European Journal of Operational Research 315 (2024) 442–453

Contents lists available at ScienceDirect

European Journal of Operational Research

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

Discrete optimization

Exact and heuristic algorithms for minimizing the makespan on a single machine scheduling problem with sequence-dependent setup times and release dates

Rafael Morais [a,*], Teobaldo Bulhões [b], Anand Subramanian [c]

[a] Universidade Federal da Paraíba, Centro de Informática, Rua dos Escoteiros s/n, Mangabeira, 58055-000, João Pessoa, Brazil
[b] Universidade Federal da Paraíba, Departamento de Computação Científica, Centro de Informática, Rua dos Escoteiros s/n, Mangabeira, 58055-000, João Pessoa, Brazil
[c] Universidade Federal da Paraíba, Departamento de Sistemas de Computação, Centro de Informática, Rua dos Escoteiros s/n, Mangabeira, 58055-000, João Pessoa, Brazil

ARTICLE INFO

Keywords:
Scheduling
Release dates
Column generation
Metaheuristics

ABSTRACT

This paper proposes efficient exact and heuristic approaches for minimizing the makespan on a single machine scheduling problem with sequence-dependent setup times and release dates. The exact procedure consists of a branch-and-price (B&P) algorithm implemented over an arc-time-indexed formulation with a pseudo-polynomial number of variables and constraints. Our B&P algorithm includes several modern features, such as a dynamic ng-path relaxation and a bidirectional labeling method for efficiently solving the pricing subproblem. The proposed heuristic algorithm is based on the iterated local search framework that employs a beam search approach adapted from the literature for generating initial solutions and an efficient scheme to perform move evaluations in amortized constant time. Computational experiments were carried out on benchmark instances containing 1800 instances ranging from 25 to 150 jobs. The results obtained attest the high performance of both the exact and heuristic algorithms in obtaining high-quality bounds when compared to existing methods. We report improved lower and upper bounds for the vast majority of the instances, as well as optimal solutions for 42.7% of the instances.

1. Introduction

The single machine configuration is one of the most studied scheduling variants, since problems in more complicated machine environments are frequently decomposed into single-machine subproblems (Pinedo, 2012). Over the years, many characteristics have been incorporated into single machine scheduling models in order to address the different situations found in real world environments.

In some manufacturing scenarios, jobs may not arrive in the system at the same time as they are often conditioned to the availability of material and personnel, or to the delivery of products coming from an unmodeled part of the factory (Fu et al., 2018). Hence, considering release dates makes the models more suitable to deal with real-life scenarios. Moreover, the presence of sequence-dependent changeover times between jobs is a significant characteristic of industries like commercial painting and metal processing, having a direct impact in the process efficiency (Ozgur & Bai, 2010). Scheduling problems considering setup times are the subject of a major review published by Allahverdi et al. (1999) and later updated in Allahverdi et al. (2008) and Allahverdi (2015). In these reviews, problems arising in different environments (e.g., single and parallel machines, flow shop, job shop and open shop) are classified into batch and non-batch, sequence-independent and sequence-dependent setup, as well as anticipatory and non-anticipatory.

A batch setup problem considers similar jobs grouped into batches, with setup times incurred in the transition between different batches. If setup operations occur for every job switch, the problem is labeled as non-batch. Setup times are said to be sequence-dependent if the time spent in the setup operation depends on both the job to be processed and the immediately preceding job. If the setup time depends only on the job being processed, it is classified as independent. When the setup operation can be processed before the job release time, the setup is considered anticipatory. In contrast, a non-anticipatory setup means that the setup operation can only start from the release date.

This paper addresses the problem of minimizing the makespan on a single machine with release dates and asymmetric non-anticipatory

* Corresponding author.
E-mail addresses: rafaelmorais@eng.ci.ufpb.br, rafaelsobralm@gmail.com (R. Morais), tbulhoes@ci.ufpb.br (T. Bulhões), anand@ci.ufpb.br (A. Subramanian).

https://doi.org/10.1016/j.ejor.2023.11.024
Received 5 February 2023; Accepted 15 November 2023
Available online 20 November 2023

sequence-dependent setup times, considering a non-batch environment. The problem is known to be strongly $\mathcal{NP}$-hard (Pinedo, 2012), and according to the three-field notation by Graham et al. (1979), it can be denoted as $1|r_j, s_{ij}|C_{max}$. It is worth emphasizing that we address the case in which an initial setup operation is considered.

For what concerns single machine scheduling problems with release dates and sequence-dependent setup times, we were not able to identify exact methods capable of systematically solving instances with more than 25 jobs to optimality. To address such instances, several heuristics were proposed in the literature, but among them we were not able to find local search-based approaches that specifically employ efficient move evaluation procedures to reduce the computational complexity of the search. In light of these gaps, the main contributions of our study are listed as follows.

- We propose a branch-and-price (B&P) algorithm for problem $1|r_j, s_{ij}|C_{max}$ that is capable of solving hundreds of open instances with up to 150 jobs.
- We develop an iterated local search (ILS) heuristic for the aforementioned problem, which uses a beam search (BS) procedure inspired in Velez-Gallego et al. (2016) to generate initial solutions. Our algorithm provides extremely competitive results when compared to existing heuristics for the problem, improving the best known solution of hundreds of benchmark instances.
- We implement an efficient move evaluation scheme based on the ideas presented in Kindervater and Savelsbergh (1997) and Vidal et al. (2013) for the vehicle routing problem with time windows. Such scheme enables the local search moves to be evaluated in amortized constant time, leading to significant gains in the performance of the algorithm regarding CPU time.

The remainder of the paper is organized as follows. Section 2 presents the related work. Section 3 formally defines problem $1|r_j, s_{ij}|C_{max}$. The proposed exact method is described in Section 4, while Section 5 explains the heuristic algorithm, including the efficient move evaluation scheme. Section 6 contains the computational experiments conducted to assess the performance of the proposed methods. Finally, Section 7 concludes.

2. Related work

To our knowledge, only a handful of works have specifically addressed problem $1|r_j, s_{ij}|C_{max}$ in the literature. Montoya-Torres et al. (2010) and Montoya-Torres et al. (2012) proposed a random insertion algorithm for a version of the problem that does not consider the initial setup time. The problem that considers the initial setup operation was tackled by Velez-Gallego et al. (2016), who presented a mixed integer linear programming (MILP) formulation and a BS heuristic. Both methods were tested on a set of instances suggested in Ovacikt and Uzsoy (1994). While the MILP model was mostly capable of solving instances of limited size (25–50 jobs) to optimality within one hour, the BS procedure managed to quickly achieve promising solutions on instances with up to 150 jobs.

Fan et al. (2019) put forward a variable block insertion heuristic (VBIH) and compared the results with those reported in Velez-Gallego et al. (2016), but only for instances with 25, 50, and 75 jobs. They also adapted an iterated greedy (IG) algorithm developed in Ruiz and Stützle (2007). The VBIH algorithm performed better than BS and IG in terms of solution quality on instances with 50 and 75 jobs, but it was outperformed by both on the 25-job instances. Moreover, Fernandez-Viagas and Costa (2021) presented a heuristic based on BS and a population-based metaheuristic for both the anticipatory and the non-anticipatory cases. The proposed algorithms were compared with reimplementations of several heuristics from the literature that were proposed for similar scheduling problems.

Other variants of this problem have been studied in the literature. Bianco et al. (1988) proposed a MILP formulation and a branch-and-bound (B&B) algorithm for the problem of minimizing the makespan with anticipatory setups and zero processing times, and they were capable of solving instances with up to 15 jobs. Ovacikt and Uzsoy (1994) and Shin et al. (2002) minimized the maximum lateness ($1|r_j, s_{ij}|L_{max}$) by means of a decomposition heuristic and a tabu search (TS) algorithm, respectively. Chang et al. (2004) implemented a heuristic approach for the problem of minimizing the total weighted tardiness ($1|r_j, s_{ij}|\sum w_j T_j$), while Chou et al. (2008) minimized the weighted completion time ($1|r_j, s_{ij}|\sum w_j C_j$) via a MILP formulation and a B&B procedure capable of solving up to 40-job instances. The problem of minimizing the total tardiness ($1|r_j, s_{ij}|\sum T_j$) was tackled by Nogueira et al. (2022) with an algorithm that combined variable neighborhood search (VNS) and Lagrangian relaxation.

Scheduling problems with release dates and sequence-dependent setup times on parallel machine environments have also been addressed in the literature. Logendran et al. (2007) studied the problem of minimizing the weighted tardiness on unrelated parallel machines ($R|r_j, s_{ijk}|w_j T_j$) with a TS-based algorithm. Gharehgozli et al. (2009) proposed a mixed-integer goal programming (MIGP) model to minimize the total weighted flow time and the total weighted tardiness simultaneously, whereas Lin and Hsieh (2014) and Diana et al. (2018) tackled problem $R|r_j, s_{ijk}|w_j T_j$ by means of metaheuristic-based algorithms. Ekici et al. (2019) proposed a mathematical formulation and some heuristic algorithms to minimize total earliness and tardiness considering the assembly line of a TV manufacturing company. Moreover, Pereira Lopes and Valério de Carvalho (2007) implemented a branch-and-price algorithm for problem $R|a_k, r_j, s^k_{ij}|\sum w_j T_j$ (a_k corresponds to the availability date of machine k) capable of solving instances with up to 150 jobs and 50 machines. Note that the characteristics of the instances of larger size considered by the authors imply in very few jobs per machine, thus explicitly favoring the performance of the pricing procedure.

Recently, Bulhões et al. (2020) proposed a branch-cut-and-price algorithm for a large class of parallel machine scheduling problems that includes a general case, more precisely, problem $R|r_j, s^k_{ij}|\sum f_j(C_j)$, where $f(C_j)$ is a generic function in terms of the completion time C_j of a job j, as well as its particular variants. The authors separated limited-memory rank-1 Chvátal-Gomory cuts to improve the lower bound (LB) values, and were capable of finding the optimal solutions of many open problems. It is worth mentioning that we attempted to replicate this approach in our implementation, but preliminary experiments demonstrated that such cuts do not seem to be effective for problem $1|r_j, s_{ij}|C_{max}$ as they are for the scheduling problems addressed by Bulhões et al. (2020). In view of this, we adopted a different mechanism to improve the LBs in the nodes of the branch-and-bound tree. In our case, the bounds are improved iteratively due to the ng-memory augmentation procedure that is run every time the column generation algorithm converges, as thoroughly explained in Section 4.

3. Problem definition

Let $J = \{1, 2, \ldots, n\}$ be the set of n jobs to be scheduled on a single machine. Each job $j \in J$ has a defined release date r_j and processing time p_j. The notation s_{ij} represents the non-anticipatory setup time incurred if job j is processed immediately after job i. If j is the first job in the schedule, an initial setup time denoted as s_{0j} is spent preparing the machine. Since setup times are non-anticipatory, the preparation of the machine can only start after the release date of the job to be processed. Setup times are assumed to be asymmetric, and preemptions are not allowed.

A solution to the problem is defined by a sequence $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ in which each job is processed exactly once. The completion time of the initial job j is given by $C_j = r_j + s_{0j} + p_j$. For the remaining jobs the completion time is given by $C_j = \max(C_i, r_j) + s_{ij} + p_j$, where i is the last job processed before j. The objective is to minimize the makespan or the maximum completion time of the entire set of jobs, denoted as $C_{max} = \max_{j \in J}\{C_j\}$.

Table 1
Example of problem instance with $n = 5$.

j	r_j	p_j
1	1	1
2	3	2
3	10	1
4	2	2
5	4	2

s_{ij}	1	2	3	4	5
0	2	5	1	5	7
1	–	9	8	1	5
2	4	–	6	5	3
3	1	3	–	8	2
4	10	2	3	–	7
5	8	1	5	7	–

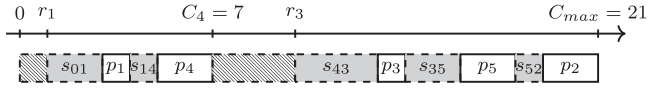


Fig. 1. Example of a solution $s = (1, 4, 3, 5, 2)$.

Table 1 contains the data of a 5-job instance of problem $1|r_j, s_{ij}|C_{max}$, while Fig. 1 shows a solution for this instance. The setups and processing times are represented by gray and white rectangles, respectively, whereas hatched rectangles represent idle time.

Job 1 ($p_1 = 1$) is the first to be processed. Since the setups are non-anticipatory, the initial setup ($s_{01} = 1$) starts only after the release date $r_1 = 1$ of such job. Job 4 ($p_4 = 2$) is the second in the sequence with the associated setup time $s_{14} = 1$. The third one to be processed is job 3 ($p_3 = 1$), however, the completion time of job 4 ($C_4 = 7$) is smaller than the release date $r_3 = 10$ of job 3, causing the machine to stay idle for 3 time units. Once the idle period is over, a setup time $s_{43} = 3$ is incurred. Moreover, a setup time $s_{35} = 2$ is required to process the next job, i.e., 5 ($p_5 = 2$), followed by a setup time $s_{52} = 1$ necessary to process the last job, i.e., 2 ($p_2 = 2$). Note that no idle times were further incurred because $C_3 = 14 > r_5 = 4$ and $C_5 = 18 > r_2 = 3$. The value of $C_{\max}$ will correspond to C_2, which is given by the sum of all processing times ($1+2+1+2+2 = 8$), the required setup times ($2+1+3+2+1 = 9$) and idle times (4). Therefore, we have that $C_{\max} = C_2 = 8 + 9 + 4 = 21$.

4. Exact approach

This section presents the proposed B&P algorithm for problem $1|r_j, s_{ij}|C_{max}$. More specifically, it describes the underlying arc-time formulations, as well as the main components of our exact approach, such as the ng-path relaxation, node resolution, the procedures implemented to solve the pricing and master problems, and the memory augmentation scheme.

4.1. Mathematical formulations

Let $J_0 = J \cup \{0\}$ (0 is a dummy job) and $G = (V = R \cup O, A = A^1 \cup A^2 \cup A^3 \cup A^4)$. Sets $R = \{(j,t) : j \in J, t = \bar{s}_j + r_j + p_j, \dots, T\}$ and $O = \{(0,t) : t = 0, \dots, T\}$ consist of vertices associated with jobs $j \in J$ and 0, respectively, where $\bar{s}_j = \min_{i \in J_0 \setminus \{j\}}\{s_{ij}\}$ and T is an upper bound (UB) on the maximum completion time of a job in some optimal solution. When the path arrives at (j,t), one assumes that both the setup operations and the processing of job j have already been carried out. However, j could have been completed in a time $t' < t$ because of a possible idle time after its processing.

Define sets $A^1 = \{((i,t),(j,t+s_{ij}+p_j)) : (i,t) \in R, (j,t+s_{ij}+p_j) \in R, t \geq r_j, i \neq j\}$, $A^2 = \{((0,t),(j,t+s_{0j}+p_j)) : (0,t) \in O, (j,t+s_{0j}+p_j) \in R, t \geq r_j\}$, $A^3 = \{((j,t),(0,T)) : (j,t) \in R\}$, and $A^4 = \{((j,t),(j,t+1)) : (j,t) \in V, (j,t+1) \in V\}$. The first three respectively contain the arcs connecting the vertices of R, from 0 to R, and from R to O, whereas the last one consists of the arcs associated with idle times. For each subset $S \subseteq V$, let $\delta^-(S)$ and $\delta^+(S)$ be the sets representing the arcs entering and leaving S, respectively. Define set $R^j = \{(i,t) \in R : i = j\}$ as the one related to job j. Let $A_j = \delta^-(R^j) \setminus A^4$ be the arc set representing the processing of job j. For an arc $a = ((i,t),(j,t+s_{ij}+p_j)) \in A_j$, in which the completion time of job j occurs at time $t + s_{ij} + p_j$, define $c_a = t + s_{ij} + p_j$ as its cost, while for the remaining arcs $a \in A^3 \cup A^4$, set $c_a = 0$. Finally, let x_a be an integer variable that counts the number of times arc $a \in A$ appears in the solution. The proposed arc-time-indexed formulation is as follows.

$$\text{(F1)} \quad \min \alpha \tag{1}$$

$$\text{s.t.} \quad \sum_{a \in A_j} x_a = 1, \qquad \forall j \in J \tag{2}$$

$$\sum_{a \in A^2} x_a = 1 \tag{3}$$

$$\sum_{a \in \delta^-(\{v\})} x_a - \sum_{a \in \delta^+(\{v\})} x_a = 0, \qquad \forall v \in V \setminus \{(0,0),(0,T)\} \tag{4}$$

$$\alpha \geq \sum_{a \in A_j} c_a x_a, \qquad \forall j \in J \tag{5}$$

$$\boldsymbol{x} \geq 0, \tag{6}$$

$$\boldsymbol{x} \text{ integer} \tag{7}$$

Fig. 2 shows the network representation of the solution depicted in Fig. 1, with the following active arcs: ((0,0),(0,1)), ((0,1),(1,4)), ((1,4),(4,7)), ((4,7),(4,8)), ((4,8),(4,9)), ((4,9),(4,10)), ((4,10),(3,14)), ((3,14),(5,18)), ((5,18),(2,21)), ((2,21),(0,21)).

Let $\mathcal{P}$ be the set of paths of graph G that start at vertex $(0,0)$ and end at vertex $(0,T)$. In addition, let b_a^P be the number of times the arc $a \in A$ is traversed by path P, which can be 0 or 1. Let λ_P be a binary variable that assumes value 1 if and only if P is in the solution. Note that variables λ and x satisfy Eq. (8) and that formulation (F2) can be obtained from a Dantzig–Wolfe decomposition of (F1):

$$x_a = \sum_{P \in \mathcal{P}} b_a^P \lambda_P \tag{8}$$

$$\text{(F2)} \quad \min \alpha \tag{9}$$

$$\text{s.t.} \quad \sum_{P \in \mathcal{P}} \Big(\sum_{a \in A_j} b_a^P \Big) \lambda_P = 1, \qquad \forall j \in J \tag{10}$$

$$\alpha \geq \sum_{a \in A_j} c_a \Big(\sum_{P \in \mathcal{P}} b_a^P \lambda_P \Big), \qquad \forall j \in J \tag{11}$$

$$\sum_{P \in \mathcal{P}} \lambda_P = 1 \tag{12}$$

$$\lambda \geq 0, \tag{13}$$

$$\lambda \text{ integer}. \tag{14}$$

4.2. ng-path relaxation

A path $P \in \mathcal{P}$ cannot contain cycles in the classical sense of the word because graph G is acyclic. Nevertheless, such path can traverse more than one arc representing the processing of the same job j, that is, more than one arc of set A_j. In this case, we assume that P has a *cycle* on job j. Of course, if P contains a cycle, it would not be part of an integer solution of (F2), but its existence weakens the linear relaxation of the formulation. Therefore, the more cycles are avoided, the better for the formulation in terms of linear relaxation.

Avoiding cycles in paths is a problem widely studied in the field of vehicle routing, and many approaches have been proposed over the

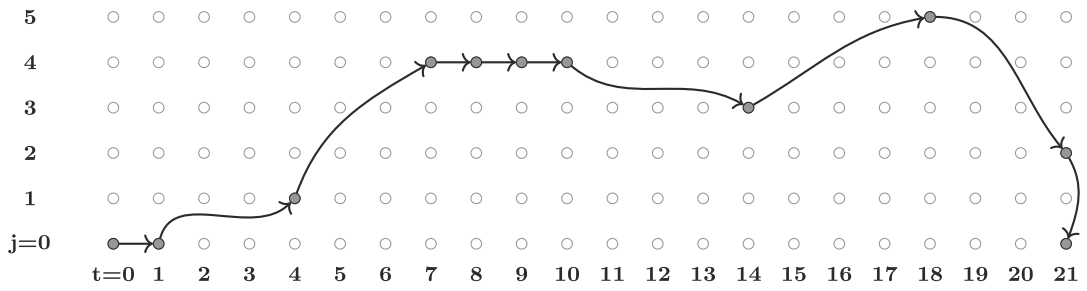


Fig. 2. Path representation of the optimal solution attained by F1 for the example instance in the layered network.

last few years. One of the most prominent and that offers the best compromise in terms of linear relaxation and CPU time is the so-called *ng-path relaxation* by Baldacci et al. (2011). In an ng-path, the cycling possibility is controlled by a memory mechanism, which is detailed in the following.

For each arc $a \in A$, one defines a memory $M_a \subseteq J$ containing the jobs that arc a can remember. Hence, an ng-path can only have a cycle associated with a certain job j if there is an arc a that does not remember job j in between two arcs representing the processing of j, i.e., $j \notin M_a$. For example, consider a path $P \in \mathcal{P}$ that traverses arcs $(a_1, \ldots, a_m)$. This path is *ng-feasible* iff there are no two arcs a_p and a_q, $p < q$, traversed by it and job $j \in J$ such that:

(I) $a_p, a_q \in A_j$, i.e., both arcs represent the processing of job j;
(II) $j \in M_{a_k}, \forall k \in \{p+1, \ldots, q-1\}$.

Condition (II) states that all intermediate arcs are capable of remembering the processing of job j, thus forbidding arc a_q to be traversed. Note that this condition does not consider a cycle associated with a job j composed of two consecutive arcs, but such cycle does not exist because of the structure of graph G. Moreover, it is worth mentioning that if $M_a = J$ for every arc $a \in A$, then only the paths that do not contain a cycle would be ng-feasible. However, the larger the memories, the larger the computational effort. In the proposed algorithm, the memories are dynamically defined throughout the optimization, in the spirit of the works by Bulhões et al. (2018), Roberti and Mingozzi (2014).

Hereafter, $\mathcal{P}(\mathcal{M}) \subseteq P$ denotes the set of paths that are ng-feasible with respect to the collection of memories $\mathcal{M}$ (one memory set M_a for each arc $a \in A$) and F2($\mathcal{M}$) represents the formulation obtained from (F2) using $\mathcal{P}(\mathcal{M})$ instead of $\mathcal{P}$.

4.3. Node resolution

The node resolution of our B&P algorithm is performed by an iterative approach described in Algorithm 1. In the *k-th* iteration, one employs *column generation* (CG) to solve an LP corresponding to an extension of the linear relaxation of formulation F2($\mathcal{M}_k$), denoted by $LP(\mathcal{M}_k)$, that includes the possible branching constraints (disregarding the root node). When the CG converges, one executes a reduced cost fixing procedure that is capable of eliminating the arcs of the pricing network. Finally, the memories are incremented so as to forbid some cycles found in the paths that make up the primal solution obtained for the LP.

4.3.1. Pricing problem and reduced cost fixing

Let μ_j and ν_j be the values of the dual variables of constraints (10) and (11), respectively, both associated with job j. The contribution given by an arc $a \in A_j$ to the reduced cost of a path is given by:

$$rc(a) = -\mu_j + c_a \nu_j \tag{15}$$

Algorithm 1 Node resolution

```
1: procedure SolveNode(node, θ, β)
2: if node is the root node then
3:     M_1 ← empty memories
4: else
5:     M_1 ← final memories of parent node
6: k ← 1
7: repeat
8:     Compute an optimal primal–dual solution pair of LP(M_k) + branching
       constraints (if any)
9:     Perform reduced cost fixing
10:    Let P' be the set of ng-paths associated with the primal solution λ̄^k
11:    M_{k+1} ← augmentNgMemories(P', M_k, θ, β)
12: until stopping criterion is met
```

Let ψ be the value of the dual variable associated with constraint (12). The reduced cost of a path $P \in \mathcal{P}$ can be expressed as follows:

$$rc(P) = \sum_{j \in J} \sum_{a \in A_j} b_a^P rc(a) - \psi \tag{16}$$

The pricing subproblem consists in finding an ng-path $P \in \mathcal{P}(\mathcal{M}_k)$ with minimum reduced cost. This subproblem is solved via a labeling algorithm, which in principle is a path enumeration. In our algorithm, a label $L = (j(L), t(L), rc(L), \Pi(L))$ represents a partial path on graph G that ends at vertex $(j(L), t(L))$ with associated reduced cost $rc(L)$. Furthermore, the label L also stores the set of jobs $\Pi(L) \subseteq J$ that have already been processed and that still were not "forgotten". This set is useful to determine the forbidden extensions for this label. This set is empty for the initial label, that is, the one whose path contains only the initial vertex $(0, 0)$. When a label L' is obtained from the extension of another label L over an arc $a \in A_j$ with memory set M_a, we have that:

$$\Pi(L') = (\Pi(L) \cap M_a) \cup \{j\} \tag{17}$$

Dominance rules are used to eliminate unpromising labels. In our case, a label L dominates a label L' if: (I) $j(L) = j(L')$; (II) $t(L) = t(L')$; (III) $rc(L) \leq rc(L')$; (IV) $\Pi(L) \subseteq \Pi(L')$.

We adopt the same implementation of the bidirectional labeling employed in Bulhões et al. (2020). Three phases are considered in this implementation. In the first one, denoted as forward labeling, one finds forward labels representing the paths that traverse the arcs in the normal direction starting at vertex $(0, 0)$. In the second phase, denoted as backward labeling, one finds backward labels representing the paths that traverses the arcs in the opposite direction starting at vertex $(0, T)$. The forward (backward) labeling only looks for labels ending at a vertex (j, t) such that $t \leq T^*$ ($t \geq T^*$), where T^* is a special value that limits the border of the bidirectional algorithm. In the last phase, *ng*-paths are generated by concatenating a forward and a backward label that end at the same vertex (j, t). The partial paths found during the search are stored in a data structure called *bucket*. Two buckets $\overrightarrow{B}(j, t)$ and $\overleftarrow{B}(j, t)$ are assigned to each vertex (j, t) of the pricing network, and they store the forward and backward labels, respectively, associated

with that vertex. For the label extensions, the forward (backward) buckets are visited in increasing (decreasing) order of time t. A new label is only inserted in its respective bucket if there is no other label in the bucket that dominates such label. In case the insertion is performed, all labels that become dominated in the bucket are removed.

In the reduced cost fixing phase, one executes the forward and backward labeling approaches without considering the border T^*, i.e., for the entire time horizon. Therefore, it is possible to determine the least-cost path that traverses a given arc of the pricing network. If the reduced cost is larger than the gap between the current lower and upper bounds, the arc can be removed from the network. The reader is referred to Bulhões et al. (2020) for a detailed description of the pricing algorithm and the reduced cost fixing procedure.

4.3.2. Solving the master problem

The master problem is solved by a two-stage CG procedure. In the first one, the labeling algorithm is executed in a heuristic fashion by modifying the dominance test of the labels. More precisely, one ignores condition (IV) of the dominance test, meaning that each bucket will keep only the label containing the best reduced cost regardless of the constraints imposed by the ng-path relaxation. If it is not possible to find a negative reduced cost path in the first stage, the procedure moves to the second stage where the labeling algorithm is executed exactly until the end of the CG. At each iteration of the first (second) stage, one includes at most 30 (150) ng-paths. Regardless of the stage, one utilizes the dual stabilization strategy proposed in Pessoa et al. (2018) to speed up the CG convergence.

4.3.3. Memory augmentation

Once again consider a path $P \in \mathcal{P}$ that traverses arcs $(a_1, \dots, a_m)$ and also arcs a_p and a_q, $p < q$, the latter two representing the processing of a same job j, that is, $a_p, a_q \in A_j$. To forbid this cycle, one should add j to the memory of arcs $a_k, \forall k \in \{p+1, \dots, q-1\}$. Nevertheless, it is possible to observe that the set of ng-paths $\mathcal{P}'$ might contain many cycles, so forbidding all of them may rapidly increase the average size of the memories and consequently the pricing time. To overcome this issue, we have adopted the same two-stage algorithm proposed in Bulhões et al. (2018) for determining which cycles are worth forbidding. The ng-paths in $\mathcal{P}'$ are traversed in decreasing order according to the value of the corresponding variable in the primal solution. For each ng-path $P \in \mathcal{P}'$, the algorithm executes up to two stages. In the first one, one forbids all cycles of P that contain up to θ arcs. The idea is to prioritize short cycles because there is a relatively small increase in the memories when forbidding them. If there is no cycle that can be forbidden in the first stage, the algorithm proceeds to the second stage in which one forbids the smallest cycle of P regardless of its size. The algorithm terminates when β ng-paths have been forbidden. In our case, we have adopted $\theta = 5$ and $\beta = 100$.

4.4. Branching

We employ a two-stage strong branching scheme similar to the one implemented in Røpke (2012). Each branching candidate corresponds to one of the precedence variables $y_{ij} = \sum_{a \in A_j \cap \delta^+(R^i)} x_a$ that determine whether a job $i \in J_0$ precedes a job $j \in J$ in the solution. In the first phase, one analyzes up to 100 candidates, half of them coming from the branching history (those with the best pseudocosts (Achterberg, 2007)) and the other half composed of variables y_{ij} whose values are closest to 0.5 in the current fractional solution. Hence, for each candidate, the procedure roughly evaluates the (LB) increase of each child node by solving the linear program (LP) of the restricted master problem (RMP) with a new branching constraint but without using column generation. Each candidate is then evaluated by the rule proposed in Achterberg (2007) that considers the product of the LB increase of the two child nodes, and the 20 best candidates are selected for phase 2. In this latter phase, the evaluation of the child nodes is more precise because one performs a column generation in a heuristic fashion. Finally, the three candidates are reevaluated with the same product rule based on the new LBs, and the best one is selected for branching.

5. Heuristic approach

The proposed heuristic approach, here denoted as ILS-BS, brings together elements of the ILS and BS methods, and its pseudocode is presented in Algorithm 2. The heuristic receives as input five parameters: I_R is the number of restarts, I_{ILS} is the number of ILS iterations without improvement, $w \in \mathbb{Z}_{>0}$, $N \in \mathbb{Z}_{>0}$ and $\gamma \in [0, 1]$ are parameters used in the constructive BS procedure. At each start, an initial solution is generated by BS (line 4) and iteratively improved by combining local search and perturbation procedures (lines 7–13). The algorithm returns the best solution found among all restarts (lines 14–17).

Algorithm 2 ILS-BS

```
1:  procedure ILS-BS(I_R, I_ILS, w, N, γ)
2:      f* ← ∞
3:      for i ← 1, ..., I_R do
4:          s ← BeamSearch(w, N, γ)
5:          s' ← s
6:          IterILS ← 0
7:          while IterILS < I_ILS do
8:              s ← LocalSearch(s)
9:              if f(s) < f(s') then
10:                 s' ← s
11:                 IterILS ← 0
12:             s ← Perturbation(s')
13:             IterILS ← IterILS + 1
14:         if f(s') < f(s*) then
15:             s* ← s'
16:             f* ← f(s')
17:     return s*
```

5.1. Constructive procedure

The BS procedure used to generate initial solutions is mostly based in Velez-Gallego et al. (2016). At first, a root node associated with an empty sequence π_0 is initialized with all decision variables free. Next, the BS tree is expanded one level at a time, from zero to $n-1$. At each level, the best N nodes are selected for branching according to Eq. (18), where $\pi = (\pi_1, \dots, \pi_{|\pi|})$ is the sequence of the node, $U(\pi)$ is the set of remaining jobs and $Q(\pi) = U(\pi) \cup \pi_{|\pi|}$. At most w branches are created for each node by fixing a job j from $U(\pi)$ at the end of a partial solution, with all resulting nodes evaluated according to their associated sum of idle and setup times, denoted as I.

$$LB(\pi) = \max\{C(\pi)\ ,\ \min_{j \in U(\pi)}\{r_j\}\} + \sum_{k \in U(\pi)} \min_{t \in Q(\pi)}\{s_{tk}\} + \sum_{k \in U(\pi)} p_k \tag{18}$$

Note that the original BS algorithm is deterministic. In our implementation, instead of selecting the best N nodes, we randomly choose N nodes among the best $\lfloor (1+\gamma)N \rfloor$ ones. This modification was performed to add a certain degree of diversity to the initial solutions.

Fig. 3 shows an example of a BS tree for the instance presented in Table 1, with parameters $w = 2$, $N = 3$, and $\gamma = 0.5$. This example is complemented by Table 2, which presents the node evaluations for each level of the tree. In the latter table, eliminated nodes due to the value of I (maximum of $w = 2$ child nodes per parent) are highlighted in gray, whereas strikethrough entries indicate the nodes that have been eliminated because of the LB value (maximum of $N = 3$ nodes per level).

Starting at the root node (level 0), one can generate n different nodes in level 1 by fixing one job j at the beginning of the sequence. Among the $n = 5$ possibilities, only $w = 2$ nodes with the least I values are kept in the tree, i.e., π_1 and π_4. In level 2, we show the best two child nodes per parent selected according to the best values of I. When the number of nodes exceeds $N = 3$, they are sorted in non-decreasing order according to the LB value, and then only $N = 3$ are selected at random from a candidate list containing the best $\lfloor (1+\gamma)N \rfloor = \lfloor (1+0.5)3 \rfloor = 4$ nodes. In level 3, one can observe that

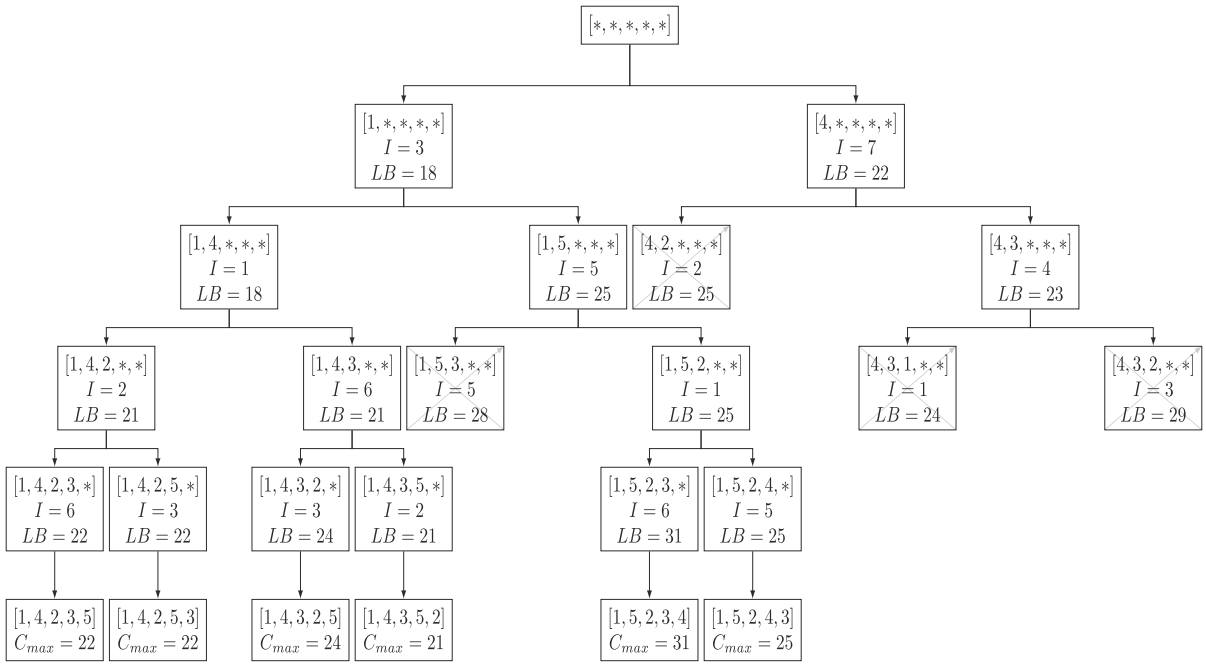


Fig. 3. Beam search tree for the instance presented in Table 1, with $w = 2$, $N = 3$ and $\gamma = 0.5$.

6 nodes passed through the first selection stage and again 3 nodes are selected for branching. The candidate list contains the best 4 candidates (π_{17}, π_{18}, π_{14}, and π_{20}) and then 3 nodes were chosen at random (π_{17}, π_{18}, and π_{20}). In level 4, all nodes generated from the previous level contain $n-1$ jobs, thus one can determine the makespan of the complete solution via the LB value. All final solutions are represented in level 5, and the one associated with the best makespan is selected as the initial solution.

5.2. Local search

The local search procedure is based on the randomized variable neighborhood descent (RVND) procedure (Mladenovic & Hansen, 1997; Subramanian et al., 2010). Firstly, a list NL of neighborhood structures is initialized. A neighborhood from this list is then randomly selected, and all neighbor solutions are explored and evaluated. If the best current solution is not improved, then the neighborhood is removed from the list and another one is randomly chosen to be explored. In case of improvement, NL is restored with all the neighborhood structures.

5.2.1. Neighborhood structures

The following neighborhood structures were considered by our algorithm.

- *Swap*: exchanges two jobs of the sequence. Fig. 4 shows an example using the data from Table 1 in which jobs 5 and 4 from sequence $(1, 5, 3, 4, 2)$ are swapped, resulting in a new sequence $(1, 4, 3, 5, 2)$. It can be observed that an improved solution was obtained even though an idle time was introduced to satisfy the release date of job 3.
- *l-block insertion*: a block of l adjacent jobs is removed from its current position and reinserted in another position of the sequence. Fig. 5 depicts an example considering $l = 2$ also using the data provided in Table 1 in which the two last consecutive jobs of the same starting sequence $(1, 5, 3, 4, 2)$ are removed and reinserted immediately after job 1, yielding an improved sequence given by $(1, 4, 2, 5, 3)$. In this case, it was not necessary to include idle times to ensure feasibility. A distinct neighborhood is associated to each different value of l.

As mentioned above, one has to adjust the sequence to satisfy the job availability constraints by including an idle time in case the release date of a job has been violated. Therefore, evaluating a move can be time consuming if performed in a straightforward fashion, i.e., by checking if the release date of all jobs affected by a move is satisfied and introducing idle times if necessary. This can rapidly compromise the performance of the method in terms of CPU time as the size of the instance increases. The next section describes an efficient way of performing the move evaluation to overcome this issue.

5.2.2. Efficient move evaluation

The move evaluation process consists in computing the makespan of the resulting sequence produced by a move as an attempt to find an improvement on the current best solution. This could be done by simply iterating through the sequence and computing the completion time of each job, thus yielding a complexity of $\mathcal{O}(n)$. In this case, since the size of all neighborhood structures considered is $\mathcal{O}(n^2)$, the overall complexity of exploring and evaluating each neighborhood is $\mathcal{O}(n^3)$. Therefore, to reduce the computational complexity, we implement an efficient move evaluation scheme based on Kindervater and Savelsbergh (1997), Vidal et al. (2013) that allows the move to be evaluated in amortized $\mathcal{O}(1)$ operations, hence implying in a reduction of the neighborhood exploration complexity to $\mathcal{O}(n^2)$.

In this scheme, the evaluation of a move is made by concatenating job subsequences. For each subsequence $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_{|\sigma|})$ in a given solution, the following attributes are stored in an auxiliary data structure:

- $F(\sigma)$ = first job;
- $L(\sigma)$ = last job;
- $E(\sigma)$ = earliest time to start the processing without idle time;
- $D(\sigma)$ = sum of processing times and setup times.

Let $\sigma = (j)$ be a unitary subsequence composed of a job $j \in J$. In this case, $F(\sigma) = j$, $L(\sigma) = j$, $E(\sigma) = r_j$ and $D(\sigma) = p_j$. One can observe that any subsequence σ, $|\sigma| \geq 1$, can be derived from the concatenation of two other subsequences σ^1 and σ^2. This process is described in Algorithm Fig. 6. The makespan of any subsequence can be defined by $C(\sigma) = E(\sigma) + D(\sigma)$. However, the makespan of a solution s represented by subsequence $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_{|\sigma|})$ does not necessarily correspond to

Table 2
Detailed beam search nodes per level.

Level	Node	I	LB (18)	$C(\pi)$
1	$\pi_1 = [1,*,*,*,*]$	$r_1 + s_{01} = 3$	18	$I + p_1 = 4$
	$\pi_2 = [2,*,*,*,*]$	$r_2 + s_{02} = 8$	23	$I + p_2 = 10$
	$\pi_3 = [3,*,*,*,*]$	$r_3 + s_{03} = 11$	24	$I + p_3 = 12$
	$\pi_4 = [4,*,*,*,*]$	$r_4 + s_{04} = 7$	22	$I + p_4 = 9$
	$\pi_5 = [5,*,*,*,*]$	$r_5 + s_{05} = 11$	25	$I + p_5 = 13$
2	$\pi_6 = [1,2,*,*,*]$	$\max(r_2 - C(\pi_1), 0) + s_{12} = 9$	30	$C(\pi_1) + I + p_2 = 15$
	$\pi_7 = [1,3,*,*,*]$	$\max(r_3 - C(\pi_1), 0) + s_{13} = 14$	33	$C(\pi_1) + I + p_3 = 19$
	$\pi_8 = [1,4,*,*,*]$	$\max(r_4 - C(\pi_1), 0) + s_{14} = 1$	18	$C(\pi_1) + I + p_4 = 7$
	$\pi_9 = [1,5,*,*,*]$	$\max(r_5 - C(\pi_1), 0) + s_{15} = 5$	25	$C(\pi_1) + I + p_5 = 11$
	$\pi_{10} = [4,1,*,*,*]$	$\max(r_1 - C(\pi_4), 0) + s_{41} = 10$	33	$C(\pi_4) + I + p_1 = 20$
	~~$\pi_{11} = [4,2,*,*,*]$~~	~~$\max(r_2 - C(\pi_4), 0) + s_{42} = 2$~~	~~25~~	~~$C(\pi_4) + I + p_2 = 13$~~
	$\pi_{12} = [4,3,*,*,*]$	$\max(r_3 - C(\pi_4), 0) + s_{43} = 4$	23	$C(\pi_4) + I + p_3 = 14$
	$\pi_{13} = [4,5,*,*,*]$	$\max(r_5 - C(\pi_4), 0) + s_{45} = 7$	29	$C(\pi_4) + I + p_5 = 18$
3	~~$\pi_{14} = [4,3,1,*,*]$~~	~~$\max(r_1 - C(\pi_{12}), 0) + s_{31} = 1$~~	~~24~~	~~$C(\pi_{12}) + I + p_1 = 16$~~
	~~$\pi_{15} = [4,3,2,*,*]$~~	~~$\max(r_2 - C(\pi_{12}), 0) + s_{32} = 3$~~	~~29~~	~~$C(\pi_{12}) + I + p_2 = 19$~~
	$\pi_{16} = [4,3,5,*,*]$	$\max(r_5 - C(\pi_{12}), 0) + s_{35} = 2$	26	$C(\pi_{12}) + I + p_5 = 18$
	$\pi_{17} = [1,4,2,*,*]$	$\max(r_2 - C(\pi_8), 0) + s_{42} = 2$	21	$C(\pi_8) + I + p_2 = 11$
	$\pi_{18} = [1,4,3,*,*]$	$\max(r_3 - C(\pi_8), 0) + s_{43} = 6$	21	$C(\pi_8) + I + p_3 = 14$
	$\pi_{19} = [1,4,5,*,*]$	$\max(r_5 - C(\pi_8), 0) + s_{45} = 7$	25	$C(\pi_8) + I + p_5 = 16$
	$\pi_{20} = [1,5,2,*,*]$	$\max(r_2 - C(\pi_9), 0) + s_{52} = 1$	25	$C(\pi_9) + I + p_2 = 14$
	~~$\pi_{21} = [1,5,3,*,*]$~~	~~$\max(r_3 - C(\pi_9), 0) + s_{53} = 5$~~	~~28~~	~~$C(\pi_9) + I + p_3 = 17$~~
	$\pi_{22} = [1,5,4,*,*]$	$\max(r_4 - C(\pi_9), 0) + s_{54} = 7$	28	$C(\pi_9) + I + p_4 = 20$
4	$\pi_{23} = [1,5,2,3,*]$	$\max(r_2 - C(\pi_{20}), 0) + s_{23} = 6$	31	$C(\pi_{20}) + I + p_3 = 21$
	$\pi_{24} = [1,5,2,4,*]$	$\max(r_5 - C(\pi_{20}), 0) + s_{24} = 5$	25	$C(\pi_{20}) + I + p_4 = 21$
	$\pi_{25} = [1,4,3,2,*]$	$\max(r_2 - C(\pi_{18}), 0) + s_{32} = 3$	24	$C(\pi_{18}) + I + p_2 = 19$
	$\pi_{26} = [1,4,3,5,*]$	$\max(r_5 - C(\pi_{18}), 0) + s_{35} = 2$	**21**	$C(\pi_{18}) + I + p_5 = 18$
	$\pi_{27} = [1,4,2,3,*]$	$\max(r_3 - C(\pi_{17}), 0) + s_{23} = 6$	22	$C(\pi_{17}) + I + p_3 = 18$
	$\pi_{28} = [1,4,2,5,*]$	$\max(r_5 - C(\pi_{17}), 0) + s_{25} = 3$	22	$C(\pi_{17}) + I + p_5 = 16$
5	$\pi_{29} = [1,5,2,3,4]$	$\max(r_5 - C(\pi_{23}), 0) + s_{34} = 8$	–	$C(\pi_{23}) + I + p_4 = 31$
	$\pi_{30} = [1,5,2,4,3]$	$\max(r_2 - C(\pi_{24}), 0) + s_{43} = 3$	–	$C(\pi_{24}) + I + p_3 = 25$
	$\pi_{31} = [1,4,3,2,5]$	$\max(r_5 - C(\pi_{25}), 0) + s_{25} = 5$	–	$C(\pi_{25}) + I + p_5 = 24$
	$\boldsymbol{\pi_{32} = [1,4,3,5,2]}$	$\max(\mathbf{r_2 - C(\pi_{26}), 0}) + \mathbf{s_{52}} = \mathbf{3}$	–	$\mathbf{C(\pi_{26}) + I + p_2 = 21}$
	$\pi_{33} = [1,4,2,3,5]$	$\max(r_5 - C(\pi_{27}), 0) + s_{35} = 4$	–	$C(\pi_{27}) + I + p_5 = 22$
	$\pi_{34} = [1,4,2,5,3]$	$\max(r_3 - C(\pi_{28}), 0) + s_{53} = 6$	–	$C(\pi_{28}) + I + p_3 = 22$

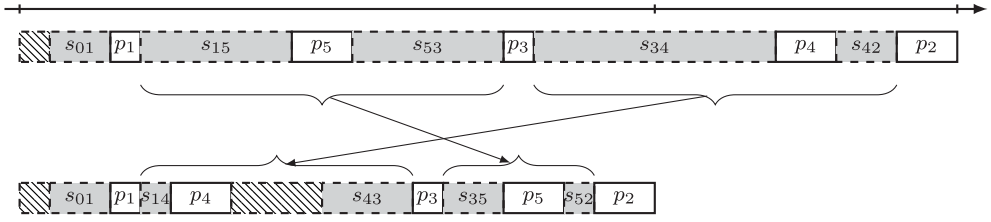


Fig. 4. Swapping jobs 5 and 4.

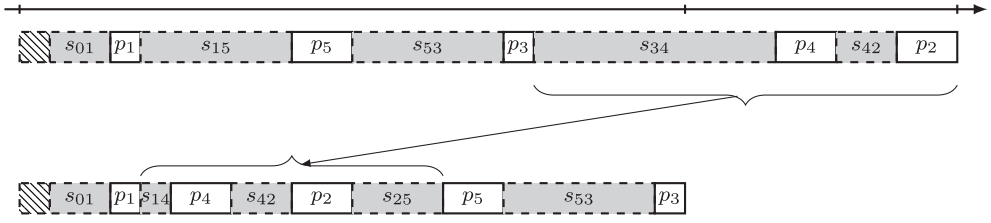


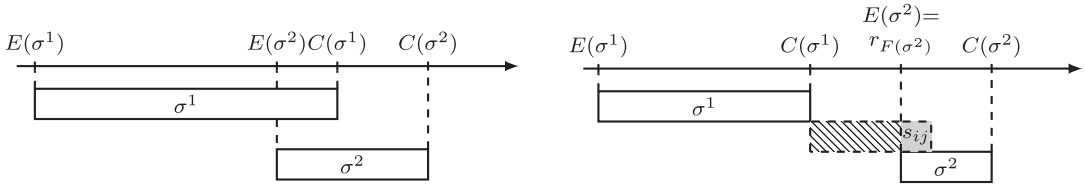
Fig. 5. l-block insertion: $l = 2$, block $(4,2)$ inserted in position $i = 1$.

$C(\sigma)$, since the initial setup time is not considered. The actual makespan of s is equal to $C(\sigma' \oplus \sigma)$, where σ' is a unitary subsequence containing a dummy job 0 with $r_0 = 0$ and $p_0 = 0$.

When performing the operation $\sigma^1 \oplus \sigma^2$, there are different cases concerning the idle times. In the first case, as shown in Fig. 6(a), the minimum time possible to conclude the first subsequence $C(\sigma^1)$ is greater than or equal to the earliest time $E(\sigma^2)$ to start σ^2 without producing idle time. In the remaining scenarios the subsequence σ^1 is concluded before $E(\sigma^2)$. In the scenario shown in Fig. 6(b), the machine stays idle between the completion of the last job of σ^1 (i) and the beginning of σ^2. However, the idle time is reduced if the setup of the first job of σ^2 (j) can be processed before the earliest time to start the sequence, as shown in Fig. 6(c).

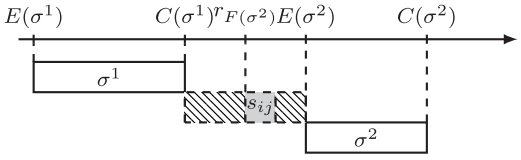
5.3. Perturbation mechanism

If the local search procedure fails to improve the current best solution, a perturbation is applied to the best solution of the current iteration. More specifically, the perturbation consists in exchanging two disjoint subsequences selected at random. This is similar to the double bridge procedure developed by Martin et al. (1991) for the traveling salesman problem.



(a) Concatenation with $C(\sigma^1) \geq E(\sigma^2)$. The resulting idle time is derived from σ^1 only.

(b) A concatenation case with $C(\sigma^1) \leq E(\sigma^2)$: the setup between the last job of σ^1 and the first job of σ^2 cannot be processed before $E(\sigma^2)$.



(c) A different concatenation case with $C(\sigma^1) \leq E(\sigma^2)$: setup between $L(\sigma^1)$ and $F(\sigma^2)$ can be processed before $E(\sigma^2)$, reducing idle time. Note that one of the idle periods (after or before the setup) can be null.

Fig. 6. Possible scenarios for subsequence concatenation.

Algorithm 3 Subsequence Concatenation $\sigma^1 \oplus \sigma^2$

procedure CONCATENATION(σ^1, σ^2)
  $F(\sigma^1 \oplus \sigma^2) \leftarrow F(\sigma^1)$;
  $L(\sigma^1 \oplus \sigma^2) \leftarrow L(\sigma^2)$;
  $I \leftarrow 0$;
  $i, j \leftarrow L(\sigma^1), F(\sigma^2)$;
  if $C(\sigma^1) < E(\sigma^2)$ **then**
    if $C(\sigma^1) < r_j$ **then**
      $I \leftarrow r_j - C(\sigma^1)$;
    if $C(\sigma^1) + I + s_{ij} < E(\sigma^2)$ **then**
      $I \leftarrow E(\sigma^2) - (C(\sigma^1) + s_{ij})$;
  $E(\sigma^1 \oplus \sigma^2) \leftarrow E(\sigma^1) + I$;
  $D(\sigma^1 \oplus \sigma^2) \leftarrow D(\sigma^1) + D(\sigma^2) + s_{ij}$;
  return $\sigma^1 \oplus \sigma^2$

6. Computational experiments

The proposed algorithms were coded in C++ (g++ 5.4.0) and the heuristic experiments were run on an Intel i7-2600 with 3.40 GHz and 16 GB of RAM running Linux Ubuntu 16.04 64 bits, whereas the B&P experiments were executed on a 2x 12-core Haswell Intel Xeon E5-2680 v3 server with 2.50 GHz and 128 GB of RAM running Linux CentOS 7. The latter algorithm was implemented over the BapCod platform (Vanderbeck et al., 2017) and the results were compared with those attained by the MILP model presented in Velez-Gallego et al. (2016). We executed both exact methods on the same machine with a time limit of 8 h and CPLEX 20.1 was used as LP (for the B&P) and MILP solver. The values of the best known UBs (including those found by our heuristics) were provided as initial primal bounds for the exact approaches.

The algorithms were tested on a set of 1800 instances proposed in Ovacikt and Uzsoy (1994). These instances are divided by the number of jobs $n \in \{25, 50, 75, 100, 125, 150\}$ and a range factor $R \in \{0.2, 0.4, 1.0, 1.4, 1.8\}$ that controls the dispersion of the release dates over time.

6.1. Results for the exact algorithm

Table 3 presents the average results obtained by the proposed exact algorithm aggregated by each group of instances divided by all possible combinations of n and R. For each group, we report the average values obtained by our B&P algorithm associated with the: *root gap*; *root time*; and number of *nodes* of the B&P tree. We also report the number of improved UBs (*imp_ub*) attained by our B&P when compared to the best known solution (BKS) considering all 5 heuristics compared in this work and the MILP formulation presented in Velez-Gallego et al. (2016). Moreover, for both our B&P and the MILP model, we report the average *gap*, the number of instances *solved* to optimality, and the average CPU *time* in seconds. It is worth noting that all gap calculations are with respect to the LB, i.e., $gap = 100 \times ((UB - LB)/LB)$. Detailed results achieved by our B&P for each individual instance are available at: https://github.com/rsobralm/SMRDSDST.

The results show that our B&P algorithm is capable of systematically obtaining very promising dual bounds, as opposed to the MILP model, which in turn achieved higher gaps in almost all groups of instances, except for two cases ($n = 150$, $R = 0.2$; and $n = 150$, $R = 1.8$). Those are the only groups where the gap obtained by the B&P is higher than 0.86%. In contrast, the MILP managed to achieve gaps lower than this value in only one group ($n = 25$, $R = 0.2$). A total of 768 (42.7%) instances were solved to optimality by B&P, most of them at the root node where in general, very strong dual bounds were achieved, against only 101 (5.6%) obtained by the MILP model. In addition, B&P could solve 96.5% of the instances with $n \leq 50$, against 13.7% of the MILP formulation. Finally, B&P was capable of improving the UBs of 123 instances, most of them involving 50-job and 75-job instances.

Table 4 presents the number of best dual bounds found by B&P and MILP on each group of instances. The results show that B&P produced the best dual bounds for almost all instances. By examining the results, we can observe that B&P achieved the best dual values in 96.7% instances against 8.8% of MILP. Surprisingly, despite the vast superiority of B&P in obtaining stronger dual bounds than MILP, the latter is still not dominated by the former. In particular, MILP obtained strictly better dual bounds in 3.3% of the instances, the vast majority for $n = 150$ and $R = 0.2$.

6.2. Results for the heuristic algorithms

We compare the performance of our ILS-BS with the best-known heuristics for problem $1|r_j, s_{ij}|C_{max}$, namely the BS by Velez-Gallego et al. (2016), as well as the VBIH and IG algorithms implemented by Fan et al. (2019). We also include in the comparison a version of the ILS algorithm, denoted as GILS, that uses an alternative approach to generate initial solutions based on the constructive phase of the

Table 3
Aggregate results obtained by the proposed exact algorithm.

Group	B&P							MILP		
	Root gap	Root time	Nodes	Gap	Solved	imp_ub	Time	Gap	Solved	Time
25/02	0.0	23.79	1.03	0.0	60	0	23.91	0.24	55	4001
25/06	0.0	22.36	1.0	0.0	60	0	22.44	3.91	14	24044
25/10	0.0	26.43	1.0	0.0	60	0	26.44	8.2	3	28091
25/14	0.0	30.45	1.2	0.0	60	0	34.82	8.95	3	27950
25/18	0.01	41.39	2.5	0.0	60	0	118	9.75	0	28800
50/02	0.01	427	1.5	0.0	60	2	505	1.29	6	26330
50/06	0.08	578	2.7	0.0	60	7	2282	3.81	0	28800
50/10	0.12	651	2.97	0.0	59	14	2909	7.17	0	28800
50/14	0.11	751	7.47	0.02	56	13	6150	6.74	0	28800
50/18	0.16	1071	8.5	0.01	44	8	11 722	7.33	1	28320
75/02	0.11	4516	3.47	0.03	46	22	13 022	1.06	1	28497
75/06	0.19	4311	4.3	0.12	25	15	19 903	2.94	1	28510
75/10	0.45	4487	6.27	0.29	15	13	25 496	6.7	0	28800
75/14	0.36	5438	5.5	0.26	16	5	27 875	5.06	0	28800
75/18	0.33	5149	4.9	0.23	18	2	24 046	6.12	3	27362
100/02	0.22	16 221	1.7	0.18	17	15	24 789	0.95	0	28719
100/06	0.33	16 496	1.83	0.32	1	1	28 740	2.69	0	28800
100/10	0.54	15 281	2.27	0.52	1	2	28 657	5.54	0	28800
100/14	0.49	16 217	1.97	0.49	5	0	26 562	4.19	3	27624
100/18	0.47	11 700	2.33	0.45	15	0	22 805	5.34	3	27362
125/02	0.34	28 191	1.0	0.34	4	4	28 191	0.88	0	28800
125/06	0.42	28 750	1.0	0.42	0	0	28 800	2.29	0	28800
125/10	0.73	28 543	1.0	0.73	0	0	28 800	5.35	0	28800
125/14	0.76	27 308	1.0	0.76	3	0	27 658	3.62	2	27842
125/18	0.51	24 193	1.0	0.51	12	0	24 717	4.39	4	26405
150/02	1.6	28 800	1.0	1.6	0	0	28 800	0.78	0	28459
150/06	0.57	28 800	1.0	0.57	0	0	28 800	2.13	0	28800
150/10	0.86	28 800	1.0	0.86	0	0	28 800	4.42	0	28800
150/14	0.84	28 496	1.0	0.84	4	0	28 496	2.98	0	28321
150/18	5.69	27 462	1.0	5.69	7	0	27 462	4.2	2	27561

Table 4
Number of best known dual bounds found.

R	$n = 25$		$n = 50$		$n = 75$		$n = 100$		$n = 125$		$n = 150$	
	B&P	MILP	B&P	MILP	B&P	MILP	B&P	MILP	B&P	MILP	B&P	MILP
0.2	60	55	60	6	59	1	59	1	60	0	26	34
0.6	60	14	60	0	59	1	60	0	60	0	58	2
1.0	60	3	60	0	60	0	60	0	60	0	60	0
1.4	60	3	60	0	60	0	58	4	57	5	56	5
1.8	60	0	60	1	60	3	59	4	59	5	51	11

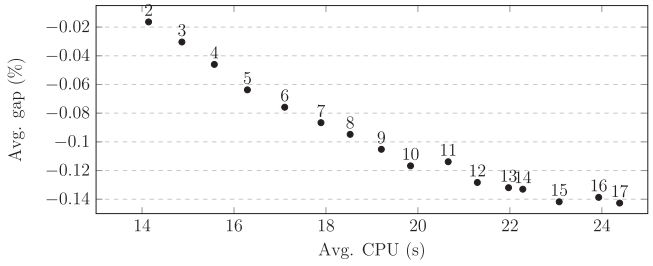


Fig. 7. Average Gap and CPU time for each value of l.

Table 5
Average Gaps per γ.

γ	0.00	0.01	0.05	0.10	0.50	1.00
Gap	−0.08	−0.12	−0.06	0.01	0.22	0.22

greedy randomized adaptive search procedure (GRASP) (Feo & Resende, 1995). The method is similar to the one presented in Silva et al. (2012), with the candidate list sorted in non-decreasing order according to the release dates for the first job selection, and the induced idle time for the remaining jobs. In all experiments reported hereafter, the ILS-based methods were executed 10 times for each instance. Detailed results obtained by ILS-BS and GILS for each individual instance are reported at: https://github.com/rsobralm/SMRDSDST.

6.2.1. Parameter tuning

Parameters I_R and I_{ILS} were calibrated using the same procedure described in Silva et al. (2021), in which the authors proposed an efficient ILS algorithm for the problem of minimizing the makespan on an identical parallel machine environment with a common server and sequence-dependent setup times. As reported in that work, we also obtained the values $I_R = 10$ and $I_{ILS} = 100$ for ILS-BS. For the GILS, on the other hand, the best setting achieved was $I_R = 20$ and $I_{ILS} = 100$.

The values of the parameter maximum block size l and the newly introduced parameter γ were tuned by running tests on a subset of 60 instances taken from the original set of 1800. More precisely, two instances from every combination of n and R were used to form this subset.

Fig. 7 shows the average CPU time and gaps for values of l from 2 to 17. The average gaps (%) were computed with respect to the objective value of the best known solution in the literature ($Best_{Lit}$), i.e., $gap = 100 \times ((Avg_{ILS\text{-}BS} - Best_{Lit})/Best_{Lit})$. We decided to adopt $l = 13$ because it yielded a good compromise between solution quality and CPU time. The same experiment was performed with GILS and similar results were achieved.

Table 5 reports the average gaps attained for different values of the parameter γ. Note that values in the range $[0, 1]$ were tested, where 0 means that only the best candidates are selected for branching in the constructive BS, while 1 means that the nodes selected for branching are randomly chosen from a list with twice of the maximum size N. We have adopted values $w = 5$ and $N = 100$ based on the promising results reported in Velez-Gallego et al. (2016). From Table 5, it can be observed that the best results were achieved with $\gamma = 0.01$.

6.2.2. Gaps obtained for the instances with up to 75 jobs

This section presents a comparison between the results obtained by the ILS-BS and GILS algorithms with those reported in Velez-Gallego et al. (2016), i.e., BS, and Fan et al. (2019), i.e., VBIH and IG, for the

Table 6
Average Gaps to the BKS considering the best solution found by each algorithm: $n \in \{25, 50, 75\}$.

R	$n = 25$					$n = 50$					$n = 75$				
	ILS-BS	GILS	BS	VBIH	IG	ILS-BS	GILS	BS	VBIH	IG	ILS-BS	GILS	BS	VBIH	IG
0.2	0.00	0.00	0.67	0.47	0.03	0.03	0.07	0.78	0.71	0.68	0.02	0.20	0.55	0.92	1.03
0.6	0.00	0.00	0.12	0.00	0.11	0.02	0.03	0.55	0.01	0.93	0.12	0.18	0.68	0.22	1.64
1.0	0.01	0.00	0.10	0.17	0.09	0.03	0.07	0.37	0.23	1.03	0.05	0.24	0.49	0.54	1.72
1.4	0.01	0.00	0.08	0.27	0.06	0.08	0.07	0.36	0.56	0.78	0.04	0.23	0.42	0.78	1.39
1.8	0.00	0.00	0.14	0.24	0.04	0.05	0.07	0.78	0.65	0.62	0.03	0.20	0.65	0.86	1.14

Table 7
Average gaps to the BKS considering the average solution found by each algorithm: $n \in \{25, 50, 75\}$.

R	$n = 25$		$n = 50$		$n = 75$	
	ILS-BS	GILS	ILS-BS	GILS	ILS-BS	GILS
0.2	0.02	0.02	0.12	0.27	0.14	0.43
0.6	0.01	0.01	0.11	0.15	0.22	0.31
1.0	0.02	0.01	0.10	0.24	0.17	0.44
1.4	0.02	0.01	0.17	0.29	0.16	0.49
1.8	0.02	0.02	0.13	0.24	0.13	0.43

Table 8
Average Gaps to the BKS considering the best solution found by each algorithm: $n \in \{100, 125, 150\}$.

R	$n = 100$			$n = 125$			$n = 150$		
	ILS-BS	GILS	BS	ILS-BS	GILS	BS	ILS-BS	GILS	BS
0.2	0.02	0.33	0.56	0.01	0.54	0.49	0.03	0.70	0.44
0.6	0.09	0.21	0.58	0.06	0.21	0.51	0.03	0.31	0.37
1.0	0.01	0.32	0.41	0.00	0.45	0.36	0.00	0.59	0.32
1.4	0.02	0.44	0.36	0.00	0.54	0.38	0.01	0.80	0.30
1.8	0.01	0.36	0.62	0.01	0.45	0.60	0.00	0.66	0.47

instances with up to 75 jobs. It is worth mentioning that Fan et al. (2019) did not report results for instances of larger size. Table 6 shows the average gap between the best solutions found by a given heuristic and the BKSs (including those found by our B&P) for each pair of $n \leq 75$ and R. Hereafter, the BKSs include those found in this paper. On average, it can be observed that the best solutions achieved by ILS-BS and GILS are of higher quality than those attained by the existing algorithms.

Table 7 reports the average gaps between the average solutions obtained by ILS-based approaches and the BKSs, for each pair of $n \leq 75$ and R. It is important to highlight that the BS algorithm is deterministic and therefore only the best results are available, whereas Fan et al. (2019) only reported the value of the best solutions found by VBIH and IG, which in turn have random components. The results attest that both algorithms are capable of consistently finding high-quality solutions, with gaps smaller than those associated with the best solutions achieved by BS, VBIH, and IG.

6.2.3. Gaps obtained for the instances greater than or equal to 100 jobs

Table 8 presents the average gaps between the best solutions found by the ILS-based procedures and the BKSs, while Table 9 exhibits the average gaps between the average solutions and the BKSs, for each pair $n \geq 100$ and R. Since the results for $n \geq 100$ can only be found in Velez-Gallego et al. (2016), the comparison considers their BS algorithm.

The results visibly show the superiority of ILS-BS compared to GILS and BS. While GILS is still capable of finding average gaps smaller than 1% in most cases, it is still not competitive with ILS-BS, and this becomes more evident as the size of the instances increases. This confirms the clear advantage of employing the BS-based constructive procedure when compared to the GRASP-based constructive approach. GILS is also dominated by BS on the 150-job instances.

Table 9
Average gaps to the BKS considering the average solution found by each algorithm: $n \in \{100, 125, 150\}$.

R	$n = 100$		$n = 125$		$n = 150$	
	ILS-BS	GILS	ILS-BS	GILS	ILS-BS	GILS
0.2	0.11	0.57	0.11	0.74	0.13	0.93
0.6	0.20	0.34	0.15	0.35	0.12	0.43
1.0	0.11	0.54	0.10	0.65	0.09	0.79
1.4	0.13	0.70	0.12	0.80	0.13	1.07
1.8	0.13	0.58	0.11	0.66	0.10	0.88

Table 10
Total number of BKSs and optimal solutions found by each algorithm.

	ILS-BS	GILS	BS	VBIH[a]	IG[a]
#BKS	1466	593	275	355	300
#Optimal[b]	547	511	229	345	292

[a] Considering only 900 instances: $n \in \{25, 50, 75\}$
[b] Total number of known optima: 786

6.2.4. Summary of the results in terms of the number of best solutions found

Table 10 presents the total number of BKSs and optimal solutions found by each heuristic algorithm. The superiority of ILS-BS compared to the other methods is clear, as the method could find 81.4% of the BKSs, considering all 1800 instances, against 32.9%, 15.3% attained by GILS and BS, respectively. When considering only the instances with $n \leq 75$, ILS-BS obtained 73.7% of the BKSs, against 55.7%, 26.6%, 39.4%, and 33.3% achieved by GILS, BS, VBIH, and IG, respectively. Furthermore, among the 786 known optima, the ILS-BS could find 69.6% of them, whereas GILS and BS obtained 65.0%, and 29.1% of the known optima, thus ratifying the competitive performance of the former when it comes to finding high-quality solutions. For $n \leq 75$, the values are 71.1%, 65.9%, 31.8%, 48.2%, and 40.8% for ILS-BS, GILS, BS, VBIH, and IG, respectively.

6.2.5. Runtime analysis

The average CPU times spent by the ILS-BS, GILS, and BS algorithms, aggregated by n, are shown in Table 11. For this experiment, the BS by Velez-Gallego et al. (2016) was carefully reimplemented in C++ and executed on the same machine as ILS-BS and GILS. The VBIH and IG algorithms presented in Fan et al. (2019) were originally coded in C++, and the authors executed them with a time limit of 25 s, for each instance, on a machine with the exact same processor as ours but with 8 GB of RAM. For brevity, we decided not to include this constant value in the table. The results clearly show that the standalone BS is much faster than ILS-BS and GILS, but at the expense of solution quality (see Sections Section 6.2.2, 6.2.3). On the other hand, both ILS-BS and GILS required much less than 25 s, on average, for $n \leq 75$, thus outperforming VBIH and IG in terms of CPU time.

We extend the comparisons between ILS-BS and the standalone BS by modifying the parameters of the latter so that the algorithm has additional nodes in the tree to find improved solutions. More precisely, we increased w from 5 to 20 and N from 500 to 2000. We refer to this alternative standalone version as BS*. As a result, BS* became slower than ILS-BS. Table 12 presents the average CPU time and average gaps achieved by BS* with respect to the average solutions found by ILS-BS. We observe that while expanding the BS tree can lead to better

Table 11
Average CPU times in seconds.

n	ILS-BS	GILS	BS
25	0.63	0.48	0.20
50	3.63	4.00	1.00
75	10.24	13.68	2.63
100	21.28	33.43	5.25
125	37.27	66.47	8.99
150	57.38	97.58	13.97

Table 12
Gaps (%) found by BS* with respect to the average solution obtained by ILS-BS.

R	$n = 25$	$n = 50$	$n = 75$	$n = 100$	$n = 125$	$n = 150$
0.2	0.01	0.24	0.26	0.22	0.21	0.14
0.6	0.00	0.12	0.15	0.12	0.12	0.07
1.0	−0.01	0.03	0.08	0.11	0.05	−0.01
1.4	0.07	0.51	0.44	0.36	0.43	0.23
1.8	0.61	0.56	0.34	0.34	0.25	0.20
Avg. Time (s)	1.35	6.74	16.48	31.62	54.70	83.59

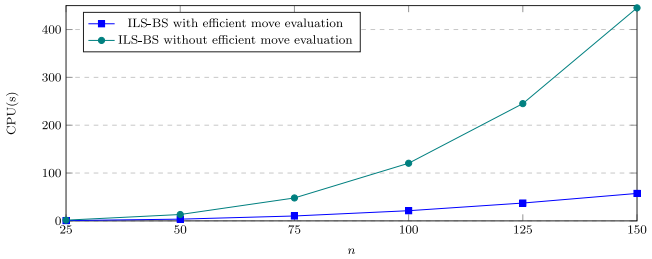


Fig. 8. Average CPU time spent by ILS-BS with and without efficient move evaluation.

solutions, the associated improvements are not enough to achieve the solution quality obtained by ILS-BS for $R = 1.4$ and $R = 1.8$. For the remaining values of R, BS* produced competitive results, most notably for $R = 1$, but in general still not better or as good as ILS-BS, with the exception of $n = 25$ and $R = 1$, as well as $n = 150$ and $R = 1$.

6.2.6. Impact of the efficient move evaluation scheme

In order to quantify the gains of adopting the efficient move evaluation scheme described in Section 5.2.2, we executed the ILS-BS algorithm without such scheme. This alternative version evaluates moves in $\mathcal{O}(n)$ time. Fig. 8 compares the runtime performance of both versions, where one can visibly observe the benefits of implementing an efficient move evaluation scheme. Moreover, the larger the instance, the larger the gains. For the 150-job instances, the ILS-BS with efficient move evaluation was approximately 7.76 times faster than the non-efficient version.

7. Conclusions

In this paper, we proposed a branch-and-price (B&P) algorithm and a heuristic procedure (ILS-BS) for solving problem $1|r_j, s_{ij}|C_{max}$. The motivation of the study was to provide very tight bounds for 1800 benchmark instances of this very challenging $\mathcal{NP}$-hard problem.

The B&P algorithm was implemented over an arc-time-indexed formulation using a dynamic ng-path relaxation scheme. The pricing subproblem aims at finding an ng-path that minimizes the reduced costs and it was solved by an efficient bidirectional labeling algorithm. The results obtained are extremely favorable compared to the MILP model from the literature. Hundreds of open instances were solved for the first time, more than 100 new best known solutions were attained, and the dual bounds of more than a thousand instances were improved.

The multi-start ILS-BS heuristic uses a BS procedure to generate initial solutions that are possibly improved by an ILS procedure that employs the l-block insertion and swap neighborhoods in the local search and a double-bridge mechanism in the perturbation phase. The local search algorithm is based on randomized variable neighborhood descent (RVND) and employs an efficient move evaluation scheme that allows neighbor solutions to be evaluated in amortized constant time. We have empirically demonstrated the substantial runtime gains of adopting such scheme, which led to an average speedup of 7.76 on the 150-job instances. The results obtained showed that ILS-BS is capable of producing high-quality solutions compared to existing heuristic methods for the problem. A considerable number of improved solutions were found, which in turn were provided to the B&P algorithm as initial primal bounds, and this enabled the exact algorithm to prove the optimality of hundreds of open instances.

Future work may include the extension of the proposed solution approaches to solve other variants of the problem with more practical and challenging assumptions (e.g., precedence constraints) or alternative objective functions.

Acknowledgments

This research was partially supported by the Brazilian research agencies CNPq, grants 309580/2021-8, 406245/2021-5, and 314088/2021-0, and Paraíba State Research Foundation (FAPESQ, Brazil), grant 2021/3182, as well as by Universidade Federal da Paraíba, Brazil through the Public Call n. 03/2020 “Produtividade em Pesquisa PROPESQ/PRPG/UFPB”, proposal codes PVL13395-202 and PVL13400-2020. We would also like to thank to Lenildo Luan Carlos and Eduardo Queiroga for their helpful contribution to this work.

References

Achterberg, T. (2007). *Constraint integer programming* (Ph.D. thesis), Technische Universitat Berlin.

Allahverdi, A. (2015). The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research*, *246*(2), 345–378.

Allahverdi, A., Gupta, J. N. D., & Aldowaisan, T. (1999). A review of scheduling research involving setup considerations. *Omega*, *27*(2), 219–239.

Allahverdi, A., Ng, C. T., Cheng, T. C. E., & Kovalyov, M. Y. (2008). A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, *187*, 985–1032.

Baldacci, R., Mingozzi, A., & Roberti, R. (2011). New route relaxation and pricing strategies for the vehicle routing problem. *Operations Research*, *59*(5), 1269–1283.

Bianco, L., Ricciardelli, S., Rinaldi, G., & Sassano, A. (1988). Scheduling tasks with sequence- dependent processing times. *Naval Research Logistics*, *35*(2), 177–184.

Bulhões, T., Sadykov, R., Subramanian, A., & Uchoa, E. (2020). On the exact solution of a large class of parallel machine scheduling problems. *Journal of Scheduling*, *23*(4), 411–429.

Bulhões, T., Sadykov, R., & Uchoa, E. (2018). A branch-and-price algorithm for the Minimum Latency Problem. *Computers & Operations Research*, *93*, 66–78.

Chang, T. Y., Chou, F. D., & Lee, C. E. (2004). A heuristic algorithm to minimize total weighted tardiness on a single machine with release dates and sequence-dependent setup times. *Journal of the Chinese Institute of Industrial Engineers*, *21*(3), 289–300.

Chou, F. D., Wang, H. M., & Chang, T. Y. (2008). Algorithms for the single machine total weighted completion time scheduling problem with release times and sequence-dependent setups. *International Journal of Advanced Manufacturing Technology*, *43*(810).

Diana, R. O., de Souza, S. R., & Filho, M. F. (2018). A variable neighborhood descent as ILS local search to the minimization of the total weighted tardiness on unrelated parallel machines and sequence dependent setup times. *Electronic Notes in Discrete Mathematics*, *66*, 191–198, 5th International Conference on Variable Neighborhood Search.

Ekici, A., Elyasi, M., Özener, O. Ö., & Sarıkaya, M. B. (2019). An application of unrelated parallel machine scheduling with sequence-dependent setups at Vestel Electronics. *Computers & Operations Research*, *111*, 130–140.

Fan, J., Öztop, H., Tasgetiren, M., & Gao, L. (2019). A variable block insertion heuristic for single machine with release dates and sequence dependent setup times for makespan minimization. In *2019 IEEE symposium series on computational intelligence* (pp. 1676–1683). Xiamen, China: IEEE.

Feo, T. A., & Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, *6*, 109–133.

Fernandez-Viagas, V., & Costa, A. (2021). Two novel population based algorithms for the single machine scheduling problem with sequence dependent setup times and release times. *Swarm and Evolutionary Computation*, *63*.

Fu, L. L., Aloulou, M. A., & Artigues, C. (2018). Integrated production and outbound distribution scheduling problems with job release dates and deadlines. *Journal of Scheduling*, *21*, 443—460.

Gharehgozli, A., Tavakkoli-Moghaddam, R., & Zaerpour, N. (2009). A fuzzy-mixed-integer goal programming model for a parallel-machine scheduling problem with sequence-dependent setup times and release dates. *Robotics and Computer-Integrated Manufacturing*, *25*(4), 853–859.

Graham, R., Lawler, E., Lenstra, J., & Kan, A. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. In E. J. P.L. Hammer, & B. Korte (Eds.), *Annals of discrete mathematics: vol. 5*, *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* (pp. 287–326). Elsevier.

Kindervater, G., & Savelsbergh, M. (1997). Vehicle routing: handling edge exchanges. In E. Aarts, & J. Lenstra (Eds.), *Local search in combinatorial optimization* (pp. 337–360). New York: Wiley.

Lin, Y. K., & Hsieh, F. Y. (2014). Unrelated parallel machine scheduling with setup times and ready times. *International Journal of Production Research*, *52*(4), 1200–1214.

Logendran, R., McDonell, B., & Smucker, B. (2007). Scheduling unrelated parallel machines with sequence-dependent setups. *Computers & Operations Research*, *34*(11), 3420–3438.

Martin, O., Otto, S., & Felten, E. (1991). Large-step markov chains for the traveling salesman problem. *Complex Systems*, *5*(3), 299–326.

Mladenovic, N., & Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, *24*(11), 1097–1100.

Montoya-Torres, J., González-Solano, F., & Soto-Ferrari, M. (2012). Deterministic machine scheduling with release times and sequence-dependent setups using random-insertion heuristics.. *International Journal of Advanced Operations Management*, *4*(1/2), 4–26.

Montoya-Torres, J., Soto-Ferrari, M., & González-Solano, F. (2010). Production scheduling with sequence-dependent setups and job release times. *Dyna*, *77*(163), 260–269.

Nogueira, T. H., Ramalhinho, H. L., de Carvalho, C. R. V., & Ravetti, M. G. (2022). A hybrid VNS-Lagrangean heuristic framework applied on single machine scheduling problem with sequence-dependent setup times, release dates and due dates. *Optimization Letters*, *16*, 59–78.

Ovacikt, I., & Uzsoy, R. (1994). Rolling horizon algorithms for a single-machine dynamic scheduling problem with sequence-dependent setup times. *International Journal of Production Research*, *32*(6), 1243–1263.

Ozgur, C., & Bai, L. (2010). Hierarchical composition heuristic for asymmetric sequence dependent single machine scheduling problems. *Operations Management Research*, *3*(3), 98–106.

Pereira Lopes, M. J., & Valério de Carvalho, J. (2007). A branch-and-price algorithm for scheduling parallel machines with sequence dependent setup times. *European Journal of Operational Research*, *176*(3), 1508–1527.

Pessoa, A., Sadykov, R., Uchoa, E., & Vanderbeck, F. (2018). Automation and combination of linear-programming based stabilization techniques in column generation. *INFORMS Journal on Computing*, *30*(2), 339–360.

Pinedo, M. L. (2012). *Scheduling: Theory, algorithms, and systems*. Nova Iorque: Springer-Verlag.

Roberti, R., & Mingozzi, A. (2014). Dynamic ng-path relaxation for the delivery man problem. *Transportation Science*, *48*(3), 413–424.

Røpke, S. (2012). Branching decisions in branch-and-cut-and-price algorithms for vehicle routing problems. Presentation In Column Generation 2012.

Ruiz, R., & Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, *177*(3), 2033–2049.

Shin, H., Kim, C.-O., & Kim, S. (2002). A tabu search algorithm for single machine scheduling with release times, due dates, and sequence-dependent set-up times. *International Journal of Advanced Manufacturing Technology*, *19*(11), 859–866.

Silva, M. M., Subramanian, A., Vidal, T., & Ochi, L. S. (2012). A simple and effective metaheuristic for the Minimum Latency Problem. *European Journal of Operational Research*, *221*(3), 513–520.

Silva, J. M. P., Teixeira, E., & Subramanian, A. (2021). Exact and metaheuristic approaches for identical parallel machine scheduling with a common server and sequence-dependent setup times. *Journal of the Operational Research Society*, *72*(2), 444–457.

Subramanian, A., Drummond, L., Bentes, C., Ochi, L., & Farias, R. (2010). A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery. *Computers & Operations Research*, *37*(11), 1899–1911, Metaheuristics for Logistics and Vehicle Routing.

Vanderbeck, F., Sadykov, R., & Tahiri, I. (2017). *BaPCod — a generic branch-and-price code*: *Technical report*, URL: https://realopt.bordeaux.inria.fr/?page_id=2.

Velez-Gallego, M. C., Maya, J., & Montoya-Torres, J. (2016). A beam search heuristic for scheduling A single machine with release dates and sequence dependent setup times to minimize the makespan. *Computers & Operations Research*, *73*, 132–140.

Vidal, T., Crainic, T. G., Gendreau, M., & Prins, C. (2013). A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows. *Computers & Operations Research*, *40*(1), 475–489.