POLITECNICO DI TORINO
Repository ISTITUZIONALE

A reduced variable neighborhood search for the just in time job shop scheduling problem with sequence dependent setup times

Original
A reduced variable neighborhood search for the just in time job shop scheduling problem with sequence dependent setup times / Brandimarte, P., Fadda, E.. - In: COMPUTERS & OPERATIONS RESEARCH. - ISSN 0305-0548. - 167:(2024), pp. 1-12. [10.1016/j.cor.2024.106634]

Availability:
This version is available at: 11583/2987691 since: 2024-04-10T06:54:50Z

Publisher:
Elsevier

Published
DOI:10.1016/j.cor.2024.106634

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

27 November 2026

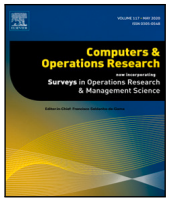
Computers & Operations Research 167 (2024) 106634



Contents lists available at ScienceDirect

Computers and Operations Research

journal homepage: www.elsevier.com/locate/cor



A reduced variable neighborhood search for the just in time job shop scheduling problem with sequence dependent setup times

Paolo Brandimarte, Edoardo Fadda *

Department of Mathematical Sciences 'Giuseppe Luigi Lagrange', DISMA, Politecnico di Torino, Torino, Italy

ARTICLE INFO

Dataset link: https://github.com/EdoF90/just-in-time-jss-setup-times

Keywords:
Applications of mathematical programming
Reduced variable neighborhood search
Just in time
Matheuristic
Dual variables
Less is more

ABSTRACT

In this paper, we deal with the just-in-time job shop scheduling problem with sequence-dependent setup times and release dates. Given a set of jobs characterized by release and due dates, the goal is to execute them by minimizing a weighted sum of their earliness, tardiness, and flow time (i.e., the difference between completion and start time of each job). We develop new destroy and repair operators by exploiting the structure of the problem, and we use them within a reduced variable neighborhood search matheuristic. Computational experiments carried out on several sets of instances show that the proposed algorithm outperforms existing solution methods.

1. Introduction

Job-shop scheduling problems are well-known optimization problems, tackled by a huge amount of literature, mostly focused on the minimization of regular objective functions like makespan. A more limited literature deals with non-regular scheduling objectives (Bürgy and Bülbül, 2018). Moreover, the majority of the papers assume that setup times are negligible or included within the job processing time (Allahverdi et al., 2008). These assumptions are not necessarily met in practice, where sequence-dependent setup times often play a major role, and non-regular objective functions must be considered.

This paper examines such a setting, tackling Just in Time (JIT) scheduling, whose aim is to complete jobs as close to their due date as possible. Therefore, JIT models feature a non-regular objective function, encompassing both earliness and tardiness penalties. Earliness captures the holding or deterioration cost of the finished jobs, while tardiness accounts for the penalty of missing the due dates and the resulting loss of customer goodwill. Nevertheless, considering only earliness and tardiness may lead to schedules in which the first operation of a job is processed as early as possible, and the last one is completed close to the due date, thus inducing waiting times and costs for keeping stock at the shop floor level, which contradicts the JIT philosophy. One example of this effect is shown in Fig. 1, where we depict a solution with zero earliness and tardiness, but with the first two operations of job 1 processed as soon as possible. As a result, there is more work in process on the shop floor than necessary.

To counter this effect, some papers introduce operation due dates by subtracting the sum of the processing times of the subsequent operations from the job due date (Abderrazzak et al., 2022; Ahmadian and Salehipour, 2020; Baptiste et al., 2008). Then, they penalize earliness and tardiness of each single operation of a job in the objective function so that, in the ideal situation, each operation completes at its due date, the job is on time, and there is no waiting time between operations. This cannot be done if sequence-dependent setup times are considered, since the sequence of operations may strongly affect the single operation due dates. To overcome this problem, we introduce in the objective function the flow time, i.e., the amount of time elapsing between the start time of the first operation and the completion time of the last operation of each job. Note that the flow time is measured with respect to the actual job start time, not its release date. In Fig. 1, the flow time of job 2 is minimal, while the flow time of job 1 is not.

Using the three-field notation $\alpha|\beta|\gamma$ of Graham et al. (1979) (where: α describes the shop (machine) environment; β collects miscellaneous information about setups, release dates, additional constraints, and details of the processing characteristics; γ defines the objective function), the problem that we address can be denoted as $J|r_j, s_{ik}|\sum_j \left[w_j^E E_j + w_j^T T_j + w_j^F (C^j - S^j)\right]$. In the following, we refer to this problem as *Just In Time Job Shop Scheduling with Sequence-dependent Set-up Times and release dates* (or $JIT - JSS - SeqST$, for short). To the best of our knowledge, while each individual feature of this problem, as well as some partial combinations, has been considered in the literature, their

* Corresponding author.
E-mail address: edoardo.fadda@polito.it (E. Fadda).

https://doi.org/10.1016/j.cor.2024.106634
Received 16 June 2023; Received in revised form 10 December 2023; Accepted 27 March 2024
Available online 4 April 2024
0305-0548/© 2024 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

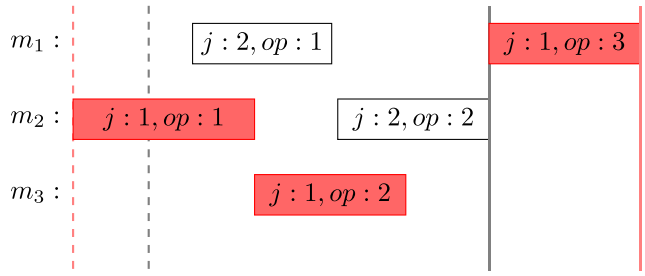


Fig. 1. Example of a schedule minimizing the sum of earliness and tardiness, but not the flow time. The dashed vertical lines correspond to release dates, whereas the continuous ones correspond to due dates (red for job 1, grey for job 2; j stands for jobs and op for operations). We could delay the start times of the first two operations of job 1, reducing the flow time, without affecting earliness and tardiness.

full combination has not been tackled. Hence, the main contribution of this paper is to fill this gap.

As in, e.g., Ahmadian and Salehipour (2020), in order to find good solutions within a reasonable amount of time, we decompose the problem into sequencing (i.e., deciding the order of the operations of all jobs to be processed on each machine) and timing (i.e., deciding the starting time of each operation) subproblems since, if a feasible sequence is provided, an optimal schedule for the given sequence can be obtained in polynomial time by solving a linear programming model called *timing problem* (Pinedo, 2008, p. 74). Then, we develop a *Reduced Variable Neighborhood Search* (RVNS) to guide the search in the space of sequences (Mladenović et al., 2016; Brimberg et al., 2023). Here, our contribution is the definition of different neighborhoods using destroy and repair operators, and the comparison of their performances. In particular, we analyze the performance of two families of destroy operators, one based on random selection, and one leveraging information from the dual variables of the timing problem. We also compare different repair operators based either on the possibly approximated solution restricted mixed-integer linear problems (MILPs), or the simulation of priority rules.

1.1. Paper contributions and limitations

The contribution of this paper is twofold. First, we consider a JIT scheduling problem encompassing sequence-dependent setup times. To the best of the authors' knowledge, both JIT setting and sequence-dependent setup times have been tackled individually, but there is no paper dealing with both features together, also including non-zero release dates. For the sake of simplicity, and in order to get better insights, we consider these features within a job shop setting, where each operation is assigned to one machine. Hence, a limitation of this paper is that we do not consider flexibility.

Second, we propose new neighborhood structures leveraging the information provided by the dual variables of the timing problem. This allows to exploit information from the timing subproblem to guide the local search in the space of sequences.

1.2. Paper organization

The paper is organized as follows. Section 2 presents a review of the literature concerning just-in-time job shop scheduling problems. Section 3 introduces the mathematical model of the problem, and Section 4 presents the proposed heuristics along with the destroy and repair operators. Section 5 reports computational experiments on a set of instances ranging from 5 to 50 jobs and 10 machines. This section is divided into two parts: the first one comparing the relative performance of pairs of destroy/repair operators, and the second one proving the effectiveness of the overall proposed algorithm. Finally, Section 6 concludes the paper and outlines future research directions.

2. Literature review

The JIT job shop scheduling models have their roots in the earliness-tardiness (ET) models that were introduced in the 1980s by Fox and Smith (1984). While this latter class of models minimizes a weighted sum of earliness and tardiness, JIT models also account for the time that jobs spend on the shop floor during their execution. How the model deals with this quantity generates two different branches of the literature, one addressing due dates at the operation level and one addressing due dates at the job level (Ahmadian and Salehipour, 2020). As mentioned above, if sequence-dependent setup times are considered, due dates at the operation level are not immediate to calculate. Therefore, we focus on the second branch of the literature.

While JIT job-shop scheduling without sequence-dependent setup times has been considered in several papers (Baptiste et al., 2008; dos Santos et al., 2010; Wang and Li, 2014; Ahmadian and Salehipour, 2020), no one deals with the JIT job-shop scheduling with sequence-dependent setup times. Therefore, in order to account for the operations waiting times, we add a term in the objective function accounting for the holding cost and the inconvenience related with the work in process. This has been done in multiple ways in other works. Leonardi and Raz (1997) include in the objective function the flow time, computed as the time between the job release date and the completion time, thus accounting for the holding cost before the start of the first operation. Instead, Brandimarte and Maiocco (1999) consider the time elapsed between the start time of the first operation and the delivery time (i.e., the maximum between the due date and the job completion time), thus accounting for the cost of holding the completed job. In this paper, we assume that the incurred holding cost is proportional to the difference between the starting time of the first operation of a job and the completion time of the last one, thus accounting for the work in process inconvenience at the shop floor level during all of the operations.

To the best of authors' knowledge, as we mentioned, no paper deals with the full set of features of the $JIT - JSS - SeqST$, and the literature dealing with JIT problems with sequence-dependent setup times is rather scarce (Allahverdi et al., 1999, 2008). Kolahan and Liang (1998) tackle a single machine JIT scheduling problem with sequence-dependent setup times, in which there are linear costs for compression or extension of job processing times. The objective function consists of a linear combination of the total weighted earliness, tardiness, and compression/extension costs. They propose tabu search as a solution method. The same solution method is adopted by Santos and França (1997), who consider a single machine problem and minimize the sum of earliness, tardiness, and setup times.

In the parallel machine setting, no JIT model with sequence-dependent setup times has been considered. The papers closest to our setting are Radhakrishnan and Ventura (2000), Feng and Lau (2007) in the identical parallel machines setting, and Zhu and Heady (2000) in the unrelated parallel machines setting. While the first paper proposes simulated annealing, the second one decomposes the problem into sequencing and timing subproblems and proposes a custom metaheuristic for dealing with sequencing. Instead, in the third paper, the authors propose a MILP model that provides optimal solutions in a reasonable time for instances up to 9 jobs and 3 machines.

In the flow shop setting a few papers deal with JIT objective functions (Shabtay, 2012; Xiong et al., 2021), but none of them consider setup times. Finally, in the other settings (e.g., job shop, flexible job shop, etc.), there seems to be no work dealing with JIT or ET models featuring sequence-dependent setup times.

Several solution methods have been applied to compute solutions for JIT and ET models. Since a comprehensive analysis is beyond the scope of the paper, we only report applications of Variable Neighborhood Search (VNS) to those problems.

In the ET context, VNS has been used in Liao and Cheng (2007), Thevenin and Zufferey (2019). In the first paper, the authors address

a single-machine weighted earliness and tardiness with a common due date for all the jobs. They hybridize VNS with tabu search and consider neighborhoods based on insertion and swap moves. In the second paper, the authors tackle a single-machine scheduling problem with rejections, sequence-dependent setup times, and earliness and tardiness penalties. They enhance VNS with a learning mechanism that helps to drive the search towards promising areas of the search space. The neighborhoods are based on pairwise swap moves.

In the JIT context, VNS has been applied to the JIT job shop scheduling without setup times in Wang and Li (2014), Ahmadian and Salehipour (2020). In both papers, the authors decompose the problem into sequencing and timing, and use a VNS to explore the sequences. Nevertheless, in the first paper, the neighborhoods used are based on swap and insertion. Instead, in the second one the neighborhoods are based on MILP problems and swap operations.

3. Mathematical model

Let $\mathcal{J} = \{1, \dots, J\}$ be the set of jobs to process on a set of machines $\mathcal{M} = \{1, \dots, M\}$. We make standard assumptions: each machine is continuously available and can process at most one job at a time, and no job preemption is allowed. Each job $j \in \mathcal{J}$ is characterized by a due date d_j, a release date r_j, and by a ordered chain of operations $\mathcal{I}_j = \{1, 2, \dots I_j\}$. Operation $i \in \mathcal{I}_j$ of job j must be processed on a given machine, denoted by $M(i, j)$, with processing time p_i^j. We call $\mathcal{J}_m$ the set of jobs having one operation to be executed on machine m, its cardinality is J_m. In other words, J_m is the number of operations that machine m has to perform. For the sake of simplicity, we rule out reentrant flows, i.e., all of the machines to be visited by a job are distinct. This assumption streamlines notation, but does not really affect the solution approach and the computational experiments. Moreover, a job need not visit all of the machines. For each machine m and job $j \in \mathcal{J}_m$, we denote by $I(j, m)$ the operation of job j to be executed on machine m.

On each machine, indexed by m, it is possible that, after the operation of job j_1 and before the operation of job j_2, a sequence-dependent setup is required, whose duration is denoted by $\delta^m_{j_1 j_2}$. Initially, each machine is configured to perform the operation of a fictitious job j_0^m.

Finally, we define w_j^E, w_j^T, and w_j^F to be weights reflecting the relative importance of the earliness, tardiness, and flow time for each job j, respectively.

The decision variables of the model are:

- C_i^j completion time of operation i of job j;
- S_i^j starting time of operation i of job j;
- E_j earliness of job j: $\max\left(d_j - C^j_{I_j}, 0\right)$;
- T_j tardiness of job j: $\max\left(C^j_{I_j} - d_j, 0\right)$;
- $y^m_{j_1 j_2}$ binary variable set to 1 if operation of job j_1 immediately precedes operation of job j_2 on machine m.

Using these definitions, the flow time of job j can be expressed as $C^j_{I_j} - S^j_1$, where $C^j_{I_j}$ is the completion time of the last operation of job j, and S^j_1 is the starting time of the first operation of job j. The mathematical model can be formulated as the following MILP problem:

$$\min \quad \sum_{j\in\mathcal{J}} \left(w_j^E E_j + w_j^T T_j + w_j^F (C^j_{I_j} - S^j_1) \right) \tag{1}$$

$$\text{s.t.} \quad C^j_{I_j} + E_j - T_j = d_j \qquad \forall j \in \mathcal{J} \tag{2}$$

$$S^j_{I(j_2,m)} \geq C^{j_1}_{I(j_1,m)} + \delta^m_{j_1 j_2} - M(1 - y^m_{j_1 j_2}) \quad \forall m \in \mathcal{M}\ \forall j_1, j_2 \in \mathcal{J}_m \tag{3}$$

$$S^j_{i+1} \geq C^j_i \qquad \forall i = 1, \dots, I_j - 1\ \forall j \in \mathcal{J} \tag{4}$$

$$S^j_1 \geq \max\left[r_j, \delta^{M(1,j)}_{j_0^{M(1,j)} j}\right] \qquad \forall j \in \mathcal{J} \tag{5}$$

$$C^j_i = S^j_i + p^j_i \qquad \forall i \in I_j\ \forall j \in \mathcal{J} \tag{6}$$

$$\sum_{j_2\in\mathcal{J}} \sum_{j_1\in\mathcal{J}\setminus\{j\}} y^m_{j_1 j_2} = J_m - 1 \qquad \forall m \in \mathcal{M} \tag{7}$$

$$\sum_{j_1\in\mathcal{J}_m} y^m_{j_1 j_2} \leq 1 \qquad \forall m \in \mathcal{M}\ \forall j_2 \in \mathcal{J}_m \tag{8}$$

$$\sum_{j_2\in\mathcal{J}_m} y^m_{j_1 j_2} \leq 1 \qquad \forall m \in \mathcal{M}\ \forall j_1 \in \mathcal{J}_m \tag{9}$$

$$y^m_{j' j} \in \{0, 1\} \qquad \forall j, j' \in \mathcal{J},\ \forall m \in \mathcal{M} \tag{10}$$

$$S^j_i, C^j_i, E_j, T_j \in \mathbb{R}^+ \qquad \forall j \in \mathcal{J} \tag{11}$$

The objective function (1) to be minimized is the weighted sum of earliness, tardiness, and flow time of each job. Constraints (2) define earliness and tardiness for each job. Constraints (3) are the disjunctive constraints forcing the starting time of one operation to be later than the completion time of the previous operation on the same machine, plus the possible setup time. Constraints (4) force the starting time of an operation of a job to be after the completion time of the previous one of the same job. Constraints (5) enforce the starting time of the first operation of each job to be after its release date and after the end of the initial setup. Constraints (6) require that the completion time of each operation must be equal to the sum of its starting time and processing time. Constraints (7) state that exactly $J_m - 1$ of the y^m_{ij} variables must be set to one for each machine m, and constraints (8) and (9) force each operation to have at most one immediate predecessor and one immediate successor. These latter constraints are asymmetric traveling salesman problem-like constraints: For each machine, there is a set of J_m operations to be executed, and we must connect their nodes with $J_m - 1$ arcs. Constraints related to sequencing and timing variables make sure that there is no subtour.

Following a common approach in the literature, we decompose the overall problem into sequencing and timing subproblems; see, e.g., Brandimarte and Maiocco (1999), Pinedo (2008). We represent the sequence of operations S as a *sequence array* including M sequences, each one describing the sequence of jobs on a single machine. We recall that we do not assume that a job must visit all of the machines; hence, S may be a "ragged" array, rather than a proper matrix. Formally, we use

$$S = \left[\begin{array}{c|ccc} m_1: & j^1_{(1)} & \dots & j^1_{(J_1)} \\ \hline \dots & \dots & \dots & \dots \\ \hline m_M: & j^M_{(1)} & \dots & j^M_{(J_M)} \end{array}\right], \tag{12}$$

where $j^m_{(n)}$ is the nth job executed on machine m. Computing the timing for a sequence S is equivalent to setting $y^m_{j_1 j_2} = 1$ if j_1 is processed immediately before j_2 on machine m, 0 otherwise, in model (1)–(11). By fixing these variables and removing the redundant constraints (i.e., all the constraints (3) for which $y^m_{j_1 j_2} = 0$, and constraints (7)–(9)), model (1)–(11) becomes:

$$\min \sum_{j\in\mathcal{J}} \left(w_j^E E_j + w_j^T T_j + w_j^F (C^j_{I_j} - S^j_1) \right) \tag{13}$$

$$\text{s.t.} \quad C^j_{I_j} + E_j - T_j = d_j \qquad \forall j \in \mathcal{J} \tag{14}$$

$$S^j_{I(j^m_{(n)},m)} \geq C^{j_1}_{I(j^m_{(n-1)},m)} + \delta^m_{j^m_{(n-1)} j^m_{(n)}} \quad \forall n = 2, \dots, J_m\ \forall m \in \mathcal{M} \tag{15}$$

$$S^j_{i+1} \geq C^j_i \qquad \forall i = 1, \dots, I_j - 1\ \forall j \in \mathcal{J} \tag{16}$$

$$S^j_1 \geq \max\left[r_j, \delta^{M(1,j)}_{j_0^{M(1,j)} j^{M(1,j)}_{(1)}}\right] \qquad \forall j \in \mathcal{J} \tag{17}$$

$$C^j_i = S^j_i + p^j_i \qquad \forall i = 1, \dots, I_j \qquad \forall j \in \mathcal{J} \tag{18}$$

$$S^j_i, C^j_i, E_j, T_j \in \mathbb{R}^+ \qquad \forall j \in \mathcal{J} \tag{19}$$

Model (13)–(19) is the so-called *timing problem*. Given a sequence array S, the solution of this timing model yields the optimal time at which the operations must start and end in order to minimize the objective function. Model (13)–(19) is a continuous LP problem, thus it can be solved in polynomial time by interior point methods.

Algorithm 1 Reduced VNS

```
while t < t_max do
    generate S randomly
    k ← 1
    while k ≤ K do
        generate S′ from N_k(S)
        if S′ is better than S then
            S ← S′
            k ← 1
        else
            k ← k + 1
        end if
        t ← CPU time
    end while
end while
```

4. Reduced variable neighborhood search

VNS is a metaheuristic first proposed in Mladenović and Hansen (1997), whose basic idea is to apply a systematic change of neighborhoods when carrying out a *descent phase* to find a local optimum, and a *perturbation phase* to escape from the corresponding valley. We use VNS to guide the search in the space of sequences. Therefore, we interpret sequences as the underlying solutions to the problem, whereas the timing subproblem maps such sequences into the value of objective function. Within this framework, for a given sequence S, we denote a finite set of pre-selected neighborhood structures by $\mathcal{N}_1(S), \ldots, \mathcal{N}_K(S)$.

In the standard VNS, the first step is to generate a random initial incumbent solution S. Then, a candidate solution S' is computed by applying the neighborhood structure $\mathcal{N}_k(S)$, after setting the counter $k = 1$. If the candidate solution is better than the incumbent solution, this is replaced ($S \leftarrow S'$), and the search proceeds with $k \leftarrow 1$; otherwise, we consider the next neighborhood ($k \leftarrow k + 1$). These steps are repeated until all of the different neighborhoods have been explored, i.e., $k = K$.

If the local search in $\mathcal{N}_k(S)$ is expensive, an alternative is to sample solutions from $\mathcal{N}_k(S)$, which leads to the RVNS. Moreover, in contrast to standard VNS, RVNS iterates the standard VNS steps until a maximum CPU time $t_{\max}$ is reached (Hansen and Mladenović, 2018). We report the pseudo-code of RVNS in Algorithm 1.

The steps of Algorithm 1 that require customization are the generation of the initial sequence S, the definition of the neighborhoods, and the generation of S' from $\mathcal{N}_k(S)$.

We generate the initial sequence by using an EDD priority rule in the first iteration. Then, from the second iteration on, we generate S by solving model (1)–(11) with a perturbed objective function (we add a uniform ±20% noise to w_j^E, w_j^T, and w_j^F), setting a time limit, and fixing as starting solution the best solution found (not necessarily optimal). In this way, we are sure to find a solution which, in the worst case, is the initial one (i.e., the best solution found).

To define the set of neighborhoods $\mathcal{N}_k(S)$, we first need to define a set $\mathcal{D}$ of D destroy operators and a set $\mathcal{R}$ of R repair operators. We define stochastic destroy operators embedding choices that depend on a random outcome ω, while the repair operators are deterministic. Therefore, we define $R \times D$ neighborhoods, each one containing all the solutions that can be obtained by applying a pair of destroy and repair operators. In formulas, setting $k = (i, j)$, we generate the set of candidates S' by applying the repair and destroy operators as follows:

$$\mathcal{N}_{(i,j)}(S) = \left\{ S' \;\middle|\; S' = \mathsf{O}_i^r\left(\mathsf{O}_j^d(S, \omega)\right) \right\}, \tag{20}$$

where $\mathsf{O}_j^d(\cdot, \omega)$ is the jth destroy operator, depending on the outcome ω, and $\mathsf{O}_i^r(\cdot)$ is the ith repair operator.

4.1. Solution representation

We use the solution representation shown in Eq. (12), even if a widely used alternative is the operation-based encoding proposed in Gen (1994). For a general problem, this representation gives a sequence of $\sum_{m \in M} J_m$ elements in which each job j appears a number of times equal to the number of machines such that $j \in \mathcal{J}_m$. Following the example in Ahmadian and Salehipour (2020), in the sequence $(1, 1, 2, 3, 2, 4, 3, 1, 3, 2, 4, 4)$ the first 1 (2 or 3) represents the first operation of job 1 (2 or 3) and so on. This representation can be read as

$$\begin{array}{r} \text{job}: \\ \text{machine}: \end{array} \left[\begin{array}{ccccc} 1 & 1 & 2 & 3 & \ldots \\ \hline 0 & 1 & 2 & 1 & \ldots \end{array}\right], \tag{21}$$

where the first row denotes the operations of the jobs and the second row represents the machines that perform those operations. Despite this representation being effective in several settings [e.g., in the heuristic proposed in Ahmadian et al. (2021) or for crossover in a genetic algorithm], we do not consider it, since different permutations may lead to the same solution, thus decreasing its effectiveness. For example,

$$\left[\begin{array}{ccccc} 1 & 1 & \mathbf{2} & 3 & \ldots \\ \hline 1 & 2 & \mathbf{3} & 2 & \ldots \end{array}\right], \left[\begin{array}{ccccc} 1 & \mathbf{2} & 1 & 3 & \ldots \\ \hline 1 & \mathbf{3} & 2 & 2 & \ldots \end{array}\right] \tag{22}$$

are equivalent since they both lead to the same sequence array:

$$S = \left[\begin{array}{c|ccc} m_1: & 1 & \ldots & \ldots \\ \hline m_2: & 1 & \ldots & \ldots \\ \hline m_3: & 2 & 3 & \ldots \\ \hline \ldots & \ldots & \ldots & \ldots \end{array}\right]. \tag{23}$$

4.2. Operators

We consider two families of destroy operators, one that destroys part of the solutions randomly or based on the solution features, and one that destroys part of the solution by leveraging information obtained from the timing problem. All of the operators depend on the parameter OP_TO_REMOVE, which specifies the number of operations to be removed, which allows to control the degree of disruption and thus the effort of the repair.

The destroy operators of the first family are:

- Remove random jobs. Randomly select $\lceil \frac{\max_m J_m}{\texttt{OP_TO_REMOVE}} \rceil$ jobs and remove all of their operations.
- Remove random operations. Randomly remove OP_TO_REMOVE operations.
- Remove random machine. Randomly select $\lceil \frac{\texttt{OP_TO_REMOVE}}{J} \rceil$ machines and remove all of the operations from the selected machines.
- Remove worst jobs. Sort the jobs in decreasing order with respect to $\left[w_j^E E_j + w_j^T T_j + w_j^F (C_{I_j}^j - S_1^j)\right]$, and remove all of the operations of the first $\lceil \frac{\max_m J_m}{\texttt{OP_TO_REMOVE}} \rceil$ jobs.
- Remove random slice. Randomly select a position between 1 and $\left(\min_m J_m\right) - \lceil \frac{\texttt{OP_TO_REMOVE}}{M} \rceil$, assuming that OP_TO_REMOVE is not too large, so that this quantity is not negative, and remove all of the operations in the selected column and in the subsequent $\lceil \frac{\texttt{OP_TO_REMOVE}}{M} \rceil$ ones. See Fig. 2(a) for a graphical representation.
- Remove random rectangle. Pick a rectangular region characterized by OP_TO_REMOVE operations and remove them. See Fig. 2(b) for a graphical representation.
- Remove setup. Sort all of the setup times in decreasing order. Then, starting from the greatest one, remove the operations before and after the selected setup until OP_TO_REMOVE operations are removed.

An alternative to randomly removing operations is to decide which operations to remove on the basis of information obtained from the timing subproblem. We use the values of the dual variables $\lambda_{j_1 j_2}^m$

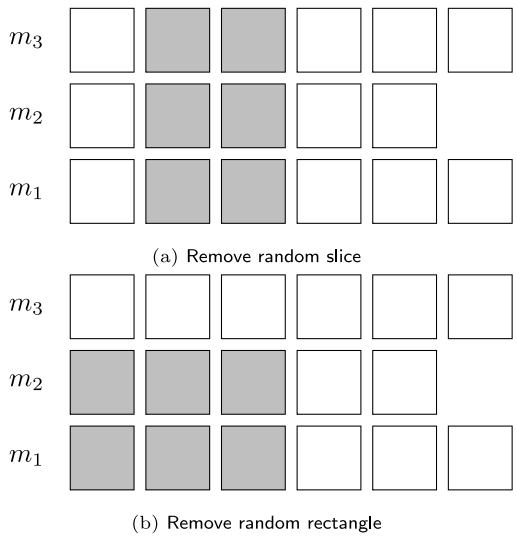


Fig. 2. Example of operations removed (in gray) by different destroy operators.

associated with each precedence constraint (Eq. (15)). The greater the dual variable, the more the constraint affects the optimal solution. In other words, given $j_1, j_2 \in \mathcal{J}_m$, if a constraint $S^j_{I(j_2,m)} \geq C^{j_1}_{I(j_1,m)} + \delta^m_{j_1 j_2}$ is characterized by a high $\lambda^m_{j_1 j_2}$, anticipating $I(j_1, m)$ may result in a better solution. On the other hand, if $\lambda^m_{j_1 j_2} = 0$, we expect that anticipating $I(j_1, m)$ will not produce any improvement in the objective function. In fact, most of the time, when $\lambda^m_{j_1 j_2} = 0$ the optimal solution presents a slack, i.e., there is some idle time between the completion time of $I(j_1, m)$ and the starting time of $I(j_2, m)$. Using the dual variables, we define the following destroy operators:

- Remove dual job. Randomly select, $\lceil \frac{\max_m J_m}{\texttt{OP_TO_REMOVE}} \rceil$ different jobs, where job j is selected with probability
$$\frac{\sum_{j_2} \sum_m \lambda^m_{j,j_2}}{\sum_j \left(\sum_{j_2} \sum_m \lambda^m_{j,j_2} \right)}.$$
Then, remove all the operations of these jobs.
- Remove dual operations. Remove OP_TO_REMOVE different operations sampling from the set of operations and assign to the operation of job j executed on machine m before the operation of job j_2 a probability equal to
$$\frac{\lambda^m_{j,j_2}}{\sum_m \sum_j \sum_{j_2} \lambda^m_{j,j_2}}.$$
- Remove dual machine. Randomly select $\lceil \frac{\texttt{OP_TO_REMOVE}}{J} \rceil$ different machines, where machine m is selected with probability
$$\frac{\sum_{j_1} \sum_{j_2} \lambda^m_{j_1 j_2}}{\sum_m \left(\sum_{j_1} \sum_{j_2} \lambda^m_{j_1 j_2} \right)}.$$

By applying a destroy operator to a sequence array S, we get a *partial sequence array*, i.e., a sequence array in which some operations have been removed. We repair it by using three possible operators: one based on the usage of priority rules, and two based on mathematical models.

Given a partial solution, the first repair operator applies priority rules to repair a solution, by fixing the assignment and the sequence of the partial solution. The rules considered are Earliest Due Date (EDD), Longest Processing Time (LPT), Shortest Processing Time (SPT), Weighted Shortest Processing Time (WSPT), Apparent Tardiness Cost with Setup (ATCS), Minimum Slack First (MSF). Hence, this is actually a family of six operators, whose application requires a sort of discrete-event simulation procedure whose pseudo-code is outlined in Algorithm 2. Given a partial sequence array S, we start by defining the set of operations that are ready to be processed at the current simulated time (as in the standard list scheduling algorithm, Pinedo (2008)). Then, until this set becomes empty, we pick the first free machine that is able to process at least one ready operation. If it already has an assigned operation (i.e., S has an operation in the position that we have to fill), we leave it. Otherwise, we assign the best operation that is ready to be processed but not yet assigned (i.e., it is not in S) according to the given priority rule. Finally, we update the set of operations ready to be processed and the time in which each machine becomes free.

Algorithm 2 Priority Rule Based Repair Operators

```
S ← partial sequence
end_times ← [0] * M
last_pos ← [0] * M
ready_ops ← {first operation of each job}
while ready_ops ≠ ∅ do
    M_a ← {m = M(i, j), i ∈ ready_ops}
    m ← argmin_{M_a}(end_times)                ▷ solve ties arbitrarily
    if S[m][last_pos[m]] is empty then
        I_a ← ready_ops \ operations in S
        j ← best(I_a|priority_rule).
    else
        j ← S[m][last_pos[m]]
    end if
    S[m][last_pos[m]] ← j
    last_pos[m]+ = 1
    Update ready_ops.
    Update end_times.
end while
```

To better explain the application of these operators, we show a simple example with the EDD priority rule. Let us consider 3 jobs j_1, j_2, and j_3 with the following characteristics:

		d_j	r_j
j_1 :	$m_1 \to m_2$	2	0
j_2 :	m_1	3	0
j_3 :	$m_1 \to m_2 \to m_1$	1	0

(24)

and with all the processing times equal to 1. Starting from the sequence array

m_1 :	j_2	j_1	j_3
m_2 :	j_1	j_3	

(25)

we apply some destroy operator and we get:

m_1 :	□	□	j_3
m_2 :	□	j_3	

(26)

Since all of the machines are free at time 0, we pick m_1, which starts processing the first operation of j_1, since this job has a smaller due date than j_2, and j_3 is already in the sequence. Since machine m_1 will complete the selected operation and be free at time 1, we now consider machine m_2. This machine could process j_3, this job is already assigned to a later position in the sequence array. Moreover, no other operations can be processed by m_2. Hence, we move to time 1, when m_1 completes the first operation of job j_1 and starts processing the first operation of job j_2, whereas m_2 starts the second operation of job j_1. Therefore, we obtain the following final sequence array

$$S = \left[\begin{array}{l|lll} m_1 : & j_1 & j_2 & j_3 \\ \hline m_2 : & j_1 & j_3 & \end{array} \right]. \tag{27}$$

The second repair operator solves model (1)–(11) by setting to 1 all the variables $y^m_{j_1 j_2}$ such that the operation of job j_1 is executed immediately before the operation of job j_2 on machine m in the partial sequence array. This results in a model of smaller size, which can be solved in a reasonable amount of time for small values of OP_TO_REMOVE. In the following, we call this repair operator Exact.

Fixing the variable $y^m_{j_1 j_2}$ to 1 implies that job j_1 must immediately precede job j_2 on machine m. This may result in a too-restrictive model. Therefore, we define an alternative model by enforcing looser precedence relations by adding to the MILP model (1)–(11) the binary decision variables $v^m_{j_1 j_2}$, set to 1 if job j_1 is executed before (but not necessarily immediately before) job j_2 on machine m. Moreover, we add the following constraints:

$$y^m_{j_1 j_2} \leq v^m_{j_1 j_2} \qquad \forall j_1, j_2 \in \mathcal{J}, \forall m \in \mathcal{M} \tag{28}$$

$$v^m_{j_1 j_2} + v^m_{j_2 j_1} = 1 \qquad \forall j_1, j_2 \in \mathcal{J}, j_1 \neq j_2, \forall m \in \mathcal{M}. \tag{29}$$

Constraint (28) links variables $v^m_{j_1 j_2}$ and $y^m_{j_1 j_2}$, and constraint (29) requires that job j_1 is executed either before or after job j_2. Then, we fix the variables $y^m_{j_1 j_2}$ if the partial solution has all the J_m operations in machine m and we set the variables $v^m_{j_1 j_2}$ according to the given partial solution, otherwise. This results in a model of smaller dimensions that can be solved in a reasonable amount of time for small values of `OP_TO_REMOVE`. It is important to notice that, due to the structure of constraints (28) and (29), the variables $v^m_{j_1 j_2}$ can be assumed to be continuous instead of binary. In the following, we call this repair operator Exact loose.

When the last two repair operators are applied, it is advisable to impose a time limit to the exact solver. In fact, it usually takes a small amount of time to reach a good solution, but a lot of time to close the gap, and spending plenty of time to obtain small improvements can be deleterious, especially in the first phases of the heuristic. Moreover, in order to speed up the computation of both operators, we fix as initial solution the one we had before the application of the destroy operator. Therefore, in the worst possible case, the operators will return the initial solution, but no model infeasibility issue has to be managed.

In conclusion, it is important to point out that some of the destroy and repair operators have already been considered in other works. For example, the pair consisting of Remove random machine and Exact is the relax-1 heuristic of Ahmadian and Salehipour (2020). However, the destroy operators leveraging information from the dual timing problem and Exact loose are contributions of this paper.

5. Computational experiments

In this section, we evaluate the performance of the proposed solution methods. Since the $JIT - JSS - SeqST$ has never been treated before, we generate a set of benchmark instances[1] by extending those available for JIT job shop scheduling in Baptiste et al. (2008). We consider instances with 10 machines, and with 5, 10, 20, and 50 jobs.

The tests have been performed on an *Intel(R) Core(TM)* i7-5500U CPU@2.40 GHz computer with 16 GB of RAM, running *Ubuntu* v22.04. Gurobi v9.5.2 is used to solve the timing models, to obtain exact benchmarks when feasible, and for the Exact and Exact loose operators. Instead, CP-SAT (the Google constraint programming solver) is employed to build the initial solution for the RVNS, since it is able to better deal with large instances.

Due to the different sizes of the instances, we set different computational time limits. In particular, we set:

- 3 s for instances with 5 jobs,
- 10 min for instances with 10 jobs,
- 15 min for instances with 20 jobs,
- 30 min for instances with 50 jobs.

[1] Problem instances are available at https://github.com/EdoF90/just-in-time-jss-setup-times.

5.1. Problem instance generation

Following the generation procedure in Baptiste et al. (2008), we denote each instance by `I-<J>x<M>-<DD>-<W>-<ID>`, where J is the number of jobs, M is the number of machines, and `DD`, `W` and `ID` are characteristics of the instance explained below.

For each machine m, the set $\mathcal{J}_m$ of jobs visiting it is generated by including each job with probability 0.9. Then, for each job, the sequence of machines that job j has to visit is generated by randomly shuffling the list of all the machines m such that $j \in \mathcal{J}_m$. Release dates are sampled uniformly in the interval $[0, 100]$, processing times in the interval $[10, 30]$, and due dates are set equal to the release date plus π times the sum of the processing times. The constant π is equal to 1.1, 1.3, 1.5, and 1.7 for the instances with `DD = extra_tight`, `DD = tight`, `DD = loose`, `DD = extra_loose`, respectively.

As to earliness and tardiness weights, we consider two possible settings, respectively denoted as `equal` and `tard`. In setting `W = equal`, both weights are sampled from a uniform distribution on the interval $[0.1, 1]$. The second generation scheme (setting `W = tard`) corresponds to a situation where tardiness cost dominates earliness cost: w^E_j is sampled from a uniform distribution on the interval $[0.1, 0.3]$, whereas w^T_j is uniformly distributed on the interval $[0.3, 1]$. Flow time weights w^F_j are set to $0.1 w^E_j$, since we expect work-in-process costs to be smaller than both earliness and tardiness penalties.

Finally, setup times are randomly selected from the set of values $\{0, 5, 10, 30\}$, with corresponding probabilities $\{0.5, 0.3, 0.15, 0.05\}$. This ensures that setup times are comparable with the processing times in 20% of the cases.

Five instances (`ID` = 0, 1, 2, 3, 4) have been randomly generated for each combination of parameters, leading to a total of 4×1×4 ×2×5 = 160 problem instances.

5.2. Operator performance analysis

In this section, we perform an analysis of the performance of the pairs of destroy and repair operators described in Section 4.2. Given a pair of operators $\mathsf{O}^d_j \in \mathcal{D}$, $\mathsf{O}^r_i \in \mathcal{R}$, we measure its suitability through a sample average defined as:

$$\rho_{ij} = \sum_{n=1}^{N} \frac{1}{N} \sum_{l=1}^{L} \frac{1}{L} \left[\frac{f(S_n) - f(\mathsf{O}^r_i(\mathsf{O}^d_j(S_n, \omega_l)))}{f(S_n)} \right], \tag{30}$$

where N is the number of different sequence arrays S_n considered, L the number of times that we run the destroy operator, ω_l is the corresponding outcome, and $f(S_n)$ is the optimal value of the timing problem computed for a fixed sequence array S_n. Notice that ρ_{ij} is positive if, on average, $\mathsf{O}^r_i(\mathsf{O}^d_j(S, \omega))$ is better than S.

Unfortunately, a comprehensive analysis is impractical, since it requires considering how ρ_{ij} varies with respect to the type and dimension of the instance, the characteristics of the sequences S_n, etc. Therefore, we focus on estimating ρ_{ij} on a set of heterogeneous instances with $I = 10$, and $M = 10$. Moreover, ρ_{ij} is strongly affected by the quality of the sequences S_n. If they are randomly generated, ρ_{ij} is large for all pairs of destroy and repair operators. Instead, if they are close to the optimal sequence, ρ_{ij} is close to zero for all pairs. Therefore, we generate sequences S_n by running for 3 s an exact solver on model (1)–(11) with a perturbed objective function (we add a random perturbation ±50% to all of the parameters), starting from an initial random solution. This procedure enables us to produce N sequences of average quality. For each of these sequences, we apply each pair of destroy and repair operators L times (to account for their intrinsic variability). The ρ_{ij} computed for $N, L = 10$, and their standard deviation are shown in Table 1. For all of the destroy operators we set `OP_TO_REMOVE` = 10. Moreover, the time limit of both Exact and Exact loose operators is 10 s, and we set S_n as initial solution. Therefore the worst possible solution returned by these operators is the starting one.

Table 1
ρ_{ij} computed with $N, L = 10$ and for each pair of operators. The standard deviation is reported within brackets.

	Exact	Exact loose	ATCS	EDD	LPT	SPT	MSF	WSPT
rnd_jobs	2.47(3.49)	18.24(8.79)	−41.79(31.05)	−34.51(21.25)	−39.01(35.12)	−32.50(35.42)	−31.66(36.60)	−36.12(26.39)
rnd_ops	0.30(0.86)	13.27(15.27)	−75.06(47.44)	−49.91(32.52)	−62.10(51.99)	−50.28(45.58)	−56.76(32.72)	−62.76(48.45)
worst_jobs	3.68(4.02)	29.81(12.31)	−43.46(47.55)	−29.38(32.13)	−43.42(45.37)	−44.84(56.52)	−19.72(35.67)	−29.11(27.61)
rnd_machines	1.59(4.50)	1.90(5.37)	−42.26(20.62)	−35.68(28.62)	−44.37(38.95)	−31.88(18.89)	−37.92(42.16)	−32.91(26.03)
rnd_slice	0.00(0.00)	25.54(14.29)	−57.53(56.12)	−50.17(42.30)	−68.42(30.19)	−60.15(58.63)	−53.90(37.90)	−61.57(50.71)
rnd_rectangle	1.43(3.07)	7.55(11.83)	−63.62(52.88)	−64.30(79.34)	−69.67(89.86)	−53.39(43.12)	−55.25(68.21)	−63.45(80.87)
dual_machines	2.91(5.41)	3.23(6.11)	−63.83(59.16)	−43.88(51.87)	−62.60(53.20)	−42.94(40.23)	−27.32(40.22)	−56.37(68.16)
dual_ops	9.79(12.05)	28.66(12.54)	−50.33(30.52)	−29.67(38.56)	−40.27(34.00)	−18.89(31.27)	−40.88(53.92)	−52.07(35.35)
dual_jobs	10.57(19.82)	32.84(22.58)	−57.55(68.69)	−62.42(81.32)	−53.49(80.27)	−52.33(51.29)	−45.98(63.81)	−54.21(62.31)
set_up	0.00(0.00)	9.21(10.63)	−48.29(44.1)	−45.3(39.08)	−46.22(52.38)	−45.92(43.69)	−31.21(29.76)	−36.25(28.24)

The results in Table 1 clearly show that the repair operators based on priority rules fail to provide improvements. Nevertheless, concluding that these operators are useless would be wrong, since they are faster than Exact and Exact loose and have good performance if the S_n are randomly generated. Therefore, they can be useful in the first steps of a local improvement procedure. However, since in the proposed RVNS we generate an initial solution of good quality, we are not going to use them.

Both Exact and Exact loose lead to good results, confirming the effectiveness of math-based neighborhoods. Since Exact loose has always better results than Exact, we use it as the only repair operator.

Considering the destroy operators, and focusing on the Exact loose column, we see that leveraging dual information is beneficial. In fact, while rnd_ops and rnd_jobs achieve average improvements of 13.27% and 18.24%, respectively, dual_ops and dual_jobs achieve improvements of 28.66% and 32.84%, respectively. The beneficial effect of dual information holds, in a relative sense, for rnd_machines and dual_machines too, where the first has an average improvement of 1.90%, while the second of 3.23%. In this latter case, the poor results of rnd_machines, and dual_machines are due to precedence constraints that allow only a small set of possible variations on partial solutions where just one machine is removed. The same happens for the operators set_up and rnd_slice, which do not lead to new solutions when the repair operator used is Exact.

Due to these results, in the RVNS, we use dual_ops, rnd_slice, worst_jobs, and dual_jobs as destroy operators. Hence, we consider four types of neighborhood.

5.3. Numerical results

In this section, we investigate the performance of RVNS and compare it against three benchmarks:

- Gurobi v9.5.2, a commercial MILP solver (Gurobi Optimization, LLC, 2023).
- Local solver[2] (LS) v11.0, a commercial mathematical optimization solver that implements proprietary heuristics to solve MILP problems of large size.
- CP-SAT,[3] a freely available constraint programming solver provided by Google's OR-Tools suite.

We split the analysis into three subsections. In the first one (Section 5.4), we show the results for instances that can be solved to optimality by Gurobi. In the second one (Section 5.5), we consider instances that Gurobi is not able to solve to optimality within the given time limit. In the third one (Section 5.6), we report the results for the instances for which Gurobi is not able to find an initial feasible solution, nor to improve an initial solution within the given time limit. In each of these subsections, we just show summary tables, but detailed results are reported in the Appendix. In all of the experiments, we set different time limits for Exact loose, and different numbers of operations to destroy (OP_TO_REMOVE). These values have been selected by carrying out preliminary experimentation.

5.4. Small size instances

The instances that can be solved to optimality by Gurobi are those characterized by 5 jobs and 10 machines (i.e., the 40 instances I-5x10-<DD>-<W>-<ID>). In Table 2 we report the average CPU time of Gurobi and of CP-SAT (the two exact methods), and we report the average gap achieved by LS and by the RVNS (the two heuristics). The time limit for all of the methods is set to 3 s, the time limit of Exact loose to 1 s, and we set OP_TO_REMOVE = 10.

The results in Table 2 show that CP-SAT requires a time comparable with that of Gurobi. In particular, CP-SAT is faster than Gurobi in the tight and extra_tight instances and slower in the loose and extra_loose ones.

LS performs well, as it achieves an average gap of 6.81% and is able to find the optimal solution in 12 out of 40 instances. The worst performances are achieved in the tard-extra_tight instances, with an average gap of 24.21%, while the best performances are for the tard-tight instances with an average gap of 0.53%. RVNS provides better performance, as it features an average gap of 0.80% and is able to find the optimal solution in 29 out of 40 instances. Its worst performances are achieved for the tard-extra_tight instances with an average gap of 3.42%, while the other average gaps are below 2%.

5.5. Medium size instances

As medium size instances, we consider the case of 10 jobs and 10 machines (i.e., the 40 instances I-10x10-<DD>-<W>-<ID>). In this case, the time limit for all the methods is set to 600 s. With this limit, Gurobi is able to find a solution, but it is not able to close the gap. For this reason, we report the average optimality gap obtained by Gurobi, and the gaps between the solution provided by Gurobi and the ones provided by the other solution methods (CP-SAT, LS, and RVNS). Therefore, a negative gap means that, on average, the solution method finds a better solution than Gurobi.

We set the time limit of Exact loose to 10 s, and OP_TO_REMOVE = 10 (as in Section 5.2). Moreover, to ensure a fair comparison, we initialize Gurobi with the same initial solution as RVNS (i.e., a solution obtained with the EDD priority rule). We report the average values for the different types of instances in Table 3.

As the reader can notice, the instances with the greatest optimality gap are the equal-extra_loose, tard-tight, and the tard-extra_tight. Therefore, the instances in which the due dates are tight are more demanding for Gurobi. In these types of instances, CP-SAT is able to outperform or to be really close to Gurobi. In more detail, CP-SAT is able to outperform Gurobi in 27 out of 40 instances (and in 8 instances it is able to find the best solution among all the methods). Its average gap with respect to Gurobi is −2.98%. All its executions are stopped by the time limit and no optimal solution is reached.

[2] https://www.localsolver.com/
[3] https://developers.google.com/optimization/cp/cp_solver

Table 2
Summary results for the small-size problem instances ($J = 5, M = 10$).

	Gurobi time [s]	CP-SAT time [s]	LS gap [%]	RVNS gap [%]
equal-extra_loose	1.30 (0.97)	1.17 (0.33)	7.94 (12.57)	0.12 (0.28)
equal-loose	0.37 (0.15)	0.71 (0.58)	1.87 (1.39)	0.34 (0.76)
equal-tight	0.53 (0.38)	0.22 (0.11)	14.15 (20.64)	1.10 (1.04)
equal-extra_tight	0.52 (0.35)	0.24 (0.21)	2.47 (3.41)	1.18 (2.63)
tard-extra_loose	1.02 (0.68)	0.95 (0.82)	1.81 (3.02)	0 (0)
tard-loose	0.30 (0.13)	0.35 (0.17)	0.53 (0.92)	0.02 (0.04)
tard-tight	0.46 (0.24)	0.21 (0.06)	1.44 (2.04)	0.21 (0.47)
tard-extra_tight	0.66 (0.60)	0.22 (0.15)	24.21 (31.25)	3.42 (5.01)
Average	**0.64 (0.44)**	**0.51 (0.31)**	**6.81 (9.41)**	**0.80 (1.28)**

Table 3
Summary results for the medium-size problem instances ($J = 10, M = 10$).

	Gurobi	CP-SAT	LS	RVNS
	opt-gap [%]	gap [%]	gap [%]	gap [%]
equal-extra_loose	29 (06)	7.62 (25.00)	43.95 (11.28)	−6.02 (10.65)
equal-loose	40 (15)	5.42 (25.56)	32.43 (24.26)	−13.61 (15.92)
equal-tight	52 (14)	−7.91 (13.89)	18.45 (31.25)	−6.07 (26.23)
equal-extra_tight	65 (04)	0.05 (16.21)	7.28 (14.41)	−28.20 (11.13)
tard-extra_loose	36 (07)	−13.25 (15.44)	38.50 (36.24)	−15.78 (9.42)
tard-loose	36 (15)	12.73 (47.61)	45.70 (31.20)	−10.81 (14.04)
tard-tight	71 (03)	−27.56 (22.23)	0.30 (16.85)	−35.85 (13.08)
tard-extra_tight	68 (05)	−2.11 (20.60)	12.53 (15.26)	−25.35 (8.67)
Average:	**49 (18)**	**−2.98 (26.35)**	**24.89 (28.25)**	**−17.54 (16.72)**

On average, LS performs worse than Gurobi for all types of instances. Its average gap with respect to Gurobi is +24.89%. Considering disaggregated data (Table A.6), LS is able to outperform Gurobi in 6 out of 40 instances, and it never finds the best solution among all the methods. This may be related to the poor efficiency of heuristic methods for relatively small instances.

Finally, the proposed RVNS achieves better average results than Gurobi in all types of instances, and it obtains the best average results among all the methods in all but 13 instances. Its average gap with respect to Gurobi is −17.54%. In more detail, RVNS is able to achieve the best results in 34 out of 40 instances.

5.6. Large size instances

As large size instances, we consider those with 20 or 50 jobs and 10 machines. For all of the methods, the time limit is 900 s for the instances with 20 jobs, and 1800 s for the instances with 50 jobs. The time limit of Exact loose is set to 30 s, and OP_TO_REMOVE = 20. In these instances, Gurobi is not able to find a feasible solution nor to improve a given one. Instead, CP-SAT is able to find good solutions, but no instance is solved to optimality. Therefore, we report the average gap between the solution of CP-SAT and RVNS, and the average gap between LS and RVNS in Table 4. These gaps are computed as:

$$\frac{f(S) - f(S^{RVNS})}{f(S)}, \tag{31}$$

where S^{RVNS} is the sequence computed by RVNS, and S is either the sequence computed by CP-SAT or LS. Therefore, a positive value means that RVNS obtains a better value than the alternative methods. To better appreciate the trends in the results, we also report the results for the instances with 10 jobs.

All the average gaps are positive, meaning that on average RVNS outperforms both benchmark methods. The average gap between RVNS and both CP-SAT and LS is around 20% and 30%, respectively. Nevertheless, while for CP-SAT the gaps have no trend, for LS the gaps reduce as the dimension of the instance increases. This may be due to the heuristic that LS is implementing, which achieves higher efficiency for large instances, or to the MILP models underneath RVNS, whose efficiency tends to decrease for large instances. Looking at the detailed results, we observe that RVNS finds the best solution among the three methods in 106 instances, CP-SAT in 10 instances, and LS just in 4 instances out of the 120 instances considered.

Table 4
Summary results for the large-size problem instances ($J > 10, M = 10$). The case of $J = 10$ is included as a reference.

instance	CP-SAT gap [%]	LS gap [%]
10 × 10-equal-extra_loose	13.36 (18.31)	55.18 (23.77)
10 × 10-equal-loose	21.86 (18.89)	33.07 (16.72)
10 × 10-equal-tight	02.82 (26.80)	18.15 (20.34)
10 × 10-equal-extra_tight	40.92 (21.65)	49.53 (11.62)
10 × 10-tard-extra_loose	2.91 (13.86)	64.13 (43.35)
10 × 10-tard-loose	22.82 (37.03)	38.13 (04.21)
10 × 10-tard-tight	14.40 (35.53)	34.46 (18.36)
10 × 10-tard-extra_tight	30.08 (17.93)	51.18 (19.53)
20 × 10-equal-extra_loose	36.35 (23.12)	54.02 (19.77)
20 × 10-equal-loose	27.29 (23.90)	29.50 (10.57)
20 × 10-equal-tight	21.26 (11.17)	22.76 (09.07)
20 × 10-equal-extra_tight	30.19 (19.53)	24.07 (15.12)
20 × 10-tard-extra_loose	31.15 (9.06)	46.66 (12.20)
20 × 10-tard-loose	30.32 (38.33)	21.60 (28.26)
20 × 10-tard-tight	22.10 (03.87)	20.41 (08.69)
20 × 10-tard-extra_tight	27.42 (16.62)	17.12 (10.41)
50 × 10-equal-extra_loose	27.77 (19.10)	21.31 (15.00)
50 × 10-equal-loose	16.69 (16.66)	10.13 (08.56)
50 × 10-equal-tight	22.11 (09.29)	06.71 (08.80)
50 × 10-equal-extra_tight	17.73 (06.34)	6.69 (08.77)
50 × 10-tard-extra_loose	25.80 (13.40)	15.42 (08.45)
50 × 10-tard-loose	22.48 (07.79)	06.72 (04.58)
50 × 10-tard-tight	27.34 (07.35)	07.36 (07.08)
50 × 10-tard-extra_tight	18.75 (07.50)	03.21 (08.29)
Average	**22.99 (19.63)**	**33.11 (28.72)**

6. Conclusions

In this paper, we have considered a just-in-time job shop scheduling problem with sequence-dependent setup times and release dates, for

Table A.5
Detailed results for small-size problem instances ($J = 5, M = 10$).

	Gurobi		CP-SAT		LS		RVNS		
Instance_name	Of	Time [s]	Of	Time [s]	Of	Time [s]	Of	Time [s]	Time to best[s]
I-5 × 10-equal-extra_loose-0	67.32	0.78	**67.32***	0.80	87.08	2.37	**67.32***	3.22	1.60
I-5 × 10-equal-extra_loose-1	70.85	0.95	**70.85***	1.15	71.22	2.18	**70.85***	2.59	2.53
I-5 × 10-equal-extra_loose-2	54.30	0.59	**54.30***	1.26	54.33	2.74	**54.30***	3.21	3.17
I-5 × 10-equal-extra_loose-3	48.21	1.19	**48.21***	0.99	48.47	2.86	48.51	2.46	2.42
I-5 × 10-equal-extra_loose-4	80.15	3.00	**80.15***	1.66	87.54	2.52	**80.15***	2.43	2.30
I-5 × 10-equal-loose-0	83.05	0.59	**83.05***	0.54	85.26	2.33	**83.05***	2.49	0.83
I-5 × 10-equal-loose-1	68.53	0.19	**68.53***	0.30	70.18	2.66	69.71	2.58	1.50
I-5 × 10-equal-loose-2	49.50	0.28	**49.50***	0.61	51.20	2.91	**49.50***	2.21	1.92
I-5 × 10-equal-loose-3	53.66	0.38	**53.66***	0.38	53.67	2.17	53.67	2.88	1.33
I-5 × 10-equal-loose-4	81.26	0.41	**81.26***	1.73	81.96	2.84	**81.26***	2.08	2.02
I-5 × 10-equal-tight-0	84.23	1.03	**84.23***	0.29	126.99	2.16	**84.23***	2.14	2.09
I-5 × 10-equal-tight-1	48.45	0.19	**48.45***	0.14	49.48	2.01	49.55	2.54	0.44
I-5 × 10-equal-tight-2	40.71	0.19	**40.71***	0.18	44.00	2.18	**40.71***	2.05	0.65
I-5 × 10-equal-tight-3	55.22	0.39	**55.22***	0.09	59.35	2.87	56.22	2.61	2.19
I-5 × 10-equal-tight-4	95.65	0.85	**95.65***	0.39	97.89	2.22	97.03	2.58	1.53
I-5 × 10-equal-extra_tight-0	93.75	0.95	**93.75***	0.61	100.08	2.56	99.27	2.38	2.35
I-5 × 10-equal-extra_tight-1	55.89	0.70	**55.89***	0.24	59.03	2.97	**55.89***	2.49	2.45
I-5 × 10-equal-extra_tight-2	96.14	0.66	**96.14***	0.10	**96.14***	2.12	**96.14***	2.14	2.11
I-5 × 10-equal-extra_tight-3	51.97	0.18	**51.97***	0.13	**51.97***	2.82	**51.97***	2.55	2.50
I-5 × 10-equal-extra_tight-4	58.70	0.14	**58.70***	0.14	**58.70***	2.72	**58.70***	2.21	0.86
I-5 × 10-tard-extra_loose-0	54.71	1.30	**54.71***	0.82	58.61	2.76	**54.71***	2.88	2.84
I-5 × 10-tard-extra_loose-1	71.25	2.06	**71.25***	2.38	71.63	2.56	**71.25***	2.33	2.29
I-5 × 10-tard-extra_loose-2	48.13	0.77	**48.13***	0.63	**48.13***	2.86	**48.13***	3.12	2.49
I-5 × 10-tard-extra_loose-3	68.45	0.62	**68.45***	0.54	**68.45***	2.88	**68.45***	3.26	2.24
I-5 × 10-tard-extra_loose-4	67.47	0.33	**67.47***	0.37	68.39	2.89	**67.47***	3.06	1.75
I-5 × 10-tard-loose-0	42.62	0.14	**42.62***	0.14	42.73	2.10	**42.62***	2.07	1.44
I-5 × 10-tard-loose-1	77.91	0.51	**77.91***	0.25	**77.91***	2.87	**77.91***	2.18	0.55
I-5 × 10-tard-loose-2	59.63	0.33	**59.63***	0.57	**59.63***	2.61	**59.63***	2.21	1.83
I-5 × 10-tard-loose-3	43.89	0.26	**43.89***	0.49	43.98	2.63	**43.89***	2.15	0.62
I-5 × 10-tard-loose-4	55.31	0.28	**55.31***	0.28	56.52	2.68	55.37	2.06	2.04
I-5 × 10-tard-tight-0	74.95	0.28	**74.95***	0.17	77.08	2.50	75.75	2.21	2.00
I-5 × 10-tard-tight-1	40.15	0.16	**40.15***	0.23	**40.15***	2.67	**40.15***	2.11	0.82
I-5 × 10-tard-tight-2	62.75	0.58	**62.75***	0.22	65.50	2.61	**62.75***	2.06	1.91
I-5 × 10-tard-tight-3	53.40	0.79	**53.40***	0.32	**53.40***	2.28	**53.40***	2.87	2.23
I-5 × 10-tard-tight-4	50.33	0.49	**50.33***	0.13	**50.33***	2.44	**50.33***	2.15	0.68
I-5 × 10-tard-extra_tight-0	60.46	0.16	**60.46***	0.16	**60.46***	2.60	**60.46***	3.04	1.47
I-5 × 10-tard-extra_tight-1	76.40	1.60	**76.40***	0.47	92.89	2.18	80.99	2.55	2.51
I-5 × 10-tard-extra_tight-2	67.29	0.10	**67.29***	0.07	**67.29***	2.19	**67.29***	3.06	1.91
I-5 × 10-tard-extra_tight-3	69.22	0.71	**69.22***	0.24	122.12	2.66	76.89	2.95	2.91
I-5 × 10-tard-extra_tight-4	72.99	0.74	**72.99***	0.16	89.82	2.79	**72.99***	2.40	2.36

which we have proposed a mathematical model and an efficient RVNS approach to tackle it. By considering 80 different neighborhoods generated from the application of 10 destroy and 8 repair operators, we find that the destroy operators leveraging information obtained from the mathematical models outperform the ones that randomly destroy part of the solution. Moreover, we define a new and effective repair operator based on a MILP model. To evaluate the performance of the proposed algorithm, as well as the quality of its solutions, we have conducted comprehensive computational experiments on a set of 160 benchmark instances. The results are compared with those obtained by Gurobi, CP-SAT, and Local Solver. The comparative analysis shows that the proposed heuristic is able to find the optimal solution in 29 out of 40 instances, and to find the best solution in 106 over 120 of the remaining instances (the ones for which we cannot compute the optimal solution).

CRediT authorship contribution statement

Paolo Brandimarte: Conceptualization, Investigation, Methodology, Writing – original draft, Writing – review & editing. **Edoardo Fadda:** Conceptualization, Software, Writing – original draft, Writing – review & editing, Investigation, Methodology.

Data availability

Problem instances are available at https://github.com/EdoF90/just-in-time-jss-setup-times. The code that supports the findings will be made available on request.

Appendix. Detailed results

In this Appendix, we provide the detailed results for each experiment, with the main aim to provide a benchmark for future studies.

In Table A.5, we show the data used to compute Table 2. We report the optimal objective function and the CPU time of Gurobi, CP-SAT, LS, and RVNS for the small instances (i.e., $J = 5, M = 10$). Moreover, we also report the time to reach the best solution for RVNS. We highlight in boldface the best result, and we add a $*$ if the solution is optimal. As expected, since CP-SAT is an exact constraint programming solver, it is able to find the optimal solution for all of the instances. Instead, LS finds the optimal solution in 12 instances, and RVNS in 29. In several instances, RVNS is able to find the optimal solution with a time-to-best far smaller than the whole execution time. This proves that RVNS is able to effectively explore the solution space.

In Table A.6, we show the data used to compute the values in Table 3 for the medium-size instances (i.e., $J = 10, M = 10$). We report the objective function of the solution found by Gurobi, its optimality gap, and the objective function of CP-SAT, LS, and RVNS (for all the methods, we set the time limit to 600 s). We highlight in boldface the best result for each instance. As the reader can notice, Gurobi is not able to close the gap in any of the instances considered. Moreover, it finds the best solution only in 5 instances, while CP-SAT in 8 and RVNS in 27.

Finally, in Table A.7, we show the data used to compute the values in Table 4. Even though Table 4 contains the data of the medium-size instances to appreciate a trend, we do not report them here, since they

Table A.6
Detailed results for the medium-size problem instances ($J = 10, M = 10$).

Instance	Of	Gap [%]	Of CP-SAT	Of LS	Of RVNS
I-10 × 10-equal-extra_loose-0	**226.18**	32	238.06	374.21	226.53
I-10 × 10-equal-extra_loose-1	125.10	25	**114.22**	165.38	115.43
I-10 × 10-equal-extra_loose-2	**170.07**	27	251.87	236.71	176.19
I-10 × 10-equal-extra_loose-3	252.50	41	188.40	359.39	**186.90**
I-10 × 10-equal-extra_loose-4	115.29	24	137.03	162.14	**115.14**
I-10 × 10-equal-loose-0	323.81	59	283.77	340.99	**235.54**
I-10 × 10-equal-loose-1	**182.09**	19	244.80	227.92	206.10
I-10 × 10-equal-loose-2	156.95	34	207.74	183.87	**135.12**
I-10 × 10-equal-loose-3	195.47	43	171.11	321.21	**163.39**
I-10 × 10-equal-loose-4	289.12	45	246.18	434.25	**220.60**
I-10 × 10-equal-tight-0	272.66	44	**243.88**	410.69	382.05
I-10 × 10-equal-tight-1	482.03	73	**368.57**	360.76	385.26
I-10 × 10-equal-tight-2	357.63	58	307.82	526.18	**276.09**
I-10 × 10-equal-tight-3	176.58	37	166.78	193.68	**158.31**
I-10 × 10-equal-tight-4	200.87	51	228.95	220.95	**166.21**
I-10 × 10-equal-extra_tight-0	411.19	65	410.05	490.68	**330.45**
I-10 × 10-equal-extra_tight-1	341.44	60	307.95	388.45	**238.08**
I-10 × 10-equal-extra_tight-2	291.31	62	274.64	354.37	**224.44**
I-10 × 10-equal-extra_tight-3	395.43	66	518.04	426.51	**319.66**
I-10 × 10-equal-extra_tight-4	411.82	73	350.30	303.91	**210.07**
I-10 × 10-tard-extra_loose-0	174.01	36	**137.68**	333.84	144.55
I-10 × 10-tard-extra_loose-1	185.25	36	**152.81**	213.13	174.99
I-10 × 10-tard-extra_loose-2	135.42	31	117.46	173.01	**111.87**
I-10 × 10-tard-extra_loose-3	212.62	50	148.47	193.50	**144.78**
I-10 × 10-tard-extra_loose-4	178.85	29	206.67	298.35	**166.11**
I-10 × 10-tard-loose-0	**106.41**	28	125.17	185.62	108.28
I-10 × 10-tard-loose-1	245.84	59	**145.78**	258.41	172.21
I-10 × 10-tard-loose-2	165.24	30	203.35	238.49	**154.90**
I-10 × 10-tard-loose-3	164.22	43	131.56	207.73	**130.05**
I-10 × 10-tard-loose-4	**152.29**	20	279.57	271.27	154.09
I-10 × 10-tard-tight-0	522.56	74	433.38	514.70	**413.62**
I-10 × 10-tard-tight-1	326.96	71	**243.38**	279.54	250.27
I-10 × 10-tard-tight-2	348.78	70	**175.79**	358.89	195.16
I-10 × 10-tard-tight-3	611.00	66	641.66	778.50	**405.09**
I-10 × 10-tard-tight-4	536.69	74	294.64	468.03	**264.68**
I-10 × 10-tard-extra_tight-0	299.46	68	283.66	371.51	**243.87**
I-10 × 10-tard-extra_tight-1	385.96	61	419.86	438.77	**289.55**
I-10 × 10-tard-extra_tight-2	351.73	68	457.56	444.76	**304.64**
I-10 × 10-tard-extra_tight-3	678.67	76	468.20	778.72	**431.84**
I-10 × 10-tard-extra_tight-4	255.32	67	221.79	213.70	**169.97**

Table A.7
Detailed results for the large-size problem instances ($J > 10, M = 10$).

Instance_name	CP-SAT		LS		RVNS	
	Of	Time [s]	Of	Time [s]	Of	Time [s]
I-20 × 10-equal-extra_loose-0	1006.86	900.07	1471.63	899.11	**841.54**	900.85
I-20 × 10-equal-extra_loose-1	1721.74	900.06	1541.99	899.04	**1026.70**	901.49
I-20 × 10-equal-extra_loose-2	1608.14	900.08	1974.95	899.80	**1484.79**	829.94
I-20 × 10-equal-extra_loose-3	1249.23	900.06	1515.35	899.40	**869.70**	897.79
I-20 × 10-equal-extra_loose-4	1219.06	900.08	1178.91	899.63	**855.59**	900.40
I-20 × 10-equal-loose-0	1304.20	901.50	1655.29	899.24	**1128.21**	900.92
I-20 × 10-equal-loose-1	1875.73	900.06	1787.33	899.19	**1374.88**	900.72
I-20 × 10-equal-loose-2	1202.63	900.03	1385.64	899.28	**733.66**	845.28
I-20 × 10-equal-loose-3	1577.04	900.05	2025.22	899.85	**1548.77**	857.55
I-20 × 10-equal-loose-4	1874.75	900.08	2024.80	899.68	**1579.68**	876.11
I-20 × 10-equal-tight-0	2057.45	901.57	2436.38	899.85	**1698.39**	900.15
I-20 × 10-equal-tight-1	1923.13	900.07	2272.32	6287.53	**1688.46**	895.71
I-20 × 10-equal-tight-2	1996.10	900.04	2327.48	899.60	**1856.68**	826.27
I-20 × 10-equal-tight-3	2353.74	900.07	2456.92	899.70	**1733.75**	857.88
I-20 × 10-equal-tight-4	1729.82	900.06	1471.77	899.42	**1351.58**	900.56
I-20 × 10-equal-extra_tight-0	1872.88	900.09	1984.48	899.06	**1685.81**	899.00
I-20 × 10-equal-extra_tight-1	2537.44	900.07	2586.27	899.19	**2354.30**	901.27
I-20 × 10-equal-extra_tight-2	2301.46	900.07	2493.28	899.59	**1671.56**	900.57
I-20 × 10-equal-extra_tight-3	2676.36	900.07	2344.20	899.59	**1862.51**	901.94
I-20 × 10-equal-extra_tight-4	3189.72	900.07	2492.58	899.05	**2116.67**	902.76
I-20 × 10-tard-extra_loose-0	1385.75	900.07	1757.53	900.01	**1141.10**	901.47
I-20 × 10-tard-extra_loose-1	1475.29	900.05	1531.87	899.71	**1148.88**	902.14
I-20 × 10-tard-extra_loose-2	1550.83	900.09	1836.68	899.29	**1148.14**	899.76

(*continued on next page*)

are already contained in Table A.6. Therefore, in Table A.7 we report the best objective function and the CPU time of CP-SAT, LS, and RVNS for large-size instances (i.e., $J > 10, M = 10$). As in the other tables, the best result for each instance is highlighted in boldface. For these instances, the best results are typically obtained by RVNS which is the best method in 74 instances, while CP-SAT is in 2 and LS in 4.

Table A.7 (*continued*).

Instance_name	CP-SAT		LS		RVNS	
	Of	Time [s]	Of	Time [s]	Of	Time [s]
I-20 × 10-tard-extra_loose-3	1650.21	900.06	1528.00	899.55	**1139.85**	898.18
I-20 × 10-tard-extra_loose-4	1811.76	900.06	2182.93	899.73	**1436.88**	890.06
I-20 × 10-tard-loose-0	1386.22	900.21	1263.26	899.37	**1218.93**	899.11
I-20 × 10-tard-loose-1	**1565.81**	900.44	1718.33	899.38	2023.20	898.99
I-20 × 10-tard-loose-2	1941.75	900.06	2274.83	899.81	**1548.70**	900.32
I-20 × 10-tard-loose-3	1833.10	900.06	1750.97	899.79	**1086.45**	875.70
I-20 × 10-tard-loose-4	1236.07	900.05	1558.90	899.89	**742.91**	865.04
I-20 × 10-tard-tight-0	2050.37	901.69	1892.16	899.79	**1739.00**	867.47
I-20 × 10-tard-tight-1	2678.78	901.84	2713.74	899.23	**2241.30**	900.15
I-20 × 10-tard-tight-2	1606.64	900.07	1645.01	900.00	**1329.62**	900.43
I-20 × 10-tard-tight-3	2429.73	900.06	2727.28	899.62	**1912.74**	877.19
I-20 × 10-tard-tight-4	1864.13	900.07	2053.54	899.17	**1488.58**	885.96
I-20 × 10-tard-extra_tight-0	2663.94	900.07	2254.73	899.35	**2034.15**	901.57
I-20 × 10-tard-extra_tight-1	3520.72	900.08	2996.17	899.81	**2447.78**	903.45
I-20 × 10-tard-extra_tight-2	2397.94	900.07	1802.70	899.99	**1726.42**	901.42
I-20 × 10-tard-extra_tight-3	1972.66	900.06	2130.92	899.95	**1621.39**	902.13
I-20 × 10-tard-extra_tight-4	2400.22	900.07	2748.43	899.12	**2358.56**	899.95
I-50 × 10-equal-extra_loose-0	17 541.61	1800.19	16 820.48	1800.02	**15 142.86**	1798.31
I-50 × 10-equal-extra_loose-1	19 149.58	1800.13	16 396.87	1799.86	**13 014.09**	1795.58
I-50 × 10-equal-extra_loose-2	18 615.07	1800.23	18 553.95	1799.90	**15 769.38**	1801.76
I-50 × 10-equal-extra_loose-3	17 258.67	1800.43	16 725.03	1799.23	**11 547.16**	1799.34
I-50 × 10-equal-extra_loose-4	17 177.62	1800.20	16 959.56	1799.03	**15 851.91**	1801.50
I-50 × 10-equal-loose-0	17567.83	1813.49	16933.15	1799.76	**15723.49**	1800.91
I-50 × 10-equal-loose-1	18563.47	1800.18	17560.57	9056.45	**13391.69**	1761.68
I-50 × 10-equal-loose-2	**16853.23**	1800.23	18049.89	1799.37	17996.77	1788.31
I-50 × 10-equal-loose-3	18019.40	1800.23	15816.82	1799.87	**14419.72**	1800.71
I-50 × 10-equal-loose-4	16150.50	1800.24	15789.73	1799.68	**14108.75**	1788.30
I-50 × 10-equal-tight-0	19655.75	1809.33	17193.25	1799.81	**16613.74**	1795.84
I-50 × 10-equal-tight-1	19043.11	1800.36	19024.38	1799.77	**15979.70**	1799.02
I-50 × 10-equal-tight-2	18338.51	1800.22	**13014.14**	1799.33	13891.73	1744.06
I-50 × 10-equal-tight-3	19072.55	1800.21	16030.53	1799.62	**14558.36**	1755.63
I-50 × 10-equal-tight-4	15419.31	1800.33	15873.92	1799.78	**14007.83**	1800.66
I-50 × 10-equal-extra_tight-0	22 478.06	1800.30	**18 414.10**	1799.75	19 983.32	1799.70
I-50 × 10-equal-extra_tight-1	17 739.85	1800.23	17 189.70	1799.15	**16 134.87**	1802.96
I-50 × 10-equal-extra_tight-2	18 688.75	1800.30	17 039.98	1799.54	**15 198.65**	1801.25
I-50 × 10-equal-extra_tight-3	22 381.51	1800.31	20 305.58	1799.27	**18 828.24**	1798.24
I-50 × 10-equal-extra_tight-4	20 187.68	1800.30	18 634.47	1799.18	**16 233.10**	1804.14
I-50 × 10-tard-extra_loose-0	18 459.53	1800.39	17 590.03	1799.63	**16 089.75**	1802.06
I-50 × 10-tard-extra_loose-1	14 834.61	1800.20	15 656.24	1799.09	**13 057.58**	1785.91
I-50 × 10-tard-extra_loose-2	16 941.54	1800.41	13 976.15	1799.92	**13 459.31**	1799.36
I-50 × 10-tard-extra_loose-3	18 437.27	1798.35	17 303.06	1799.91	**14 406.77**	1795.25
I-50 × 10-tard-extra_loose-4	16 904.87	1800.20	14 265.26	1799.25	**11 512.95**	1801.91
I-50 × 10-tard-loose-0	21188.58	1800.04	17228.17	1799.69	**17205.28**	1790.48
I-50 × 10-tard-loose-1	22788.54	1800.46	19217.73	1799.76	**16897.47**	1801.48
I-50 × 10-tard-loose-2	19167.28	1800.27	17753.04	1799.27	**16873.75**	1796.41
I-50 × 10-tard-loose-3	17567.82	1800.36	16272.71	1799.23	**14696.68**	1800.34
I-50 × 10-tard-loose-4	19755.13	1800.33	17475.88	1799.29	**16293.69**	1794.84
I-50 × 10-tard-tight-0	24063.68	1800.90	18433.78	3308.39	**17848.88**	1788.17
I-50 × 10-tard-tight-1	19718.57	1800.43	19085.70	1799.46	**15571.34**	1800.49
I-50 × 10-tard-tight-2	19002.23	1800.23	15976.37	1799.90	**15204.04**	1799.74
I-50 × 10-tard-tight-3	19419.60	1800.19	16725.15	1799.18	**16650.01**	1800.44
I-50 × 10-tard-tight-4	20991.90	1800.19	17446.00	1800.00	**15709.61**	1800.96
I-50 × 10-tard-extra_tight-0	16 369.09	1800.46	14 408.46	1799.55	**13 201.38**	1801.09
I-50 × 10-tard-extra_tight-1	18 720.66	1800.72	16 909.03	1800.01	**14774.83**	1751.71
I-50 × 10-tard-extra_tight-2	21 122.72	1800.30	**18 822.68**	1799.63	19 685.03	1797.32
I-50 × 10-tard-extra_tight-3	21 796.83	1800.29	18 455.03	1799.86	**18 341.98**	1805.53
I-50 × 10-tard-extra_tight-4	19 173.64	1800.25	**15774.99**	1800.01	16 397.02	1802.62

References

Abderrazzak, S., Hamid, A., Omar, S., 2022. Adaptive large neighborhood search for the just-in-time job-shop scheduling problem. In: 2022 International Conference on Control, Automation and Diagnosis. ICCAD, IEEE, pp. 1–6. http://dx.doi.org/10.1109/iccad55197.2022.9853973.

Ahmadian, M.M., Salehipour, A., 2020. The just-in-time job-shop scheduling problem with distinct due-dates for operations. J. Heuristics 27 (1–2), 175–204. http://dx.doi.org/10.1007/s10732-020-09458-6.

Ahmadian, M.M., Salehipour, A., Cheng, T., 2021. A meta-heuristic to solve the just-in-time job-shop scheduling problem. European J. Oper. Res. 288 (1), 14–29. http://dx.doi.org/10.1016/j.ejor.2020.04.017.

Allahverdi, A., Gupta, J.N., Aldowaisan, T., 1999. A review of scheduling research involving setup considerations. Omega 27 (2), 219–239. http://dx.doi.org/10.1016/s0305-0483(98)00042-5.

Allahverdi, A., Ng, C., Cheng, T., Kovalyov, M.Y., 2008. A survey of scheduling problems with setup times or costs. European J. Oper. Res. 187 (3), 985–1032. http://dx.doi.org/10.1016/j.ejor.2006.06.060.

Baptiste, P., Flamini, M., Sourd, F., 2008. Lagrangian bounds for just-in-time job-shop scheduling. Comput. Oper. Res. 35 (3), 906–915. http://dx.doi.org/10.1016/j.cor.2006.05.009.

Brandimarte, P., Maiocco, M., 1999. Job shop scheduling with a non-regular objective: A comparison of neighbourhood structures based on a sequencing/timing decomposition. Int. J. Prod. Res. 37 (8), 1697–1715. http://dx.doi.org/10.1080/002075499190969.

Brimberg, J., Salhi, S., Todosijević, R., Urošević, D., 2023. Variable neighborhood search: The power of change and simplicity. Comput. Oper. Res. 155, 106221. http://dx.doi.org/10.1016/j.cor.2023.106221.

Bürgy, R., Bülbül, K., 2018. The job shop scheduling problem with convex costs. European J. Oper. Res. 268 (1), 82–100. http://dx.doi.org/10.1016/j.ejor.2018.01.027.

dos Santos, A.G., Araujo, R.P., Arroyo, J.E.C., 2010. A combination of evolutionary algorithm, mathematical programming, and a new local search procedure for the just-in-time job-shop scheduling problem. In: Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 10–24. http://dx.doi.org/10.1007/978-3-642-13800-3_2.

Feng, G., Lau, H.C., 2007. Efficient algorithms for machine scheduling problems with earliness and tardiness penalties. Ann. Oper. Res. 159 (1), 83–95. http://dx.doi.org/10.1007/s10479-007-0284-z.

Fox, M.S., Smith, S.F., 1984. ISIS a knowledge-based system for factory scheduling. Expert Syst. 1 (1), 25–49. http://dx.doi.org/10.1111/j.1468-0394.1984.tb00424.x.

Gen, M., 1994. Solving job-shop scheduling problem using genetic algorithms. In: Proceedings of the 16th International Conference on Computer and Industrial Engineering, Ashikaga, Japan, 1994. pp. 93–98.

Graham, R., Lawler, E., Lenstra, J., Kan, A., 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey. In: Discrete Optimization II, Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium Co-Sponsored By IBM Canada and SIAM Banff, Aha. and Vancouver. Elsevier, pp. 287–326. http://dx.doi.org/10.1016/s0167-5060(08)70356-x.

Gurobi Optimization, LLC, 2023. Gurobi Optimizer Reference Manual. URL https://www.gurobi.com.

Hansen, P., Mladenović, N., 2018. Variable neighborhood search. In: Handbook of Heuristics. Springer International Publishing, pp. 759–787. http://dx.doi.org/10.1007/978-3-319-07124-4_19.

Kolahan, F., Liang, M., 1998. An adaptive TS approach to JIT sequencing with variable processing times and sequence-dependent setups. European J. Oper. Res. 109 (1), 142–159. http://dx.doi.org/10.1016/s0377-2217(97)00098-2.

Leonardi, S., Raz, D., 1997. Approximating total flow time on parallel machines. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing - STOC '97. ACM Press, pp. 875–891. http://dx.doi.org/10.1145/258533.258562.

Liao, C.J., Cheng, C.C., 2007. A variable neighborhood search for minimizing single machine weighted earliness and tardiness with common due date. Comput. Ind. Eng. 52 (4), 404–413. http://dx.doi.org/10.1016/j.cie.2007.01.004.

Mladenović, N., Hansen, P., 1997. Variable neighborhood search. Comput. Oper. Res. 24 (11), 1097–1100. http://dx.doi.org/10.1016/s0305-0548(97)00031-2.

Mladenović, N., Todosijević, R., Urošević, D., 2016. Less is more: Basic variable neighborhood search for minimum differential dispersion problem. Inform. Sci. 326, 160–171. http://dx.doi.org/10.1016/j.ins.2015.07.044.

Pinedo, M.L., 2008. Modeling and solving scheduling problems in practice. In: Scheduling. Springer New York, New York, NY, pp. 427–454.

Radhakrishnan, S., Ventura, J.A., 2000. Simulated annealing for parallel machine scheduling with earliness-tardiness penalties and sequence-dependent set-up times. Int. J. Prod. Res. 38 (10), 2233–2252. http://dx.doi.org/10.1080/00207540050028070.

Santos, H.C., França, P.M., 1997. Scheduling with sequence-dependent setup times and early-tardy penalties. IFAC Proc. Vol. 30 (19), 239–244. http://dx.doi.org/10.1016/s1474-6670(17)42305-6.

Shabtay, D., 2012. The just-in-time scheduling problem in a flow-shop scheduling system. European J. Oper. Res. 216 (3), 521–532. http://dx.doi.org/10.1016/j.ejor.2011.07.053.

Thevenin, S., Zufferey, N., 2019. Learning variable neighborhood search for a scheduling problem with time windows and rejections. Discrete Appl. Math. 261, 344–353. http://dx.doi.org/10.1016/j.dam.2018.03.019.

Wang, S., Li, Y., 2014. Variable neighbourhood search and mathematical programming for just-in-time job-shop scheduling problem. Math. Probl. Eng. 2014, 1–9. http://dx.doi.org/10.1155/2014/431325.

Xiong, F., Chu, M., Li, Z., Du, Y., Wang, L., 2021. Just-in-time scheduling for a distributed concrete precast flow shop system. Comput. Oper. Res. 129, 105204. http://dx.doi.org/10.1016/j.cor.2020.105204.

Zhu, Z., Heady, R.B., 2000. Minimizing the sum of earliness/tardiness in multi-machine scheduling: a mixed integer programming approach. Comput. Ind. Eng. 38 (2), 297–305. http://dx.doi.org/10.1016/s0360-8352(00)00048-6.