European Journal of Operational Research 286 (2020) 115–128

Contents lists available at ScienceDirect

European Journal of Operational Research

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

Discrete Optimization

Minimizing makespan on a single machine with release dates and inventory constraints

Morteza Davari[a,b], Mohammad Ranjbar[c], Patrick De Causmaecker[a], Roel Leus[d,*]

[a] CODeS, Department of Computer Science & imec-ITEC, KU Leuven KULAK Belgium
[b] Research Centre for Operations Management, Faculty of Economics and Business, KU Leuven Campus Brussels Belgium
[c] Department of Industrial Engineering, Ferdowsi University of Mashhad Iran
[d] ORSTAT, Faculty of Economics and Business, KU Leuven Belgium

ARTICLE INFO

Article history:
Received 10 March 2019
Accepted 9 March 2020
Available online 17 March 2020

Keywords:
Single-machine scheduling
Inventory constraint
Complexity
Branch-and-bound
Dynamic programming
Lagrangian relaxation

ABSTRACT

We consider a single-machine scheduling problem with release dates and inventory constraints. Each job has a deterministic processing time and has an impact (either positive or negative) on the central inventory level. We aim to find a sequence of jobs such that the makespan is minimized while all release dates and inventory constraints are met. We show that the problem is strongly NP-hard even when the capacity of the inventory is infinite. To solve the problem, we introduce a time-indexed formulation and a sequence-based formulation, a branch-and-bound algorithm, and a dynamic-programming-based guess-and-check (GC) algorithm. From extensive computational experiments, we find that the GC algorithm outperforms all other alternatives.

1. Introduction

We consider a single-machine scheduling problem in which a set of jobs must be performed. Each job is characterized by a processing time, a release date and an inventory modification. A job with negative inventory modification can be processed only if the central inventory level is sufficiently high, and a job with positive inventory modification can be executed only if the inventory level is low enough. The goal is to find a feasible sequence of jobs with the minimum makespan, which is the completion time of the latest job.

Our problem has some interesting applications in scheduling warehousing operations, where shipments arrive in a loading/unloading dock either to deliver (unload) or to pick up (load) a certain number of products. The warehouse includes a storage space that is used to store these products in inventory. Obviously, jobs with positive inventory modification imply unloading operations and jobs with negative inventory modification constitute loading operations.

There is a very limited number of papers considering machine scheduling subject to inventory constraints. Briskorn, Choi, Lee, Leung, and Pinedo (2010) prove the complexity of many variants of our problem without release dates and with the assumption that the inventory has an infinite capacity. Also, Briskorn, Jaehn, and Pesch (2013) propose exact algorithms for the problem with inventory constraints and with the objective of minimizing the total weighted completion times. We also cite Briskorn and Leung (2013), who develop a number of branch-and-bound (BB) algorithms that solve instances of a single-machine scheduling problem with inventory constraints to minimize the maximum lateness, and Briskorn and Pesch (2013), who propose a number of variable neighborhood algorithms for single-machine scheduling with inventory constraints to optimize a number of well-known regular objective functions, including the minimization of the maximum lateness, the minimization of the total completion time and the minimization of the total weighted tardiness. In this paper, we consider a variant of the above-mentioned problems that incorporates release dates and minimizes makespan.

Inventory constraints are generalizations of non-renewable resource constraints. Non-renewable resources are consumed while processing jobs; typical examples are raw materials, energy and financial resources (Slowiński, 1984; Toker, Kondakci, & Erkip, 1991). At specific moments in time, additional amounts of new resources can become available (Carlier & Rinnooy Kan, 1982). We also refer to Gafarov, Lazarev, and Werner (2011), Györgyi and Kis (2014, 2015), Kis (2015), and Györgyi (2017), who propose exact and approximation algorithms for a number of single-machine scheduling problems with non-renewable resource constraints, with and

* Corresponding author.
E-mail address: roel.leus@kuleuven.be (R. Leus).

https://doi.org/10.1016/j.ejor.2020.03.029
0377-2217/

without release dates and with objective functions such as minimization of the makespan and minimization of the total weighted completion times.

Our problem is also related to a number of problems within the area of scheduling with so-called "inventory releasing" jobs, which has been motivated by applications in Just-In-Time manufacturing (Boysen, Bock, & Fliedner, 2013; Drótos & Kis, 2013), where processing jobs leads to the release of a predefined number of product units into inventory. The objective in these problems is to minimize the resulting product inventory. Another related research area is project scheduling with inventory constraints (Bartels & Zimmermann, 2015; Neumann & Schwindt, 2003; Neumann, Schwindt, & Trautmann, 2005; Schwindt & Trautmann, 2000), where activity execution implies consumption or replenishment of certain amounts of non-renewable resources.

The contents of this article are as follows. We first provide a formal problem statement followed by some complexity results in Section 2. Subsequently, we propose two mixed-integer programming (MIP) formulations (in Section 3), a BB algorithm (in Section 4) and a guess-and-check (GC) algorithm (in Section 5) to solve the problem to optimality. Finally, we discuss our computational results (in Section 6) and state some conclusions (in Section 7).

2. Problem statement

In this section, we formally describe the problem under study (in Section 2.1), present some complexity results (in Section 2.2), and discuss a situation where a given solution can be guaranteed to be optimal (in Section 2.3).

2.1. Description of the problem

We are given a set $J = \{1, \ldots, n\}$ of jobs, partitioned into set J^- of loading jobs and set J^+ of unloading jobs. Each job $j \in J$ has a processing time $p_j \in \mathbb{R}^+$, a release date $r_j \in \mathbb{R}^+$ and an inventory modification $\delta_j \in \mathbb{R}$, where $\delta_j \geq 0$ for $j \in J^+$ and $\delta_j < 0$ for $j \in J^-$. All jobs are to be executed by a single machine, which can only serve one job at a time. The initial inventory is denoted by I_0 and the capacity of the inventory storage is I_C. The objective is to sequence the jobs in J such that the makespan (the completion time of the last job in the sequence) is minimized while the inventory is never below 0 nor above I_C.

A solution to our problem is a sequence of jobs; we use the terms *sequence* and *solution* interchangeably throughout this paper. Given a sequence, one can build a *schedule* with specific job starting times. Let $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$ be a sequence. This sequence is feasible iff $0 \leq I_0 + \sum_{\kappa=1}^{s} \delta_{\sigma_\kappa} \leq I_C$ for each $s = 1, \ldots, n$. Let Ω be the set of all feasible sequences and let $C_j(\boldsymbol{\sigma})$ be the completion time of job j if we schedule all jobs as soon as possible based on $\boldsymbol{\sigma}$. The makespan of a sequence $\boldsymbol{\sigma}$ is denoted by either $C_{\max}(\boldsymbol{\sigma})$ or $C_{\sigma_n}(\boldsymbol{\sigma})$ throughout this text. Our problem can be stated as follows:

$$\min_{\boldsymbol{\sigma} \in \Omega} C_{\sigma_n}(\boldsymbol{\sigma}).$$

The problem can be referred to as $1|inv, r_j|C_{\max}$ in the standard three-field notation.

Example. Consider an example with $n = 5$, $I_0 = 6$ and $I_C = 8$. Table 1 contains the remaining data for this problem instance. An optimal solution with a makespan of 27 is (3,1,5,4,2), which is illustrated in Fig. 1.

2.2. Complexity results

We have the following results.

Theorem 1. *$1|inv, r_j|C_{\max}$ is strongly NP-hard even when $I_C = +\infty$.*

Table 1
Data for the example instance.

j	1	2	3	4	5
p_j	1	1	8	4	8
r_j	7	1	4	18	14
δ_j	−1	−4	−2	5	−2

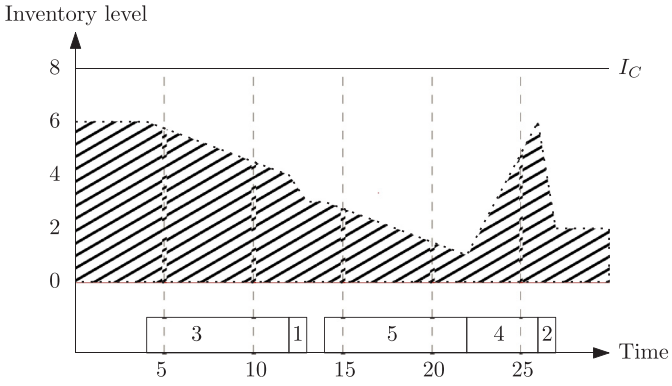

Fig. 1. An optimal solution for the example instance.

Theorem 2. *The verification of the existence of a feasible solution to $1|inv|C_{\max}$ is strongly NP-complete.*

Proofs of all theorems are provided in the Appendix. It is worth mentioning that $1|inv, r_j|C_{\max}$ is closely related to $1|inv|L_{\max}$, which is also known to be strongly NP-hard (Briskorn et al., 2010; Briskorn & Pesch, 2013). More specifically, one can show that any instance of $1|inv, r_j|C_{\max}$ can be solved to optimality by solving a polynomial number of associated instances of $1|inv|L_{\max}$, and likewise any instance of $1|inv|L_{\max}$ can be solved to optimality by a polynomial number of calls of an optimal procedure for $1|inv, r_j|C_{\max}$.

Studying the complexity of $1|inv, r_j|C_{\max}$ is important because, on the one hand, $1|r_j|C_{\max}$ is polynomially solvable (Lawler, 1973), and on the other hand $1|inv|C_{\max}$ is also polynomially solvable if $I_C/2 \geq |\delta_j|$ for every $j \in J$. Although finding a feasible solution to $1|inv|C_{\max}$ is strongly NP-complete in the general case (see Theorem 2), one can trivially show that any feasible solution to $1|inv|C_{\max}$ (if there exists one) is also optimal. Despite the complexity status, it also is not difficult to come up with some necessary conditions for the existence of a feasible solution. A trivial example is that any instance with at least one feasible solution must satisfy $0 \leq I_0 + \sum_{k \in J} \delta_k \leq I_C$. There is an $O(n)$-time algorithm that finds a feasible solution to $1|inv|C_{\max}$ if $I_C/2 \geq |\delta_j|$ for every $j \in J$ (Briskorn and Pesch, 2013, Theorem 2). Let $\delta^+_{\max} = \max_{j \in J^+}\{\delta_j\}$ and $\delta^-_{\max} = \max_{j \in J^-}\{-\delta_j\}$. A stronger result can be derived as follows:

Theorem 3. *There is an $O(n)$-time algorithm for problem $1|inv|C_{\max}$*

(a) *when $I_C \geq \delta^+_{\max} + \delta^-_{\max}$,*
(b) *or even when $I_C \geq \delta^+_{\max} + \delta^-_{\max} - 1$ if all δ_j are integers.*

2.3. Early optimality

Since our problem is strongly NP-hard, finding an optimal solution is expected to be difficult. Given a feasible solution, however, we may be able to immediately conclude optimality provided that the solution fulfills a specific condition:

Theorem 4. *A feasible solution $\boldsymbol{\sigma}$ is optimal if $\exists k \in \{1, \ldots, n\}$ for which the following conditions hold:*

(a) $r_{\sigma_k} \leq r_{\sigma_l}$ *for every* $l = k+1, \ldots, n$.
(b) $r_{\sigma_k} + \sum_{s=k}^{l-1} p_{\sigma_s} \geq r_{\sigma_l}$ *for every* $l = k+1, \ldots, n$.
(c) $C_{\sigma_{k-1}}(\boldsymbol{\sigma}) \leq r_{\sigma_k}$.

We refer to a solution that satisfies the conditions of Theorem 4 as an *early-opt* solution. Any algorithm that scans the solution space can be halted as soon as an early-opt solution is encountered. Determining whether or not an instance has an early-opt solution is not straightforward, however.

3. Mixed integer programming formulations

In this section, we develop two MIP formulations (in Sections 3.1 and 3.2).

3.1. Time-indexed formulation

We define decision variables x_{jt}, which take value one if the processing of job j starts at time t, and value zero otherwise. We also introduce variable y_t, which represents the inventory level at time t. The time horizon $T = \{t_{\min}, t_{\min}+1, \ldots, t_{\max}\}$ is the set of potential starting times, where

$$t_{\min} = \min_{j\in J}\{r_j\}$$

and $t_{\max} = \max_{j\in J}\{r_j\} + \sum_{j\in J} p_j - \min_{j\in J}\{p_j\}$.

We determine a smaller set $T_j = \{r_j, r_j+1, \ldots, t^j_{\max}\}$ for each job j, where

$$t^j_{\max} = \max\left\{r_j\,;\ \max_{i\in J\setminus\{j\}}\{r_i\} + \sum_{i\in J\setminus\{j\}} p_i\right\}.$$

A time-indexed formulation (TIF, for short) for our problem can be given as follows:

TIF : min z

subject to

$$\sum_{t\in T_j} x_{jt} = 1 \qquad j\in J \qquad (1)$$

$$\sum_{j\in J}\sum_{\tau=t-p_j}^{t-1} x_{j\tau} \le 1 \qquad t\in T \qquad (2)$$

$$z \ge \sum_{t\in T_j} t x_{jt} + p_j \qquad j\in J \qquad (3)$$

$$y_{t_{\min}} = I_0 + \sum_{j=1}^{n} \delta_j x_{jt_{\min}} \qquad (4)$$

$$y_t = y_{t-1} + \sum_{j=1}^{n} \delta_j x_{jt} \qquad t\in T\setminus\{t_{\min}\} \qquad (5)$$

$$0 \le y_t \le I_C \qquad t\in T \qquad (6)$$

In this formulation, the objective is to minimize the makespan z. Constraints (1) state that each job should start exactly once. Constraints (2) ensure that there are no overlaps in the execution of the jobs. Constraints (3) compute the makespan for each choice for $\mathbf{x}$. Finally, constraints (4)–(6) impose the inventory constraints. We note that this formulation correctly represents our problem only when the time horizon can be discretized (i.e., when all processing times and ready times are integers).

3.2. Sequence-based formulation

We define decision variables x_{js}, which take value one if job j is the sth job processed and zero otherwise. We also introduce variables y_s, which represent the inventory level after finishing the job at position s, and variables z_s, which are the completion times of the job in position s, for $s = 1, \ldots, n$. A sequence-based formulation (SBF) can then be stated as follows:

SBF : min z_n

subject to

$$\sum_{s=1}^{n} x_{js} = 1 \qquad j\in J \qquad (7)$$

$$\sum_{j\in J} x_{js} = 1 \qquad s = 1,\ldots,n \qquad (8)$$

$$z_s \ge \sum_{j\in J} x_{js}(r_j + p_j) \qquad s = 1,\ldots,n \qquad (9)$$

$$z_s \ge z_{s-1} + \sum_{j\in J} x_{js} p_j \qquad s = 2,\ldots,n \qquad (10)$$

$$y_1 = I_0 + \sum_{j=1}^{n} \delta_j x_{j1} \qquad (11)$$

$$y_s = y_{s-1} + \sum_{j=1}^{n} \delta_j x_{js} \qquad s = 2,\ldots,n \qquad (12)$$

$$0 \le y_s \le I_C \qquad s = 1,\ldots,n \qquad (13)$$

In this formulation, the objective is to minimize the completion time z_n of the last job in the sequence. Constraints (7) and (8) enforce assignment of jobs to positions. Constraints (9) and (10) compute the completion time of the job in each position. Finally, constraints (11) and (12) impose the inventory constraints. Unlike the time-indexed formulation, this SBF works even when the time horizon is not discrete.

4. Branch and bound

Before discussing the main algorithmic ingredients of the BB algorithm in Sections 4.3–4.5, we first develop a number of useful concepts in Sections 4.1 and 4.2.

4.1. Preliminary concepts

We consider an alternative *block-based* representation of a solution, which consists of a series of *blocks* and a set of sequences of jobs within the blocks. A block is a set of jobs that are executed consecutively without intermediate idle time. For ease of reference, we refer to a solution with a block-based representation as a *block solution*. Clearly, a feasible sequence can be translated into a feasible block solution in linear time, and vice versa. The idea of decomposition based on blocks of jobs is not new; see, for example, Pan and Shi (2005), who use block decomposition to obtain tighter bounds in the presence of release dates and deadlines, and Baptiste and Sadykov (2009), who use interval decomposition to establish a new MIP formulation for single-machine scheduling with a piecewise-linear objective function.

A block solution is denoted as $\Pi = (\boldsymbol{\sigma}_1, \ldots, \boldsymbol{\sigma}_\vartheta)$, where $\boldsymbol{\sigma}_k$ is the sequence of jobs in the kth block. A block solution Π also implies an *assignment* of jobs to blocks, which is written as $\mathbf{A}(\Pi) = (B_1(\Pi), \ldots, B_\vartheta(\Pi))$. Throughout this paper, we use both $\mathbf{A}(\Pi)$ and $\mathbf{A}$ to refer to an assignment, and both $B_k(\Pi)$ and B_k as the set of jobs in the kth block. Obviously, the sets B_1, B_2, ..., B_ϑ need to constitute a partition of J.

Example. An optimal solution to the example instance of Section 1 is given in Fig. 1, which corresponds with the block solution $\Pi^* = ((3,1),(5,4,2))$; the associated assignment is $\mathbf{A}^* = \mathbf{A}(\Pi^*) = (\{1,3\},\{2,4,5\})$.

Let $I_{in}(B_k) = I_0 + \sum_{v=1}^{k-1}\delta(B_v)$ be the initial inventory at the start of block B_k, with $\delta(B_k) = \sum_{j\in B_k}\delta_j$ the net inventory modification

of block B_k, and let $C(B_k)$ be the earliest possible completion time of the last job in block B_k. We say that a block solution $\Pi = (\boldsymbol{\sigma}_1, \ldots, \boldsymbol{\sigma}_\vartheta)$ is feasible if the following conditions hold for each sequence $\boldsymbol{\sigma}_k = (\sigma_{k1}, \sigma_{k2}, \ldots, \sigma_{k|B_k|})$, $k = 1, \ldots, \vartheta$:

Condition 1. $r_{\sigma_{k1}} > C(B_{k-1})$ if $k \in \{2, \ldots, \vartheta\}$.

Condition 2. $r_{\sigma_{ks}} \leq r_{\sigma_{k1}} + \sum_{v=2}^{s-1} p_{\sigma_{kv}}$ for each $s = 2, \ldots, |B_k|$.

Condition 3. $0 \leq I_{in}(B_k) + \sum_{j=1}^{s} \delta_j \leq I_C$ for each $s = 1, \ldots, |B_k|$.

Condition 1 implies that the release date of the first job is strictly greater than the earliest completion time of the previous block. Condition 2 ensures that the jobs within the block can be scheduled without idle time, and Condition 3 guarantees that the inventory constraints are not violated. An assignment $\mathbf{A}$ is said to be feasible if at least one feasible block solution exists for the assignment.

Example. The earlier-described block solution $\Pi^* = ((3,1),(5,4,2))$ to the example instance is feasible because $7 = r_1 \leq r_3 + p_3 = 12$ (Condition 2) and $0 \leq 6 - 2 = 4 \leq 8$ and $0 \leq 6 - 2 - 1 = 3 \leq 8$ (Condition 3). Similarly, for the second block we have $14 = r_5 > C(B_1^*) = 13$ (Condition 1), $18 = r_4 \leq r_5 + p_5 = 22$ and $1 = r_2 \leq r_5 + p_5 + p_4 = 26$ (Condition 2) and $0 \leq 3 - 2 = 1 \leq 8$, $0 \leq 3 - 2 + 5 = 6 \leq 8$ and $0 \leq 3 - 2 + 5 - 4 = 2 \leq 8$ (Condition 3). The corresponding assignment $\mathbf{A}^* = \mathbf{A}(\Pi^*) = (\{1,3\},\{2,4,5\})$ is then also feasible.

Let $\mathcal{A}$ be the set of all feasible assignments. The minimum completion time $C(\mathbf{A})$ of a feasible assignment $\mathbf{A}$ is the minimum completion time of its last block. We can then reformulate the scheduling problem as follows:

$$\min_{\mathbf{A} \in \mathcal{A}} C(\mathbf{A}).$$

This requires the evaluation of $C(\mathbf{A}) = C(B_\vartheta)$, which can be achieved via the following subproblem:

$$\text{SUBP}: C(B_k) = \min_{\boldsymbol{\sigma}_k \in \Omega(B_k)} \{r_{\sigma_{k1}} + p(B_k)\}.$$

The set $\Omega(B_k)$ contains all the sequences that respect Conditions 1, 2 and 3, for given values $C(B_{k-1})$ and $I_{in}(B_k)$ (which are input parameters to SUBP). While $I_{in}(B_k)$ is computed in linear time, the quantities $C(B_{k-1}), C(B_{k-2}), \ldots, C(B_1)$ must be computed recursively by calling SUBP. Consequently, in order to compute $C(\mathbf{A})$ we should solve ϑ instances of SUBP. Upon computing $C(\mathbf{A})$, a unique Π is automatically constructed.

4.2. The complexity of SUBP

SUBP can be shown to be strongly NP-hard by a reduction from 3-PARTITION that is similar to the proof of Theorem 1. Nevertheless, it can be solved in polynomial time in certain situations. Based on Theorem 3, there is an $O(|B_k|)$-time algorithm for SUBP if the following two conditions hold.

(a) $I_C \geq \max_{j \in B_k \cap J^+}\{\delta_j\} + \max_{j \in B_k \cap J^-}\{-\delta_j\}$.
(b) $\max_{j \in B_k}\{r_j\} \leq r_{j'} + p_{j'}$ for all $j' \in B_k$.

Moreover, any instance of SUBP for which at least one of the following conditions holds, does not have any feasible solution:

(a) $\max_{j \in B_k}\{r_j\} \leq C(B_{k-1})$.
(b) $I_{in}(B_k) + \delta(B_k) < 0$ or $I_{in}(B_k) + \delta(B_k) > I_C$.

To solve SUBP, we first verify whether the instance is infeasible according to the foregoing conditions. If not, then we check whether SUBP is solvable in $O(|B_k|)$-time. If this is not the case, then we proceed as follows.

We define the following decision problem:

Problem DECP
Instance: An instance of SUBP and a target job $\hat{j} \in B_k$.
Question: Does there exist a sequence $\boldsymbol{\sigma} \in \Omega(B_k)$ such that $\sigma_1 = \hat{j}$?

SUBP and DECP are closely related. In fact, SUBP can be solved by solving a series of instances of DECP as described in Algorithm 1. DECP is strongly NP-complete, since even the decision counterpart of $1|inv|C_{\max}$ is already strongly NP-complete (see the proof of Theorem 2).

In spite of its complexity status, DECP can often be solved in a relatively efficient manner. We solve DECP using an implicit enumeration method referred to as eB-DECP (for "embedded branching" procedure), which is briefly described below.

In the search tree of eB-DECP, each node represents a partial sequence of jobs in B_k. The partial sequence $\tilde{\boldsymbol{\sigma}}_k^0 = (\hat{j})$ is associated

Algorithm 1 Successive check.

Input: A SUBP instance.

1: Create a list of the jobs in B_k in non-decreasing order of the release dates.
2: Choose the first job in the list and consider it as the target job.
3: Solve the associated DECP instance. If the answer is YES then the resulting sequence is an optimal solution to SUBP and the procedure ends. If the answer is NO then the next job in the list becomes the target job and we repeat step 3, either until the procedure ends with an optimal solution or until all jobs in the list have been selected. In the latter case, SUBP has no feasible solution.

Output: YES (in which case a sequence is also output) or NO.

with the root node, with $\hat{j}$ the target job in DECP. The children of the root node correspond with partial sequences $(\hat{j}, i_1)$, where $i_1 \in B_k \setminus \{\hat{j}\}$. The children of these nodes then generate $(\hat{j}, i_1, i_2)$ with $i_2 \in B_k \setminus \{\hat{j}, i_1\}$, etc. A node is fathomed if its associated partial sequence violates any of the Conditions 1–3. We also eliminate partial sequences based on the dominance theorem of dynamic programming (DP) (Davari, Demeulemeester, Leus, & Talla Nobibon, 2016; Jouglet, Baptiste, & Carlier, 2004): among the two nodes with the same set of scheduled jobs but with different partial sequences, the one which has lexicographically larger release dates is dominated. In case of ties, the one with lexicographically larger job indices is dominated. For practical reasons, each partial sequence is only compared with the alternative sequence in which the order of the last two jobs is different.

The search tree is traversed in a depth-first manner. The child node whose last job (in the sequence) has the smallest release date is visited first. In case of ties, the node whose last job has the smallest job index is visited first. The procedure eB-DECP is halted as soon as a feasible sequence for B_k is found, in which case the answer to DECP is YES. If no feasible sequence is found after traversing the entire tree, the answer to DECP is NO.

4.3. Branching scheme

A node N^u in the BB tree corresponds to a (possibly partial) assignment $\mathbf{A}^u$, i.e., a partition of a subset of J (u is the index of the node). When $\mathbf{A}^u$ is a complete assignment (containing all jobs in J, which only happens in a leaf node) then $C(\mathbf{A}^u)$ can be computed using SUBP. Otherwise, $\mathbf{A}^u$ represents multiple (either or not feasible) complete assignments, which are located in the leaf nodes reachable from N^u.

The root node N^0 (at level 0) corresponds to the empty partial block solution $\mathbf{A}^0 = \emptyset$. Partial solutions are augmented using an *add operator* $\leftarrow$, defined as follows: $\mathbf{A} \leftarrow (j, b)$ is the solution $\mathbf{A}$ with the *target job* $j \in J$ appended to the bth block (referred to as *target block*). The root node is branched into a number of child nodes N^u

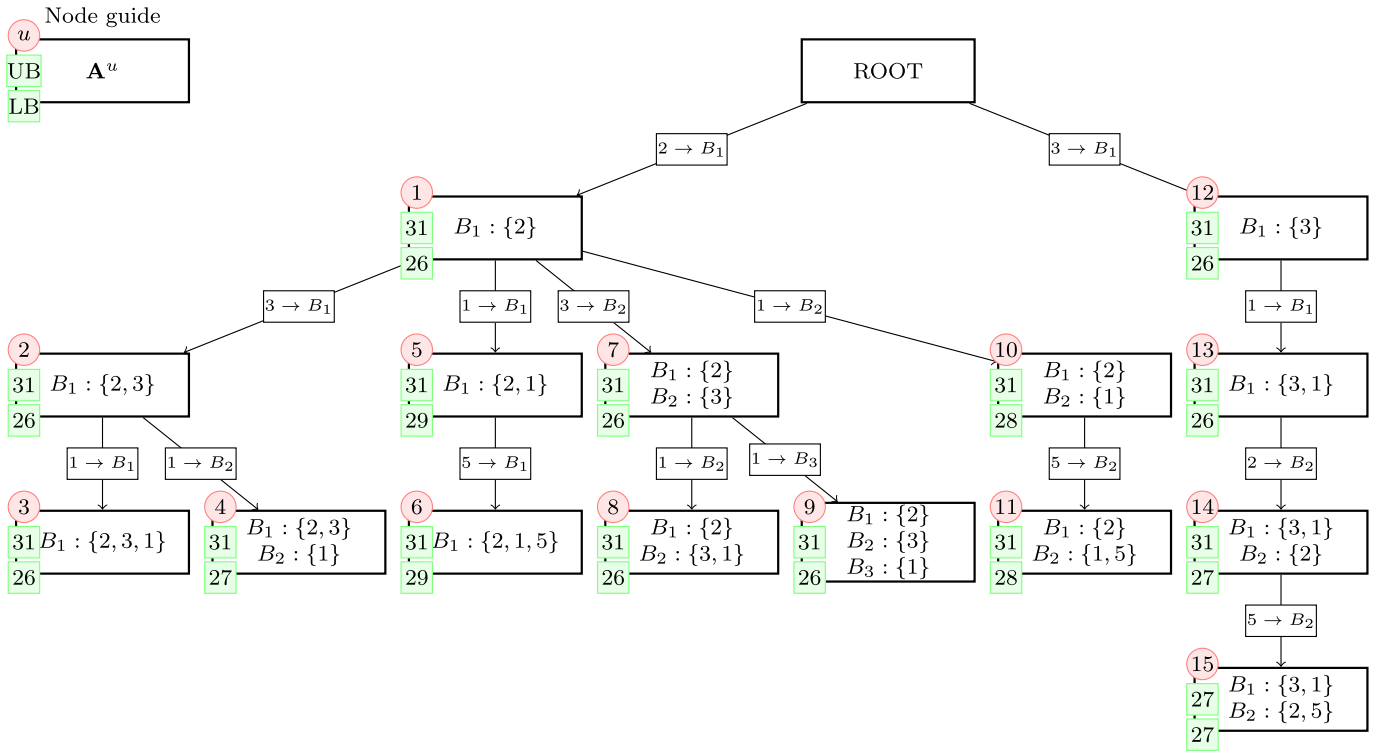

Fig. 2. The BB tree for the example instance. Nodes eliminated by dominance rules (see Section 4.5) are not included in the tree.

with $\mathbf{A}^u = \mathbf{A}^0 \leftarrow (j, 1)$, which constitutes the first level of the BB tree. We identify the parent of a node N^u by $pa(N^u)$, the target job of N^u by $tj(N^u)$ and the target block of N^u by $tb(N^u)$. Nodes at level l are branched into nodes in level $l+1$. The following two properties hold for nodes at levels $l \geq 2$:

(a) Blocks are filled sequentially: $tb(N^u) \in \{tb(pa(N^u)), tb(pa(N^u)) + 1\}$.
(b) We avoid duplicate assignments: if the target blocks of N^u and its parent are the same ($tb(N^u) = tb(pa(N^u))$) then the position of $tj(N^u)$ in a given job list $\boldsymbol{\rho}$ is greater than the position of $tj(pa(N^u))$.

The list $\boldsymbol{\rho}$ is constructed as follows: the jobs are sorted in non-decreasing order of their release dates; in case of ties, the job with the lower job index appears first. The tree is explored in a depth-first manner. Among the unvisited children of a node, the child with the lowest-indexed target block is visited first; in case of ties, the target job with lowest position in $\boldsymbol{\rho}$ is considered first.

Example. For the example instance used in the previous sections, we have $\boldsymbol{\rho} = (2, 3, 1, 5, 4)$. Fig. 2 shows the resulting search tree. From the root note we generate only two children; the other three potential children are fathomed by dominance rules discussed in Section 4.5.

4.4. Lower and upper bounds

To compute an initial upper bound, we use the heuristic algorithm introduced by Ghorbanzadeh, Ranjbar, and Jamili (2019). We denote this upper bound by UB_{ini}. This upper bound is also used in our GC algorithm (see Section 5). In each node N^u of the tree we maintain a lower bound $\mathcal{L}(N^u)$ on the completion time of the last block in $\mathbf{A}^u$, as follows:

$$\mathcal{L}(N^u) = \begin{cases} \mathcal{L}(pa(N^u)) + p_{tj(N^u)} & \text{if } tb(N^u) = tb(pa(N^u)), \\ \max\left\{\mathcal{L}(pa(N^u)) + 1; r_{tj(N^u)}\right\} + p_{tj(N^u)} & \text{if } tb(N^u) = tb(pa(N^u)) + 1. \end{cases}$$

If $\mathbf{A}^u$ is feasible then $\mathcal{L}(N^u) = C(\mathbf{A}^u)$. Let $U(N^u)$ be the set of unscheduled jobs in node N^u. Starting from $\mathcal{L}(N^u)$, we compute a lower bound $LB(N^u)$ by appending all jobs from $U(N^u)$ as soon as possible after $\mathcal{L}(N^u)$, while respecting the release dates but ignoring the inventory constraints. When two or more jobs are eligible to be scheduled, we schedule the job with the smallest release date. If there is still a tie, we arbitrarily choose one of the jobs. The value $LB(N^u)$ is the completion time of the last job scheduled in this manner.

In our BB procedure, we compute three upper bounds. The first upper bound $UB_1(N^u) = C(\mathbf{A}^u)$ is computed only in leaf nodes. For each non-leaf node N^u, if $\max_{j \in U(N^u)}\{r_j\} \leq \mathcal{L}(N^u)$ then we construct a complete assignment $\tilde{\mathbf{A}}^u$ by adding all jobs in $U(N^u)$ to the last block of $\mathbf{A}^u$. If $\tilde{\mathbf{A}}^u$ is feasible then we compute $UB_2(N^u) = C(\tilde{\mathbf{A}}^u)$. For each non-leaf node N^u, we also construct an assignment $\tilde{\mathbf{A}}^u$ corresponding with the schedule obtained in the computation of $LB(N^u)$. If $\tilde{\mathbf{A}}^u$ is feasible then we let $UB_3(N^u) = C(\tilde{\mathbf{A}}^u)$. Note that the schedule will always satisfy Conditions 1 and 2, so only the verification of Condition 3 suffices to check the feasibility of $\tilde{\mathbf{A}}^u$. Throughout the search, we maintain a global upper bound UB, which equals the best upper bound found so far.

4.5. Dominance rules

The following dominance rule is obvious.

Dominance rule 1. Any node N^u with $LB(N^u) \geq UB$ is fathomed.

The second dominance rule is associated with $UB_3(N^u)$. As explained in Section 4.4, if $\max_{j \in U(N^u)}\{r_j\} \leq \mathcal{L}(N^u)$, then we compute a tentative upper bound by constructing $\tilde{\mathbf{A}}^u$. If $\tilde{\mathbf{A}}^u$ is feasible then it is the only feasible assignment in any of the nodes below N^u, otherwise no nodes below N^u have a feasible assignment. Therefore, the following dominance rule is valid.

Dominance rule 2. All children of any node N^u with $\max_{j \in U(N^u)}\{r_j\} \leq \mathcal{L}(N^u)$ are fathomed.

The following rule attempts to fathom nodes with a target block different from their parent.

Dominance rule 3. Consider a node N^u. All children N^y of this node for which $tb(N^y) = tb(N^u) + 1$ are fathomed if at least one of the following three conditions holds:

(a) $\mathbf{A}^u$ is not feasible.
(b) $r_{j^*} + \sum_{j \in U(N^u)} p_j \geq \text{UB}$ where $j^* = \min_{j \in U(N^u)}\{r_j | r_j > C(N^u)\}$.

If Condition (a) is true, then any assignment associated with the children N^y of N^u for which $tb(N^y) = tb(N^u) + 1$ is also infeasible, as it essentially includes all blocks in $\mathbf{A}^u$. If Condition (b) holds, then the block $tb(N^u) + 1$ need not be constructed in any child of N^u because the naive lower bound $r_{j^*} + \sum_{j \in U(N^u)} p_j$ is dominated by UB. Note that neither condition can conclude the elimination of those children N^y of node N^u for which $tb(N^y) = tb(N^u)$ because by adding jobs to the last block of $\mathbf{A}^u$, the two conditions may no longer be true.

The final dominance rule, similarly to Dominance rule 3, fathoms children of a node with a different target block. The idea here is to move jobs from the last block of the associated partial assignment $B^u_{tb(N^u)}$ to the previous block $B^u_{tb(N^u)-1}$. If the resulting assignment is feasible and has a smaller makespan then the new assignment dominates the original one. Consider the set $\mathcal{E}^u = \left\{ j \in B^u_{tb(N^u)} | r_j \leq C\left(B^u_{tb(N^u)-1}\right) \right\}$. For each $E \subseteq \mathcal{E}^u$, let $\mathbf{A}^u(E)$ be the new assignment obtained by moving all jobs in E from the last to the previous block.

Dominance rule 4. Consider a node N^u. All children N^y of this node for which $tb(N^y) = tb(N^u) + 1$ are fathomed if there is a subset $E \subseteq E^u$ for which $\mathbf{A}^u(E)$ is feasible and $C(\mathbf{A}^u(E)) \leq C(\mathbf{A}^u)$.

Example. Consider the search tree in Fig. 2. Among the five possible children of the root node, three (those that assign jobs 1, 4 and 5) are fathomed by Dominance rule 1. All children of N^{11} are eliminated by Dominance rule 2 because $\max\{4; 18\} = 18 \leq 23 = \mathcal{L}(N^{11})$. All children N^y of N^5 for which $tb(N^y) = tb(N^5) + 1 = 2$ are dominated by Dominance rule 3 because for this node $j^* = 5$ and thus $r_{j^*} + \sum_{j \in U(N^5)} p_j = 14 + 20 = 34 \geq 31 = \text{UB}$. Also, children of N^6 with target block $tb(N^6) + 1 = 2$ are fathomed by Dominance rule 3 because $\mathbf{A}^6$ is not feasible. The remaining children of N^6 are eliminated by Dominance rule 1. For this specific instance, Dominance rule 4 does not eliminate any node because all nodes that can be dominated by Dominance rule 4 are removed by other dominance rules.

5. A guess-and-check method

In this section, we introduce a *guess-and-check algorithm* (GC) that solves our problem to optimality. The idea is to iteratively guess a minimum makespan value Γ for our problem and then, using an iterative DP approach, test the correctness of our guesses. The guess Γ is correct if there is at least one feasible sequence of jobs with a makespan less than or equal to Γ and no sequence with a makespan not exceeding $\Gamma - 1$. This method only works if the time horizon and all possible inventory levels are discrete.

Algorithm 2 summarizes our GC algorithm. The verification of each guess is done by function *verify*(Γ), which returns either a sequence $\boldsymbol{\sigma}$ with $C_{\max}(\boldsymbol{\sigma}) \leq \Gamma$ or *null*; the latter means that no sequence with a makespan smaller than or equal to Γ exists. This function is explained in Section 5.1. Initially, we set $\Gamma = \text{UB}_{ini}$ (see the first paragraph of Section 4.4). In each iteration, if there is a feasible sequence $\boldsymbol{\sigma}$ with $C_{\max}(\boldsymbol{\sigma}) \leq \Gamma$, we update our guess and set it to $C_{\max}(\boldsymbol{\sigma}) - 1$. If, at any point, $\boldsymbol{\sigma}$ is proved to be an early-opt solution, the algorithm halts.

Algorithm 2 Guess-and-check.

Input: Instance I of our problem.
1: $\Gamma = \text{UB}_{ini}$
2: **while** *verify*(Γ) outputs a feasible sequence $\boldsymbol{\sigma}$ **do**
3: $\quad \Gamma = C_{\max}(\boldsymbol{\sigma}) - 1$
4: $\quad$ **if** $\boldsymbol{\sigma}$ is early-opt **then**
5: $\quad\quad$ **exit while**
6: $\quad$ **end if**
7: **end while**
Output: $\Gamma + 1$

In Algorithm 2, we deliberately opt for an incremental search rather than a binary search because, for a given Γ, the procedure associated with the function *verify*(Γ) is computationally much less expensive when there is a feasible solution (*verify*(Γ) returns $\boldsymbol{\sigma}$) than when there is no feasible solution (*verify*(Γ) returns *null*). The current incremental search approach guarantees that Algorithm 2 stops immediately after the first incident where *verify*(Γ) returns *null*.

5.1. An equivalent graph problem

We introduce a graph problem to verify Γ. Given Γ, we construct the following *directed acyclic graph* (DAG) $G_\Gamma = (V_\Gamma, A_\Gamma)$. Let $t_0 = \Gamma - \sum_{j \in J} p_j$ and $t_n = \Gamma$. There is a vertex $v_{t,I} \in V_\Gamma$ for each $t \in \{t_0, \ldots, t_n\}$ and $I \in \{0, \ldots, I_C\}$. Also, there is an arc $a_{t,I,j} \in A_\Gamma$ connecting $v_{t,I}$ and $v_{t+p_j, I+\delta_j}$ if and only if $r_j \leq t \leq \Gamma - p_j$ and $0 \leq I + \delta_j \leq I_C$. Vertices in this DAG refer to time and inventory levels, whereas arcs correspond to execution of jobs. Fig. 3 depicts the graph G_{26} for the instance of Section 2.

Let $I_n = I_0 + \sum_{j \in J} \delta_j$. We refer to an arc that corresponds to executing job j as a *j-arc* and to a path from v_{t_0,I_0} to v_{t_n,I_n} that includes exactly one j-arc for each $j \in J$ as a *sequence-feasible* path. Finding a feasible sequence with a makespan equal to Γ is equivalent to finding a sequence-feasible path in G_Γ. Thus, we formulate the following decision problem:

Problem SEQFEAS
Instance: G_Γ
Question: Is there a sequence-feasible path in G_Γ?

If there is a sequence-feasible path in G_Γ (G_Γ is a YES-instance) then *verify*(Γ) outputs the associated sequence, and if G_Γ has no sequence-feasible path (G_Γ is a NO-instance) then *verify*(Γ) outputs *null*. Unfortunately, SEQFEAS is NP-complete (since it is equivalent to DECP). In what follows, we propose an iterative algorithm to solve SEQFEAS.

5.2. An iterative algorithm to solve SEQEFAS

The algorithm iteratively applies DP-based tools, either to prove that G_Γ is a NO-instance, or to find a sequence-feasible path in G_Γ. This algorithm includes four main elements: a NO-instance detection procedure, a YES-instance detection procedure, a pruning procedure, and a brute-force search procedure. The YES- and NO-instance detection procedures are subroutines that may detect YES- and NO-instances, respectively, the pruning procedure prunes arcs in G_Γ that are certainly not in any sequence-feasible path, and the brute-force search procedure thoroughly searches G_Γ to find a sequence-feasible path. The first three procedures and a truncated version of the last procedure are called in each iteration until either a conclusive answer to SEQEFAS is achieved or we lose hope in achieving an answer by continuing to iterate. If no conclusive answer is obtained, a non-truncated version of the brute-force search procedure is called.

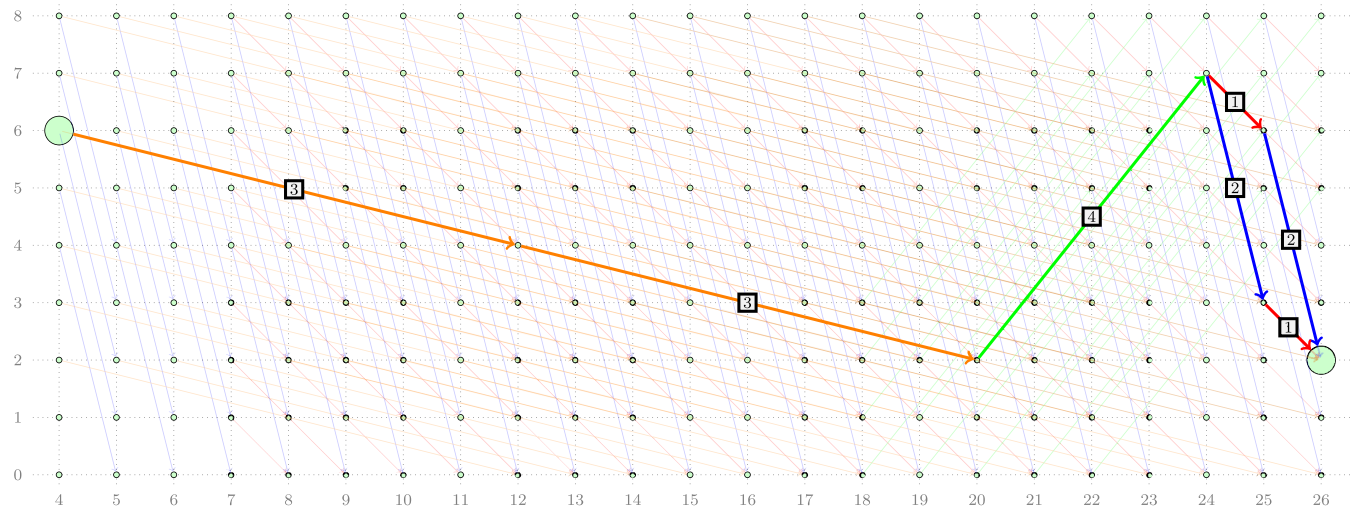

Fig. 3. Graph G_{26} for the instance of Section 2, where $t_0 = 4$. The horizontal dimension of the grid corresponds with time, the vertical dimension with inventory.

5.2.1. NO-instance detection

We will say that a path is *inventory-feasible* if it starts from v_{t_0,I_0} and ends at v_{t_n,I_n}. Inventory-feasible paths may have more than one j-arc for some jobs j and no j'-arc for some other jobs j'. Note that all sequence-feasible paths are also inventory-feasible. In Fig. 3, two inventory-feasible paths are highlighted in bold. These two paths include two 3-arcs and no 5-arc and are therefore not sequence-feasible.

We are only interested in inventory-feasible paths that are also sequence-feasible. We devise a Lagrangian-based approach to penalize paths that are inventory-feasible but not sequence-feasible. We introduce Lagrangian multipliers μ_j for each job j and associate a cost to each arc $a_{t,I,j}$ (if it exists) as follows:

$$c(a_{t,I,j}) = \begin{cases} -\mu_j + \sum_{i \in J} \mu_i & \text{if } t = t_0 \text{ and } I = I_0 \\ -\mu_j & \text{otherwise} \end{cases}.$$

We refer to this weighted version of G_Γ as $G_\Gamma^{\boldsymbol{\mu}}$. We also denote the shortest path from v_{t_0,I_0} to v_{t_n,I_n} in $G_\Gamma^{\boldsymbol{\mu}}$ and its length by $P_\Gamma^{\boldsymbol{\mu}}$ and $L_\Gamma^{\boldsymbol{\mu}}$, respectively. We observe that the length of every sequence-feasible path in $G_\Gamma^{\boldsymbol{\mu}}$ is zero, regardless of the choice for $\boldsymbol{\mu}$. Thus, no sequence-feasible path exists in G_Γ if $L_\Gamma^{\boldsymbol{\mu}} > 0$ for some $\boldsymbol{\mu}$. As an illustration, for the graph of Fig. 3, let us choose $\boldsymbol{\mu}_1 = (0, 0, -1, 0, 1)$. We have $L_{26}^{\boldsymbol{\mu}_1} = 2 > 0$, which guarantees the non-existence of a sequence-feasible path. Remark that this is a sufficient condition for non-existence, but not a necessary one.

In order to compute $P_\Gamma^{\boldsymbol{\mu}}$ and $L_\Gamma^{\boldsymbol{\mu}}$, we use the following DP recursion.

$$g_0(t, I) = \begin{cases} 0 & \text{if } t = t_n \text{ and } I = I_n \\ +\infty & \text{if } A_{t,I}^+ = \emptyset \\ \min_{a_{t,I,j} \in A_{t,I}^+} \left\{ c(a_{t,I,j}) + g_0(t + p_j, I + \delta_j) \right\} & \text{otherwise} \end{cases} \tag{14}$$

where $A_{t,I}^+$ is the set of all arcs leaving $v_{t,I}$. Clearly, $L_\Gamma^{\boldsymbol{\mu}}$ equals $g_0(t_0, I_0)$; $P_\Gamma^{\boldsymbol{\mu}}$ can be retrieved in linear time. The recursion runs in time $O(nI_C\Sigma_{j \in J} p_j)$. When there is no inventory-feasible path in $G_\Gamma^{\boldsymbol{\mu}}$ then $L_\Gamma^{\boldsymbol{\mu}} = \infty$.

Ideally, we would like to find $\boldsymbol{\mu}^* := \arg\max_{\boldsymbol{\mu}} \{L_\Gamma^{\boldsymbol{\mu}}\}$, but this is not a trivial task because it requires solving a rather complicated two-stage problem. We therefore iteratively improve our choice of $\boldsymbol{\mu}$ using an adapted version of the *conjugate sub-gradient algorithm* described in Tanaka and Fujikuma (2012). Let $\boldsymbol{\mu}^k$ denote the vector of multipliers in the kth iteration. In iteration k, we compute the shortest path in $G_\Gamma^{\boldsymbol{\mu}^k}$ using recursion (14). Let $o_j^{\boldsymbol{\mu}^k}$ be the number of j-arcs in $P_\Gamma^{\boldsymbol{\mu}^k}$. We compute $\boldsymbol{\mu}^{k+1}$ as follows:

$$\mu_j^{k+1} = \mu_j^k + \frac{V^k d_j^{k+1}}{\sum_{j \in J} {d_j^{k+1}}^2},$$

where

$$d_j^{k+1} = \begin{cases} \xi d_j^k + (1 - o_j^{\boldsymbol{\mu}^k}) & \text{if } k \neq 0 \\ (1 - o_j^{\boldsymbol{\mu}^k}) & \text{if } k = 0 \end{cases} \text{ and } \xi = \frac{\sqrt{\sum_{j \in J} \left(1 - o_j^{\boldsymbol{\mu}^k}\right)^2}}{\sqrt{\sum_{j \in J} d_j^{k^2}}}.$$

Also, $\boldsymbol{\mu}^0 = (0, \ldots, 0)$, $\mathbf{d}^0 = (0, \ldots, 0)$, and V^k stands for the lowest encountered violation cost so far (see Section 5.2.2).

5.2.2. YES-instance detection

The value $L_\Gamma^{\boldsymbol{\mu}^k}$ can be less than or equal to zero in iteration k of the sub-gradient algorithm, which suggests (but does not guarantee) the existence of a sequence-feasible path. We then attempt to construct such a path. We construct an initial sequence $\boldsymbol{\sigma}$ of jobs by following the order of their associated j-arcs in $P_\Gamma^{\boldsymbol{\mu}}$. If there are two or more arcs for the same job j, we only consider the first occurrence. If there are no arcs for some jobs, we add those jobs to the end of the sequence in the order of their release dates. We execute jobs based on this sequence between t_0 and t_n without intermediate idle time. If the sequence is feasible, G_Γ is a YES-instance. If not then we penalize the infeasiblities and try to find a feasible solution using local search techniques.

We introduce violation costs $\theta_{jtI} = \max\{0, r_j - (t - p_j)\} + \max\{0, I - I_C, -I\}$ that are incurred if job j is completed at time t when the inventory level reaches I. For a given sequence $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, we compute its total violation cost as follows:

$$\theta_{\boldsymbol{\sigma}} = \theta_{\sigma_1, t_0 + p_{\sigma_1}, I_0 + \delta_{\sigma_1}} + \theta_{\sigma_2, t_0 + p_{\sigma_1} + p_{\sigma_2}, I_0 + p_{\sigma_1} + p_{\sigma_2}} + \cdots + \theta_{\sigma_n, t_0 + \sum_{i \in J} p_i, I_0 + \sum_{i \in J} \delta_i}.$$

We apply both a steepest descent local search and a time-window heuristic (Davari et al., 2016, Section 7) to minimize this violation cost.

In the time-window heuristic, jobs are first partitioned into $\lceil n/10 \rceil$ sets $J_{W_1}, \ldots, J_{W_{\lceil n/10 \rceil}}$ as follows:

$$J_{W_1} = \{\sigma_1, \ldots, \sigma_{10}\};$$

$$\ldots$$

$$J_{W_{\lceil n/10 \rceil}} = \{\sigma_{10(\lceil n/10 \rceil - 1) + 1}, \ldots, \sigma_n\}.$$

For each $s \in \{1, \ldots, \lceil n/10 \rceil\}$, we build a subproblem W_s with job set J_{W_s}. The starting time for subproblem W_s is $t_{W_s} = t_0 + \sum_{j=1}^{(s-1)\cdot 10} p_{\sigma_j}$ and the starting inventory level is $I_{W_s} = I_0 + \sum_{j=1}^{(s-1)\cdot 10} \delta_{\sigma_j}$. Each subproblem W is then solved by the following DP recursion:

$$g_1(S) = \begin{cases} 0 & \text{if } S = \emptyset \\ \min_{j \in S} \left\{ \theta_{j, t_W(S), I_W(S)} + g_1(S \setminus \{j\}) \right\} & \text{otherwise} \end{cases} \quad (15)$$

where $t_W(S) = t_W + \sum_{j \in S} p_j$ and $I_W(S) = I_W + \sum_{j \in S} \delta_j$. Recursion (15) outputs an optimal sequence for each subproblem W with lowest violation cost, assuming that the ordering of all the other jobs in $J \backslash J_W$ is maintained.

Algorithm 3 is a combination of a steepest descent procedure and the time-window heuristic for minimizing the violation cost. If the resulting sequence $\boldsymbol{\sigma}$ has zero violation, we conclude that G_Γ is a YES-instance; otherwise we compute $V^k = \min\{V^{k-1}, \theta_{\boldsymbol{\sigma}}\}$ (with $V^0 = \infty$), which is used in kth iteration of the conjugate subgradient algorithm described in Section 5.2.1.

Algorithm 3 A YES-instance detection procedure.

Input: $P_\Gamma^{\boldsymbol{\mu}}$.

Construct a feasible sequence $\boldsymbol{\sigma}$ from $P_\Gamma^{\boldsymbol{\mu}}$.
$\boldsymbol{\sigma} \leftarrow steepestdescent(\boldsymbol{\sigma})$
while $\theta_{\boldsymbol{\sigma}} > 0$ **do**
  for $k = 1$ to $\lceil n/10 \rceil$ **do**
    $\boldsymbol{\sigma}_W \leftarrow g_1(W_k)$
  end for
  $\boldsymbol{\sigma}' = (\boldsymbol{\sigma}_{W_1}, \ldots, \boldsymbol{\sigma}_{W_{\lceil n/10 \rceil}})$
  if $\theta_{\boldsymbol{\sigma}'} = \theta_{\boldsymbol{\sigma}}$ **then**
    return $\boldsymbol{\sigma}$
  else
    $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma}'$
  end if
end while

Output: $\boldsymbol{\sigma}$

5.2.3. Pruning G_Γ

There will typically be a number of arcs in G_Γ that are not part of any sequence-feasible path, and thus can be pruned. Let $L_\Gamma^{\boldsymbol{\mu}}(v \to v')$ be the length of a shortest path from vertex v to v' in $G_\Gamma^{\boldsymbol{\mu}}$. An arc $a_{t,I,j}$ satisfying the following inequality can be pruned:

$$c(a_{t,I,j}) + L_\Gamma^{\boldsymbol{\mu}}(v_{t_0,I_0} \to v_{t,I}) + L_\Gamma^{\boldsymbol{\mu}}(v_{t+p_j, I+\delta_j} \to v_{t_n,I_n}) > 0.$$

Pruning G_Γ can be achieved in $O(nI_C \Sigma_{j \in J} p_j)$ time.

5.2.4. A brute-force search procedure

It is possible that the combination of the NO-instance and YES-instance recognition procedures and the pruning fails to provide a conclusive answer to SEQFEAS. In such a case, and as a last resort, we apply a brute-force search procedure to solve SEQFEAS. We search G_Γ for a sequence-feasible path by traversing the graph from v_{t_0,I_0} to v_{t_n,I_n}. In this process, for each visited vertex $v_{t,I}$, we keep track of the partial sequence $\hat{\boldsymbol{\sigma}}_{v_{t,I}}$ and the corresponding set $S_{v_{t,I}}$ of executed jobs. From vertex $v_{t,I}$, we backtrack to the previous vertex and choose a different arc if

1. $\hat{\boldsymbol{\sigma}}_{v_{t,I}}$ includes more than one occurrence of some job j, or
2. $L_\Gamma^{\boldsymbol{\mu}}(v_{t+p_j,I+\delta_j} \to v_{t_n,I_n}) + \sum_{j \in J \backslash S_{v_{t,I}}} \mu_j > 0$, where $\boldsymbol{\mu}$ is the last vector of Lagrangian multipliers, or
3. $\hat{\boldsymbol{\sigma}}_{v_{t,I}}$ is dominated by another partial sequence for the same job set $S_{v_{t,I}}$ based on the dominance theorem of DP (Davari et al., 2016; Jouglet et al., 2004): among two partial sequences that are both feasible, the one which has lexicographically larger job indices is dominated. For practical reasons, each partial sequence is only compared with the alternative sequences in which the order of the last five jobs is different.

The procedure halts either when it reaches v_{t_n,I_n}, which means a sequence-feasible path is found, or when there is no more path to traverse. In the former case the procedure outputs *found*; in the latter case it returns *none*. This procedure is computationally demanding and is therefore only invoked after we lose hope in the success of the combined iterative YES- and NO-detection procedure. We do, however, exploit a truncated version of this procedure in each iteration: this truncated version is interrupted if the number of vertices visited reaches one hundred thousand, in which case it outputs *null*. Note that a vertex can be visited more than once while visiting different paths and that each visit counts.

5.2.5. Overall scheme of the algorithm

Algorithm 4 describes the overall scheme of the iterative algorithm that solves SEQFEAS. In this algorithm, $dp(G_\Gamma^{\boldsymbol{\mu}})$ is the DP recursion (14) applied to $G_\Gamma^{\boldsymbol{\mu}}$, $heuristics(P_\Gamma^{\boldsymbol{\mu}})$ is Algorithm 3 applied to $P_\Gamma^{\boldsymbol{\mu}}$, $prune(\boldsymbol{\mu}, G_\Gamma)$ is the process of pruning G_Γ using its weighted

Algorithm 4 The iterative algorithm to solve SEQFEAS.

Input: G_Γ as an instance of SEQFEAS.

$\boldsymbol{\mu}^0 = (0, \ldots, 0)$
$k = 0$
$itrwtimp = 0$
$L_{\text{best}} = -\infty$
while $k \leq 1000$ and $itrwtimp \leq 200$ **do**
  $G_\Gamma^{\boldsymbol{\mu}^k} \leftarrow G_\Gamma, \boldsymbol{\mu}^k$
  $P_\Gamma^{\boldsymbol{\mu}^k}, L_\Gamma^{\boldsymbol{\mu}^k} \leftarrow dp(G_\Gamma^{\boldsymbol{\mu}^k})$
  if $L_\Gamma^{\boldsymbol{\mu}^k} > 0$ **then**
    return NO-instance
  end if
  if $k \geq 10$ **then**
    if $L_{\text{best}} < L_\Gamma^{\boldsymbol{\mu}^k}$ **then**
      $L_{\text{best}} = L_\Gamma^{\boldsymbol{\mu}^k}$ and $itrwtimp = 0$
    else
      $itrwtimp = itrwtimp + 1$
    end if
  end if
  $\boldsymbol{\sigma} \leftarrow heuristics(P_\Gamma^{\boldsymbol{\mu}^k})$
  if $\theta_{\boldsymbol{\sigma}} = 0$ **then**
    return YES-instance
  end if
  $result \leftarrow bruteTruncated(\boldsymbol{\mu}^k, G_\Gamma)$
  if $result = found$ **then**
    return YES-instance
  else if $result = none$ **then**
    return NO-instance
  end if
  $G_\Gamma \leftarrow prune(\boldsymbol{\mu}^k, G_\Gamma)$
  $\boldsymbol{\mu}^{k+1} \leftarrow update(\boldsymbol{\mu}^k, P_\Gamma^{\boldsymbol{\mu}^k})$
  $k = k + 1$
end while
$result \leftarrow brute(\boldsymbol{\mu}^k, G_\Gamma)$
if $result = found$ **then**
  return YES-instance
else
  return NO-instance
end if

Output: NO-instance or YES-instance

counterpart $G_{\Gamma}^{\boldsymbol{\mu}}$, *update*$(\boldsymbol{\mu}, P_{\Gamma}^{\boldsymbol{\mu}})$ is our *conjugate sub-gradient procedure* as described in Section 5.2.1, and finally *brute*$(\boldsymbol{\mu}^k, G_{\Gamma})$ and *bruteTruncated*$(\boldsymbol{\mu}^k, G_{\Gamma})$ are the brute-force search procedure and its truncated version, respectively.

In more detail, lines 6–10 of Algorithm 4 are devoted to the NO-instance detection, lines 11–17 compute the number of iterations without improvement, lines 18–21 relate to the YES-instance detection procedure, lines 22–27 comprise the truncated brute-force search, lines 28–29 are the pruning and the conjugate sub-gradient procedures, and finally lines 32–37 relate to the complete brute-force search.

6. Computational results

All algorithms have been implemented in VC++ 2015, and Cplex 12.7.1 is used to solve the MIP formulations. All computational results were obtained on a laptop Dell Latitude with 2.6 GHz Core(TM) i7-3720QM processor, 8GB of RAM, running under Windows 10.

6.1. Instance generation

To the best of our knowledge, there are no publicly available instance sets of our problem. We therefore generate our own instances, with $n = 10$, 20, 30, 40 and 50 jobs. The values p_i $(1 \le i \le n)$ are sampled from a uniform integer distribution on interval $[1, \alpha]$, where $\alpha \in \{10, 100\}$. Release dates r_i are drawn from a uniform integer distribution on $[0, \tau \Sigma_{i \in J} p_i]$ with $\tau \in \{0.5, 1, 1.5, 2\}$, and the absolute values $|\delta_i|$ of the inventory modifications stem from a uniform integer distribution on [1,10]. The inventory capacity I_C is selected randomly from $[10\eta, 20\eta]$, where $\eta = \{1, 3, 5\}$. The assignment of jobs to the two subsets J^- and J^+ and the value I_0 of the initial inventory might give rise to many infeasible instances. For this reason, we assign jobs and choose the value I_0 randomly but with extra attention. We first assign each job either to J^- or to J^+, each with a 50% probability. Subsequently, if $\Sigma_{j \in J} \delta_j > I_C$ or $\sum_{j \in J} \delta_j < -I_C$ then we repeat the assignment procedure, until both $\Sigma_{j \in J} \delta_j \le I_C$ and $\sum_{j \in J} \delta_j \ge -I_C$. Finally, we choose an integer value for the initial inventory I_0 from the following interval:

$$\left[\min \left\{ I_C; \max \left\{ 0; 0 - \sum_{j \in J} \delta_j \right\} \right\}, \ \max \left\{ 0; \min \left\{ I_C; I_C - \sum_{j \in J} \delta_j \right\} \right\} \right].$$

In conclusion, for each combination of (n, α, τ, η), four instances are generated; the total number of instances is thus $5 \times 2 \times 4 \times 3 \times 4 = 480$. None of the generated instances turns out to be infeasible, which is not very surprising because our instance generation scheme ensures that $0 \le I_0 + \sum_{j \in J} \delta_j \le I_C$. Instances with this property might still be infeasible, but such instances will be quite rare, especially for larger n values.

In all our experiments, the time limit is set to 1000 seconds. If an instance is not solved to guaranteed optimality, it is said to be 'unsolved' for the procedure. Throughout this section, we report averages computed over all instances, both solved and unsolved (for an unsolved instance, we report a CPU time of 1000 seconds).

6.2. Overall results

We compare four different solution methods, namely the time-indexed formulation (TIF), the sequence-based formulation (SBF), the block-based branch-and-bound algorithm (BB), and the guess-and-check algorithm (GC). The overall results are given in Table 2.

We first compare the two MIP formulations TIF and SBF. In general, SBF performs better than TIF. The main reason is that the number of variables in TIF strongly increases with $\Sigma_{j \in J} p_j$, leading to a significant growth of the associated branch-and-cut tree generated by the solver. We will see in Section 6.3, nevertheless, that TIF is slightly better than SBF when job processing times are small and $n \ge 30$.

BB is very fast for instances of size $n = 10$, can solve all instances with size $n = 20$, and struggles to solve instances of size $n \ge 30$. GC, finally, clearly outperforms all the other solution methods. Unlike the other three methods, GC solves all instances of size $n = 30$ in only a few seconds. It fails to solve only three instances of size $n = 40$ and only seven instances with $n = 50$ within the time limit.

Figs. 4–7 depict the number of instances solved to optimality within different time limits for the different solution methods. Fig. 4, for instance, shows that TIF does not solve any instance of size $n = 50$ within the first 10 seconds, whereas the other three methods all solve at least some instances even within one second. BB, in particular, performs remarkably well from the start (see Fig. 6): it solves more than 60% of the instances within the first second. This performance could be explained by the early detection of early-opt solutions and the tightness of the bounds in some instances. This good performance in the first second, however, is not continued with larger runtimes: BB does not solve many extra instances after the first seconds, especially when $n > 30$. GC, by contrast, exhibits a good performance during the first seconds, but also successfully solves more and more instances as the time limit increases (see Fig. 7). When comparing the performance of BB and GC for the first second, we observe that GC performs slightly better for instances with $n \le 40$, while it performs worse for instances with $n \ge 50$. To support this observation, we refer to Figs. 6 and 7, and also to Table 6 in Section 6.4, where results are reported also for $n > 50$.

6.3. Sensitivity analysis

We now examine the performance of the different solution methods under specific parameter settings. We first study the effect of the range from which processing times are generated (via parameter α). Table 3 compares all four solution methods when $\alpha = 10$ and 100. We find that the performance of TIF and GC significantly deteriorate with increasing α, whereas SBF and BB are only slightly sensitive to α. This is not surprising, as both TIF and the first DP recursion in GC are time-indexed. We also observe that TIF performs better than SBF when $\alpha = 10$ and $n \ge 30$.

Table 4 shows the effect of varying the parameter τ. SBF has the best performance when release dates are tight ($\tau = 0.5$) and the worst when release dates are neither tight nor loose ($\tau = 1.0$ or 1.5). This is not illogical: the linear relaxation of SBF for instances with tight release dates is generally tight. Interestingly, SBF performs better than the other three methods when $n = 50$ and $\tau = 0.5$, and worse than the other three when $n = 50$ and $\tau \ge 1.0$. TIF improves as τ goes up, whereas BB and GC are less performant with increasing τ. For small instances ($n \le 20$), BB is the most efficient method when $\tau \le 1.0$, while for $\tau \ge 1.5$, GC is the best.

The parameter η controls the capacity of the inventory storage. Table 5 shows the results when $\eta = 1$ (low storage space), 3 (medium), and 5 (large). All methods become faster with increasing η.

6.4. The performance of BB and GC on larger instances

In order to investigate the behavior of BB and GC on larger instances, we generate another dataset, following the same proce-

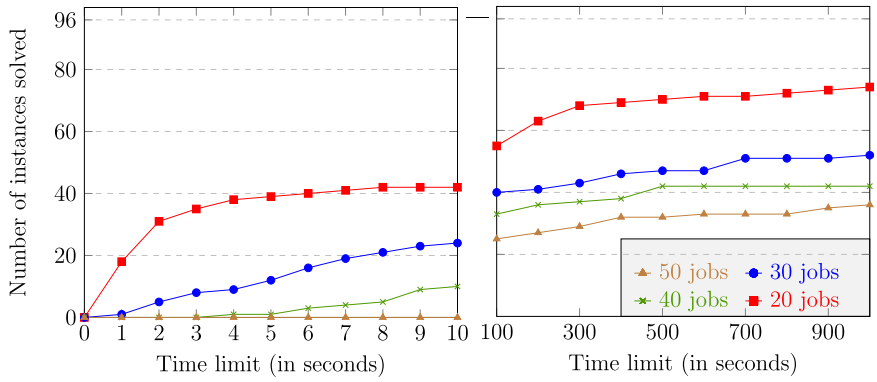

Fig. 4. Number of instances solved to optimality by TIF within different time limits.

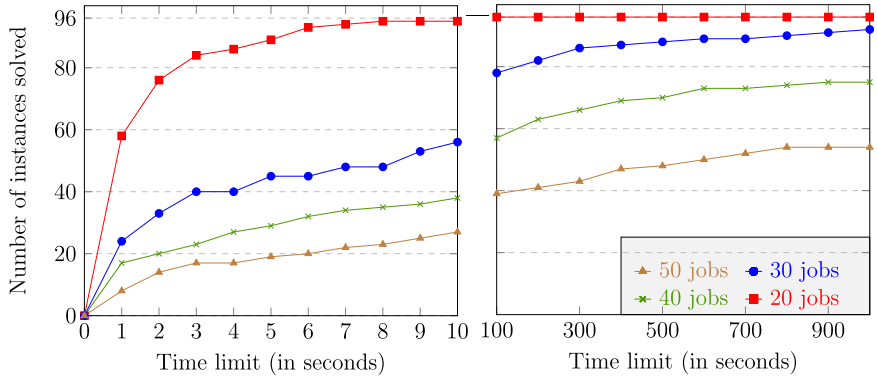

Fig. 5. Number of instances solved to optimality by SBF within different time limits.

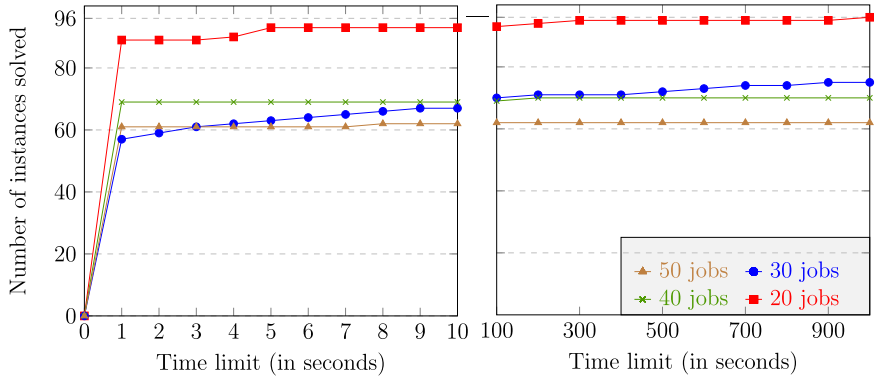

Fig. 6. Number of instances solved to optimality by BB within different time limits.

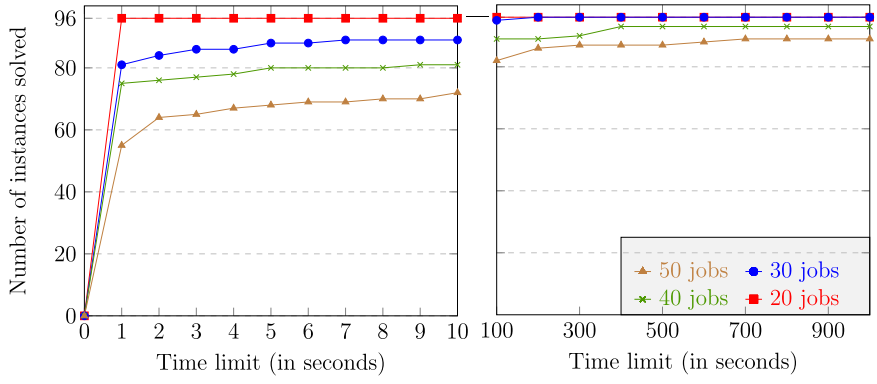

Fig. 7. Number of instances solved to optimality by GC within different time limits.

Table 2
Average CPU times (in seconds) and number of unsolved instances within the time limit (out of 96, between parentheses) for $n = 10, 20, 30, 40$ and 50.

Method	n									
	10		20		30		40		50	
TIF	10.22	(0)	299.93	(22)	525.11	(44)	600.27	(67)	690.07	(86)
SBF	0.10	(0)	1.35	(0)	109.42	(4)	292.47	(21)	512.31	(42)
BB	**< 0.01**	(0)	1.94	(0)	193.86	(17)	271.60	(26)	354.25	(34)
GC	**< 0.01**	(0)	**0.09**	**(0)**	**5.02**	**(0)**	**48.04**	**(3)**	**99.15**	**(7)**

Table 3
Average CPU times (in seconds) and number of unsolved instances within the time limit (out of 48, between parentheses) for $n = 10, 20, 30, 40$ and 50, and $\alpha = 10$ and 100.

Method	α	n									
		10		20		30		40		50	
TIF	10	0.26	(0)	4.42	(0)	90.54	(2)	199.56	(6)	379.13	(12)
	100	20.19	(0)	595.45	(22)	959.69	(42)	-	(48)	-	(48)
SBF	10	0.11	(0)	1.55	(0)	131.57	(2)	347.90	(12)	565.74	(23)
	100	0.10	(0)	1.15	(0)	87.27	(2)	237.04	(9)	458.88	(19)
BB	10	**< 0.01**	(0)	3.75	(0)	130.41	(6)	187.95	(9)	312.67	(15)
	100	**< 0.01**	(0)	0.13	(0)	257.30	(11)	354.17	(17)	395.83	(19)
GC	10	**< 0.01**	**(0)**	**0.05**	**(0)**	**0.57**	**(0)**	**6.94**	**(0)**	**72.17**	**(3)**
	100	**< 0.01**	**(0)**	**0.12**	**(0)**	**9.47**	**(0)**	**89.14**	**(3)**	**126.13**	**(4)**

Table 4
Average CPU times (in seconds) and number of unsolved instances within the time limit (out of 24, between parentheses) for $n = 10, 20, 30, 40$ and 50, and $\tau = 0.5, 1.0, 1.5$ and 2.0.

Method	τ	n									
		10		20		30		40		50	
TIF	0.5	22.03	(0)	505.30	(12)	593.99	(13)	730.00	(16)	895.77	(20)
	1.0	7.41	(0)	404.26	(7)	579.40	(13)	596.19	(13)	739.43	(16)
	1.5	4.50	(0)	186.82	(2)	492.06	(11)	564.35	(13)	562.09	(12)
	2.0	6.96	(0)	103.35	(1)	435.00	(7)	510.55	(12)	562.98	(12)
SBF	0.5	0.11	(0)	0.17	(0)	0.80	(0)	**1.16**	**(0)**	**4.15**	**(0)**
	1.0	0.10	(0)	2.22	(0)	124.97	(2)	330.33	(6)	785.58	(16)
	1.5	0.10	(0)	1.44	(0)	118.83	(0)	656.62	(14)	678.59	(14)
	2.0	0.11	(0)	1.58	(0)	193.08	(2)	181.79	(1)	580.91	(12)
BB	0.5	**< 0.01**	**(0)**	**< 0.01**	**(0)**	3.31	(0)	88.33	(2)	250.00	(6)
	1.0	**< 0.01**	**(0)**	**0.02**	**(0)**	52.24	(0)	459.23	(11)	666.67	(16)
	1.5	**< 0.01**	**(0)**	2.96	(0)	344.88	(8)	250.00	(6)	208.33	(5)
	2.0	**< 0.01**	**(0)**	4.79	(0)	375.00	(9)	291.67	(7)	292.00	(7)
GC	0.5	**< 0.01**	**(0)**	0.14	(0)	**0.40**	**(0)**	6.13	(0)	11.06	(0)
	1.0	**< 0.01**	**(0)**	0.04	(0)	**0.23**	**(0)**	**11.91**	**(0)**	**128.97**	**(2)**
	1.5	**< 0.01**	**(0)**	**0.06**	**(0)**	**7.24**	**(0)**	**75.78**	**(1)**	**72.11**	**(1)**
	2.0	**< 0.01**	**(0)**	**0.11**	**(0)**	**12.21**	**(0)**	**98.36**	**(2)**	**184.46**	**(4)**

Table 5
Average CPU times (in seconds) and number of unsolved instances within the time limit (out of 32, between parentheses) for $n = 10, 20, 30, 40$ and 50, and $\eta = 1, 3$ and 5.

Method	η	n									
		10		20		30		40		50	
TIF	1	15.76	(0)	384.49	(11)	608.47	(18)	735.86	(22)	817.29	(23)
	3	8.26	(0)	242.54	(5)	490.31	(13)	550.54	(16)	659.09	(20)
	5	6.65	(0)	272.77	(6)	476.55	(13)	514.42	(16)	593.82	(17)
SBF	1	0.11	(0)	1.67	(0)	153.92	(3)	412.22	(10)	502.58	(13)
	3	0.12	(0)	1.40	(0)	96.97	(0)	248.33	(6)	559.85	(15)
	5	0.08	(0)	0.98	(0)	77.37	(1)	77.37	(5)	474.50	(14)
BB	1	**< 0.01**	**(0)**	4.94	(0)	289.94	(8)	563.17	(18)	656.50	(21)
	3	**< 0.01**	**(0)**	0.16	(0)	128.88	(4)	187.50	(6)	281.25	(9)
	5	**< 0.01**	**(0)**	0.72	(0)	162.76	(5)	62.50	(2)	125.00	(4)
GC	1	**< 0.01**	**(0)**	**0.07**	**(0)**	**2.51**	**(0)**	**76.55**	**(2)**	**137.78**	**(3)**
	3	**< 0.01**	**(0)**	**0.07**	**(0)**	**3.92**	**(0)**	**64.45**	**(1)**	**115.38**	**(3)**
	5	**< 0.01**	**(0)**	**0.12**	**(0)**	**8.64**	**(0)**	**3.13**	**(0)**	**44.30**	**(1)**

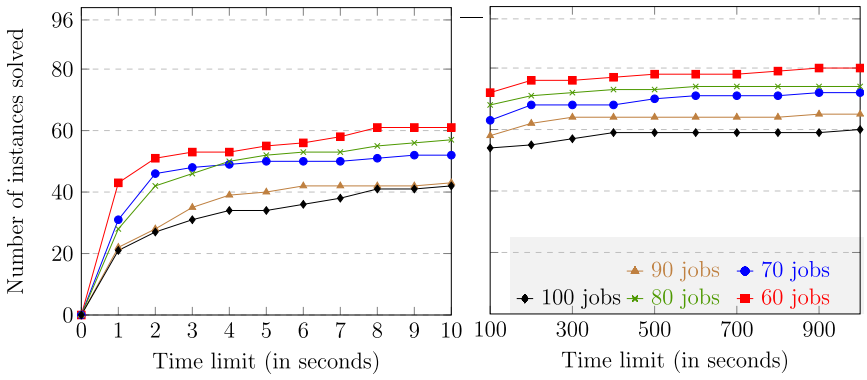

Fig. 8. Number of large instances solved to optimality by GC within different time limits.

Table 6
Number of instances solved within the first second (out of 96) for large instances.

Method	n				
	60	70	80	90	100
BB	56	63	60	59	56
GC	43	31	28	22	21

dure as in Section 6.1 but now with $n = 60$, 70, 80, 90 and 100 jobs.

We have observed (the details are not reported here) that BB rarely solves additional instances after the first second when $n \geq 50$. We therefore first run both BB and GC on the new instances with a time limit of one second. Table 6 shows that, within this time limit, BB solves more instances than GC. More interestingly, the proportion of the instances that BB solve in this timespan is always around 60%, regardless of n.

In another experiment, we apply GC to this dataset with a time limit of up to 1000 seconds; Table 7 contains the detailed computational results. The table shows that GC solves the majority of the instances, although obviously this becomes more difficult as n gets closer to 100. The number of instances solved within different time limits is plotted in Fig. 8 to provide a more elaborate picture of the performance of GC for large n.

7. Summary and conclusion

In this paper, we have studied single-machine scheduling with release dates and inventory constraints to minimize the makespan. We have shown that the problem is strongly NP-hard, and we have proposed two MIP formulations, a branch-and-bound algorithm and a guess-and-check algorithm. The novelty of our branch-and-bound method is its block-based representation of a solution, while the specific character of the guess-and-check algorithm resides in its dynamic-programming-based verification procedures. We have compared the computational performance of the branch-and-bound procedure, the guess-and-check algorithm, and the two MIP formulations on a set of test instances. We observe that the guess-and-check algorithm outperforms the other methods for most of the problem settings.

Acknowledgments

The authors thank prof. dr. Dirk Briskorn from Bergische Universität Wuppertal for sharing some interesting ideas on the complexity of the problem. The authors are also grateful to the anonymous reviewers for their constructive feedback on this work, especially to Reviewer 2, who suggested to test the SBF formulation. This research has been partially supported by COMEX (Project P7/36) and the BELSPO/IAP Programme.

Appendix

Proof of Theorem 1. We show the NP-hardness of $1|inv, r_j|C_{\max}$ by a reduction from 3-PARTITION. This reduction shows that determining the feasibility of an instance of $1|inv, r_j|C_{\max}$ is already strongly NP-complete.

Problem 3-PARTITION
Instance: $3m+1$ integers $\alpha_1, \alpha_2, \ldots, \alpha_{3m}$ and β such that $\beta/4 < \alpha_j < \beta/2$ and $\sum_{i=1}^{3m} \alpha_i = m\beta$.
Question: Does there exist a partition $\{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ such that $\sum_{i \in \mathcal{A}_s} \alpha_i = \beta$ for all $s \in \{1, \ldots, m\}$?

Given an instance of 3-PARTITION, we can construct an instance of $1|inv, r_j|C_{\max}$ as follows:

- $J^- = \{1, 2, \ldots, 3m\}$, $J^+ = \{3m+1, 3m+2, \ldots, 4m\}$, $I_0 = I_C = \beta$.

Table 7
Average CPU times and number of unsolved large instances within the time limit for GC.

	n									
	60		70		80		90		100	
Overall	203.55	(16)	286.36	(24)	252.01	(22)	354.34	(31)	406.00	(36)
$\alpha = 10$	136.79	(6)	262.85	(12)	215.95	(10)	314.93	(13)	413.13	(19)
$\alpha = 100$	270.32	(10)	309.87	(12)	288.08	(12)	393.75	(18)	398.88	(17)
$\tau = 0.5$	75.26	(1)	160.94	(2)	235.84	(5)	249.43	(5)	403.10	(9)
$\tau = 1.0$	224.56	(3)	357.10	(8)	500.54	(11)	572.52	(12)	452.04	(9)
$\tau = 1.5$	341.17	(8)	232.25	(5)	222.02	(5)	463.09	(11)	304.83	(7)
$\tau = 2.0$	173.23	(4)	395.15	(9)	49.66	(1)	132.27	(1)	464.03	(11)
$\eta = 1$	201.83	(6)	406.14	(12)	254.61	(7)	363.00	(11)	391.98	(12)
$\eta = 3$	190.76	(5)	347.71	(10)	230.26	(7)	301.25	(9)	552.46	(16)
$\eta = 5$	218.07	(5)	105.23	(2)	271.17	(8)	398.77	(11)	273.57	(8)

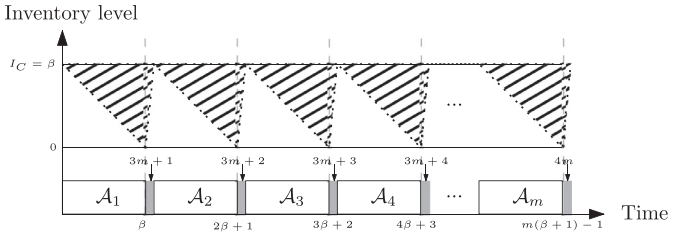

Fig. A1. The schedule associated with the proof of Theorem 1.

- For each job $j \in J^-$, we set $p_j = \alpha_j$, $\delta_j = -\alpha_j$ and $r_j = 0$.
- For each job $j \in J^+$, we set $p_j = 1$, $\delta_j = \beta$ and $r_j = (j - 3m)(\beta + 1) - 1$.

Any feasible solution to this instance of $1|inv, r_j|C_{\max}$ will have the structure of the solution in Fig. A.1, where $\mathcal{A}_s \subset J^-$ for every $s \in \{1, \ldots, m\}$ and $\mathcal{A}_{s_1} \cap \mathcal{A}_{s_2} = \emptyset$ for every pair $(s_1, s_2) \in \{1, \ldots, m\}^2$ with $s_1 \neq s_2$. Thus, if there exists a feasible solution to the instance of $1|inv, r_j|C_{\max}$ then the answer to the associated instance of 3-PARTITION is YES, because $\{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ will be a valid partition, with $\sum_{i \in \mathcal{A}_s} \alpha_i = \beta$ for all $s \in \{1, \ldots, m\}$ (as is shown in Fig. A.1). Conversely, if there is no feasible solution to the $1|inv, r_j|C_{\max}$ instance then the answer to the 3-PARTITION instance is NO. To see this, it suffices to argue that if such a valid partition $\{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ with $\sum_{i \in \mathcal{A}_s} \alpha_i = \beta$ for all $s \in \{1, \ldots, m\}$ existed, then we could have constructed a feasible solution to the associated instance of $1|inv, r_j|C_{\max}$ using that partition. We conclude that $1|inv, r_j|C_{\max}$ is NP-hard in the strong sense.

The problem will remain strongly NP-hard even if the inventory capacity is unlimited. The proof is very similar, but with $I_C = +\infty$. Let $m(\beta + 1)$ be a threshold on the objective function. If the optimal objective value for the foregoing instance is $m(\beta + 1)$ then the associated $\{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ is a valid partition. In this case, since the inventory level cannot go below zero, it also never passes β in any optimal solution. If the optimal objective value for this instance is larger than $m(\beta + 1)$, on the other hand, then there is no valid partition to the 3-PARTITION instance. □

Proof of Theorem 2. The NP-hardness of $1|inv|C_{\max}$ is proved similarly to Theorem 1. Given an instance of 3-PARTITION, we can construct an instance of $1|inv|C_{\max}$ in the same way as before, but without release dates. Due to the choice of I_C, any feasible solution will still resemble the schedule in Fig. A.1, and thus determining the feasibility of an instance of $1|inv|C_{\max}$ is NP-complete in the strong sense. □

Proof of Theorem 3. Clearly, every feasible sequence (if any exists) leads to a semi-active schedule with the same makespan $\Sigma_{j \in J}\{p_j\}$ and is optimal. Therefore, to prove the theorem, it suffices to describe an O(n)-time algorithm that finds a feasible solution. If $I_0 + \sum_{j \in J} \delta_j < 0$ or $I_0 + \sum_{j \in J} \delta_j > I_C$ then there is no feasible solution. Otherwise, we construct a sequence with the following procedure.

Case (a): we construct two sets L^+ and L^-. Set L^+ initially contains all the jobs in J^+ and $L^- := J^-$. We start with an empty sequence of jobs, and we stepwise append jobs from the start of the sequence, as follows. For the first position in the sequence, if $I_0 \geq \delta^-_{\max}$ then we (randomly) select a job $j_1 \in L^-$, place it in the first position and remove it from L^-. Otherwise, $I_0 + \delta^+_{\max} \leq I_C$ and j_1 is selected (and removed) from L^+. In both cases, the bounds in the inventory position are respected. In the following steps, we assign a job to the kth position in the sequence ($k = 2, \ldots, n$). If $I_0 + \sum_{\kappa=1}^{k-1} \delta_{j_\kappa} \geq \delta^-_{\max}$ then we choose $j_k \in L^-$, otherwise $j_k \in L^+$. Fig. A.2a provides a schematic illustration of a sequence produced by this procedure, which always outputs a feasible solution. If in any of the above steps $|L^+| = 0$ or $|L^-| = 0$ then we are obliged to choose a job from the other set, but this will not influence the feasibility of the resulting sequence.

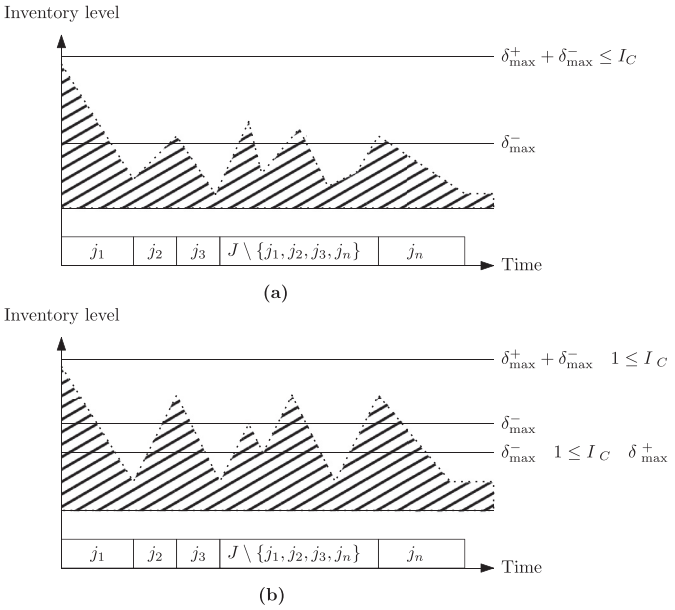

Fig. A2. Illustrations for the proof of Theorem 3. (a) Illustration of case (a) and (b) Illustration of case (b).

The construction of the two sets requires O(n)-time and the construction of the sequence also takes O(n)-time. Therefore, the procedure can be done in O(n)-time.

Case (b): If $I_C \geq \delta^+_{\max} + \delta^-_{\max} - 1$ and all inventory modifications as well as I_0 and I_C are integers, then the same procedure as in Case (a) can be used to generate a feasible sequence, because at any step k, $I_0 + \sum_{\kappa=1}^{k-1} \delta_{j_\kappa} < \delta^-_{\max}$ implies $I_0 + \sum_{\kappa=1}^{k-1} \delta_{j_\kappa} \leq I_C - \delta^+_{\max}$ and then we can start any job j_k from L^+. Even if I_C and I_0 are not integer but all δ_j are integer, then one can modify I_C and I_0 (in polynomial time) to an integer value in an equivalent instance. □

Proof of Theorem 4. Consider a feasible sequence $\boldsymbol{\sigma}$ that satisfies Conditions (a), (b) and (c). Following Conditions (b) and (c), we have:

$$C_{\max}(\boldsymbol{\sigma}) = C_{\sigma_n}(\boldsymbol{\sigma}) = r_{\sigma_k} + \sum_{s=k}^{n} p_{\sigma_s}.$$

Conversely, Condition (a) ensures that none of the jobs $\sigma_k, \ldots, \sigma_n$ can be started earlier than r_{σ_k}, which implies that

$$C_{\max}(\boldsymbol{\sigma}') \geq r_{\sigma_k} + \sum_{s=k}^{n} p_{\sigma_s} = C_{\max}(\boldsymbol{\sigma})$$

for any feasible sequence $\boldsymbol{\sigma}'$. We therefore conclude that $\boldsymbol{\sigma}$ is optimal. □

References

Baptiste, P., & Sadykov, R. (2009). On scheduling a single machine to minimize a piecewise linear objective function: A compact MIP formulation. *Naval Research Logistics, 56*(6), 487–502. doi:10.1002/nav.20352.

Bartels, J.-H., & Zimmermann, J. (2015). *Scheduling tests in automotive r&d projects using a genetic algorithm*. In C. Schwindt, & J. Zimmermann (Eds.) (Vol. 2, pp. 1157–1185)). Cham: Springer International Publishing.

Boysen, N., Bock, S., & Fliedner, M. (2013). Scheduling of inventory releasing jobs to satisfy time-varying demand: An analysis of complexity. *Journal of Scheduling, 16*(2), 185–198. doi:10.1007/s10951-012-0266-0.

Briskorn, D., Choi, B.-C., Lee, K., Leung, J., & Pinedo, M. (2010). Complexity of single machine scheduling subject to nonnegative inventory constraints.. *European Journal of Operational Research, 207*(2), 605–619.

Briskorn, D., Jaehn, F., & Pesch, E. (2013). Exact algorithms for inventory constrained scheduling on a single machine. *Journal of Scheduling, 16*(1), 105–115.

Briskorn, D., & Leung, J. Y.-T. (2013). Minimizing maximum lateness of jobs in inventory constrained scheduling. *Journal of the Operational Research Society, 64*(12), 1851–1864. doi:10.1057/jors.2012.155.

Briskorn, D., & Pesch, E. (2013). Variable very large neighbourhood algorithms for truck sequencing at transshipment terminals. *International Journal of Production Research, 51*, 7140–7155.

Carlier, J., & Rinnooy Kan, A. (1982). Scheduling subject to nonrenewable-resource constraints. *Operations Research Letters, 1*(2), 52–55. doi:10.1016/0167-6377(82)90045-1.

Davari, M., Demeulemeester, E., Leus, R., & Talla Nobibon, F. (2016). Exact algorithms for single-machine scheduling with time windows and precedence constraints. *Journal of Scheduling, 19*, 309–334. doi:10.1007/s10951-015-0428-y.

Drótos, M., & Kis, T. (2013). Scheduling of inventory releasing jobs to minimize a regular objective function of delivery times. *Journal of Scheduling, 16*(3), 337–346. doi:10.1007/s10951-012-0297-6.

Gafarov, E. R., Lazarev, A. A., & Werner, F. (2011). Single machine scheduling problems with financial resource constraints: Some complexity results and properties. *Mathematical Social Sciences, 62*(1), 7–13. doi:10.1016/j.mathsocsci.2011.04.004.

Ghorbanzadeh, M., Ranjbar, M., & Jamili, N. (2019). Transshipment scheduling at a single station with release date and inventory constraints. *Journal of Industrial and Production Engineering, 36*(5), 301–312. doi:10.1080/21681015.2019.1647300.

Györgyi, P. (2017). A PTAS for a resource scheduling problem with arbitrary number of parallel machines. *Operations Research Letters, 45*(6), 604–609. doi:10.1016/j.orl.2017.09.007.

Györgyi, P., & Kis, T. (2014). Approximation schemes for single machine scheduling with non-renewable resource constraints. *Journal of Scheduling, 17*(2), 135–144. doi:10.1007/s10951-013-0346-9.

Györgyi, P., & Kis, T. (2015). Approximability of scheduling problems with resource consuming jobs. *Annals of Operations Research, 235*(1), 319–336. doi:10.1007/s10479-015-1993-3.

Jouglet, A., Baptiste, P., & Carlier, J. (2004). *Handbook of scheduling: Algorithms, models and performance analysis*. CRC Press, Boca Raton, FL, USA.

Kis, T. (2015). Approximability of total weighted completion time with resource consuming jobs. *Operations Research Letters, 43*(6), 595–598. doi:10.1016/j.orl.2015.09.004.

Lawler, E. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Management Science, 19*(5), 544–546.

Neumann, K., & Schwindt, C. (2003). Project scheduling with inventory constraints. *Mathematical Methods of Operations Research, 56*(3), 513–533. doi:10.1007/s001860200251.

Neumann, K., Schwindt, C., & Trautmann, N. (2005). Scheduling of continuous and discontinuous material flows with intermediate storage restrictions. *European Journal of Operational Research, 165*(2), 495–509. doi:10.1016/j.ejor.2004.04.018.

Pan, Y., & Shi, L. (2005). Dual constrained single machine sequencing to minimize total weighted completion time. *IEEE Transactions on Automation Science and Engineering, 2*(4), 344–357.

Schwindt, C., & Trautmann, N. (2000). Batch scheduling in process industries: An application of resource-constrained project scheduling. *OR-Spectrum, 22*(4), 501–524. doi:10.1007/s002910000042.

Slowiński, R. (1984). Preemptive scheduling of independent jobs on parallel machines subject to financial constraints. *European Journal of Operational Research, 15*(3), 366–373. doi:10.1016/0377-2217(84)90105-X.

Tanaka, S., & Fujikuma, S. (2012). A dynamic-programming-based exact algorithm for general single-machine scheduling with machine idle time. *Journal of Scheduling, 15*, 347–361.

Toker, A., Kondakci, S., & Erkip, N. (1991). Scheduling under a non-renewable resource constraint. *Journal of the Operational Research Society, 42*(9), 811–814. doi:10.1057/jors.1991.152.