence to process a set of jobs with given processing times possibly subject to additional constraints concerning, for instance, release dates, due dates, etc.

✉ Gaia Nicosia
gaia.nicosia@uniroma3.it

Andrea Pacifici
andrea.pacifici@uniroma2.it

Ulrich Pferschy
ulrich.pferschy@uni-graz.at

Julia Resch
julia.resch@uni-graz.at

Giovanni Righini
giovanni.righini@unimi.it

[1] Dipartimento di Ingegneria, Università Roma Tre, Via della Vasca Navale 79, 00146 Rome, Italy

[2] Dipartimento di Ingegneria Civile e Ingegneria Informatica, Università di Roma “Tor Vergata”, Via del Politecnico 1, 00133 Rome, Italy

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

[4] Dipartimento di Informatica, Università degli Studi di Milano, Via Celoria 18, 20100 Milan, Italy

In several industrial settings, different unforeseen phenomena, such as data-obsolescence or disruptions, could deteriorate the performance (or optimality) of the planned ahead schedule. In this case, it is sometimes possible—or even necessary—to compute a new schedule by rearranging the previous job sequence. A similar situation frequently occurs, e.g., in lot production or in operating-rooms scheduling. In the first case, lots must typically go through several working stages: Between two of them, it may be beneficial to reorganize the sequence, owing to, for instance, different characteristics of the lots in the next stage. In the second case, a tentative schedule for a certain planning period is built in advance and, later on, the final schedule is output trying to minimize changes with respect to the original plan.

In this context, we are interested in the following problem: We are provided with an initial job sequence to feed a single processing resource; we need to rearrange the jobs such that the new sequence performs well in terms of some given criterion. Depending on the considered setting, we also need to deal with a given set of feasible reconfigurations (such restrictions may be imposed, e.g., by the physical handling system of the plant) which are, somehow, not too distant from the original sequence. A special version of this problem is also considered in Alfieri et al. (2018a, b). The authors studied a rescheduling problem with the constraint that the jobs extracted from the given initial sequence can be re-inserted only in later positions, i.e., jobs can be postponed but not moved ahead of the schedule. This corresponds to a physical setting where jobs are transported on a conveyor that feeds a single processor and a robot or worker can pick them up and reinsert them later in the queue.

In this paper, we adopt a similar setting but with an additional set of restrictions. In particular, we consider the scenario in which the handling mechanism consists of a conveyor that feeds a single machine in a given sequence and a robot, placed along the line, that is able to alter this sequence. The robot may pick parts from the conveyor as they are moving, stacks them on a buffer of finite capacity from which it takes the parts and places them back on the conveyor, in their final positions (which is later in the original sequence due to the conveyor movement). Since the stack is managed according to a Last-In-First-Out (LIFO) policy, only the last part put into the stack can be extracted and re-inserted in the new sequence. This setting has been introduced in Nicosia et al. (2019) where the authors present some preliminary results on the corresponding rescheduling problem.

An area of research which is strictly related to the problem we address in this article, deals with *sequence coordination* in supply chain (SC). One of the main tasks in SC management is indeed coordination of several activities performed at different stages of the chain. An obvious overall goal consists in successfully meeting customers needs and achieving a good level of efficiency and performance. Usually, in a coordinated SC, two or more processes subject to their mutual coordination are considered. This involves fitting the schedules of different manufacturers together when some planned or unexpected schedule changes are experienced by one or more of them (Ivanov and Sokolov 2015). In this context, Agnetis et al. (2006) consider two consecutive stages of a supply chain where ideal job sequences (typically, different for the two stages) are given. The authors address a supply chain scheduling coordination problem consisting in finding a trade-off schedule that takes into account the ideal schedules of both stages. They propose a number of polynomial-time algorithms for different versions of the problem, namely from the point of view of the manufacturer, then from that of the supplier, and, finally, they consider the situation when both stages cooperate to obtain a satisfactory compromise schedule. A related coordination problem is addressed in Agnetis et al. (2001) where two departments in a manufacturing facility process the jobs in batches. In a first department, a setup is paid whenever a certain attribute changes from one batch to another, whereas in a second department, the setup is associated to a different attribute. The problem of finding a unique sequence of jobs in order to minimize the overall setup cost arises. The authors prove that the problem is NP-hard and propose an effective heuristic approach. The above coordination problems are also tightly connected to those addressed in *multi-agent scheduling*, a research field which received great attention more recently: Two or more agents have to agree on a fair, i.e., acceptable schedule of their distinct sets of jobs on a common processing resource [see, for instance, Leung et al. (2010), Perez-Gonzalez and Framinan (2014) and Agnetis et al. (2019)].

Another important and fruitful research stream, connected to the problem addressed here, concerns the so-called *rescheduling* (or dynamic scheduling). In many real-world scenarios, scheduling is an activity requiring frequent revisions due to unexpected changes such as, for instance, machine breakdown or unavailability, delay in the arrival of materials, job cancellation, due date changes, etc. With the terms rescheduling and dynamic scheduling, many authors indicate the problem of scheduling in the presence of real-time events; this includes the process of updating the current schedule to face previously unknown events such as the arrival of new jobs (Hall et al. 2007), disruptions (Nouiri et al. 2018), perturbation of the originally given or estimated data (Hall and Potts 2010), etc. A recent and effective application of these concepts in the health care sector can be found in Ballestín et al. (2019). Two different reviews of the state-of-the-art of currently developing research on dynamic scheduling are given in Ouelhadj and Petrovic (2009) and Vieira et al. (2003).

The most common strategies (called predictive-reactive), when facing any unexpected change in the scenario, consider both the possibility of local adjustments and a whole re-computation of the current schedule with the aim of (locally) improving shop efficiency. Together with the latter objective, it is also of interest to measure how much the new schedule deviates from the original schedule. This concept (usually referred to as *stability*) is important since, typically, modification costs increase with the magnitude of such deviation, whereas big and frequent schedule changes often may cause undesired nervousness phenomena due to a lack of continuity.

From the pioneering works by Daniels and Kouvelis (1995) and Wu et al. (1993) up to the most recent papers (see, e.g., Detti et al. 2019; Niu et al. 2019), *robustness* is a widely adopted concept in scheduling and it is an alternative (pro-active) approach that tries to design a schedule which *a priori* guarantees a certain level of efficiency, given a set of possible scenarios. This way stability is preserved while performance is kept above a fixed level.

Indeed, our problem can be viewed in the framework of the so-called *recoverable robustness*, see Liebchen et al. (2009). A recoverable robust solution is not necessarily feasible in all scenarios of a robust optimization problem, but it can be made feasible by applying a (simple, quick) recovery algorithm to it. This concept has been investigated in several application contexts (mostly in transportation problems) and there is a limited literature also in scheduling. For instance, in van den Akker et al. (2018) an initial solution of a scheduling problem is given and in each scenario some recovery actions are performed to make the solution acceptable again.

The algorithms presented in this work can be regarded as recovery algorithms to achieve certain objective benchmarks (rather than feasibility) in different problem settings which are illustrated below.

In the special rescheduling problem addressed in this paper, we consider several objective functions, namely total weighted completion time, maximum lateness, number of late jobs and weighted number of late jobs. We devise strictly polynomial-time optimization algorithms based on dynamic programming recursions for the first three objectives. In contrast, we prove that the problem is NP-hard when minimizing the weighted number of late jobs, but still permits a pseudo-polynomial dynamic programming algorithm in that case. The remainder of this work is organized as follows: After a rigorous statement of the problem in Sects. 2, 3, 4 and 5 are devoted to the first three objective functions and describe the corresponding *efficient* solution algorithms. For the problem with the weighted number of late jobs objective, in Section 6, we prove its complexity and propose a pseudo-polynomial algorithm. Section 7 illustrates by a short experimental study the behavior of the rescheduling process depending on the stack size. Finally, concluding remarks are given in Sect. 8.

2 Problem statement

In this section, we give a formal statement of the problem under consideration and introduce the notation used throughout the paper. Hereafter, we use the term "job" to refer to both, a physical piece of material which is processed by some machine or resource and the process itself (characterized by a certain duration and possible additional data).

Let us consider a deterministic single-machine environment where we are given a set J of n jobs that have to be scheduled according to a *regular*, i.e., non-decreasing, objective function $f(\sigma)$ of the job completion times $C_j(\sigma)$, $j = 1, \ldots, n$. For such a schedule $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ with $\sigma_k \in J$, $k = 1, \ldots, n$, if $i \leq j$, we refer to the ordered set of jobs $\langle \sigma_i, \sigma_{i+1}, \ldots, \sigma_j \rangle$ as the *subsequence* $\sigma(i, j)$. Moreover, for each job $j \in J$ we know its processing time p_j and, possibly, a due date d_j and a weight w_j. As usual for scheduling problems, we will assume that all these values are nonnegative integers. However, this property will be required only for the dynamic programming algorithm in Sect. 6. Additionally, we are given an initial sequence σ_0 in which the n jobs of set J are numbered from 1 to n. So, $\sigma_0 = \langle 1, 2, \ldots, n \rangle$ and we say that *job j is placed in the j-th position* to indicate that it is the j-th job of the sequence σ_0. Clearly, if $i, j \in J$ and $i \leq j$, we have $\sigma_0(i, j) = \langle i, i+1, \ldots, j \rangle$.

In the problem addressed here, we look for a new job sequence σ such that

(i) $f(\sigma)$ is minimum and
(ii) σ can be derived from σ_0 by applying a (constrained) number of *feasible moves*.

Any move in this scheduling environment is performed by a physical device (e.g., a robot arm) that operates on a sequence of parts, each associated to one job, arranged in an ordered sequence along a line (e.g., on a moving conveyor). The initial sequence on this line corresponds to σ_0. The considered device (i) picks up a job j; (ii) places j in the *stack* with bounded capacity S; (iii) possibly picks up other jobs and places them in the stack; (iv) picks the jobs from the stack, according to a LIFO policy, and places them back (on the conveyor) at a suitable position *later* in the sequence. We assume that there is always enough space between jobs to place even the whole content of the stack between any two jobs on the line.

In this setting, moved jobs can only be postponed, as in fact the robot arm picks the jobs from the line while the conveyor feeding the processor moves ahead. Hence, reinsertion can only happen in a later position of the sequence.

Definition 1 *A move $i \to j$, $i < j$, consists of deleting job i from a given sequence and reinserting it immediately after all jobs of the subsequence $\sigma_0(i + 1, j)$.*

Figure 1 illustrates the process for three consecutive moves on a sequence σ_0 with $n = 9$ jobs. The picture emphasizes the characteristics of two types of feasible moves for the above mentioned physical device. A move $1 \to 3$ in which job 1 is placed immediately after job 3 is performed first. Then, moves $5 \to 9$ and $7 \to 9$ are done: Job 5 is loaded into the stack directly before 7, which is then extracted from the stack and placed just after job 9. After that, 5 is placed back on the line, right after 9 and 7. It is clear that the latter operations require that the stack capacity S is at least 2.

Observe that, if we start from σ_0, due to the stack LIFO policy, Definition 1 is consistent even after a number of moves has been performed.

Following the above considerations, we are now characterizing the compatibility of two moves.

Definition 2 *Two moves $i_1 \to j_1$ and $i_2 \to j_2$ with $i_1 < i_2$ are feasible if:*

(i) either $j_1 < i_2$, i.e., $i_1 < j_1 < i_2 < j_2$, and the moves are called sequential*;*
(ii) or $i_1 < i_2 < j_2 \leq j_1$, and move $i_2 \to j_2$ is nested *in move $i_1 \to j_1$.*

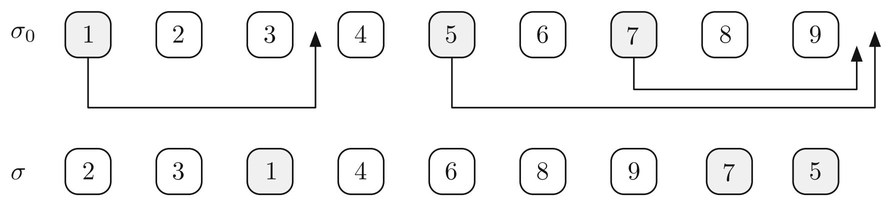


Fig. 1 Three feasible moves

Note that, as a consequence of Definition 1, it is easy to see that

- when move $i_2 \rightarrow j_2$ is nested in move $i_1 \rightarrow j_1$, i_2 precedes i_1 in the final sequence even if $j_1 = j_2$;
- although the device picks jobs in increasing index order, the sequence σ, obtained after a set of feasible moves considering the LIFO policy of the stack, is independent from the particular order in which the moves are performed.
- given a sequence $\sigma \neq \sigma_0$, if there exists a set of feasible moves to reach σ starting from σ_0, then such a set is unique.

We next define the *level* of a move:

Definition 3 *A move that does not contain nested moves is said to be at level* 1. *Recursively, a move m is at level* $\ell > 1$ *if* ℓ *is the smallest value such that m contains feasible nested moves at level up to* $\ell - 1$.

Clearly, a move at level ℓ is feasible only if $\ell \leq S$. In the example shown in Fig. 1, $1 \rightarrow 3$ and $5 \rightarrow 9$ are sequential moves while $7 \rightarrow 9$ is nested in $5 \rightarrow 9$. In particular, $5 \rightarrow 9$ is at level 2.

In the remainder of this paper, we adopt the following definition for a move.

Definition 4 *A move* $i \rightarrow j$ *at level* ℓ *is denoted as* (i, j, ℓ). *For notational convenience, we also define* (i, i, ℓ) *for every level* ℓ*, meaning that job i is not moved at all.*

Note that in our setting the stack capacity constraint imposes the index ℓ to be an integer within the range 1 to S, i.e., there must not be more than S moves nested inside each other.

Definition 5 *The set of feasible schedules* $\mathcal{F}_S$ *for the rescheduling problem where the stack capacity is limited by S, is comprised of all the schedules resulting from a set of feasible moves at levels up to a maximum of S starting from* σ_0.

In this paper, we consider the minimization of three classical objective functions in scheduling theory to evaluate the sequence obtained from σ_0 through the LIFO-constrained moves: total weighted completion time, maximum lateness (with extension to any regular function) and number of late jobs. For the latter, we consider both, the weighted and unweighted case.

For any feasible schedule σ of the jobs of J and any job $j \in J$, we define the following quantities: $C_j(\sigma)$ is the completion time of j in σ, $L_j(\sigma) = C_j(\sigma) - d_j$ is the lateness of job j in σ and

$$U_j(\sigma) = \begin{cases} 1, & \text{if } C_j(\sigma) > d_j, \\ 0, & \text{otherwise,} \end{cases}$$

indicates if job j is late. Note that the completion time of job j in the initial sequence σ_0 is given by $C_j(\sigma_0) = \sum_{k=1}^{j} p_k$ and its lateness by $L_j(\sigma_0)$. We indicate with $P(i, j) = \sum_{k=i}^{j} p_k$ the total processing time of the subsequence $\sigma_0(i, j)$ and with $W(i, j) = \sum_{k=i}^{j} w_k$ its total weight. Moreover, for a given subsequence $\sigma(i, j)$, the maximum lateness within the subsequence is

$$L_{\max}(\sigma(i, j)) = \max_{k=\sigma_i,\ldots,\sigma_j} \{L_k(\sigma)\}.$$

Furthermore, if $\phi_j : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$, $j \in J$, are given regular functions, we denote the maximum of regular function objective as

$$\Phi_{\max}(\sigma) = \max_{j \in J}\{\phi_j(C_j(\sigma))\}.$$

In accordance with the standard Graham's three field notation, we indicate the problems addressed in this paper as follows: $1|\text{resch-LIFO}|\sum w_j C_j$, $1|\text{resch-LIFO}|L_{\max}$, $1|\text{resch-LIFO}|\Phi_{\max}$, $1|\text{resch-LIFO}|\sum U_j$ and $1|\text{resch-LIFO}|\sum w_j U_j$ where the last field refers to the particular objective we want to minimize in rearranging the given initial schedule σ_0 through the above described mechanism.

In this paper, we also consider the following special generalization of the above problems. Let $\Omega \subseteq J$ be a given subset of *movable* jobs and consider as feasible solutions only the schedules obtained from σ_0 by a set of feasible moves (at levels up to a maximum of S), where only moves $i \rightarrow j$ with $i \in \Omega$ are involved. Jobs from $J \setminus \Omega$ cannot be moved, whereas jobs from Ω *may* be moved but it is not mandatory to move them. In the remainder of the paper, we will refer to this variant as Ω-*constrained* problem. Observe that our original rescheduling problems are special cases of these Ω-constrained problems in which $\Omega = J$.

3 Total weighted completion time

We start our study by considering the minimization of the weighted sum of job completion times, for which we present a polynomial time dynamic program.

First, we can observe that for any partition of a schedule $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ in Q adjacent subsequences $\sigma(u_q, v_q) = \langle \sigma_{u_q}, \ldots, \sigma_{v_q} \rangle$ with $u_q \leq v_q$ such that $u_1 = 1$, $v_Q = n$ and $u_q = v_{q-1} + 1$ for all $q = 2, \ldots, Q$, the objective function can be expressed as follows:

$$f^{(1)}(\sigma) = \sum_{j \in J} w_j C_j(\sigma) = \sum_{q=1}^{Q} f^{(1)}(\sigma(u_q, v_q)), \tag{1}$$

where the term $f^{(1)}\left(\sigma(u_q, v_q)\right)$ denotes the total weighted completion time for the subsequence $\sigma(u_q, v_q)$, which in turn is the sum of the weighted completion times of the jobs in the subsequence. Denoting by J_q the set of jobs of subsequence $\sigma(u_q, v_q)$, clearly we have:

$$f^{(1)}\left(\sigma(u_q, v_q)\right) = \sum_{j \in J_q} w_j C_j(\sigma) \quad \forall q = 1, \ldots, Q. \tag{2}$$

Consider the set $\mathcal{M}$ of all feasible moves (i, j, ℓ) that allows to reach σ starting from σ_0. Let $\widetilde{\mathcal{M}} \subseteq \mathcal{M}$ indicate the set of moves (i, j, ℓ) which are not nested in any other move $(h, k, \ell') \in \mathcal{M}$ at a higher level $\ell' > \ell$. By definition, the moves $\widetilde{\mathcal{M}}$ are sequential moves. In $\widetilde{\mathcal{M}}$, we also include *identity* moves $(i, i, 1)$ which correspond to not moving job i. (Clearly, parts of σ which are unchanged with respect to σ_0, i.e., if the subsequence $\sigma(i, j) = \sigma_0(i, j)$, can be represented by a set of consecutive identity moves $(k, k, 1)$ with $k = i, \ldots, j$.) Let us denote $\widetilde{\mathcal{M}}$ as follows:

$$\widetilde{\mathcal{M}} = \{(u_1, v_1, \ell_1), (u_2, v_2, \ell_2), \ldots, (u_r, v_r, \ell_r)\} \tag{3}$$

with $1 = u_1 \leq v_1$, $v_r = n$, $v_{q-1} + 1 = u_q \leq v_q$, $q = 2, \ldots, r$. (Note that r in (3) plays a different role than Q in (2).) Because of the additivity principle (1), the cost $f^{(1)}(\sigma)$ of schedule σ can be computed by adding—to the cost of σ_0—the contributions of the r feasible moves of $\widetilde{\mathcal{M}}$. Note that in (3) each move (u_q, v_q, ℓ_q) may contain nested moves, so that its contribution depends on these nested moves as well. For instance, the sequential moves of $\widetilde{\mathcal{M}}$ in the example illustrated by Fig. 1 are $(1, 3, 1)$, $(4, 4, 1)$ and $(5, 9, 2)$. Move $(5, 9, 2)$ contains the nested move $(7, 9, 1)$, and therefore, its contribution must also account for the one of $(7, 9, 1)$.

Hereafter, we present a dynamic programming algorithm for determining the optimal set of moves yielding a minimum cost schedule σ^*, starting from σ_0. The correctness of this algorithm straightforwardly follows from the following observation which corresponds to a standard optimality principle: In any optimal schedule σ^*, there exists a partition as in (3), such that each subsequence in the partition is optimal for the subproblem containing only the jobs of that subsequence. The basic step of the dynamic program consists of computing the cost of a move (i, j, ℓ), at level ℓ based on the knowledge of optimal costs for any subsequence of $\sigma(i, j)$ in which the stack capacity is $\ell - 1$.

Definition 6 *The cost $c(i, j, \ell)$ is defined as the* minimum *total weighted completion time* variation *when a move (i, j, ℓ') at some level $\ell' \leq \ell$ is performed. The cost $\mu^*(i, j, \ell)$ is the minimum cost variation of subsequence $\sigma_0(i, j)$, when the stack capacity is equal to ℓ.*

Observe that $c(i, j, \ell)$ includes the optimal cost variation produced by all possible nested moves at lower levels, and hence, it takes into account information of all $c(h, k, \ell - 1)$ for $i < h \leq k \leq j$ which in turn are summed up into $\mu^*(i + 1, j, \ell - 1)$. While $\mu^*(i, j, \ell)$ gives the optimal cost variation for the subproblem implied by i and j, $c(i, j, \ell)$ considers the particular solution where the move $i \rightarrow j$ must be performed.

Hereafter, we show how to compute the variations in the schedule overall cost, for different moves and their combinations. We first consider the effect of *single moves* on the total weighted completion time. In this context, the following statement results directly from the definition of the completion time.

Observation 7 *The completion time of each job preceding i and following j in the initial sequence σ_0 is not affected by the move (i, j, ℓ).*

Based on the initial schedule σ_0, the *cost variation* $m(i, j)$ due to moving job i after job j is expressed as

$$m(i, j) = w_i \sum_{k=i+1}^{j} p_k - p_i \sum_{k=i+1}^{j} w_k \quad \forall i, j \in J, \ i < j. \tag{4}$$

The first term in (4) indicates the increase of the weighted completion time of job i, while the second term indicates the total decrease of the weighted completion times of the jobs in $\sigma_0(i + 1, j)$. In addition, because of the definition of the cost of a move (4), it is easy to verify the following:

Observation 8 *The effect of a move (i, j, ℓ) at any level $\ell > 1$ does not depend on the order of the jobs in the subsequence $\sigma_0\,(i + 1, j)$.*

As a consequence, because of the LIFO constraint, the cost variation due to a *single* move (i, j, ℓ) is always equal to $m(i, j)$, for all possible levels ℓ.

As far as *sequential moves* are concerned, we recall that, owing to Eq. (1), the effect of two sequential moves (i_1, j_1, ℓ_1) and (i_2, j_2, ℓ_2), $j_1 < i_2$, on the objective $f^{(1)}$ equals the sum of the effects of each move.

Moves at level $\ell = 1$ cannot contain any nested moves, so the cost $c(i, j, 1)$ of a move is equal to $m(i, j)$. Obviously, when a job i is not moved, the associated cost is null and $c(i, i, 1) = 0$. Hence, we may give a recursive expression of the optimal cost variation for the subsequence $\sigma_0(i, j)$ as follows:

$$\begin{cases} \mu^*(i,i,1) = 0 & \forall i \in J \\ \mu^*(i,j,1) = \min\{\min_{k \in \sigma_0(i,j-1)}\{c(i,k,1) & \forall i, j \in J,\ i < j. \\ \quad + \mu^*(k+1,j,1)\}, c(i,j,1)\} & \end{cases} \tag{5}$$

Even when dealing with *nested moves*, i.e., for $\ell > 1$, the cost variation of a set of moves is additive: Following the same line of arguments yielding to Observation 8 and Eq. (4), the overall effect on $f^{(1)}$ of two (or more) nested moves is equal to the sum of the effects of each move.

In order to compute the cost $c(i, j, \ell)$ for a move (i, j, ℓ') with $\ell' \le \ell$, one clearly has to take into account the possibility of optimally rearranging the subsequence $\sigma_0(i + 1, j)$ with moves at levels lower than ℓ'. Thus, after computing all optimal cost variations $\mu^*(i, j, \ell - 1)$ obtainable from a set of moves up to level $\ell - 1$ for the subsequence $\sigma_0(i, j)$, it is possible to determine the (optimal) cost $c(i, j, \ell)$. This in turn allows to compute the optimal cost variation $\mu^*(i, j, \ell)$ for the subsequence $\sigma_0(i, j)$ at level ℓ. Formally, the following recursion holds:

$$\begin{cases} c(i,j,1) = m(i,j) & \forall i, j \in J,\ i < j \\ c(i,i,\ell) = 0 & \forall i \in J,\ \forall \ell = 1, \ldots, S \\ c(i,j,\ell) = m(i,j) & \forall i, j \in J,\ i < j,\ \forall \ell = 2, \ldots, S. \\ \quad + \mu^*(i+1,j,\ell-1) & \end{cases} \tag{6}$$

In Eq. (6), $\mu^*(i, j, \ell)$ is computed in a similar way as in (5):

$$\begin{cases} \mu^*(i,i,\ell) = 0 & \forall i \in J \\ \mu^*(i,j,\ell) = \min\{\min_{k \in \sigma_0(i,j-1)}\{c(i,k,\ell) & \forall i, j \in J,\ i < j. \\ \quad + \mu^*(k+1,j,\ell)\}, c(i,j,\ell)\} & \end{cases} \tag{7}$$

Finally, the optimal solution value is found by computing $\mu^*(1, n, S)$ such that the objective function value of the optimal schedule equals

$$f^{(1)}(\sigma^*) = f^{(1)}(\sigma_0) + \mu^*(1, n, S).$$

As for the computational complexity of the above dynamic program, we observe that the computation of all $m(i, j)$, i.e., the costs at level $\ell = 1$, requires $O(n^2)$ time. For each level $\ell = 1, \ldots, S$ and for each ordered pair of jobs (i, j), the algorithm must compute the cost $c(i, j, \ell)$ of the corresponding move and the optimal subsequence cost $\mu^*(i, j, \ell)$. Computing all c values requires $O(n^2)$ time at each level, since each value is computed in $O(1)$, while the computation of $\mu^*(i, j, \ell)$ in Eq. (7) has a cost of $O(n^3)$ time for each level ℓ. Hence, the overall complexity of the algorithm is $O(n^3 S)$ which is upper bounded by $O(n^4)$.

The results above can be summarized as follows.

Theorem 9 *Problem $1|resch\text{-}LIFO|\sum w_j C_j$ is polynomially solvable in time $O(n^4)$.*

3.1 Extension to the Ω-constrained problem

It is easy to handle the special case in which only jobs from a given subset $\Omega \subseteq J$ of the jobs can be moved. For this purpose, we just make moves of jobs in Ω extremely costly. Hence, we can still use Recursions (6) and (7) but with the following adjusted costs at level $\ell = 1$ for a large enough constant M:

$$c(i, j, 1) = \begin{cases} M, & \text{if } i \in J \setminus \Omega,\ i < j, \\ \text{as in Expression (6)}, & \text{otherwise}, \end{cases}$$

for all $i, j \in J$ with $i \le j$.

4 Maximum of regular functions minimization

In this section, we first focus on the minimization of the maximum lateness of the jobs and present a dynamic programming solution algorithm related to the one presented in the previous Sect. 3. We then discuss how the same algorithm can be used to solve the more general problem $1|\text{resch-LIFO}|\Phi_{\max}$.

4.1 Minimization of maximum lateness

Recall that for a given schedule σ, we indicate by $L_j(\sigma) = C_j(\sigma) - d_j$, $L_{\max}(\sigma) = \max_{j \in J}\{L_j(\sigma)\}$, and $L_{\max}(\sigma(i, j)) = \max_{k=\sigma_i,\ldots,\sigma_j}\{L_k(\sigma)\}$.

Here, the objective function for any partition of a schedule $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ in Q adjacent subsequences $\sigma(u_q, v_q) = \langle \sigma_{u_q}, \ldots, \sigma_{v_q} \rangle$ with $u_q \le v_q$ such that $u_1 = 1$, $v_Q = n$ and $u_q = v_{q-1} + 1$ for all $q = 2, \ldots, Q$, fulfills the following decomposition property:

$$L_{\max}(\sigma) = \max_{q=1,\ldots,Q} \left\{ L_{\max}(\sigma(u_q, v_q)) \right\}.$$

We first consider the effect of single moves on the maximum lateness. A straightforward consequence of Observation 7 is that a move (i, j, ℓ) does not affect the lateness of any job preceding i and following j in the initial sequence σ_0. Hence, even in this case, moves involving disjoint subsequences can be evaluated independently.

Differently from Eq. (4), we are not looking at the *variation* in the objective function due to a move (i, j, ℓ), but rather at the optimal value that the objective function would take if move (i, j, ℓ) were executed. Hence, we define the cost $g(i, j, \ell)$ as the *minimum* value of the maximum lateness in subsequence $\sigma(i, j)$ when moves up to level ℓ are done within it. For a single move (i, j, ℓ), at any level, the lateness of all jobs in $\sigma_0(i+1, j)$ decreases by p_i while the lateness of job i increases by $P(i+1, j) = \sum_{k=i+1}^{j} p_k$. As a consequence, we may compute the cost at level 1 as follows:

$$g(i, j, 1) = \max\{L_i(\sigma_0) + P(i+1, j),\\ L_{\max}(\sigma_0(i+1, j)) - p_i\} \quad \forall i, j \in J,\ i < j.$$

The term $L_i(\sigma_0) + P(i+1, j)$ indicates the new lateness of job i while the term $L_{\max}(\sigma(i+1, j)) - p_i$ indicates the new maximum lateness of the subsequence $\sigma_0(i+1, j)$. We also define the cost of not moving a job i:

$$g(i, i, 1) = L_i(\sigma_0) \quad \forall i \in J.$$

Similar to the problem with total weighted completion time objective, also in this case the dynamic programming algorithm is based on computing the cost of a move (i, j, ℓ), at level ℓ starting from the optimal costs of any subsequence of $\sigma(i, j)$ at level $\ell - 1$.

Definition 10 *The cost $\lambda^*(i, j, \ell)$ is the value of the optimal solution of the subproblem restricted to subsequence $\sigma_0(i, j)$, when the stack capacity is equal to ℓ.*

Note that, as before, for $\ell > 1$ the computation of $g(i, j, \ell)$ takes into account the lateness values produced by all possible nested moves at lower levels, i.e., the quantities $g(h, k, \ell-1)$ for $i < h \le k \le j$ which in turn yield the value $\lambda^*(i+1, j, \ell-1)$.

In general, different orders of the jobs in the subsequence $\sigma_0(i+1, j)$ imply different effects of a move (i, j, ℓ) on the objective function. Similar to the total weighted completion time case, when computing the cost $g(i, j, \ell)$ we need to take into account the possibility of optimally rearranging the subsequence $\sigma_0(i+1, j)$ with moves at lower levels. Thus, only after computing the optimal cost $\lambda^*(i+1, j, \ell-1)$, obtainable from a set of moves at level at most $\ell - 1$ for the subsequence $\sigma_0(i+1, j)$, it is possible to determine $g(i, j, \ell)$.

Starting from the $g(\cdot)$ values at level ℓ, it is then possible to compute the optimal costs $\lambda^*(i, j, \ell)$ for each subsequence $\sigma_0(i, j)$ at level ℓ.

In conclusion, the following recursion holds:

$$\begin{cases} g(i, j, 1) = \max\{L_i(\sigma_0) + P(i+1, j), \\ \quad L_{\max}(\sigma_0(i+1, j)) - p_i\} & \forall i, j \in J,\ i < j \\ g(i, i, \ell) = L_i(\sigma_0) & \forall i \in J,\ \forall \ell = 1, \dots, S \\ g(i, j, \ell) = \max\{L_i(\sigma_0) + P(i+1, j), \\ \quad \lambda^*(i+1, j, \ell-1) - p_i\} & \forall i \in J,\ i < j,\ \forall \ell = 2, \dots, S. \end{cases} \tag{8}$$

With Eq. (8), $\lambda^*(i, j, \ell)$ is computed in an analogous way as done in (5):

$$\begin{cases} \lambda^*(i, i, \ell) = L_i(\sigma_0) & \forall i \in J,\ \forall \ell = 1, \dots, S \\ \lambda^*(i, j, \ell) = \min\{\min_{k \in \sigma_0(i, j-1)} & \forall i, j \in J,\ i < j,\ \forall \ell = 1, \dots, S. \\ \quad \{\max\{g(i, k, \ell), \lambda^*(k+1, j, \ell)\}\}, \\ \quad g(i, j, \ell)\} \end{cases} \tag{9}$$

In the above Eq. (9), $\lambda^*(i, i, \ell)$ always equals the "initial" value $L_i(\sigma_0)$ while, only when we consider a (proper) subsequence with more than one job, the actual costs associated with moves are accounted for. In particular, the optimal solution value $\lambda^*(i, j, \ell)$ can be computed as the best alternative among the solutions in which a move (i, k, ℓ), $k = i, \dots, j$, is done: option $k = i$ corresponds to a solution in which job i is not moved; if the best alternative is $k = j$, then $\lambda^*(i, j, \ell) = g(i, j, \ell)$; if $i < k < j$, the maximum lateness is the largest between the cost of move $i \to k$ at level at most ℓ and the optimal cost of the subsequence $\sigma_0(k+1, j)$ at level at most ℓ. Finally, the optimal solution value is found by computing $f^{(2)}(\sigma^*) = \lambda^*(1, n, S)$.

The computational complexity can be computed as in Section 3. The initialization, i.e., the computation of costs at level $\ell = 1$, requires $O(n^2)$ time. For each level $\ell = 1, \dots, S$ the algorithm must compute the costs $g(\cdot)$ and $\lambda^*(\cdot)$ for each pair of jobs (i, j) (with i preceding j in σ_0). Computing all g values requires $O(n^2)$ time at each level since each value is computed in $O(1)$. The optimal costs $\lambda^*(i, j, \ell)$ can be computed, through Recursion (9), in $O(n^3)$ for each level ℓ. Hence, the overall complexity of the algorithm is $O(n^3 S)$ which is upper bounded by $O(n^4)$.

Summarizing, we have the following result:

Theorem 11 *Problem $1|resch\text{-}LIFO|L_{\max}$ is polynomially solvable in time $O(n^4)$.*

4.2 Extension to maximum of regular functions objective

The correctness of the above dynamic programming algorithm is based on a decomposition principle, ensuring that parts of an optimal schedule are also optimal for the subproblems containing only the jobs of those parts. This in turn implies that an optimal subsequence is independent of its starting time. In other words, consider an optimal schedule σ^* and one of its subsequences $\sigma^*(i, j)$ starting at, say, time

t. Then, the same sequence $\sigma^*(i, j)$ is optimal for the subproblem pertaining only jobs $\{i, i+1, \ldots, j\}$ (starting, e.g., at time $t = 0$).

Unfortunately, when dealing with the minimization of the maximum of regular functions of the job completion times, such a property does not hold anymore. In fact, consider as an example the following trivial (sub-)problem with only two jobs $\{1, 2\}$, with $p_1 = 1$, $p_2 = 4$ and linear regular functions $\phi_1(x) = 0.2x + 15$, $\phi_2(x) = 2x$. The optimal schedule starting, e.g., at time $t = 0$ is $\langle 1, 2\rangle$, while when the starting time is $t = 10$, the optimal sequence becomes $\langle 2, 1\rangle$. As a consequence, no immediate generalization of the efficient dynamic programming approach illustrated Recursions (8) and (9) may guarantee optimality.

However, problem $1|\text{resch-LIFO}|\Phi_{\max}$ is still efficiently solvable by using a simple alternative procedure presented hereafter. The idea is to perform a binary search over the optimal objective values $\Phi_{\max}$, each time solving a suitable instance of $1|\text{resch-LIFO}|L_{\max}$.

First observe that we may compute an interval in which the optimal value of the objective varies:

$$\alpha = \max_{j\in J}\{\phi_j(p_j)\} \leq \Phi_{\max} \leq \max_{j\in J}\{\phi_j(P(1, j))\} = \omega. \quad (10)$$

Let I be an instance of $1|\text{resch-LIFO}|\Phi_{\max}$ and $\bar{\Phi}$ be a fixed target value for the optimal objective value of I. Clearly, $\bar{\Phi}$ must be chosen in the interval $[\alpha, \omega]$. We are asking if there exist solution schedules for our instance of $1|\text{resch-LIFO}|\Phi_{\max}$ with optimal objective value not larger than $\bar{\Phi}$. We refer to such an instance as a YES-instance for $\bar{\Phi}$.

For all jobs $j \in J$, compute a deadline $\tau_j(\bar{\Phi}) = \max_{t\geq 0}\{\phi_j(t) \leq \bar{\Phi}\}$. In general, since $\phi_j(\cdot)$ is non-decreasing, each such deadline may be efficiently computed in time, say, c. In the worst case, c is $O(\log(\sum_{j\in J} p_j))$; however, it is a reasonable to think of special cases where c is constant. Then solve an instance of $1|\text{resch-LIFO}|L_{\max}$ with due date $d_j = \tau_j(\bar{\Phi})$ for $j \in J$. Let $\sigma^*(\bar{\Phi})$ be an optimal solution and $L_{\max}(\bar{\Phi})$ be the corresponding optimal objective value. If $L_{\max}(\bar{\Phi}) \leq 0$, then I is a YES-instance for $\bar{\Phi}$ and $\sigma^*(\bar{\Phi})$ is a solution of $1|\text{resch-LIFO}|\Phi_{\max}$ with objective not larger than $\bar{\Phi}$.

Then, an optimal solution for our instance I of $1|\text{resch-LIFO}|\Phi_{\max}$ can be found by looking for the minimum value of $\bar{\Phi}$ such that $L_{\max}(\bar{\Phi}) \leq 0$. As we already mentioned, this can be done by performing a binary search over the interval $[\alpha, \omega]$ of Expression (10).

The overall complexity of the above procedure depends on the computational costs of the dynamic programming for the $1|\text{resch-LIFO}|L_{\max}$ problem, the deadlines computation and the binary search. In conclusion, assuming that we are able to efficiently compute the values $\phi_j(\cdot)$ and $\tau_j(\cdot)$ for all $j \in J$, and the values of suitable bounds for the objective α and ω, we may state that:

Theorem 12 *Problem $1|\text{resch-LIFO}|\Phi_{\max}$ is polynomially solvable in time $O((n \cdot c + n^4)\log(\omega - \alpha))$ where c is the cost for computing a job deadline.*

4.3 Extension to the Ω-constrained problem

Similar to Sect. 3.1, in order to handle the special case where not all jobs are movable, the initialization in Recursion (8) is adjusted as follows (for a large enough constant M):

$$g(i, j, 1) = \begin{cases} M, & \text{if } i \in J \setminus \Omega,\ i < j, \\ \text{as in Expression (8)}, & \text{otherwise,} \end{cases}$$

for all $i, j \in J$ with $i \leq j$.

5 Minimization of the number of late jobs

This section provides a dynamic programming algorithm to minimize the number of late jobs, $f^{(3)}(\sigma) = \sum_{j\in J} U_j(\sigma)$, where the quantity $U_j(\sigma)$ assumes value 1 if and only if job j is late in schedule σ, i.e., if its lateness $L_j(\sigma)$ is strictly positive.

Given a subsequence of a schedule, the number of late jobs in this subsequence depends not only on the processing times and due dates of the jobs in this subsequence, but generally also on its starting time. As a consequence, an (optimal) rearrangement of a subsequence that minimizes the number of late jobs depends on its starting time. This is illustrated in the following example: consider the sequence $\sigma = \langle 1, 2, 3\rangle$ with $p_1 = 7$, $p_2 = p_3 = 10$, $d_1 = d_2 = 25$ and $d_3 = 15$. It is easy to see that, in order to minimize the number of late jobs in the subsequence $\sigma(2, 3)$, the ordering $\langle 3, 2\rangle$ should be preferred if the sequence starts at $t = 0$ (i.e., move $1 \to 3$ is performed), whereas for $t = 7$ (i.e., job 1 is not moved) the arrangement $\langle 2, 3\rangle$ is better. However, the following result indicates that the relative order of the lateness values is independent from the starting time of a subsequence.

Observation 13 *Let $\sigma_q(i, j)$ be a feasible arrangement of the subsequence $\sigma(i, j)$ that minimizes the q-th largest lateness value, $q = 1, \ldots, j - i + 1$. Then, $\sigma_q(i, j)$ does not depend on the starting time of $\sigma_q(i, j)$.*

Note that in general the arrangements minimizing the q-th largest lateness are different for different values of q. However, when a subsequence is moved earlier (or delayed), all lateness values decrease (or increase) by the same amount, and hence, their order remains unchanged. Observation 13 holds for any possible definition of "feasible arrangement"; in particular, in this paper we clearly refer to the set of sequences resulting from a set of feasible moves at levels up to a given maximum.

In the reminder of this section, we refer to *ℓ-rearrangement* to indicate a feasible rearrangement of a subsequence that can be obtained by moves up to level ℓ.

5.1 Dynamic programming

By exploiting Observation 13, we subsequently devise a strongly polynomial time dynamic program for minimizing the number of late jobs.

In the dynamic program, we store for each subsequence of jobs not only the current number of late jobs, but we also record the minimum required reduction of the starting time for this subsequence to reach any number of late jobs, as we formally define below:

Definition 14 *For each $i, j \in J$ with $i \leq j, m = 0, \ldots, j - i$ and $\ell = 0, \ldots, S$, let $s(i, j, m, \ell)$ be the* minimum decrease of the starting time *of the subsequence $\sigma_0(i, j)$ to obtain m late jobs in it when the subsequence can be rearranged with moves up to level ℓ.*

Hereinafter, we will refer to this information as the *state* of the dynamic programming algorithm. Note that $s(i, j, m, \ell)$ may also be negative (i.e., an increase of the starting time) if the number of late jobs in $\sigma_0(i, j)$ is smaller than m.

Definition 14 does not include the case $m = j - i + 1$ since the case where *all* jobs in a subsequence are late never requires reducing the starting time of the subsequence and thus is not relevant for a LIFO rearrangement.

Due to Observation 13, all ℓ-rearrangements of a subsequence $\sigma_0(i, j)$ that allow us to obtain m late jobs in it, when the subsequence is moved earlier by s', are dominated by the ℓ-rearrangements of $\sigma_0(i, j)$ that allow to obtain the same number of m late jobs in it when the subsequence is moved by $s'' < s'$. By this dominance relation between rearrangements, we can restrict ourselves to recording the minimum decrease of the starting time[1] and thus avoid the combinatorial explosion of the problem.

For every given number m, the value of state $s(i, j, m, \ell)$ 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 $\sigma_0(i, j)$. (Note that if $k = i$ we are looking at an identity move, while if $k > i$ we are considering all possible moves nested in $i \rightarrow k$.)

For each candidate k, we compute three multi-sets of lateness values, coming from three job subsets: namely the jobs nested in the move, the moved job and the other jobs. From the multi-set obtained by the union of these multi-sets, we select the minimum reduction in time that is needed to have m late jobs in $\sigma_0(i, j)$ for each value of m. In particular, some late jobs can be obtained from subsequence $\sigma_0(i+1, k)$ after reducing its starting time by p_i; some late jobs can be obtained from subsequence $\sigma_0(k+1, j)$ whose starting time has not changed; and an additional late job is possibly the moved job i that is delayed by $P(i+1, k)$.

Hereafter, we use the notation $\max_{[q]}(\mathcal{A})$ to indicate the q-th largest value in the multi-set of integer values $\mathcal{A}$. To be precise, we refer to the q-th entry in the non-increasingly sorted multi-set.

At level $\ell = 0$, the values $s(i, j, m, 0)$ are evaluated on the initial sequence using the lateness values of the jobs in it. Let $\mathcal{L}(\sigma_0(i, j))$ be the multi-set of lateness values $\{L_k(\sigma_0) : k \in \sigma_0(i, j)\}$. With the notation introduced above, we initialize

$$\begin{aligned} &s(i, j, m, 0) = \max_{[m+1]}(\mathcal{L}(\sigma_0(i, j))) \\ &\quad \forall i, j \in J,\ i \leq j,\ \forall m = 0, \ldots, j - i. \end{aligned} \tag{11}$$

For levels $\ell \geq 1$, the recursive extension rule is as follows. For each subsequence $\sigma_0(i, j)$ and for each level $\ell = 1, \ldots, S$, we first consider all possible ways in which this subsequence can be feasibly rearranged with moves up to level ℓ. For this purpose, we compute the multi-set $\mathcal{S}(i, j, k, \ell)$ containing, for each possible move (i, k, ℓ') at any level $\ell' \leq \ell$, the $s(\cdot)$ values for all numbers $m = 0, \ldots, j - i$ of late jobs:

$$\begin{aligned} \mathcal{S}(i, j, k, \ell) = &\bigcup_{u=0}^{k-i-1} \{s(i+1, k, u, \ell - 1) - p_i\} \\ &\cup \bigcup_{u=0}^{j-k-1} \{s(k+1, j, u, \ell)\} \\ &\cup \{C_k(\sigma_0) - d_i\}. \end{aligned} \tag{12}$$

In (12), we distinguish the union of three multi-sets: The first one contains $k - i$ values computed from the non-dominated $(\ell-1)$-rearrangements of the subsequence $\sigma_0(i+1, k)$ whose jobs, due to the move $i \rightarrow k$, start earlier by p_i time units. The second multi-set refers to the $s(\cdot)$ values of the jobs from $k+1$ to j whose starting times are not affected by the move. Finally, the third term corresponds to the lateness of job i after move $i \rightarrow k$ is performed. Recall that Observation 13 guarantees that a $(\ell - 1)$-rearrangement of subsequence $\sigma_0(i+1, k)$ producing (at most) m late jobs does not change when the subsequence is moved earlier by p_i.

The $(m + 1)$-largest entry of the multi-set $\mathcal{S}(i, j, k, \ell)$ represents the necessary amount of time by which we need to bring forward $\sigma_0(i, j)$ after a move (i, k, ℓ') with $\ell' \leq \ell$, in order to have m late jobs. Thus, the value of the new state corresponds to the move $i \rightarrow k$ such that this decrease of the

[1] Clearly, the values $s(i, j, m, \ell)$ are finite, even if they are negative. Since $m < j - i + 1$, there always remains at least one on-time job which forbids an arbitrarily large increase of the starting time.

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

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

Table 2 State values $s(i, j, m, 0)$ for all $i, j \in \{1, 2, 3, 4\}$ with $i \le j$ and $m \in \{0, \dots, j - i\}$

			m			
			0	1	2	3
$j = 1$	$i = 1$	$s(1, 1, m, 0)$	− 20			
$j = 2$	$i = 2$	$s(2, 2, m, 0)$	20			
	$i = 1$	$s(1, 2, m, 0)$	20	− 20		
$j = 3$	$i = 3$	$s(3, 3, m, 0)$	30			
	$i = 2$	$s(2, 3, m, 0)$	30	20		
	$i = 1$	$s(1, 3, m, 0)$	30	20	− 20	
$j = 4$	$i = 4$	$s(4, 4, m, 0)$	20			
	$i = 3$	$s(3, 4, m, 0)$	30	20		
	$i = 2$	$s(2, 4, m, 0)$	30	20	20	
	$i = 1$	$s(1, 4, m, 0)$	30	20	20	− 20

starting time is minimum and can be computed as follows:

$$s(i, j, m, \ell) = \min_{k \in \sigma_0(i,j)} \left\{ \max_{[m+1]}(\mathcal{S}(i, j, k, \ell)) \right\}. \qquad (13)$$

In computing the $s(i, j, m, \ell)$ values, we keep a specific order for i, j, and ℓ: This is illustrated in the RECURSION procedure of Algorithm 1. The optimal solution value, $f^{(3)}(\sigma^*) = \sum_{j\in J} U_j(\sigma^*)$, is finally given by $\min\{m : s(1, n, m, S) \le 0\}$.

To illustrate the non-trivial handling of multi-sets in the dynamic programming recursion, we give the following example.

Example Consider a schedule $\sigma_0 = \langle 1, 2, 3, 4\rangle$ with 4 jobs. The respective processing times and due dates are given in Table 1. Since the example is intended to illustrate how a state value is computed, we restrict ourselves to the case $S = 1$. An analogous procedure is to be used for larger values of S.

In order to compute the minimum number of late jobs in the sequence σ_0 with moves up to level 1, we have to calculate all state values $s(i, j, m, \ell)$ with $i, j \in \{1, 2, 3, 4\}$, $i \le j$, $m = 0, \dots, j - i$ and $\ell \in \{0, 1\}$.

At level $\ell = 0$, the initialization according to (11) needs to be performed. Since the lateness value of each job in σ_0 is already given in Table 1, the $s(\cdot)$ values can easily be obtained. For example, $s(1, 4, 2, 0)$ corresponds to the third largest value in the multi-set $\{30, 20, 20, -20\}$ which is 20. All state values at level 0 are presented in Table 2.

At level $\ell = 1$, the subsequences are examined in a specific order as shown by the rows in Table 3: In an outer loop, index j spans the range 1 to 4 while in an inner loop index i runs from j down to 1. In order to compute $s(i, j, m, 1)$ with (13), we first have to compute $\mathcal{S}(i, j, k, 1)$ for all $k \in \sigma_0(i, j)$ according to (12).

As $\mathcal{S}(i, j, k, 1)$ is the union of three multi-sets, each of the three sets is given separately in the middle part of Table 3: On the left, indicated by $s(i + 1, k, u, 0) - p_i$ from (12), the lateness values of the jobs in the subsequence, which are moved to an earlier starting time, are shown. In the middle, the lateness values $s(k + 1, j, u, 1)$ of the jobs in the later part of the subsequence are given. These values come from the previous rows in the table. For this reason, the $s(\cdot)$ values need to be evaluated in the order stated above. On the right, the lateness value $C_k(\sigma_0) - d_i$ corresponding to the moved job is given.

The detailed description of the computation of a selected multi-set follows on the basis of Figure 2: As illustrated, when move $1 \to 2$ at level $\ell = 1$ is performed in the sequence $\sigma_0(1, 4)$, the values of the multi-set $\mathcal{S}(1, 4, 2, 1)$ are computed according to (12) as follows: The first multi-set contains the lateness value of job 2 when it is moved forward by p_1 due to the movement of job 1. In this example $\bigcup_{u=0}^{0}\{s(2, 2, u, 0) - p_1\} = \{-5\}$. The second multi-set contains the lateness values of jobs 3 and 4 when they are optimally rearranged by moves up to level $l = 1$. Therefore, $s(3, 4, 0, 1) = 30$ (obtained by the identical move $3 \to 3$) and $s(3, 4, 1, 1) = 15$ (obtained by move $3 \to 4$). The third term in (12) is the lateness value of the moved job 1 in its new position which is $C_2(\sigma_0) - d_1 = 35 - 45 = -10$. Thus, (12) defines the multi-set $\mathcal{S}(1, 4, 2, 1) = \{30, 15, -5, -10\}$ which is also presented in Table 3 together with all other multi-sets $\mathcal{S}(\cdot)$.

After calculating $\mathcal{S}(i, j, k, 1)$ for all $k \in \sigma_0(i, j)$, the state values $s(i, j, m, 1)$ can be obtained from (13). For this purpose, the values in the multi-set $\mathcal{S}(i, j, k, 1)$ are sorted in non-increasing order, as shown in the right part of Table 3. The value $s(i, j, m, 1)$ is then obtained by talking the minimum value of all $(m + 1)$-largest values from the multi-sets $\mathcal{S}(i, j, k, 1)$. Using the table, this can be performed by calculating the minimum value of the corresponding $j - i + 1$ rows in the column with the corresponding m-value. For example, $s(1, 4, 2, 1)$ corresponds to the minimum of the multi-set $\{10, -5, -5, -5\}$ which is -5.

As we can see in the last row of Table 3, the optimal solution for stack capacity $S = 1$ has two late jobs since $m = 2$ is the minimum value which fulfills $s(1, 4, m, 1) \le 0$, i.e., a schedule with two late jobs can be obtained at level $\ell = 1$ starting at time 0. In this example, multiple optimal schedules exist. One of them, for instance, is the schedule $\sigma = \langle 2, 1, 3, 4\rangle$ in which jobs 3 and 4 are late. It can be obtained by the single move $1 \to 2$.

Table 3 State values $s(i, j, m, 1)$ for all $i, j \in \{1, 2, 3, 4\}$ with $i \leq j$ and $m \in \{0, \dots, j - i\}$

		move $i \to k$	$s(i+1, k, u, 0) - p_i$	$s(k+1, j, u, 1)$	$C_k(\sigma_0) - d_i$	m			
			$u = 0, \dots, k-i-1$	$u = 0, \dots, j-k-1$		0	1	2	3
$j = 1$	$i = 1$	$k = 1$			-20	-20			
		$\mathbf{s(1, 1, m, 1)}$				**−20**			
$j = 2$	$i = 2$	$k = 2$			20	20			
		$\mathbf{s(2, 2, m, 1)}$				**20**			
	$i = 1$	$k = 1$		20	-20	20	-20		
		$k = 2$	-5		-10	-5	-10		
		$\mathbf{s(1, 2, m, 1)}$				**−5**	**−20**		
$j = 3$	$i = 3$	$k = 3$			30	30			
		$\mathbf{s(3, 3, m, 1)}$				**30**			
	$i = 2$	$k = 2$		30	20	30	20		
		$k = 3$	20		25	25	20		
		$\mathbf{s(2, 3, m, 1)}$				**25**	**20**		
	$i = 1$	$k = 1$		25, 20	-20	25	20	-20	
		$k = 2$	-5	30	-10	30	-5	-10	
		$k = 3$	5, -5		-5	5	-5	-5	
		$\mathbf{s(1, 3, m, 1)}$				**5**	**−5**	**−20**	
$j = 4$	$i = 4$	$k = 4$			20	20			
		$\mathbf{s(4, 4, m, 1)}$				**20**			
	$i = 3$	$k = 3$		20	30	30	20		
		$k = 4$	15		40	40	15		
		$\mathbf{s(3, 4, m, 1)}$				**30**	**15**		
	$i = 2$	$k = 2$		30, 15	20	30	20	15	
		$k = 3$	20	20	25	25	20	20	
		$k = 4$	20, 10		35	35	20	10	
		$\mathbf{s(2, 4, m, 1)}$				**25**	**20**	**10**	
	$i = 1$	$k = 1$		25, 20, 10	-20	25	20	10	-20
		$k = 2$	-5	30, 15	-10	30	15	-5	-10
		$k = 3$	5, -5	20	-5	20	5	-5	-5
		$k = 4$	5, -5, -5		5	5	5	-5	-5
		$\mathbf{s(1, 4, m, 1)}$				**5**	**5**	**−5**	**−20**

5.1.1 Algorithmic realization of the dynamic program

In this section, we illustrate a straightforward implementation of the algorithm sketched in Section 5.1. The corresponding pseudocode is presented in Algorithm 1.

The recursion requires the generation of the states from level $\ell = 0$ up to S to ensure that the effects of moves nested in any move of level ℓ have already been examined. Furthermore, for each level, the subsequences are examined in a given order: In an outer loop (Steps 8–17), the index j spans the range 1 to n while in an inner loop the index i runs from j down to 1 (Steps 9–17). This guarantees that when move (i, k, ℓ) is considered, the effects of all moves (i', k', ℓ) with $k < i' \leq k'$, have already been computed. For a given triple (i, j, ℓ), all multi-sets $\mathcal{S}(i, j, k, \ell)$, $k = i, \dots, j$, are temporarily stored in the lists $\mathcal{S}_k$. Note that Steps 12–14 implement the union of the three multi-sets of Equation (12) and the procedure *Append*, used in the algorithm, simply adds an element to the end of a linked list.

The running time complexity of Algorithm 1 can be analyzed as follows. The main computation, executed in Loop 1 (Steps 10–14) and Loop 2 (Steps 15–17) of procedure RECURSION, is performed $O(Sn^2)$ times, namely for all levels ℓ and over all ordered pairs (i, j). Both, Loop 1 and Loop 2, have total cost $O(n^2)$ since each iteration in those loops requires $O(n)$ time. Thus, the overall worst-case time complexity is $O(n^4 S)$ which can be bounded by $O(n^5)$. The above discussion can be summed up in the following theorem.

Theorem 15 *Algorithm 1 determines an optimal solution for problem* $1|$*resch-LIFO*$|\sum U_j$ *in* $O(n^5)$ *time.*

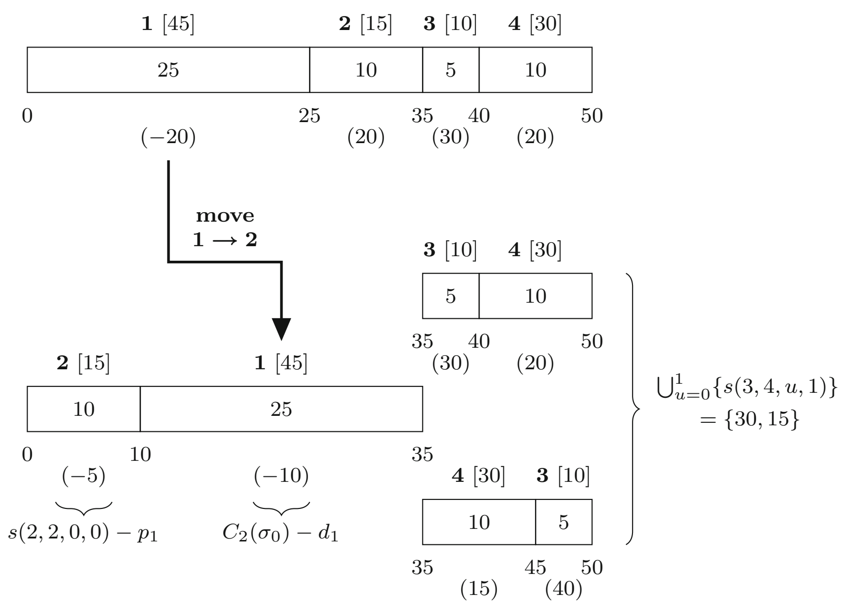


Fig. 2 Illustration of computing the multi-set $\mathcal{S}(1,4,2,1)$ in the given example where processing times are indicated within the rectangles representing the jobs, due dates are in square brackets and lateness values are in parentheses

Algorithm 1 Dynamic Program for minimizing the number of late jobs

```
1: procedure INITIALIZATION (for $\ell = 0$)
2:     for $i = 1, \dots, n$ do
3:         for $j = i, \dots, n$ do
4:             for $m = 0, \dots, j - i$ do
5:                 $s(i, j, m, 0) \leftarrow (m+1)$-largest value of a list with values $L_i(\sigma_0), \dots, L_j(\sigma_0)$
6: procedure RECURSION (for $\ell \geq 1$)
7:     for $\ell = 1, \dots, S$ do
8:         for $j = 1, \dots, n$ do
9:             for $i = j, \dots, 1$ do
10:                for $k = i, \dots, j$ do                                   ▷ Loop 1
11:                    Let $\mathcal{S}_k$ be an empty list
12:                    for $u = 0, \dots, k-i-1$ do Append $s(i+1, k, u, \ell-1) - p_i$ to the list $\mathcal{S}_k$
13:                    for $u = 0, \dots, j-k-1$ do Append $s(k+1, j, u, \ell)$ to the list $\mathcal{S}_k$
14:                    Append $C_k(\sigma_0) - d_i$ to the list $\mathcal{S}_k$
15:                for $m = 0, \dots, j - i$ do                               ▷ Loop 2
16:                    Let $\mathcal{S}$ be a list with the $(m+1)$-largest values of $\mathcal{S}_i, \dots, \mathcal{S}_j$
17:                    $s(i, j, m, \ell) \leftarrow \min(\mathcal{S})$
18: $f^{(3)}(\sigma^*) \leftarrow \min\{m : s(1, n, m, S) \leq 0\}$
```

5.2 Extension to the Ω-constrained problem

To deal with the problem where not all jobs are movable, we need to reconsider Expressions (11) and (13) since the multisets $\mathcal{S}(i, j, k, \ell)$ have no meaning anymore for $i \in J \setminus \Omega$. In particular, Recursion (13) can be rewritten as follows:

$$s(i,j,m,\ell) = \begin{cases} \max_{[m+1]}\left(\bigcup_{u=0}^{j-i-1}\{s(i+1,j,u,\ell)\} \cup \{C_i(\sigma_0) - d_i\}\right), & \text{if } i \in J \setminus \Omega, \\ \text{as in Expression (13)}, & \text{if } i \in \Omega, \end{cases}$$

for all $i, j \in J$, $i \leq j$, $m = 0, \dots, j - i$, and $\ell = 1, \dots, S$. Since job i cannot be moved, its lateness value is $C_i(\sigma_0) - d_i$, whereas the necessary reductions of the starting times for the subsequence $\sigma_0(i+1, j)$ are the already computed values of the states $s(i+1, j, u, \ell)$, $u = 0, \dots, j - i - 1$.

6 Minimization of the weighted number of late jobs

Here, we deal with a generalization of the problem introduced in the previous Sect. 5, and consider different job priorities. We therefore introduce a cost, associated to each job, which is paid if that job is late. This type of objective is studied in a number of papers (see, e.g., Baptiste 1999; Blazewicz et al. 2005). We first prove that the resulting problem, namely $1|\text{resch-LIFO}|\sum w_j U_j$, is NP-hard and then present a pseudo-polynomial exact solution algorithm based on dynamic programming.

Theorem 16 *Problem $1|\text{resch-LIFO}|\sum w_j U_j$ is binary NP-hard.*

Proof We consider the following decision problem $\mathcal{D}$: Given an instance $\bar{I}$ of $1|\text{resch-LIFO}|\sum w_j U_j$ and an integer Q, is there a feasible solution σ of $\bar{I}$ such that $\sum_j w_j U_j(\sigma) \leq Q$? We prove that $\mathcal{D}$ is NP-complete, thus proving the hardness of $1|\text{resch-LIFO}|\sum w_j U_j$.

The reduction is from EQUAL- CARDINALITY PARTITION: Given a set of $2n$ positive integers $\{a_1, a_2, \dots, a_{2n}\}$, is there a subset A' of the index set $A = \{1, \dots, 2n\}$ with $\sum_{i \in A'} a_i = \sum_{i \in A \setminus A'} a_i$ and $|A'| = n$? Given an instance I of EQUAL-CARDINALITY PARTITION, we can build an instance $\tilde{I}$ of $\mathcal{D}$ as follows. There are $2n$ jobs, and for each job $i = 1, \dots, 2n$

we set $w_i = p_i = a_i$ and $d_i = \frac{1}{2}\sum_{k=1}^{2n} a_k$. Moreover, the stack capacity is $S = n$ and $Q = \frac{1}{2}\sum_{i=1}^{2n} a_i$.

It is easy to observe that $\tilde{I}$ is a YES-instance of $\mathcal{D}$, i.e., there is a schedule σ such that $\sum_j w_j U_j(\sigma) \le Q$, if and only if I is a YES-instance of Equal- Cardinality Partition. Note that if such a schedule σ exists, then it can be obtained by moving at most n jobs (the late ones) to the right end of the schedule. □

6.1 Dynamic programming

In this section, we show that a pseudo-polynomial algorithm exists for $1|\text{resch-LIFO}|\sum w_j U_j$, hence complementing the result given in Theorem 16.

First of all, let us recall that all data (i.e., processing times, weights and due dates) are positive integer values. Unlike the dynamic program for the unweighted case, where for the optimal ℓ-rearrangement of each subsequence the minimum reduction of the starting times for different number of late jobs is recorded, we now store the *minimum weighted number of late jobs* for different reductions of the starting time.

Definition 17 *For each $i, j \in J$ with $i \le j$, $t = 0, \ldots, P(1, i-1)$, and $\ell = 0, \ldots, S$, let $r(i, j, t, \ell)$ be the* minimum weighted number of late jobs *that can be achieved by rearranging the subsequence $\sigma_0(i, j)$ with moves up to level ℓ, after decreasing the starting time of the subsequence by t.*

As an initialization, we have:

$$r(i, j, t, 0) = \sum_{k \in \sigma_0(i,j):\, L_k(\sigma_0) > t} w_k \qquad \forall i, j \in J,\ i \le j,\ \forall t = 0, \ldots, P(1, i-1). \tag{14}$$

Similar to the algorithm for the unweighted case, in the recursion, the value of state $r(i, j, t, \ell)$ is obtained by taking the minimum weighted number of late jobs over all possible moves (i, k, ℓ') for $k = i, \ldots, j$ at some level $\ell' \le \ell$. The effect of such a move is given by the sum of (at most) three terms expressing the weighted number of late jobs of the optimal $(\ell - 1)$-rearrangement of subsequence $\sigma_0(i+1, k)$ with a starting time decreased by p_i, the weighted number of late jobs of the optimal ℓ-rearrangement of subsequence $\sigma_0(k+1, j)$ and the weighted number of late jobs contributed by the moved job i which is given by

$$\bar{r}(i, k, t) = \begin{cases} w_i, & \text{if } t < C_k(\sigma_0) - d_i, \\ 0, & \text{otherwise.} \end{cases}$$

Therefore, the value for the new state can be computed with the following recursion:

$$r(i, j, t, \ell) = \begin{cases} \bar{r}(i, i, t), & \text{if } i = j, \\ \min_{k \in \sigma_0(i,j)} \{\mathcal{R}(i, k, j, t, \ell)\}, & \text{if } i < j, \end{cases} \tag{15}$$

for all $i, j \in J$, $i \le j$, for all $t = 0, \ldots, P(1, i-1)$ and for all $\ell = 1, \ldots, S$ where

$$\mathcal{R}(i, k, j, t, \ell) = \begin{cases} r(i+1, j, t, \ell) + \bar{r}(i, i, t), & \text{if } k = i, \\ r(i+1, k, t + p_i, \ell - 1) + r(k+1, j, t, \ell) + \bar{r}(i, k, t), & \text{if } i < k < j, \\ r(i+1, j, t + p_i, \ell - 1) + \bar{r}(i, j, t), & \text{if } k = j. \end{cases}$$

The optimal solution value, $f^{(4)}(\sigma^*) = \sum_{j \in J} w_j U_j(\sigma^*)$, is found by computing $r(1, n, 0, S)$.

6.1.1 Algorithmic realization of the dynamic program

The pseudocode in Algorithm 2 straightforwardly follows the approach described in the previous section. The recursion requires, analogous to Algorithm 1, the generation of the states from level $\ell = 0$ up to S. Additionally, for each level ℓ, the subsequences $\sigma_0(i, j)$ are also examined in two loops where index j runs from 1 up to n in an outer loop, while in an inner loop index i runs from j down to 1 to ensure that certain states are already computed.

Algorithm 2 Dynamic Program for minimizing the weighted number of late jobs

```
1: procedure Initialization (for $\ell = 0$)
2:   for $i = 1, \ldots, n$ do
3:     for $j = i, \ldots, n$ do
4:       for $t = 0, \ldots, P(1, i-1)$ do
5:         $r(i, j, t, 0) \leftarrow 0$
6:         for $k = i, \ldots, j$ with $L_k(\sigma_0) > t$ do
7:           $r(i, j, t, 0) \leftarrow r(i, j, t, 0) + w_k$
8: procedure Recursion (for $\ell \ge 1$)
9:   for $\ell = 1, \ldots, S$ do
10:    for $j = 1, \ldots, n$ do
11:      for $i = j, \ldots, 1$ do
12:        for $t = 0, \ldots, P(1, i-1)$ do
13:          $r_{\min} \leftarrow \infty$
14:          for $k = i, \ldots, j$ do
15:            if $t < C_k(\sigma_0) - d_i$ then $\bar{r} \leftarrow w_i$
16:            else $\bar{r} \leftarrow 0$
17:            if $i = j$ then $r_k \leftarrow \bar{r}$
18:            else if $k = i$ then $r_k \leftarrow r(i+1, j, t, \ell) + \bar{r}$
19:            else if $k = j$ then $r_k \leftarrow r(i+1, j, t + p_i, \ell - 1) + \bar{r}$
20:            else $r_k \leftarrow r(i+1, k, t + p_i, \ell - 1) + r(k+1, j, t, \ell) + \bar{r}$
21:            $r_{\min} \leftarrow \min\{r_{\min}, r_k\}$
22:          $r(i, j, t, \ell) \leftarrow r_{\min}$
23: $f^{(4)}(\sigma^*) \leftarrow r(1, n, 0, S)$
```

It is easy to see that the proposed algorithm has a pseudo-polynomial running time of $O(n^3 S P)$ where $P = P(1, n) = \sum_{j \in J} p_j$ is the sum of the processing times of all jobs.

Of course, Algorithm 2 can also be used for the unweighted case (with $w_j = 1$ for all $j \in J$). Clearly, we then still have a pseudo-polynomial running time instead of the strongly polynomial running time stated in Theorem 15.

6.2 Extension to the Ω-constrained problem

To solve the Ω-constrained problem, we can still use Equations (14) as an initialization but we need to adapt Recursion (15) as follows:

$$r(i, j, t, \ell) = \begin{cases} r(i+1, j, t, \ell) + \overline{r}(i, i, t), & \text{if } i \in J \setminus \Omega,\ i < j, \\ \text{as in Expression (15)}, & \text{otherwise}, \end{cases}$$

for all $i, j \in J, i \leq j$, for all $t = 0, \ldots, P(1, i-1)$ and for all $\ell = 1, \ldots, S$. If job i is not movable, its total weighted number of late jobs, depending on t, is $\overline{r}(i, i, t)$ and the weighted number of late jobs for the subsequence $\sigma_0(i+1, j)$ is simply the already computed value $r(i+1, j, t, \ell)$.

6.3 Alternative dynamic program

An alternative solution approach for $1|\text{resch-LIFO}|\sum w_j U_j$ can be derived by generalizing Algorithm 1 to the weighted setting. In this case, we define $\widetilde{s}(i, j, m, \ell)$ as a generalization of $s(i, j, m, \ell)$ where m now denotes the weighted number of late jobs. Consequently, variable m can take every weight value of the objective function, and therefore can be upper-bounded by the value $W = W(1, n) = \sum_{j \in J} w_j$.

Definition 18 *For each $i, j \in J$ with $i \leq j$, $m = 0, \ldots, W(i, j)$ and $\ell = 0, \ldots, S$, let $\widetilde{s}(i, j, m, \ell)$ be the minimum decrease of the starting time of the subsequence $\sigma_0(i, j)$ that ensures a value of* at most *m for the weighted number of late jobs when the subsequence can be rearranged with moves up to level ℓ.*

The generalization of Eqs. (11) to (13) is as follows: the initialization for all $i, j \in J$, $i \leq j$, and for all $m = 0, \ldots, W(i, j) - 1$ is given by

$$\widetilde{s}(i, j, m, 0) = argmin_{t \in \mathcal{T}} \left\{ \sum_{\substack{k \in \sigma_0(i,j): \\ L_k(\sigma_0) > t}} w_k \right\}$$

where $\mathcal{T} = \{t \in \{L_i(\sigma_0), \ldots, L_j(\sigma_0)\} : \sum_{k \in \sigma_0(i,j):\, L_k(\sigma_0) > t} w_k \leq m\}$.

The recursive extension rule for all $i, j \in J, i \leq j, m = 0, \ldots, W(i, j) - 1$ and $\ell = 1, \ldots, S$ is

$$\widetilde{s}(i, j, m, \ell) = \min_{k \in \sigma_0(i,j)} \left\{ \max_{[m+1]}(\widetilde{\mathcal{S}}(i, j, k, \ell)) \right\} \tag{16}$$

with the modified multi-set

$$\begin{aligned} &\widetilde{\mathcal{S}}(i, j, k, \ell) \\ &= \bigcup_{u=0}^{W(i+1,k)-1} \{\widetilde{s}(i+1, k, u, \ell - 1) - p_i\} \\ &\cup \bigcup_{u=0}^{W(k+1,j)-1} \{\widetilde{s}(k+1, j, u, \ell)\} \\ &\cup \bigcup_{u=0}^{w_i - 1} \{C_k(\sigma_0) - d_i\}. \end{aligned}$$

Obviously, this generalization of the algorithm sketched in Sect. 5.1 results in a worst-case computational complexity of $O(n^3 S W)$ which can be bounded by $O(n^4 W)$.

Summing up the above argumentation and those of Sect. 6.3, we obtain the following theorem.

Theorem 19 *Problem $1|\text{resch-LIFO}|\sum w_j U_j$ is pseudo-polynomial solvable within $O(n^4 B)$ running time where $B = \min \left\{ \sum_{j \in J} p_j, \sum_{j \in J} w_j \right\}$.*

7 Empirical analysis of the stack size effect

In this section, we illustrate the effect of rescheduling and in particular the influence of the stack size for the three polynomial time algorithms presented in Sects. 3 to 5. We do not report running times. We only observe that the dynamic programming algorithms show quite small deviations in running times for instances of the same size.

As a test bed, we follow the data generation scheme by Potts and Van Wassenhove (1988). We choose processing times and weights independent and identically, uniformly distributed with $p_j \sim U(1, 100)$ and $w_j \sim U(1, 100)$. For two parameters d^ℓ and d^u we choose $d_j \sim U(P(1, n)\, d^\ell,\ P(1, n)\, d^u)$. We consider four choices for d^ℓ, namely $d^\ell \in \{0.2, 0.4, 0.6, 0.8\}$, and all possibilities for $d^u \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ with $d^\ell \leq d^u$, which yields 14 pairs of parameter values. For the number of jobs, we take $n = 50$ and $n = 100$. For each of the 28 resulting classes of instances, 20 instances were randomly generated, i.e., 560 instances in total.

The goal of our analysis is twofold: We want to analyze the potential improvement of the objective function obtained through rescheduling and we want to illustrate the utilization of the stack. Naturally, both aspects strongly depend on the

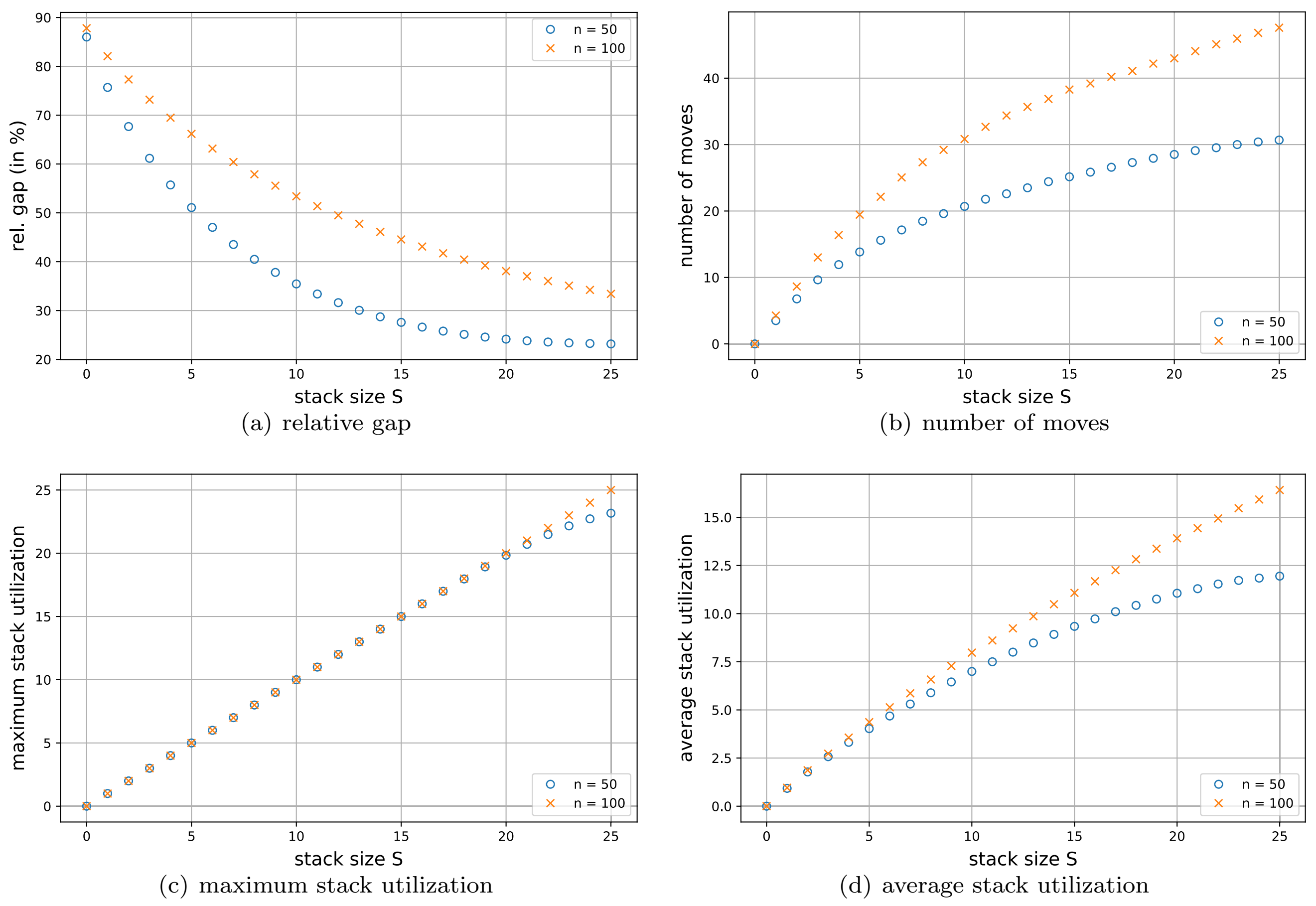


Fig. 3 Effect of increasing stack size S for $1|\text{resch-LIFO}|\sum w_j C_j$

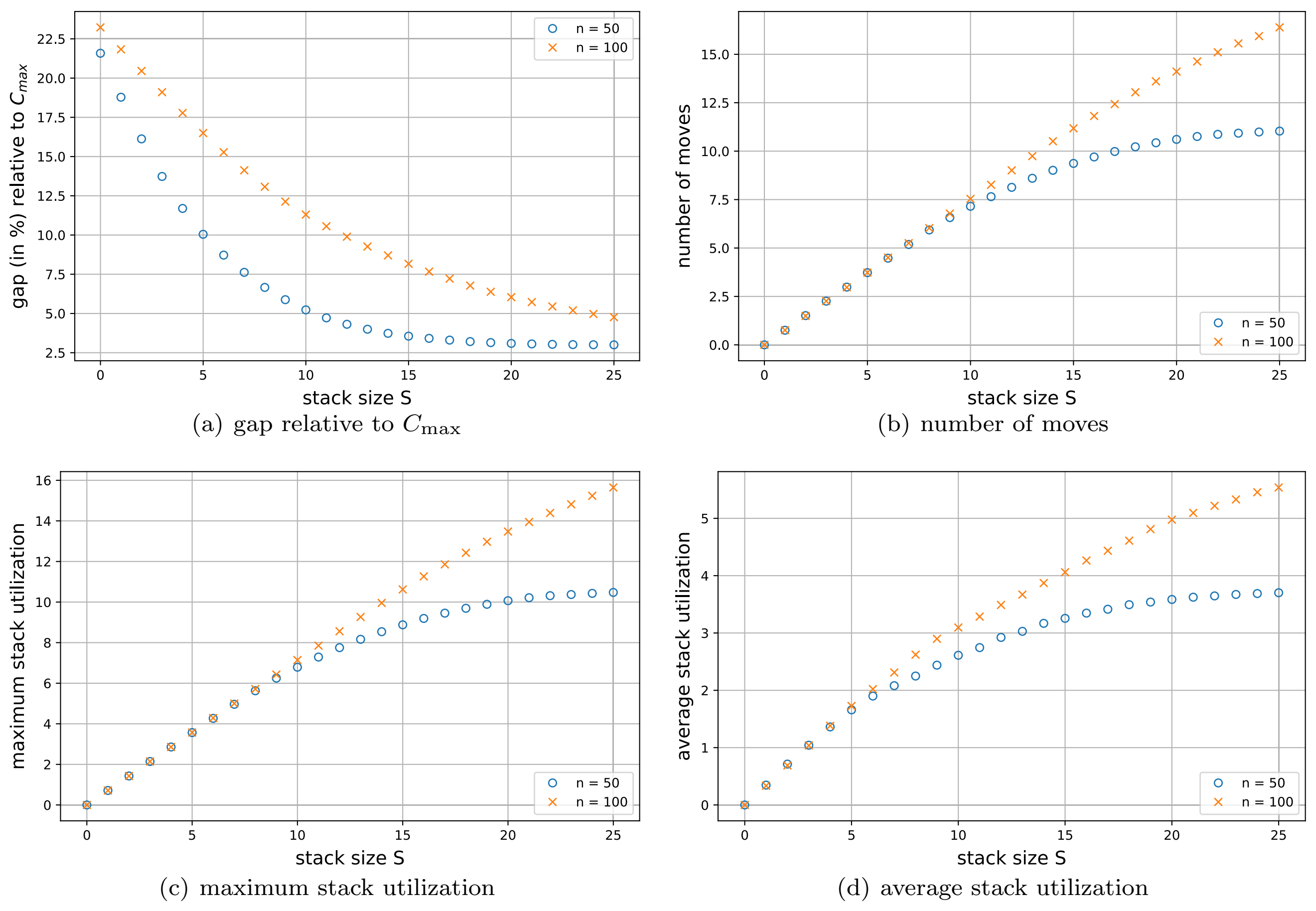
stack size. Therefore, all our evaluations are performed for increasing stack sizes, starting from $S = 1$ up to $S = 25$. Note that for practical applications in a production setting, the LIFO stack can be expected to be of moderate size with $S \le 10$, but for the empirical analysis we consider also larger values of S. It is also clear that S does not depend on the number of jobs.

The results of our tests are given in Figs. 3, 4 and 5. All values are taken as averages over the 280 instances for $n = 50$ and $n = 100$, respectively.

The two graphs in the upper row of each figure describe the effect of rescheduling on the solution. In Figs. 3a, 4a and 5a, we give the gap between the solution obtained from rescheduling under LIFO constraints with a certain stack size (given on the x-axis) and the optimal schedule (obtained by completely reordering the given jobs). For $1|\text{resch-LIFO}|\sum w_j C_j$, the gap in Fig. 3a is the difference relative to the total weighted completion time of an optimal schedule expressed in percent. For $1|\text{resch-LIFO}|L_{\max}$, it is not obvious how to scale the difference to the optimal schedule since the latter could have a positive, zero or negative maximum lateness. Thus, we take the difference between lateness after rescheduling and lateness of an optimal schedule and divide it by $C_{\max} = P(1, n)$, see Fig. 4a. Finally, for $1|\text{resch-LIFO}|\sum U_j$ the difference of the number of late jobs is given as an absolute value in Fig. 5a (note that the optimal value might be zero). It can be seen that there is a certain limit on the effect reachable by rescheduling, e.g., for total weighted completion time the gap hardly gets below 20%. This is possibly due to the LIFO constraint.

In Figs. 3b, 4b and 5b, we give the number of moves, i.e., the total number of jobs inserted into the stack during the rescheduling process. As can be expected, a larger stack size permits much more improvement of the solution and reschedules a larger proportion of the jobs. But from a certain stack size, the objective improves only marginally and thus also the number of moves ceases to grow.

The two graphs in the lower row of each figure describe the stack utilization. In Figs. 3c, 4c and 5c, the maximum stack utilization is given, i.e., the largest number of jobs contained in the stack at any time during the rescheduling process. It turns out that for the total weighted completion time the stack size is almost always fully exploited at some point of the execution. This is not the case when minimizing maximum



(a) gap relative to $C_{\max}$
(b) number of moves
(c) maximum stack utilization
(d) average stack utilization

Fig. 4 Effect of increasing stack size S for $1|\text{resch-LIFO}|L_{\max}$

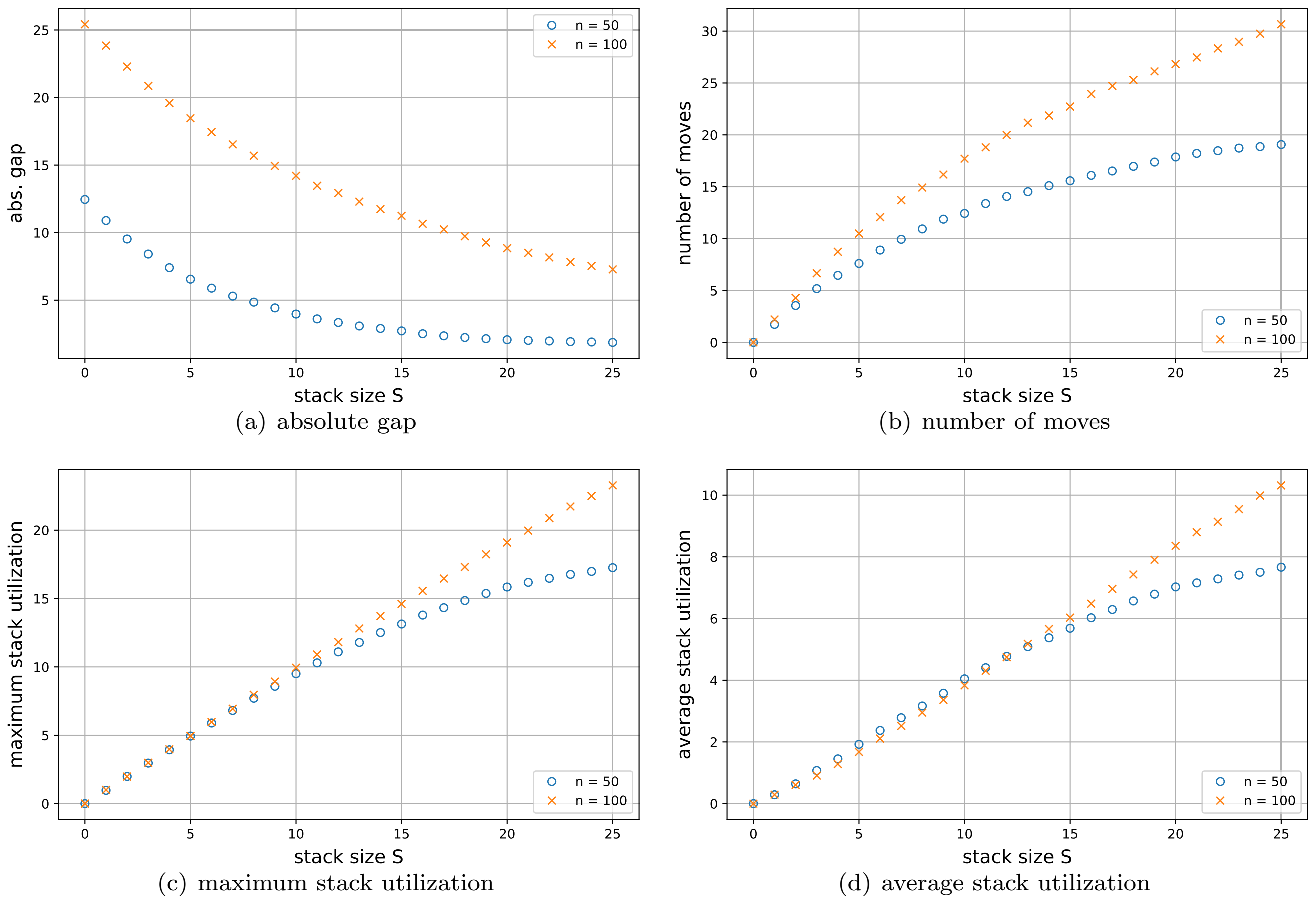
lateness, especially for the smaller instances with $n = 50$, where a close to optimal solution is reached with a moderate stack size, but further improvements are reachable only for a few instances as the stack size increases. If the number of late jobs is minimized, the stack size can be mostly fully exploited for the larger instances with $n = 100$, while for $n = 50$ an almost optimal solution is reachable even with $S \approx 15$ (as can be seen from Fig. 5a), and thus, the stack is not fully utilized (see Fig. 5c).

Finally, Figs. 3d, 4d and 5d show the average stack utilization. This means that we track the number of jobs contained in the stack in each of the $n-1$ steps of the rescheduling process and take their average. Clearly, this includes a certain phasing-in and phasing-out effect since the empty stack only starts accepting jobs at the beginning and has to be empty again until the end of the sequence. The concave shape of the corresponding figures is due to this effect which is naturally much stronger for $n = 50$ than for $n = 100$. It turns out that for maximum lateness the average stack utilization is considerably smaller than for total weighted completion time (compare Figs. 3d and 4d) with number of late jobs as a close follower. We believe that this is due to the fact that for total weighted completion time the optimal solution is unique (for distinct input values) while for the other two objectives there usually exist various different sequences with the same solution value. Once an optimal or very good solution value is reached, further rearrangements of the sequence are not beneficial any more.

8 Conclusions

In this paper, motivated by questions arising in manufacturing applications, we study the problem of rearranging a given sequence of jobs on a single machine in order to minimize one of four different objectives. We rely on the standard scheduling parameters of a job, namely processing time, weight (importance) and due date. From a practical point of view, also operating costs arising from the rearrangement steps, in particular energy consumption, could be taken explicitly into account.

The new sequence can be obtained respecting certain technological constraints: In particular, our jobs are associated to physical parts, sequenced on a conveyor that feeds a pro-



(a) absolute gap (b) number of moves (c) maximum stack utilization (d) average stack utilization

Fig. 5 Effect of increasing stack size S for $1|\text{resch-LIFO}|\sum U_j$

cessing resource, which can be picked up by a robot. A job taken from the conveyor by the robot is first put into a buffer and then placed again on the conveyor in a later position of the original sequence. The buffer is managed as a stack with limited capacity so the last job entering the buffer is the first taken by the robot to be put down again on the conveyor. Due to this LIFO mechanism, only a certain set of sequences can be reached starting from the initial one and this set constitutes all the feasible solutions of our problems.

Our contribution is focused on settling the computational complexity and providing exact solution algorithms. In particular, we are able to provide strongly polynomial (dynamic programming) solution algorithms for the minimization of (i) total weighted completion time of the jobs, (ii) maximum of regular functions of the completion times, in particular maximum lateness, of the jobs, and (iii) number of late jobs. We also prove that (iv) if we want to minimize the weighted number of late jobs, then the problem becomes (weakly) NP-hard. For the latter problem, we present a pseudo-polynomial solution algorithm (again based on a dynamic program).

Future research should consider an empirical study concerning both the characterization of optimal solutions, that may be obtained starting from different input sequences, and the performance of exact solution algorithms in terms of numerical efficiency, with possible comparisons with alternative enumeration schemes.

In more general terms, it would clearly be interesting to consider other regimes for buffer management. From a practical point of view, the most relevant setting would be a queue environment implied by a FIFO mechanism. Of course, also a random access buffer where any job can be taken from the buffer, may be well justified, although technically more complicated. These two aspects will be subject of future research. Moreover, it would also be interesting to consider alternative performance figures. We dealt with the four most common, standard objectives functions, but in the future also other choices might be relevant, in particular in the context of a real-world application (Li et al. 2021).

Funding Open access funding provided by Università degli Studi Roma Tre within the CRUI-CARE Agreement. This study was partially funded by MIUR PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data), by the University of Graz under the Field of Excellence "COLIBRI" and by the Regione Lombardia, grant agreement n. E97F17000000009, Project AD-COM.

Data availability Not applicable.

Declarations

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

References

Agnetis, A., Detti, P., Meloni, C., & Pacciarelli, D. (2001). Coordination between two stages of a supply chain. *Annals of Operations Research, 107*(1), 15–32.

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

Agnetis, A., Chen, B., Nicosia, G., & Pacifici, A. (2019). Price of fairness in two-agent single-machine scheduling problems. *European Journal of Operational Research, 276*(1), 79–87.

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.

Alfieri, A., Nicosia, G., Pacifici, A., & Pferschy, U. (2018a). Constrained job rearrangements on a single machine. AIRO Springer seriesIn P. Daniele & L. Scrimali (Eds.), *New trends in emerging complex real life problems* (Vol. 1, pp. 33–41). Berlin: Springer.

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

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.

Baptiste, P. (1999). Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling, 2*(6), 245–252.

Blazewicz, J., Pesch, E., Sterna, M., & Werner, F. (2005). The two-machine flow-shop problem with weighted late work criterion and common due date. *European Journal of Operational Research, 165*(2), 408–415.

Daniels, R. L., & Kouvelis, P. (1995). Robust scheduling to hedge against processing time uncertainty in single-stage production. *Management Science, 42*(2), 363–737.

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.

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

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

Ivanov, D., & Sokolov, B. (2015). Coordination of the supply chain schedules with re-scheduling considerations. *IFAC-PapersOnLine, 48*(3), 1509–1514.

Leung, J. Y. T., Pinedo, M., & Wan, G. (2010). Competitive two-agent scheduling and its applications. *Operations Research, 58*(2), 458–469.

Li, X., Ventura, J. A., & Bunn, K. A. (2021). A joint order acceptance and scheduling problem with earliness and tardiness penalties considering overtime. *Journal of Scheduling, 24*, 49–68.

Liebchen C, Lübbecke M, Möhring R, 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, pp. 1–27). Springer.

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).

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.

Perez-Gonzalez, P., & Framinan, J. M. (2014). A common framework and taxonomy for multicriteria scheduling problems with interfering and competing jobs: Multi-agent scheduling problems. *European Journal of Operational Research, 235*(1), 1–16.

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.

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.

Wu, S. D., Storer, R. H., & Chang, P. C. (1993). One machine rescheduling heuristics with efficiency and stability as criteria. *Computers & Operations Research, 20*(1), 1–14.