



Research article

Rescheduling involving due dates on a single machine due to the arrival of new jobs with rejection

Xuecong Xu¹, Shuaijing Yu² and Wenchang Luo^{1,*}

¹ School of Mathematics and Statistics, Ningbo University, Ningbo, Zhejiang 315211, China

² Yiwu Industrial and Commercial College, Yiwu, Zhejiang 322000, China

* **Correspondence:** Email: luowenchang@nbu.edu.cn.

Abstract: We investigated a novel rescheduling problem in which the original jobs have already been scheduled on a single machine with the aim of minimizing either maximum lateness or maximum tardiness. Before the execution of this planned schedule, a set of new jobs arrives, making it necessary to reconsider the schedule. The decision-maker needs to determine a revised sequence for the original and new jobs while avoiding excessive disruption to the planned schedule. For original jobs, the extent of disruption is defined as the deviation of completion time between its planned and adjusted schedule. To preserve an adequate service level for the original jobs, one possible approach is to reject some new jobs by paying rejection costs. We proposed general models in which the maximum completion time deviation appears not only as a constraint but also as a part of the optimization objective. The overall optimization objective is to minimize the sum of three components: the scheduling objective (maximum lateness or maximum tardiness) of the adjusted schedule, which contains original jobs and accepted new jobs, the maximum disruption penalty, and the total rejection cost. Two pseudo-polynomial time dynamic programming exact algorithms were developed to provide optimal solutions. In addition, genetic algorithms were designed to handle the problem more efficiently. The performance of both methods was evaluated through numerical experiments.

Keywords: rescheduling; rejection; lateness; tardiness; exact algorithm; genetic algorithm

Mathematics Subject Classification: 90B35, 90C39

1. Introduction

Recently, both rescheduling in accommodation to some unexpected events that occur in dynamic production, manufacture, and service environments and scheduling with job rejection due to the processing capacity restriction have received considerable attention, but separately, from research and industrial communities. The first research stream concerns rescheduling in response to unexpected events.

Different rescheduling problems have been shown in many practical applications such as automotive manufacturing [1], space shuttle missions [2], ship building [3], short-haul route planning [4], and liberalized power markets [5]. The popular unexpected events creating the necessity for rescheduling come from the arrival of new orders [6], changes in order availability [7], machine failures [8], etc. The literature on rescheduling is extensive. For representative surveys, we refer the reader to Vieira et al. [9], Ouelhadj and Petrovic [10], and Uhlmann and Frazzon [11].

The second research stream concerns scheduling with job rejection. Job-rejection scheduling primarily occurs in make-to-order production systems, where production capacity is limited and delivery requirements are stringent. In such production systems, manufacturers may choose to reject certain job orders. Consequently, before finalizing the scheduling plan, a high-level decision must be made to classify orders into accepted and rejected categories. The scheduling model with job rejection was first proposed by Bartal et al. [12], who addressed the problem on identical parallel machines with the objective of minimizing the sum of the makespan of accepted jobs and the rejection costs of rejected jobs. In the single-machine setting, Sengupta [13] investigated scheduling with job rejection, where the objective is to minimize the maximum lateness or maximum tardiness of the accepted jobs together with the total rejection cost. He established NP-hardness results and developed a pseudo-polynomial-time exact algorithm and a fully polynomial time approximation scheme (FPTAS) for the maximum tardiness problem, as well as a polynomial-time ϵ -approximation scheme for the maximum lateness problem. Other related studies on scheduling with job rejection include the work of Engels et al. [14], Ou and Zhong [15], Shabtay [16], Nobibon and Leus [17], and Thevenin et al. [18], which focused on optimizing the total (or weighted) completion time in various scheduling environments. For further review studies in the field of scheduling with job rejection, readers are referred to the works of Slotnick [19] and Shabtay et al. [20].

At the intersection of these two research streams, several studies have integrated disruption-driven rescheduling with job-rejection decisions. These studies are reviewed chronologically below. Wang et al. [21] were among the first to incorporate job rejection into a multi-objective dynamic rescheduling framework involving the continuous arrival of new jobs. In their model, the processing times of accepted jobs could be compressed through additional resource allocation. The scheduling cost combined the total completion time, rejection cost, and resource consumption, while the disruption cost was measured by total virtual tardiness, with the completion times in the original schedule treated as virtual due dates. They developed a genetic algorithm with a directed search strategy to approximate the Pareto front. Subsequently, Wang et al. [22] studied rescheduling with job rejection on identical parallel machines, where the disruption was caused by the unavailability of a subset of jobs. Their model minimized the total completion time of the accepted jobs subject to upper bounds on the maximum completion-time deviation and the total rejection cost. For this problem, they proposed a dynamic programming algorithm for a fixed number of machines and an improved branch-and-price algorithm.

Luo et al. [23] later investigated a rescheduling problem with job rejection in which some jobs failed to arrive at their scheduled release times. Their objective was to minimize the sum of the total weighted completion time, the maximum tardiness in the revised schedule, and the total rejection cost. They developed a pseudo-polynomial-time dynamic programming algorithm and an FPTAS for the case without an upper bound on the total rejection cost. More recently, Fang et al. [24] and Ren et al. [25] studied single-machine rescheduling with job rejection caused by the arrival of new jobs. Fang et al. minimized the sum of the total weighted completion time, a penalty proportional to the maximum

completion-time deviation, and the total rejection cost, subject to an upper bound on the maximum deviation. They proved that the problem is NP-hard even in the unit-weight case. Ren et al. considered a similar model in which the total weighted completion time was replaced by the maximum delivery completion time. Both studies developed pseudo-polynomial-time dynamic programming algorithms and FPTASs.

Overall, the above studies that jointly consider rescheduling and job rejection differ mainly in terms of the disruption source and scheduling criterion. Wang et al. [21], Fang et al. [24], and Ren et al. [25] considered disruptions caused by new job arrivals, whereas Wang et al. [22] and Luo et al. [23] examined different forms of job unavailability. The scheduling criteria considered in these studies include the total or total weighted completion time, maximum tardiness, and maximum delivery completion time.

The above comparison, together with the work of Sengupta [13], helps clarify the research gap. Sengupta considered two single-machine job-rejection problems under the maximum-lateness and maximum-tardiness criteria, respectively, but his models did not involve rescheduling. Therefore, to the best of our knowledge, no study has jointly considered new job arrivals, job rejection, control of the maximum completion-time deviation, and either the maximum lateness or maximum tardiness criterion in the single-machine setting.

To address this gap, this article investigates single-machine rescheduling problems in which a set of original jobs has already been arranged to minimize the maximum lateness or the maximum tardiness. However, the arrival of new jobs before the planned schedule is executed necessitates rescheduling. The scheduler is required to construct a revised schedule in response to the arrival of the new jobs without excessively disrupting the original plan, where the disruption is measured as the maximum deviation in completion time between the original and adjusted schedules for any given original job. To ensure service quality for the original jobs, in rescheduling, one has an option to reject some new jobs by paying their corresponding rejection costs. We consider two models in which disruption is treated both as a constraint and as a component of the objective function. In both models, the overall objective is to minimize the sum of the scheduling objective for the original and accepted new jobs, the disruption penalty, and the total rejection cost. The scheduling objective is maximum lateness in the first model and maximum tardiness in the second.

In deadline-sensitive production and service environments, due dates often represent delivery or service commitments, and an excessive delay of even one job may have serious consequences. The maximum-lateness criterion measures the worst due-date performance among the original and accepted new jobs. When some deterioration of the original schedule is unavoidable after new jobs are accepted, including $L_{\max}$ in the objective serves to control the worst lateness in the revised schedule. Since accepting and integrating new jobs may affect existing commitments, whereas rejecting them may incur economic or service costs, the rescheduling decision needs to balance worst-case due-date performance, disruption to the original schedule, and rejection cost.

As a motivating example, we examine a scenario of medical procedure rescheduling. In healthcare delivery, efficient scheduling of medical procedures (e.g., surgeries, scans, treatments) on shared resources (operating rooms, MRI machines, radiotherapy devices) is critical for patient outcomes, resource utilization, and service quality. Often, an initial schedule is carefully designed to ensure timely treatment of patients while respecting medical urgency (due dates). However, in dynamic clinical environments, new urgent cases frequently arrive, making it necessary to determine whether and how they should be scheduled together with the originally planned cases. This creates a natural **rescheduling problem**.

The arrival of emergency or high-priority procedures disrupts the pre-arranged elective schedule. Simply accommodating all new arrivals may excessively delay planned procedures, compromising patient safety, satisfaction, and contractual obligations. Therefore, hospital managers face a trade-off: accept new cases to serve urgent needs, but potentially delay others; or reject some new requests (e.g., redirect to other facilities, postpone non-urgent cases) by incurring logistical or financial **rejection costs**. In practice, the rejection cost does not necessarily represent an immediate contractual penalty. Rather, it can be interpreted as a generalized cost associated with declining a newly arrived case, such as patient transfer expenses, outsourcing fees, administrative costs, revenue loss, or the deterioration of future service relationships. Even when a rejected case is postponed to a later planning cycle or redirected to another provider, such a decision may still incur operational or economic consequences. Therefore, following the standard practice in scheduling with job rejection, these consequences are modeled through a rejection-cost parameter. It should be noted that the originally scheduled procedures have already been accepted and confirmed by the healthcare provider. Although accepting and scheduling new cases may delay the originally planned procedures, canceling those procedures in response to the new arrivals would be far less acceptable to the affected patients, who might need to seek alternative providers, rearrange their treatment plans, and face additional time and administrative burdens. Therefore, in the proposed model, rejection decisions are considered only for newly arriving cases, while all originally scheduled procedures are retained in the revised schedule.

The deviation between originally planned completion times and the revised ones for existing patients can be viewed as a measure of **service disruption**. Excessive delays can increase clinical risks and patient dissatisfaction. Hence, it is necessary to bound the maximum allowable delay (disruption threshold Δ) for any original procedure. Applying such optimized rescheduling frameworks can enhance hospital responsiveness, balance resource use, and maintain service levels, ultimately supporting better healthcare logistics and patient care delivery.

The objective is to minimize a composite cost that includes:

- The **scheduling objective**, such as minimizing maximum lateness or maximum tardiness across all accepted patients.
- The **disruption penalty** proportional to the maximum delay experienced by originally scheduled patients.
- The total **rejection penalty** for declined new cases.

To establish the connection between the medical scenario and the scheduling model, each patient requiring a medical procedure is represented as a job, and the processing time of the job corresponds to the duration of the associated procedure. The shared medical resource (e.g., an operating room, MRI machine, or radiotherapy device) is modeled as a single machine. Patients already included in the original treatment plan correspond to the original job set, whereas newly arriving emergency or urgent patients correspond to the new job set. Throughout the remainder of the paper, the scheduling terminology of jobs and machines is adopted for consistency.

This medical rescheduling scenario directly aligns with the models $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum e_j$ and $1|rej, \delta_{\max} \leq \Delta|T_{\max} + \mu\delta_{\max} + \sum e_j$ presented in the paper.

Beyond healthcare systems, similar rescheduling situations also arise in manufacturing and service operations. In make-to-order production, for instance, urgent orders may arrive after an initial production plan has been generated. The manufacturer then needs to decide which new orders to accept and how

to schedule them together with the previously accepted orders, balancing potential delays to existing orders against the penalties incurred for rejecting new ones. Likewise, in service appointment systems, incoming high-priority requests may require adjustment of previously arranged appointments. Accepting such requests may improve responsiveness to urgent demand, but excessive changes may reduce the service level of existing customers. Therefore, the proposed models are not limited to medical procedure rescheduling, but are applicable to a general class of rescheduling problems involving new job arrivals, disruption control, and rejection decisions.

The remainder of the article is organized as follows. We present a formal formulation of the problems under consideration in Section 2. In Section 3, we analyze the structural properties of the problems and develop exact algorithms based on dynamic programming. In Section 4, we introduce a genetic algorithm to solve these problems. In Section 5, we conduct numerical experiments to evaluate the performance of our dynamic programming algorithms and the improved genetic algorithms against the standard genetic algorithm. In Section 6, we present the conclusions and discuss potential directions for future research.

2. Problem formulations and notations

In this section, we formally state our problems as follows.

Let $\mathcal{J}_O = \{1, 2, \dots, n_O\}$ denote the set of original non-preemptive jobs to be processed on a single machine. An optimal schedule $\pi^* = (1, 2, \dots, n_O)$, ordered by the earliest due date (EDD) rule, where job i precedes job j if $d_i \leq d_j$, is already planned to minimize the maximum lateness or tardiness. Before actual processing begins, a set of new jobs arrives, denoted by $\mathcal{J}_N = \{n_O + 1, n_O + 2, \dots, n_O + n_N\}$. We need to construct a revised schedule in response to the arrival of the new jobs while ensuring that the service level for the original jobs remains acceptable. This may allow some of the new jobs to be rejected. The goal of the problem is to minimize the total cost, which comprises the scheduling objective, disruption penalty, and total rejection costs.

For ease of description, let $\mathcal{J} = \mathcal{J}_O \cup \mathcal{J}_N$ and $n = n_O + n_N$. The processing time and due date of job j are denoted by p_j and d_j , respectively, for $j = 1, 2, \dots, n$. In addition, the rejection cost of each new job j is represented by e_j for $j = n_O + 1, n_O + 2, \dots, n_O + n_N$. Following Hall and Potts [6], we assume that all these input parameters are nonnegative integers and known in advance.

According to the planned schedule π^* , the completion time $C_j(\pi^*)$ of each original job has been determined for $j = 1, 2, \dots, n_O$. In the rescheduling, some new jobs may be accepted for processing, which may change the completion times of the original jobs and cause disruption. Let σ represent the rescheduled sequence containing all original jobs and the accepted new jobs. The relative order of the original jobs is not required to be preserved in σ . For each original job j , define $C_j(\sigma)$ as its completion time in the new schedule σ and $\delta_j = |C_j(\sigma) - C_j(\pi^*)|$. To keep an acceptable service level, we require $\delta_{\max} = \max_{j=1,2,\dots,n_O} \delta_j \leq \Delta$, where Δ is a given threshold that indicates the service level.

Let the sets of accepted jobs and rejected jobs from the new jobs $\mathcal{J}_N$ be $\mathcal{J}_N^A$ and $\mathcal{J}_N^R$, respectively. For the rejected new jobs in set $\mathcal{J}_N^R$, we need to pay the corresponding rejection costs. For the accepted new jobs in set $\mathcal{J}_N^A$, an interruption penalty must be paid due to the disruptions incurred. The scheduling objective is to minimize the maximum lateness $L_{\max}$ or the maximum tardiness $T_{\max}$, which are respectively defined as $L_{\max} = \max_{j \in \mathcal{J}_O \cup \mathcal{J}_N^A} \{C_j(\sigma) - d_j\}$ and $T_{\max} = \max_{j \in \mathcal{J}_O \cup \mathcal{J}_N^A} \max\{C_j(\sigma) - d_j, 0\}$ in the adjusted schedule σ .

Following the well-known three-field notation for scheduling introduced by Graham et al. [26], our problems can be denoted as

$$1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$$

and

$$1|rej, \delta_{\max} \leq \Delta|T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j,$$

where $\mu \geq 0$ is the relative cost of one unit of disruption with respect to the scheduling cost and $\sum_{j \in \mathcal{J}_N^R} e_j$ denotes the total rejection cost of the rejected jobs.

Each of these two problems admits three corresponding special cases:

(a) the general scheduling problem with job rejection

$$1|rej|L_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$$

and

$$1|rej|T_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j,$$

when Δ is sufficiently large and $\mu = 0$;

(b) the constrained scheduling problem under job rejection

$$1|rej, \delta_{\max} \leq \Delta|L_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$$

and

$$1|rej, \delta_{\max} \leq \Delta|T_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j,$$

when $\mu = 0$;

(c) the overall cost problem

$$1|rej|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$$

and

$$1|rej|T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j,$$

when Δ is sufficiently large.

It is clear that problems $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ and $1|rej, \delta_{\max} \leq \Delta|T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ are NP-hard, because their special cases $1|rej|L_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ and $1|rej|T_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ (by setting $\mathcal{J}_O = \emptyset$) have been proved as NP-hard by Sengupta [13]. For ease of reference, the main notation used throughout the paper is summarized in Table 1.

Table 1. Summary of main notation.

Notation	Description
$\mathcal{J}_O, \mathcal{J}_N, \mathcal{J}$	Sets of original jobs, new jobs, and all jobs, respectively.
$\mathcal{J}_N^A, \mathcal{J}_N^R$	Sets of accepted and rejected new jobs, respectively.
n_O, n_N, n	Numbers of original jobs, new jobs, and all jobs, respectively.
p_j, d_j, e_j	Processing time, due date, and rejection cost of job j , respectively.
P	Total processing time of all jobs, i.e., $P = \sum_{j \in \mathcal{J}} p_j$.
π^*, σ	Planned schedule and rescheduled schedule, respectively.
$\delta_{\max}$	Maximum completion-time deviation of original jobs.
Δ	Maximum allowable disruption threshold.
μ	Disruption penalty coefficient.
$L_{\max}, T_{\max}$	Maximum lateness and maximum tardiness, respectively.
l_o	Last original job in the EDD-ordered sequence.
N_P	Population size.
P_c	Crossover probability.
P_m	Mutation probability.

3. Structural properties and exact algorithms

In this section, we present the structural properties of the optimal solutions for the problems introduced in Section 2. These properties enable the design of exact algorithms to obtain the optimal solution. For convenience of exposition, the jobs in $\mathcal{J}_O \cup \mathcal{J}_N$ are indexed in nondecreasing order of their due dates, i.e., $d_1 \leq d_2 \leq \dots \leq d_n$. Ties among the original jobs are broken according to their relative order in the planned schedule π^* . Furthermore, the last original job in this indexing is denoted by l_o .

For the $1|rej, \delta_{\max} \leq \Delta | L_{\max} + \mu \delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ problem, we have the following lemma.

Lemma 1. *For problem $1|rej, \delta_{\max} \leq \Delta | L_{\max} + \mu \delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$, there exists an optimal schedule σ^* satisfying the following properties:*

- *The accepted new jobs from $\mathcal{J}_N$ that are scheduled before the last job l_o in $\mathcal{J}_O$, together with all original jobs in $\mathcal{J}_O$, preserve their index order under the EDD rule;*
- *The accepted new jobs from $\mathcal{J}_N$ that are scheduled after the last job l_o in $\mathcal{J}_O$ also preserve their index order under the EDD rule;*
- *Any accepted job $j \in \mathcal{J}_N$ with $j > l_o$ must be scheduled after the last job l_o in $\mathcal{J}_O$.*

Proof. Let σ be an optimal schedule, and let $\mathcal{J}_N^R$ and $\mathcal{J}_N^A$ denote the sets of rejected and accepted new jobs in σ , respectively. Fixing $\mathcal{J}_N^R$, consider the subproblem of scheduling the jobs in $\mathcal{J}_O \cup \mathcal{J}_N^A$ to minimize $L_{\max} + \mu \delta_{\max}$ subject to $\delta_{\max} \leq \Delta$. According to the structural result of Hall and Potts [6], this subproblem admits an optimal schedule in which the original jobs are sequenced in EDD order as in π^* and no idle time occurs. Since the rejection set is fixed, this schedule is also optimal for the original problem. Therefore, without loss of generality, we assume that the original jobs in σ are sequenced in EDD order as in π^* and that σ contains no idle time. Let l_o denote the last original job in schedule σ . Define δ as the total processing time of the accepted new jobs that are scheduled before job l_o in σ . By construction, we have $\delta = \delta_{l_o}(\sigma) = \delta_{\max}(\sigma) \leq \Delta$. We now transform σ into a new schedule

σ^* through two steps: (a) rearrange job l_o together with all jobs preceding l_o in σ according to their index order, yielding an intermediate schedule $\hat{\sigma}$, and let $\hat{l}_o$ denote the last original job in $\hat{\sigma}$, and (b) rearrange all accepted new jobs that appear after $\hat{l}_o$ in $\hat{\sigma}$ in their index order, which results in σ^* . For every original job $j \in \mathcal{J}_O$, the quantity $\delta_j(\sigma^*) (= \delta_j(\hat{\sigma}))$ equals the total processing time of accepted new jobs placed before j in $\hat{\sigma}$. Consequently, $\delta_{\max}(\sigma^*) = \delta_{\max}(\hat{\sigma}) \leq \delta = \delta_{\max}(\sigma) \leq \Delta$, implying that both σ^* and $\hat{\sigma}$ remain feasible. Since the EDD rule guarantees optimality in minimizing $L_{\max}$, the sequence determined by $d_1 \leq d_2 \leq \dots \leq d_n$ ensures that $\hat{\sigma}$ performs no worse than σ and σ^* performs no worse than $\hat{\sigma}$. Thus, σ^* is a desired optimal schedule. $\square$

Based on the structural properties established in Lemma 1, we develop a dynamic programming (DP) exact algorithm for the problem $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$.

The key idea of the proposed DP is to exploit the two-segment structure of an optimal schedule established in Lemma 1. First, we sequence all jobs in $\mathcal{J}$ according to the EDD rule such that $d_1 \leq d_2 \leq \dots \leq d_{n_O+n_N}$. Let $P_O = \sum_{j \in \mathcal{J}_O} p_j$. For convenience of exposition, we denote by l_o the last original job in the ordered sequence. According to Lemma 1, there exists an optimal schedule in which the accepted jobs before and after l_o preserve the EDD order within their respective segments. The partial schedule consisting of l_o and all jobs preceding it is referred to as the left schedule, while the accepted new jobs scheduled after l_o form the right schedule. In the end, the right schedule will be appended to the left schedule. Under this structure, original jobs are always included in the left schedule and are processed according to the EDD order, whereas new jobs require status decisions subject to the structural restrictions in Lemma 1. Once these decisions are made, the corresponding schedule is uniquely determined. Therefore, the DP does not explicitly enumerate complete job sequences. Different from a standard job-rejection DP, the proposed DP must determine not only whether a new job is accepted or rejected, but also whether an accepted new job is scheduled before or after l_o .

A 7-dimensional vector

$$(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a, E_j)$$

is used to indicate a state, which is feasible if and only if there is a partial schedule on the first j jobs.

The meaning of each component is as follows:

- j is the index of the job under consideration, which specifies the set of jobs that have been processed in the dynamic programming recursion;
- C_j^b is the current completion time of the left schedule. It determines the starting time of the right schedule after concatenation and is used to compute the overall maximum lateness through $\max\{L_j^b, C_j^b + L_j^a\}$;
- t_j^b is the current total processing time of the new jobs from $\mathcal{J}_N$ before job l_o . This value represents the disruption imposed on the original jobs, and is used to check the constraint $\delta_{\max} \leq \Delta$ and to compute the disruption penalty $\mu\delta_{\max}$;
- t_j^a is the current total processing time of the new jobs from $\mathcal{J}_N$ after job l_o in the right schedule, which is used to compute the normalized completion times of jobs in the right schedule and hence update L_j^a ;
- L_j^b is the maximum lateness of jobs in the left schedule, which is used to evaluate the contribution of the left schedule to the final maximum lateness;
- L_j^a represents the current maximum normalized lateness of jobs scheduled after job l_o in the right schedule, where the starting time is normalized to zero. After the right schedule is appended to the

- left schedule, its contribution to the final maximum lateness is evaluated by $C_j^b + L_j^a$;
- E_j is the current total rejection cost, which is a component of the objective function.

The initial state

$$(0; C_0^b, t_0^b, t_0^a, L_0^b, L_0^a, E_0) = (0; 0, 0, 0, -d_{\max}, -d_{\max}, 0)$$

is true, where $d_{\max} = \max_{j \in \mathcal{J}} d_j$.

In brief, for each original job, there exists only one option, namely, to accept it and place it in the left schedule. For each new job, however, up to three options are available: (i) accept it and assign it in the left schedule, (ii) accept it and assign it in the right schedule, or (iii) reject it. Consequently, a given true state $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a, E_j)$ can generate at most three feasible successor states with the form $(j+1; C_{j+1}^b, t_{j+1}^b, t_{j+1}^a, L_{j+1}^b, L_{j+1}^a, E_{j+1})$. The four cases are described in detail below.

Case 1. If job $j+1 \in \mathcal{J}_O$, then job $j+1$ is appended to the end of the left schedule, leading to a partial schedule on the first $j+1$ jobs. The completion time C_{j+1}^b in the left schedule is updated to $C_j^b + p_{j+1}$. Since no new job from $\mathcal{J}_N$ is involved, the total processing time of new jobs arranged before or after l_o remains unchanged, i.e., $t_{j+1}^b = t_j^b$, $t_{j+1}^a = t_j^a$. Regarding lateness, the maximum lateness of jobs in the left schedule is updated by $L_{j+1}^b = \max\{L_j^b, C_{j+1}^b - d_{j+1}\}$, while the maximum lateness of jobs in the right schedule remains the same, i.e., $L_{j+1}^a = L_j^a$, where the starting time of the right schedule is set to zero. Moreover, the total rejection cost is $E_{j+1} = E_j$. Hence, the recurrence relations for Case 1 can be summarized as follows.

$$\begin{aligned} C_{j+1}^b &= C_j^b + p_{j+1}, & t_{j+1}^b &= t_j^b, & t_{j+1}^a &= t_j^a, \\ L_{j+1}^b &= \max\{L_j^b, C_{j+1}^b - d_{j+1}\}, & L_{j+1}^a &= L_j^a, & E_{j+1} &= E_j. \end{aligned} \quad (3.1)$$

Case 2. If job $j+1 \in \mathcal{J}_N$ and $j+1 < l_o$ with $t_j^b + p_{j+1} \leq \Delta$, then job $j+1$ is accepted and appended to the end of the left schedule, leading to a partial schedule on the first $j+1$ jobs. The completion time C_{j+1}^b in the left schedule is updated to $C_j^b + p_{j+1}$. Since the new job is scheduled before l_o , the total processing time of new jobs preceding l_o increases, i.e., $t_{j+1}^b = t_j^b + p_{j+1}$, while the total processing time of new jobs assigned after l_o remains unchanged, i.e., $t_{j+1}^a = t_j^a$. Regarding lateness, the maximum lateness of jobs in the left schedule is updated as $L_{j+1}^b = \max\{L_j^b, C_{j+1}^b - d_{j+1}\}$, whereas the maximum lateness of jobs in the right schedule remains unchanged, that is, $L_{j+1}^a = L_j^a$, with the starting time of the right schedule set to zero. Moreover, the total rejection cost is $E_{j+1} = E_j$. In summary, the recurrence relations for Case 2 can be written as

$$\begin{aligned} C_{j+1}^b &= C_j^b + p_{j+1}, & t_{j+1}^b &= t_j^b + p_{j+1}, & t_{j+1}^a &= t_j^a, \\ L_{j+1}^b &= \max\{L_j^b, C_{j+1}^b - d_{j+1}\}, & L_{j+1}^a &= L_j^a, & E_{j+1} &= E_j. \end{aligned} \quad (3.2)$$

Case 3. If $j+1 \in \mathcal{J}_N$, then job $j+1$ is accepted and appended to the end of the right schedule, leading to a partial schedule on the first $j+1$ jobs. In this case, the completion time in the left schedule is given by $C_{j+1}^b = C_j^b$. Since the job is arranged after l_o , the total processing time of new jobs scheduled before l_o is preserved, that is, $t_{j+1}^b = t_j^b$, whereas the total processing time of new jobs scheduled after l_o increases by its processing time. Specifically, $t_{j+1}^a = t_j^a + p_{j+1}$. Regarding lateness, the maximum lateness of jobs in the left schedule remains the same, that is, $L_{j+1}^b = L_j^b$, while the maximum lateness of jobs after l_o in the right schedule is updated according to $L_{j+1}^a = \max\{L_j^a, t_j^a + p_{j+1} - d_{j+1}\}$, where the

starting time of the right schedule is set to zero. Also, the total rejection cost is $E_{j+1} = E_j$. In summary, the recurrence relations for Case 3 can be expressed as

$$\begin{aligned} C_{j+1}^b &= C_j^b, & t_{j+1}^b &= t_j^b, & t_{j+1}^a &= t_j^a + p_{j+1}, \\ L_{j+1}^b &= L_j^b, & L_{j+1}^a &= \max\{L_j^a, t_j^a + p_{j+1} - d_{j+1}\}, & E_{j+1} &= E_j. \end{aligned} \quad (3.3)$$

Case 4. If $j + 1 \in \mathcal{J}_N$, job $j + 1$ is rejected, leading to a partial schedule on the first $j + 1$ jobs. Evidently, apart from the rejection cost $E_{j+1} = E_j + e_{j+1}$, all other components remain unchanged. Hence, we obtain the recurrence relations for Case 4 as follows.

$$\begin{aligned} C_{j+1}^b &= C_j^b, & t_{j+1}^b &= t_j^b, & t_{j+1}^a &= t_j^a, \\ L_{j+1}^b &= L_j^b, & L_{j+1}^a &= L_j^a, & E_{j+1} &= E_j + e_{j+1}. \end{aligned} \quad (3.4)$$

For each state with the same $(j, C_j^b, t_j^b, t_j^a, L_j^b, L_j^a)$, only the one with the minimum value of E_j is retained. This dominance rule is valid because states sharing identical values of $(j, C_j^b, t_j^b, t_j^a, L_j^b, L_j^a)$ have exactly the same feasible extensions. Therefore, a state with a larger rejection cost can never lead to a better final solution and may be safely discarded.

Define $p_{\max} = \max_{j \in \mathcal{J}} p_j$, $d_{\max} = \max_{j \in \mathcal{J}} d_j$, and $e_{\max} = \max_{j \in \mathcal{J}_N} e_j$.

With the initial true state $(0; 0, 0, 0, -d_{\max}, -d_{\max}, 0)$ and the recurrences in Eqs. (3.1), (3.2), (3.3), and (3.4), one can construct a dynamic programming table $DP(j; C^b, t^b, t^a, L^b, L^a, E)$, where $0 \leq j \leq n$, $0 \leq C^b \leq np_{\max}$, $0 \leq t^b \leq \Delta$, $0 \leq t^a \leq n_N p_{\max}$, $-d_{\max} \leq L^b \leq np_{\max}$, $-d_{\max} \leq L^a \leq np_{\max}$, and $0 \leq E \leq n_N e_{\max}$, to compute all the true states.

Based on the preceding state transition equations, we now provide a detailed description of the dynamic programming exact algorithm for the problem $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$, which is deemed Algorithm 1.

Algorithm 1 Exact algorithm for the problem $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$

- 1: Sort all the jobs in $\mathcal{J}_O \cup \mathcal{J}_N$ in the EDD order, i.e., $d_1 \leq d_2 \leq \dots \leq d_n$. Let the last job in $\mathcal{J}_O$ be l_o .
 - 2: Build a dynamic programming table $DP1(j; C^b, t^b, t^a, L^b, L^a, E)$, where $0 \leq j \leq n$, $0 \leq C^b \leq np_{\max}$, $0 \leq t^b \leq \Delta$, $0 \leq t^a \leq n_N p_{\max}$, $-d_{\max} \leq L^b \leq np_{\max}$, $-d_{\max} \leq L^a \leq np_{\max}$, and $0 \leq E \leq n_N e_{\max}$.
 - 3: Start from the initial true state $(0; 0, 0, 0, -d_{\max}, -d_{\max}, 0)$. For each true state $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a, E_j)$, generate the successor states of the form $(j + 1; C_{j+1}^b, t_{j+1}^b, t_{j+1}^a, L_{j+1}^b, L_{j+1}^a, E_{j+1})$ according to the recurrence relations given in Eqs. (3.1), (3.2), (3.3), and (3.4).
 - 4: For each fixed combination $(j + 1; C_{j+1}^b, t_{j+1}^b, t_{j+1}^a, L_{j+1}^b, L_{j+1}^a)$, retain only a state with the minimum E_{j+1} .
 - 5: Among all the true states of the form $(n; C_n^b, t_n^b, t_n^a, L_n^b, L_n^a, E_n)$, find the one with the minimum value of $\max\{L_n^b, C_n^b + L_n^a\} + \mu t_n^b + E_n$, and by backtracking determine the corresponding subset of accepted jobs from $\mathcal{J}_N$ before job l_o and the subset of accepted jobs from $\mathcal{J}_N$ after job l_o .
 - 6: Schedule the above jobs according to the rules as described in Lemma 1.
-

Clearly, the running time of Algorithm 1 is in $O(n^2 n_N p_{\max}^2 (np_{\max} + d_{\max})^2 \Delta)$, as formally stated in the following theorem.

Theorem 1. Algorithm 1 finds an optimal solution to the scheduling problem $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ within running time $O(n^2 n_N p_{\max}^2 (np_{\max} + d_{\max})^2 \Delta)$.

Proof. By Lemma 1, there exists an optimal schedule satisfying the two-segment structure stated therein. Hence, it suffices to consider only schedules with this structure.

We first show that the state representation correctly models the overlapping subproblems and that the dominance rule is valid. Consider two feasible states having the same values of $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a)$ and rejection costs E_j and $\bar{E}_j$, respectively. Since the two states have the same value of j , they have the same remaining jobs. For job $j + 1$, the admissible decisions are determined by whether it is an original or new job and, for a new job, whether $j + 1 < l_o$. The feasibility of assigning a new job to the left schedule is determined by $t_j^b + p_{j+1} \leq \Delta$. Therefore, the two states admit the same feasible decisions for job $j + 1$.

If the same decision is applied to both states, Eqs. (3.1)–(3.4) produce successor states having the same values of $(j + 1; C_{j+1}^b, t_{j+1}^b, t_{j+1}^a, L_{j+1}^b, L_{j+1}^a)$ and add the same amount to their respective rejection costs. Repeating this argument for the remaining jobs shows that the two states admit exactly the same feasible sequences of decisions for jobs $j + 1, \dots, n$. Under any such sequence, they produce the same final values of $(C_n^b, t_n^b, t_n^a, L_n^b, L_n^a)$ and incur the same additional rejection cost. Thus, the state representation correctly models the overlapping subproblems in Algorithm 1: partial schedules arising from different preceding decision sequences but having the same values of $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a)$ induce the same subproblem for the remaining jobs $j + 1, \dots, n$. Since the terminal objective value is $\max\{L_n^b, C_n^b + L_n^a\} + \mu t_n^b + E_n$, the difference between the final objective values obtained from the two states under any common feasible sequence of subsequent decisions is $E_j - \bar{E}_j$. Therefore, if $E_j > \bar{E}_j$, the former state always yields a larger objective value and can be safely discarded. If $E_j = \bar{E}_j$, either state may be retained. Hence, retaining only a state with the minimum E_j for each combination $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a)$ is valid.

We next verify the correctness and completeness of the transitions. Under the structure established in Lemma 1, each original job is assigned to the left schedule, each accepted new job with index smaller than l_o is assigned to either the left or right schedule, and each accepted new job with index larger than l_o is assigned to the right schedule; a new job may also be rejected. These are exactly the decisions represented by Eqs. (3.1)–(3.4). The transition conditions preserve the EDD order within the two segments and enforce $t_j^b + p_{j+1} \leq \Delta$ whenever a new job is assigned to the left schedule. Thus, every complete path generated by the DP represents a feasible schedule with the structure specified in Lemma 1. Conversely, every feasible schedule with this structure specifies one admissible decision for each EDD-indexed job and therefore determines a complete path generated by Eqs. (3.1)–(3.4). Hence, the transitions neither generate infeasible schedules nor omit any schedule that must be considered.

We next establish the optimal substructure. Let σ^* be an optimal schedule satisfying the structure in Lemma 1. For any j , suppose that the decisions of σ^* for the first j EDD-indexed jobs produce the state $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a, E_j)$. If another feasible partial solution for the first j EDD-indexed jobs produced the same values of $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a)$ with rejection cost $\bar{E}_j < E_j$, then, by the property established above, its decisions for the first j jobs could replace the corresponding decisions of σ^* , while the decisions of σ^* for jobs $j + 1, \dots, n$ remained unchanged. The resulting schedule would be feasible and would have an objective value smaller than that of σ^* by $E_j - \bar{E}_j$, contradicting the optimality of σ^* . Hence, for every j , the partial solution induced by σ^* on the first j EDD-indexed jobs has the minimum rejection cost among all feasible partial solutions producing the same values of $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a)$. Thus, an optimal complete schedule is composed of optimal partial solutions for the corresponding states, which establishes the required optimal substructure. Together with the completeness of the transitions and the validity of the dominance rule, this shows that Algorithm 1 retains an optimal terminal state and

therefore returns an optimal solution.

Finally, by the dominance rule, at each stage j , Algorithm 1 retains at most one state for each combination $(C_j^b, t_j^b, t_j^a, L_j^b, L_j^a)$, so E_j does not contribute an additional factor. According to the ranges of these variables, the number of retained states at each stage is $O(nn_N p_{\max}^2 (np_{\max} + d_{\max})^2 \Delta)$. There are n stages, and each retained state generates at most four successor states in constant time. Therefore, the running time of Algorithm 1 is $O(n^2 n_N p_{\max}^2 (np_{\max} + d_{\max})^2 \Delta)$. This completes the proof. $\square$

Now, we turn to investigate the problem $1|rej, \delta_{\max} \leq \Delta |T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$. We also have the similar structural properties on the optimal solution for the problem $1|rej, \delta_{\max} \leq \Delta |T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ as shown in Lemma 2, so we omit the similar proof.

Lemma 2. *For problem $1|rej, \delta_{\max} \leq \Delta |T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$, there exists an optimal schedule σ^* satisfying the following properties:*

- *The accepted new jobs from $\mathcal{J}_N$ that are scheduled before the last job l_o in $\mathcal{J}_O$, together with all original jobs in $\mathcal{J}_O$, preserve their index order under the EDD rule;*
- *The accepted new jobs from $\mathcal{J}_N$ that are scheduled after the last job l_o in $\mathcal{J}_O$ also preserve their index order under the EDD rule;*
- *Any accepted job $j \in \mathcal{J}_N$ with $j > l_o$ must be scheduled after the last job l_o in $\mathcal{J}_O$.*

With some minor modifications of Algorithm 1, we can also design the dynamic programming exact algorithm for the problem $1|rej, \delta_{\max} \leq \Delta |T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ as below, which is deemed Algorithm 2.

Algorithm 2 Exact algorithm for the problem $1|rej, \delta_{\max} \leq \Delta |T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$

- 1: Sort all the jobs in $\mathcal{J}_O \cup \mathcal{J}_N$ in the EDD order, i.e., $d_1 \leq d_2 \leq \dots \leq d_n$. Let the last job in $\mathcal{J}_O$ be l_o .
 - 2: Build a dynamic programming table $DP2(j; C^b, t^b, t^a, L^b, L^a, E)$, where $0 \leq j \leq n$, $0 \leq C^b \leq np_{\max}$, $0 \leq t^b \leq \Delta$, $0 \leq t^a \leq n_N p_{\max}$, $-d_{\max} \leq L^b \leq np_{\max}$, $-d_{\max} \leq L^a \leq np_{\max}$, and $0 \leq E \leq n_N e_{\max}$.
 - 3: Start from the initial true state $(0; 0, 0, 0, -d_{\max}, -d_{\max}, 0)$. For each true state $(j; C_j^b, t_j^b, t_j^a, L_j^b, L_j^a, E_j)$, generate the successor states of the form $(j+1; C_{j+1}^b, t_{j+1}^b, t_{j+1}^a, L_{j+1}^b, L_{j+1}^a, E_{j+1})$ according to the recurrences given in Eqs. (3.1), (3.2), (3.3), and (3.4).
 - 4: For each fixed combination $(j+1; C_{j+1}^b, t_{j+1}^b, t_{j+1}^a, L_{j+1}^b, L_{j+1}^a)$, retain only a state with the minimum E_{j+1} .
 - 5: Among all the true states of the form $(n; C_n^b, t_n^b, t_n^a, L_n^b, L_n^a, E_n)$, find the one with the minimum value of $\max\{0, L_n^b, C_n^b + L_n^a\} + \mu t_n^b + E_n$, and by backtracking determine the corresponding subset of accepted jobs from $\mathcal{J}_N$ before job l_o and the subset of accepted jobs from $\mathcal{J}_N$ after job l_o .
 - 6: Schedule the above jobs according to the rules as described in Lemma 2.
-

Finally, we obtain the following theorem.

Theorem 2. *Algorithm 2 finds an optimal solution to the scheduling problem $1|rej, \delta_{\max} \leq \Delta |T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ within running time $O(n^2 n_N p_{\max}^2 (np_{\max} + d_{\max})^2 \Delta)$.*

Proof. By Lemma 2, it suffices to consider schedules satisfying the two-segment structure stated therein. Algorithm 2 differs from Algorithm 1 only in the objective function; its state representation, transitions,

feasibility conditions, and dominance rule are identical to those of Algorithm 1. Moreover, at a terminal state, $T_{\max} = \max\{0, L_n^b, C_n^b + L_n^a\}$, whereas $L_{\max} = \max\{L_n^b, C_n^b + L_n^a\}$. Thus, $T_{\max}$ can be evaluated directly from the same state variables as $L_{\max}$. Consequently, the arguments in the proof of Theorem 1 concerning the completeness of the transitions, overlapping subproblems, validity of the dominance rule, and optimal substructure apply without change. Therefore, Algorithm 2 finds an optimal solution.

Since Algorithms 1 and 2 have the same state ranges and the same number of transitions from each state, as well as the same running time. Hence, Algorithm 2 runs in $O(n^2 n_N p_{\max}^2 (np_{\max} + d_{\max})^2 \Delta)$. $\square$

4. Genetic algorithm

In this section, we provide a detailed description of the genetic algorithm developed to solve the problems $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$ and $1|rej, \delta_{\max} \leq \Delta|T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$, which is named the genetic algorithm based on optimal structure (GABO). The effectiveness of the GABO mainly comes from incorporating the structural properties established in Lemma 1 into the chromosome encoding scheme (the same idea applies to the $T_{\max}$ problem based on Lemma 2), thereby reducing the search space explored by the genetic algorithm. The description of the algorithm starts with the chromosome encoding strategy, where encoding rules are formulated based on the structural properties of the optimal solution. Next, we introduce the population update process of the GABO, including population initialization, selection, crossover, and mutation. This process is outlined in Algorithm 3.

Algorithm 3 GABO: The genetic algorithm with chromosome encoding based on optimal solution structure

- 1: **Input:** The parameters of jobs, including original jobs and new jobs;
 - 2: **Output:** A job sequence and its objective function value;
 - 3: Initialize the chromosomes of individuals in the population;
 - 4: Verify and correct chromosomes based on the optimal solution structure to ensure feasibility for all chromosomes;
 - 5: Set G as the maximum number of generations;
 - 6: **for** $i = 1 : G$ **do**
 - 7: Evaluate the fitness of every individual;
 - 8: Record the best individual (bi) and its fitness across generations;
 - 9: Perform the roulette-wheel-based selection operation;
 - 10: Conduct crossover with a certain probability between individuals;
 - 11: Conduct mutation with a certain probability;
 - 12: Repeat Step 4;
 - 13: **end**
 - 14: Return the job sequence decoded from bi and its objective function value.
-

4.1. Chromosome representation

For the problem $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$, we already know that there exists an optimal schedule with the structure as described in Lemma 1, which means that we can find the optimal schedule from schedules that satisfy Lemma 1, reducing the search space for feasible solutions. In an optimal schedule, the set of original jobs cannot be rejected and should observe their index order EDD. Therefore,

for all the jobs in $\mathcal{J}$, we sequence them in an EDD order. Without loss of generality, we assume that $d_1 \leq d_2 \leq \dots \leq d_{n_o+n_N}$. First, we need to clarify which jobs are accepted and which are rejected in $\mathcal{J}_N$. Second, for the accepted jobs, we need to clarify which ones are processed before l_o and which ones are processed after l_o . Since both the jobs before and after l_o need to satisfy the EDD rule, respectively, we can obtain a unique schedule based on the structural properties shown in Lemma 1.

Based on the above description, we adopt a hybrid binary-integer chromosome encoding, and the specific encoding rules are as follows. The chromosome length is defined as $2|\mathcal{J}|$, where the first $|\mathcal{J}|$ genes represent the acceptance and rejection of jobs in $\mathcal{J}$: 1 represents acceptance, 0 represents rejection, and the last $|\mathcal{J}|$ genes represent the processing state of the job in $\mathcal{J}$: If the job is accepted, then 1 represents that it is processed on the left schedule and 2 represents that it is processed on the right schedule. If the job is rejected, then 0 represents that it is not processed. It should be noted that only the accepted jobs need to consider the processing position. Considering the chromosome encoding scheme, the problem constraints, and the structural characteristics of the optimal solution, we define the feasibility of a chromosome (i.e., whether the schedule decoded from the chromosome satisfies the problem constraints) based on the following three rules:

1. All original jobs must be accepted; hence, their acceptance genes must be set to 1;
2. All original jobs must be assigned to the left schedule; hence, their position genes must be set to 1;
3. Every accepted new job with an EDD index greater than l_o must be assigned to the right schedule; hence, its position gene must be set to 2.

For example, suppose the job set $\mathcal{J} = \{1, 2, 3, 4, 5, 6\}$, where the jobs 1, 3, 4 are the original jobs, the remaining jobs are new jobs, and $d_1 \leq d_2 \leq \dots \leq d_6$. Because jobs 1, 3, 4 are the original jobs and cannot be rejected, the corresponding acceptance or rejection status code is 1, and the processing status code is 1. The chromosome encoding of a feasible solution is $X = (111101121102)$, which indicates that jobs 1, 2, 3, 4, and 6 are accepted and job 5 is rejected. Among the accepted jobs, jobs 2 and 6 are processed on the right schedule, while the rest of the jobs are processed on the left schedule. For convenience, we divide the first 6 genes and the last 6 genes into two rows. The first row represents the acceptance or rejection status of the jobs, and the second row represents the processing status of the accepted jobs or rejection, as shown in Table 2.

Table 2. Example of encoding a chromosome.

Job	1	2	3	4	5	6
Accept or Reject	1	1	1	1	0	1
Position or Rejection	1	2	1	1	0	2

The corresponding sequence (i.e., solution) of the above-mentioned chromosome X is (1, 3, 4, 2, 6) (the new job 5 is rejected). This encoding rule also applies to the problem $1|rej, \delta_{\max} \leq \Delta|T_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$.

4.2. Population initialization

Population initialization is the initial stage of the genetic algorithm and is one of its critical steps, which affects the initial search state and global search performance of the algorithm. There are several common initialization strategies, such as random initialization, initialization based on a uniform distribution, and local search initialization. We adopt a random initialization strategy to generate

the initial population, during which individuals are allowed to violate feasibility constraints. Then, based on the three feasibility rules of the chromosome, each individual in the population undergoes a feasibility check, followed by chromosome repair if any constraint violations are detected. Since both problems have the constraint $\delta_{\max} \leq \Delta$, we need to ensure that the schedule decoded from the final optimal chromosome obtained by the algorithm satisfies the constraint. Individuals that do not meet the constraint may be removed from the population. However, checking the constraint for each individual upon creating it would greatly increase the computational burden. Therefore, we incorporate this constraint into the calculation of the fitness function. When calculating the fitness value of an individual, a sufficiently large penalty is added for individuals that violate the constraint, while the penalty is 0 for those that satisfy the constraint.

4.3. Selection

Genetic operations include selection, crossover, and mutation. Selection refers to selecting dominant individuals, crossover involves exchanging genes between mates to produce offspring, and mutation involves changes in an individual's gene information. Selection is the first step of genetic operations. In each iteration, a set of outstanding individuals is chosen from the current population $P(t)$ based on their fitness, and these individuals are transferred to the next generation population $P(t + 1)$. The commonly used selection operators are roulette wheel selection and tournament selection. In this paper, we adopt roulette wheel selection [27], one of the earliest and most widely used selection methods in genetic algorithms due to its simplicity and effectiveness. In roulette wheel selection, each individual is selected probabilistically based on their fitness. Let FV_i denote the penalized objective value of individual i , i.e., $FV_i = L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j + d_{p,i}$ for the maximum-lateness problem, where $L_{\max}$ is replaced by $T_{\max}$ for the maximum-tardiness problem, and $d_{p,i}$ is the penalty term associated with the violation of $\delta_{\max} \leq \Delta$. Let $FV_{\min}^t = \min_{k \in P(t)} FV_k$ and define the nonnegative shift as

$$s_t = \begin{cases} 1 - FV_{\min}^t, & FV_{\min}^t \leq 0, \\ 0, & FV_{\min}^t > 0. \end{cases}$$

The fitness value of individual i is then defined as

$$f_i = \frac{1}{FV_i + s_t}. \quad (4.1)$$

This transformation guarantees that all fitness values are positive, even when the penalized objective value is zero or negative.

From this definition, it follows that as FV_i decreases, the corresponding fitness f_i increases. Consequently, individuals with smaller penalized objective values are more likely to be selected, which is consistent with the minimization objective. If the population size is N_P , the probability of selecting individual i is given by

$$p_i = f_i / \sum_{k=1}^{N_P} f_k, \quad i = 1, 2, \dots, N_P. \quad (4.2)$$

Divide the roulette wheel into N_P parts according to the above probability. In each round of selection, first generate a random number between $[0, 1]$ and use this number as a pointer to point to the roulette wheel. The individuals corresponding to the area pointed to by the pointer are the ones we choose to

transfer to the next generation. Repeat the N_p rounds of selection to ensure that the population size of the next generation is N_p .

4.4. Crossover

The crossover operator selects a pair of chromosomes from the parent population and exchanges genetic information with a crossover probability of P_c , thereby generating new offspring and enhancing the exploration capability of the algorithm. In this study, we adopt a uniform crossover operator. For each selected pair of parent chromosomes, crossover is first performed with probability P_c . If crossover is not activated, the two chromosomes remain unchanged and are directly passed to the next operation. If crossover is activated, each gene position is independently considered for exchange with probability 0.5. Specifically, the corresponding genes of the two parent chromosomes are exchanged when the randomly generated binary mask takes a value of 1, and remain unchanged otherwise. In this way, two offspring chromosomes are generated.

Since the chromosome consists of both acceptance genes and position genes, the crossover operation may change job acceptance decisions and schedule assignments. Consequently, the generated offspring chromosomes may violate the encoding feasibility rules. The treatment of such infeasible chromosomes is described together with the mutation operator.

4.5. Mutation

The mutation operator mainly plays an auxiliary search role in genetic algorithms, which can improve population diversity. Low-frequency mutations can preserve important genes in the population, while high-frequency mutations may cause the algorithm to behave like a random search. Since the chromosome contains both binary acceptance genes and integer-valued position genes, different mutation operations are applied to the two types of genes. Each gene of every chromosome undergoes mutation independently with a probability of P_m . If an acceptance gene is selected for mutation, its value is flipped between 0 and 1. If a position gene is selected for mutation, its value is replaced uniformly at random by one of the other two values in $\{0, 1, 2\}$.

Crossover and mutation operations may produce chromosomes that violate the encoding feasibility rules. Performing feasibility checks and repairs after each individual genetic operation would incur additional computational cost. Therefore, in each generation, once crossover and mutation have been completed for the entire offspring population, all offspring chromosomes are checked and repaired together. Specifically, the repair procedure is conducted according to the following rules: (i) For every original job, its acceptance gene and position gene are set to 1, so that it is accepted and assigned to the left schedule. (ii) If a new job is rejected, its position gene is set to 0. (iii) If an accepted new job with an EDD index smaller than l_o has position gene 0, its position gene is randomly set to 1 or 2. (iv) Every accepted new job with an EDD index greater than l_o is assigned to the right schedule by setting its position gene to 2.

It should be noted that the repair procedure is only used to maintain chromosome feasibility with respect to the encoding rules. The disruption constraint $\delta_{\max} \leq \Delta$ is handled separately during fitness evaluation. For each chromosome, the violation amount is calculated as $v = \max\{0, \delta_{\max} - \Delta\}$. If the disruption constraint is violated, a quadratic penalty term λv^2 is added to the objective value, where λ is a sufficiently large penalty coefficient. This penalty mechanism increases the penalized objective value

of infeasible chromosomes and reduces their selection probability, thereby guiding the search toward feasible regions of the solution space.

5. Numerical experiments

In this section, we focus on the problem $1|rej, \delta_{\max} \leq \Delta|L_{\max} + \mu\delta_{\max} + \sum_{j \in \mathcal{J}_N^R} e_j$. To evaluate the performance of the proposed algorithms, a mixed-integer linear programming (MILP) formulation is developed and solved using Gurobi Optimizer 13.0.1 as a benchmark. The MILP formulation is provided in Appendix A. The proposed dynamic programming-based exact algorithm (Algorithm 1) and the genetic algorithm based on optimal structure (GABO) are coded in Python. In addition, a standard genetic algorithm (SGA) is implemented as a baseline algorithm to evaluate the effectiveness of the problem-specific design of the GABO. All algorithms are coded in Python 3.7 and run in PyCharm Professional 2024.2. All experiments are conducted on a computer equipped with a 2.60 GHz 13th Gen Intel® Core™ i5-13500H processor and 16 GB of RAM, running the Windows 11 Home operating system.

Since the considered problem is NP-hard, obtaining optimal solutions becomes increasingly difficult as the input size grows. As a result, only relatively small instances can be solved to optimality within a limited computational time. Our preliminary computational results show that most instances with $n_O \geq 20$ and $n_N \geq 20$ cannot be solved to optimality within 30 minutes by either the MILP model or the proposed exact algorithm.

Based on the above observations, we first consider instances where the number of original jobs is $n_O \in \{5, 10, 15, 20\}$ and the number of new jobs is $n_N \in \{5, 10, 15, 20\}$. For each combination of (n_O, n_N) , 10 instances are randomly generated, resulting in a total of $4 \times 4 \times 10 = 160$ instances. The parameter settings follow Fang et al. [24]. For each job j , the processing time p_j and due date d_j are randomly generated from the discrete uniform distributions $U[1, 50]$ and $U[\lfloor P/4 \rfloor, P]$, respectively. The rejection cost of each new job is generated as $e_j = \beta_j p_j$, where β_j is an integer uniformly selected from $\{1, 2, \dots, 10\}$. Under this setting, new jobs with longer processing times tend to incur larger rejection penalties. Furthermore, the disruption penalty coefficient is fixed at $\mu = 1$, so that one unit of maximum completion-time deviation has the same weight as one unit of maximum lateness in the objective function. The disruption threshold is set as $\Delta = \lfloor P/4 \rfloor$. For the MILP model and Algorithm 1, a time limit of 1800 seconds is imposed on each instance. For the GABO and SGA, each run terminates when either the maximum number of generations, set to 2000, is reached or the best objective value is not improved for 100 consecutive generations. The number of independent runs used in each experiment is specified below.

Before comparing the GABO with other methods, a sensitivity analysis is conducted to determine suitable values for the algorithmic parameters. The combination $(n_O, n_N) = (20, 15)$ is selected for this analysis because all 10 instances under this combination can be solved to optimality within the time limit. Therefore, the optimal objective values provide a reliable benchmark for evaluating the performance of the GABO under different parameter settings. For each parameter setting, the GABO is independently executed 10 times on each of the 10 instances. We report the average CPU time (Avg Time) and the average percentage deviation (Avg Gap) from the optimal objective value obtained by Algorithm 1, where $\text{Gap} = \frac{Z - Z^*}{Z^*} \times 100\%$, Z is the objective value obtained by the GABO, and Z^* is the optimal objective value. A one-factor-at-a-time strategy is adopted. The ranges considered for the three

parameters are based on commonly used settings in the genetic algorithm literature. Specifically, N_P varies from 20 to 200, P_c from 0.6 to 0.9, and P_m from 0.01 to 0.10. When one parameter is varied, the remaining parameters are fixed at their default values. Specifically, the default settings are $N_P = 50$, $P_c = 0.8$, and $P_m = 0.05$. The results are reported in Table 3.

Table 3. Sensitivity analysis of the GABO parameters.

Parameter	Value	Avg Gap	Avg Time
Population Size (N_P)	20	9.98%	0.78
	50	6.81%	2.15
	100	8.09%	4.26
	150	9.16%	8.96
	200	9.84%	8.76
Crossover Probability (P_c)	0.6	8.10%	1.42
	0.7	7.01%	1.57
	0.8	6.81%	1.74
	0.9	10.46%	1.99
Mutation Probability (P_m)	0.01	8.23%	1.89
	0.03	8.19%	3.10
	0.05	6.81%	3.79
	0.07	10.37%	1.69
	0.10	15.21%	1.97

As shown in Table 3, the performance of the GABO is influenced by the parameter settings. For the population size N_P , the average gap decreases when N_P increases from 20 to 50, whereas larger values do not lead to further improvement and instead increase the computational time. Regarding the crossover probability, the best solution quality is obtained at $P_c = 0.8$. For the mutation probability, $P_m = 0.05$ yields the smallest average gap, while a larger value ($P_m = 0.10$) significantly deteriorates the solution quality. Overall, $N_P = 50$, $P_c = 0.8$, and $P_m = 0.05$ provide the best balance between solution quality and computational efficiency among the considered parameter settings.

Tables 4 and 5 report the computational results on small-sized instances, where n_O and n_N denote the numbers of original and new jobs, respectively. For the GABO and SGA, each small-sized instance is solved independently 20 times. “Avg Time” denotes the average CPU time in seconds. The notation “< 0.01” indicates that all 10 instances are solved within 0.01 second. If an instance cannot be solved to optimality within the prescribed time limit of 1800 seconds, a CPU time of 1800 seconds is used when computing the average, and the resulting value is prefixed by the symbol “>”. “Avg Gap” denotes the average percentage deviation from the optimal objective value Z^* obtained by Algorithm 1. For the MILP model, if optimality is not proven within the time limit but a feasible solution exists, the best feasible objective value found within 1800 seconds is used to compute Avg Gap. “# Solved” denotes the number of instances solved to optimality within the time limit. For the GABO and SGA, “Avg Success Rate” denotes the average proportion of runs that attain Z^* .

Table 4. Performance comparison of the MILP model and Algorithm 1.

n_O	n_N	MILP			Algorithm 1	
		Avg Time	Avg Gap	# Solved	Avg Time	# Solved
5	5	0.42	0	10	< 0.01	10
5	10	7.29	0	10	0.06	10
5	15	114.28	0	10	2.71	10
5	20	> 643.77	0	9	338.79	10
10	5	2.00	0	10	< 0.01	10
10	10	35.85	0	10	0.09	10
10	15	> 857.95	0	8	12.13	10
10	20	> 1476.20	0	3	> 305.59	9
15	5	6.58	0	10	< 0.01	10
15	10	212.67	0	10	0.14	10
15	15	> 1476.20	0	3	8.92	10
15	20	> 1800	0.43%	0	> 481.43	9
20	5	31.69	0	10	< 0.01	10
20	10	> 914.20	0	8	0.14	10
20	15	> 1800	8.10%	0	17.65	10
20	20	> 1800	11.79%	0	> 334.84	9

Based on Table 4, both the MILP model and Algorithm 1 are capable of obtaining optimal solutions for relatively small combinations ($n_O \leq 10, n_N \leq 10$). However, Algorithm 1 is substantially more efficient. At $(n_O, n_N) = (10, 10)$, the average CPU time of the MILP model is 35.85 seconds, whereas Algorithm 1 requires only 0.09 second. As the problem size increases, the performance of the MILP model deteriorates rapidly. In particular, no instance can be solved to optimality within the time limit for $(15, 20)$, $(20, 15)$, and $(20, 20)$, and the corresponding average gaps increase to 0.43%, 8.10%, and 11.79%, respectively. By comparison, Algorithm 1 solves all instances for most tested combinations and still solves 9 out of 10 instances for $(10, 20)$, $(15, 20)$, and $(20, 20)$. These results demonstrate the superior computational efficiency and scalability of Algorithm 1 compared with the MILP model on the tested instance set. The results also indicate that the number of new jobs has a stronger impact on computational difficulty than the number of original jobs. When $n_O = 20$, the average CPU time of Algorithm 1 increases from less than 0.01 second to more than 334.84 seconds as n_N increases from 5 to 20. Conversely, when $n_N = 10$, increasing n_O from 5 to 20 only increases the average CPU time from 0.06 second to 0.14 second.

Table 5. Performance comparison of the GABO and SGA on small-sized instances.

n_O	n_N	GABO			SGA		
		Avg Time	Avg Gap	Avg Success Rate	Avg Time	Avg Gap	Avg Success Rate
5	5	0.54	0	100.00%	0.76	11.06%	74.00%
5	10	0.65	0.30%	98.00%	2.26	32.95%	43.00%
5	15	1.29	1.87%	82.00%	5.63	84.51%	7.50%
5	20	2.47	12.75%	51.00%	6.54	135.94%	0
10	5	0.55	0.28%	92.00%	1.81	22.25%	29.00%
10	10	0.89	0.87%	89.00%	5.26	50.76%	4.50%
10	15	1.82	19.30%	58.00%	4.81	94.40%	0
10	20	2.26	19.28%	40.00%	6.39	293.27%	0
15	5	0.68	0.59%	91.00%	2.66	37.43%	12.00%
15	10	2.63	3.07%	84.00%	3.27	76.18%	0
15	15	5.43	9.64%	57.00%	3.04	161.50%	0
15	20	3.59	19.94%	34.44%	5.64	309.23%	0
20	5	0.81	0	100.00%	5.90	63.76%	5.00%
20	10	1.69	4.43%	63.00%	4.82	79.97%	0.50%
20	15	5.55	6.82%	49.00%	4.40	195.73%	0
20	20	3.54	26.41%	2.22%	5.65	303.54%	0

Table 5 reports the performance comparison between the GABO and SGA on small-sized instances. Overall, the GABO clearly outperforms the SGA in terms of solution quality and reliability. For all tested combinations, the GABO obtains a smaller average gap and a higher average success rate than the SGA. Averaged over all 16 combinations, the average gap of the GABO is only 7.85%, whereas that of the SGA reaches 122.03%. Meanwhile, the average success rate of the GABO is 68.17%, compared with only 10.97% for the SGA. The advantage of the GABO becomes more evident as the problem size increases. For example, when $(n_O, n_N) = (20, 20)$, the average gap of the GABO is 26.41%, while that of the SGA increases to 303.54%. In the same case, the GABO still reaches the optimal solution in 2.22% of the runs, whereas the SGA fails to obtain the optimal solution in any run. Similar observations can be made for other relatively difficult instances, such as $(15, 20)$ and $(10, 20)$, where the SGA has very large average gaps and zero success rate, while the GABO still maintains a considerably better solution quality. In terms of computational time, both algorithms require only a few seconds on average for all small-sized instances. The GABO is faster than the SGA in most tested combinations, with an overall average CPU time of 2.15 seconds compared with 4.30 seconds for the SGA. However, the main advantage of the GABO lies in solution quality rather than computational time. These results indicate that incorporating the optimal structural properties into the genetic algorithm effectively improves the search efficiency and robustness of the algorithm, especially for instances with larger numbers of new jobs.

To further investigate the scalability and stability of the GABO, we test its performance on large-sized instances, where the number of original jobs equals that of new jobs, i.e., $n_O = n_N \in \{100, 300, 500, 700, 1000\}$. For each problem size, 10 independent instances are randomly

generated using the same data generation scheme described earlier. Each instance is solved independently 10 times, and the reported results are averaged over all runs and instances. In addition to the average objective value (Avg Obj) and average CPU time (Avg Time), we report the average standard deviation (Avg Std Dev) and the relative standard deviation (RSD) of the objective values across repeated runs. Specifically, the standard deviation is first computed for each instance over 10 independent runs, and Avg Std Dev is the average of these standard deviations over the 10 instances. The relative standard deviation is calculated as $RSD = \frac{\text{Avg Std Dev}}{\text{Avg Obj}} \times 100\%$, which is used to evaluate the relative stability of the GABO under different problem sizes. Table 6 presents the computational results on large-sized instances.

Table 6. Performance of the GABO on large-sized instances.

n_O	n_N	Avg Obj	Avg Std Dev	RSD	Avg Time
100	100	5119.18	786.18	15.36%	39.62
300	300	20609.86	1224.00	5.93%	101.22
500	500	38612.99	1862.85	4.82%	116.35
700	700	56523.76	1892.70	3.35%	254.50
1000	1000	82867.50	2343.36	2.83%	313.60

The results in Table 6 show that the GABO can handle large-sized instances efficiently. As expected, the average objective value increases steadily with the problem size, reflecting the increasing complexity of larger instances. As the problem size increases from (100, 100) to (1000, 1000), the average CPU time increases from 39.62 seconds to 313.60 seconds. Considering that the total number of jobs increases from 200 to 2000, the computational time remains within a few minutes even for the largest instances. This indicates good scalability of the proposed algorithm. Regarding solution stability, the RSD shows a clear decreasing trend as the problem size increases. Specifically, the RSD decreases from 15.36% for the (100, 100) instances to 2.83% for the (1000, 1000) instances. This indicates that the variability among independent runs becomes relatively smaller on larger instances. Therefore, the GABO exhibits stable and robust performance on large-sized instances.

Overall, the computational results lead to three conclusions. First, Algorithm 1 is more efficient and reliable than the MILP model for obtaining exact solutions on small-sized instances. Second, the GABO consistently outperforms the SGA in terms of solution quality and reliability, while requiring comparable or lower computational time in most cases. Third, the GABO can solve large-sized instances involving up to 2000 jobs within reasonable computational time and exhibits good relative stability. These results demonstrate the effectiveness of the proposed exact and heuristic algorithms, and further confirm the benefit of incorporating the optimal structural properties of the problem into the genetic algorithm framework.

6. Concluding remarks

This work addresses a single-machine rescheduling problem with due dates, where newly arriving jobs may be either accepted or rejected. The objective is to minimize the maximum lateness or tardiness, while jointly considering the scheduling cost of original and accepted jobs, the disruption penalty, and the total rejection penalty, subject to a bound on the maximum disruption. We develop pseudo-polynomial

time dynamic programming algorithms together with a genetic algorithm to solve the proposed models, and conduct computational experiments to assess their effectiveness. The numerical results confirm the effectiveness of the proposed algorithms. For the $L_{\max}$ -based model, Algorithm 1 is more effective than the MILP model in obtaining exact solutions on small-sized instances. In the largest tested small-sized case $(n_O, n_N) = (20, 20)$, Algorithm 1 solves 9 out of 10 instances to optimality, whereas the MILP model solves none within the time limit. Moreover, the GABO substantially outperforms the SGA: averaged over all 16 small-sized combinations, the average gap is reduced from 122.03% to 7.85%, and the average success rate is improved from 10.97% to 68.17%. The large-scale experiments further show that the GABO can handle instances with up to 2000 jobs within 313.60 seconds on average, with an RSD of only 2.83% for the largest instances. Potential extensions include applying the framework to more complex machine environments, such as flow shops and parallel machines, or exploring alternative forms of disruption.

Author contributions

Xuecong Xu, Shuaijing Yu, and Wenchang Luo designed the methodology; Wenchang Luo and Xuecong Xu wrote the original draft; Xuecong Xu developed the genetic algorithm and conducted the computational experiments; Wenchang Luo supervised the study; and all authors reviewed and edited the manuscript. All authors have read and approved the final version of the manuscript.

Use of Generative-AI tools declaration

During the preparation of this work, the authors used ChatGPT to improve the clarity and readability of the language. All AI-assisted content was carefully reviewed and revised by the authors, who take full responsibility for the final version of the manuscript.

Acknowledgments

The authors acknowledge their gratitude to the editor and the anonymous reviewers for their valuable and constructive suggestions that helped in improving the earlier version of the manuscript.

Conflict of interest

All authors declare no conflicts of interest in this paper.

Funding

The research work of the second author is supported by the Visiting Scholar Teacher Professional Development Project of the Zhejiang Provincial Department of Education (Grant No. FX2023132), and that of the third author by the National Natural Science Foundation of China (Grant No. 12471302) and the Natural Science Foundation of Ningbo (Project ID 2024J195).

Supplementary

Appendix A, containing the MILP formulation used in the computational experiments, is provided as a separate supplementary file.

References

1. J. C. Bean, J. R. Birge, J. Mittenenthal, C. E. Noon, Matchup scheduling with multiple resources, release dates and disruptions, *Oper. Res.*, **39** (1991), 470–483. <https://doi.org/10.1287/opre.39.3.470>
2. M. Zweben, E. Davis, B. Daun, M. J. Deale, Scheduling and rescheduling with iterative repair, *IEEE Trans. Syst. Man Cybern.*, **23** (1993), 1588–1596. <https://doi.org/10.1109/21.257756>
3. J. Clausen, J. Larsen, A. Larsen, J. Hansen, Disruption management–operations research between planning and execution, *OR/MS Today*, **28** (2001), 40–43. Available from: <https://optimization-online.org/?p=9178>.
4. G. Yu, M. Argüello, G. Song, S. M. McCowan, A. White, A new era for crew recovery at Continental Airlines, *Interfaces*, **33** (2003), 5–22. <https://doi.org/10.1287/inte.33.1.5.12720>
5. K. Dahal, K. Al-Arfaj, K. Paudyal, Modelling generator maintenance scheduling costs in deregulated power markets, *Eur. J. Oper. Res.*, **240** (2015), 551–561. <https://doi.org/10.1016/j.ejor.2014.07.008>
6. N. G. Hall, C. N. Potts, Rescheduling for new orders, *Oper. Res.*, **52** (2004), 440–453. <https://doi.org/10.1287/opre.1030.0101>
7. N. G. Hall, C. N. Potts, Rescheduling for job unavailability, *Oper. Res.*, **58** (2010), 746–755. <https://doi.org/10.1287/opre.1090.0751>
8. Z. Liu, Y. K. Ro, Rescheduling for machine disruption to minimize makespan and maximum lateness, *J. Sched.*, **17** (2014), 339–352. <https://doi.org/10.1007/s10951-014-0372-2>
9. G. E. Vieira, J. W. Herrmann, E. Lin, Rescheduling manufacturing systems: A framework of strategies, policies, and methods, *J. Sched.*, **6** (2003), 39–62. <https://doi.org/10.1023/A:1022235519958>
10. D. Ouelhadj, S. Petrovic, A survey of dynamic scheduling in manufacturing systems, *J. Sched.*, **12** (2009), 417–431. <https://doi.org/10.1007/s10951-008-0090-8>
11. I. R. Uhlmann, E. M. Frazzon, Production rescheduling review: Opportunities for industrial integration and practical applications, *J. Manuf. Syst.*, **49** (2018), 186–193. <https://doi.org/10.1016/j.jmsy.2018.10.004>
12. Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, L. Stougie, Multiprocessor scheduling with rejection, *SIAM J. Discrete Math.*, **13** (2000), 64–78. <https://doi.org/10.1137/S0895480196300522>
13. S. Sengupta, Algorithms and approximation schemes for minimum lateness/tardiness scheduling with rejection, in *Algorithms and Data Structures*, F. Dehne, J. R. Sack, M. Smid, Eds., Lecture Notes in Computer Science, vol. 2748, Springer, Berlin, Heidelberg, 2003, 79–90. https://doi.org/10.1007/978-3-540-45078-8_8
14. D. W. Engels, D. R. Karger, S. G. Kolliopoulos, S. Sengupta, R. N. Uma, J. Wein, Techniques for scheduling with rejection, *J. Algorithms*, **49** (2003), 175–191. [https://doi.org/10.1016/S0196-6774\(03\)00078-6](https://doi.org/10.1016/S0196-6774(03)00078-6)

15. J. Ou, X. Zhong, Bicriteria order acceptance and scheduling with consideration of fill rate, *Eur. J. Oper. Res.*, **262** (2017), 904–907. <https://doi.org/10.1016/j.ejor.2017.04.036>
16. D. Shabtay, The single machine serial batch scheduling problem with rejection to minimize total completion time and total rejection cost, *Eur. J. Oper. Res.*, **233** (2014), 64–74. <https://doi.org/10.1016/j.ejor.2013.08.013>
17. F. T. Nobibon, R. Leus, Exact algorithms for a generalization of the order acceptance and scheduling problem in a single-machine environment, *Comput. Oper. Res.*, **38** (2011), 367–378. <https://doi.org/10.1016/j.cor.2010.06.003>
18. S. Thevenin, N. Zufferey, M. Widmer, Metaheuristics for a scheduling problem with rejection and tardiness penalties, *J. Sched.*, **18** (2015), 89–105. <https://doi.org/10.1007/s10951-014-0395-8>
19. S. A. Slotnick, Order acceptance and scheduling: A taxonomy and review, *Eur. J. Oper. Res.*, **212** (2011), 1–11. <https://doi.org/10.1016/j.ejor.2010.09.042>
20. D. Shabtay, N. Gaspar, M. Kaspi, A survey on offline scheduling with rejection, *J. Sched.*, **16** (2013), 3–28. <https://doi.org/10.1007/s10951-012-0303-z>
21. D. J. Wang, F. Liu, Y. Jin, A multi-objective evolutionary algorithm guided by directed search for dynamic scheduling, *Comput. Oper. Res.*, **79** (2017), 279–290. <https://doi.org/10.1016/j.cor.2016.04.024>
22. D. Wang, Y. Yin, T. C. E. Cheng, Parallel-machine rescheduling with job unavailability and rejection, *Omega*, **81** (2018), 246–260. <https://doi.org/10.1016/j.omega.2018.04.008>
23. W. Luo, M. Jin, B. Su, G. Lin, An approximation scheme for rejection-allowed single-machine rescheduling, *Comput. Ind. Eng.*, **146** (2020), 106574. <https://doi.org/10.1016/j.cie.2020.106574>
24. K. Fang, W. Luo, M. L. Pinedo, M. Jin, L. Lu, Rescheduling for new orders on a single machine with rejection, *J. Oper. Res. Soc.*, **75** (2024), 346–360. <https://doi.org/10.1080/01605682.2023.2197000>
25. Y. X. Ren, S. S. Yu, L. S. Tan, W. C. Luo, Rescheduling involving deliver times on a single machine due to the arrival of new jobs with rejection, *J. Oper. Res. Soc. China*, (2025), 1–18. <https://doi.org/10.1007/s40305-025-00607-x>
26. R. L. Graham, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: A survey, *Ann. Discrete Math.*, **5** (1979), 287–326. [https://doi.org/10.1016/S0167-5060\(08\)70356-X](https://doi.org/10.1016/S0167-5060(08)70356-X)
27. M. Srinivas, L. M. Patnaik, Genetic algorithms: A survey, *Computer*, **27** (1994), 17–26. <https://doi.org/10.1109/2.294849>



AIMS Press

©2026 the Author(s), licensee AIMS Press. This is an open access article distributed under the terms of the Creative Commons Attribution License (<https://creativecommons.org/licenses/by/4.0>)