Constraints journal manuscript No.
(will be inserted by the editor)

The Item Dependent StockingCost Constraint

Vinasetan Ratheil Houndji · Pierre Schaus · Laurence Wolsey

Received: date / Accepted: date

Abstract We recently introduced a global `StockingCost` constraint to compute the total number of periods between the production periods and the due dates in a multi-order capacitated lot-sizing problem. Here we consider a more general case in which each order can have a different per period stocking cost and the goal is to minimise the total stocking cost. In addition the production capacity, limiting the number of orders produced in a given period, is allowed to vary over time. We propose an efficient filtering algorithm in $O(n \log n)$ where n is the number of orders to produce. On a variant of the capacitated lot-sizing problem, we demonstrate experimentally that our new filtering algorithm scales well and is competitive wrt the `StockingCost` constraint when the stocking cost is the same for all orders.

Keywords StockingCost Constraint · Production Planning · Lot-Sizing · Scheduling · Constraint Programming · Global Constraint · Optimization contraint · Cost-based filtering

V. R. Houndji
Institute of Information and Communication Technologies, Electronics and Applied Mathematics (ICTEAM), Université catholique de Louvain (UCL), Louvain la Neuve, Belgium.
Institut de Formation et de Recherche en Informatique (IFRI), Ecole Doctorale des Sciences de l'Ingénieur (ED-SDI), Université d'Abomey-Calavi, Abomey-Calavi (UAC), Benin.
E-mail: vinasetan.houndji@uclouvain.be, ratheil.houndji@uac.bj

P. Schaus
Institute of Information and Communication Technologies, Electronics and Applied Mathematics (ICTEAM), Université catholique de Louvain (UCL), Louvain la Neuve, Belgium.
E-mail: pierre.schaus@uclouvain.be

L. Wolsey
Institute for Multidisciplinary Research in Quantitative Modelling and Analysis (IMMAQ), Université catholique de Louvain (UCL), Louvain la Neuve, Belgium.
E-mail: laurence.wolsey@uclouvain.be

1 Introduction

In production planning, one of the most important and difficult tasks is the determination of the size of the production lots[9,1]. Lot-Sizing (LS) problems have been well studied since their introduction by [?]. There are many variants of LS problems depending on their characteristics: single or multiple item, capacitated or uncapacitated, single level or multiple levels, set up costs, changeover costs, storage/stocking costs, etc. We refer to [12,2,8,17,9] for some reviews on this family of problems. The Capacitated Lot-Sizing Problem (CLSP) treated here is a production planning problem which consists of determining a minimal cost production schedule for multiple items (production costs, setup costs, changeover costs, stocking costs, etc.) over a discrete and finite planning horizon, such that machine capacity restrictions are respected and all demands are satisfied. Exact solution approaches are based on mixed integer programming formulations to which one adds strong valid inequalities or extended formulations involving additional variables (see for example [?,?,?,?,?]). We refer to [9] for operations research approaches to the CLSP proposed in the literature.

Here our goal is to show that Constraint Programming (CP) has a role to play in solving some variants of the CLSP. It is well known that Constraint Programming (CP) can be effective in solving some hard combinatorial problems. For example, CP is one of the best approaches to tackle scheduling problems. Surprisingly lot-sizing has only very recently become a field of research in Constraint Programming. In [6] German et al. proposed the `LotSizing` constraint for a single item problem in which the different costs depend on the period of the production of the order. Our work [7] introduced the `StockingCost` constraint to compute the total number of periods between the production periods and the due dates in a capacitated lot-sizing problem. This constraint is well suited to compute the stocking cost when the per period stocking cost is the same for every order. Unfortunately, in many problems the stocking cost is order dependent since some order types (items) are more or less expensive to hold in stock. In this case, each order has the per period stocking cost of the corresponding item. This work generalizes the `StockingCost` constraint allowing a per period stocking cost that is potentially different for each order. The new constraint is denoted `IDStockingCost` (ID stands for Item Dependent) for the rest of the paper . Cost-based filtering algorithms for this kind of optimization constraint are often based on the following pieces of information [4]:

- a relaxed or less constrained problem;
- the value of an optimal solution to this relaxed problem. This value is a lower bound on the original problem objective function. It is used to 1) check the consistency of the constraint and 2) filter the objective variable;
- an optimistic evaluation of the cost increase if a value is assigned to a decision variable X_i. This cost increase is often called marginal cost or reduced cost or regret and is used to filter decision variables.

We use this approach to derive a filtering algorithm for the `IDStockingCost` constraint. We relax the problem by only considering the due dates of orders and the capacity restrictions. This relaxation allows us to have an $O(n \log n)$ algorithm to achieve a filtering of the `IDStockingCost` constraint based on a lower bound on the marginal costs. The filtering algorithm introduced does not achieve bound consistency but the experimental results show that it scales well.

The remainder of this paper is organized as follows: Section 2 presents the item-dependent problem, gives a formal definition of the item dependent stockingCost constraint `IDStockingCost`, and shows how one can achieve pruning with the state-of-the-art constraints; Section 3 describes a filtering algorithm for the cost variable H based on a relaxed problem; Section 4 shows how to filter the date variables X based on an optimal solution of the relaxed problem and a lower bound on the marginal costs; Section 5 presents some computational experiments on an NP-Hard variant of the CLSP involving changeover costs when switching from production of one order to another; and Section 6 concludes.

2 The Item Dependent StockingCost Problem and Constraint

One has a time horizon T, a set $i = 1, \ldots, n$ of orders each with a due date $d_i \in [1, \ldots, T]$ and a per period stocking cost h_i. There is a machine which has c_t units of production capacity in period t. Producing an order in period t consumes one unit of machine capacity. The problem is to produce each order by its due date at latest without exceeding the machine capacity and to minimize the sum of the stocking costs of the orders. Below we formally define the Item Dependent StockingCost Constraint and shows some decompositions of this constraint.

The `IDStockingCost`[1] constraint takes the following form:

$$\texttt{IDStockingCost}([X_1, \ldots, X_n], [d_1, \ldots, d_n], [h_1, \ldots, h_n], H, [c_1, \ldots, c_T])$$

in which:

- n is the total number of orders to produce;
- T is the total number of periods over the planning horizon $[1, \ldots, T]$;
- the variable X_i is the date of production of order i on the machine, $\forall i \in [1, \ldots, n]$. Let $X_i^{\min}$ (resp. $X_i^{\max}$) denote the minimal (resp. maximal) value in the finite domain D_i of variable X_i;
- the integer d_i is the due-date for order i, $\forall i \in [1, \ldots, n]$;
- the integer $h_i \geq 0$ is the stocking cost for order i, $\forall i \in [1, \ldots, n]$;
- the integer $c_t \geq 0$ is the maximum number of orders the machine can produce during the period t (production capacity for t), $\forall t \in [1, \ldots, T]$;
- the variable H is the maximum value of the total stocking cost. Let $H^{\min}$ (resp. $H^{\max}$) denote the minimal (resp. maximal) value in the finite domain of variable H.

The `IDStockingCost` constraint holds when:

$$X_i \leq d_i, \forall i \tag{1}$$

$$\sum_i (X_i = t) \leq c_t, \forall t \tag{2}$$

$$\sum_i (d_i - X_i) \cdot h_i \leq H \tag{3}$$

[1] In typical applications of this constraint, assuming that c_t is $O(1)$, the number of orders n is on the order of the horizon T: $n \sim O(T)$.

This decomposition imposes that (1) each order i is produced before or on its due date, (2) the capacity of the machine is respected at any period t and (3) H is the maximum value of the total stocking cost.

As proposed in [7], the T constraints in equation (2) can be replaced by a global cardinality constraint `gcc` [14,13]. Note that for $c_t = 1, \forall t \in [1, \ldots, T]$, the `gcc` constraint can be replaced by an `allDifferent` [10] constraint. The bound consistency of `gcc` constraint can be obtained in $O(n)$ plus the time for sorting the n variables [13]. An even stronger model is obtained by replacing the constraints in (2) and (3) by an arc-consistent `cost-gcc` [15] constraint. Similarly, for the unit capacity case one can use the `minimumAssignment` [4,3] constraint with filtering based on reduced costs. The filtering algorithms for the `minimumAssignment` and `cost-gcc` execute in $O(T^3) \approx O(n^3)$. This paper presents a fast filtering algorithm for `IDStockingCost` running in $O(n \log n)$.

In the rest of the paper, without loss of generality, we assume that:

1. $X_i \leq d_i, \forall i$;
2. the `gcc` constraint is bound consistent[2]. Here the `gcc` constraint is bound consistent means that for each $X_i, \forall v_i \in \{X_i^{\min}, X_i^{\max}\}$ and $\forall X_j \neq X_i : \exists v_j \in [X_j^{\min}, \ldots, X_j^{\max}]$ such that $\sum_k (v_k = t) \leq c_t, \forall t$. For example, consider three orders such that $X_1 \in [3,4]$, $X_2 \in [3,4]$, $X_3 \in [1,4]$ and $c_1 = 0, c_2 = c_3 = c_4 = 1$. We can see that X_3 can neither take the value 4 nor 3 because the interval $[3,4]$ must be reserved for X_1 and X_2. On the other hand, X_3 cannot take value 1 because $c_1 = 0$. Thus `gcc` is bound consistent if $X_1 \in [3,4]$, $X_2 \in [3,4]$ and $X_3 = \{2\}$.
   The `gcc` bound consistent propagator (in $O(n)$ plus the time for sorting the n variables) is triggered before any filtering from the `IDStockingCost` constraint.

3 Filtering of the cost variable H

This section explains how to filter the lower bound on H in $O(n \log n)$. First we define the problem associated to the optimal cost of the problem (denoted $\mathcal{P}$) and a relaxed version of this problem (denoted $\mathcal{P}^r$). Problem $\mathcal{P}^r$ is used to filter the `IDStockingCost` constraint in order to have a scalable filtering algorithm. After establishing the condition for optimality of $\mathcal{P}^r$, we give an $O(n \log n)$ algorithm to compute an optimal solution of $\mathcal{P}^r$.

3.1 The problem definition

By considering the definition of the `IDStockingCost` constraint, the best lower bound H^{opt} of the global stocking cost variable H can be obtained by solving the following problem:

[2] A constraint is bound consistent if, for each minimum and maximum values, there exists a solution wrt the constraint by considering the domains of other variables without holes.

$$(\mathcal{P}) \qquad \begin{aligned} H^{opt} &= \min \sum_i (d_i - X_i) \cdot h_i \\ &\sum_i (X_i = t) \le c_t, \forall t \\ &X_i \in D_i, \forall i \end{aligned}$$

in which D_i is the domain of the variable X_i that is the set of values $\in [1, \ldots, T]$ that X_i can take.

The problem $\mathcal{P}$ can be solved with a max-flow min-cost algorithm on the bipartite graph linking orders and periods [15]. Indeed the cost of assigning $X_i \leftarrow t$ can be computed as $(d_i - t) \cdot h_i$ if $t \in D_i$, $+\infty$ otherwise. With unit capacity, it is a min-assignment problem that can be solved in $O(T^3)$ with the Hungarian algorithm. The costs on the arcs have the particularity to evolve in a convex way (linearly) along the values, but even so, we are not aware of a faster min-assignment algorithm. Since our objective is to design a fast scalable filtering, we now introduce the relaxed problem.

The relaxation we make is to assume that X_i can take any value $\le X_i^{\max}$ without holes: $D_i = [1, \ldots, X_i^{\max}]$. Our filtering algorithm is thus based on a relaxed problem in which the orders can be produced in any period before their minimum values (but not after their maximum values). Let $\mathcal{P}^r$ denote this new relaxed problem and $(H^{opt})^r$ denote its optimal value. $(H^{opt})^r$ gives a valid lower bound on H^{opt} allowing us to possibly increase $H^{\min}$.

$$(\mathcal{P}^r) \qquad \begin{aligned} (H^{opt})^r &= \min \sum_i (d_i - X_i) \cdot h_i \\ &\sum_i (X_i = t) \le c_t, \forall t \\ &X_i \le X_i^{\max}, \forall i \end{aligned}$$

We will show below that one can compute $(H^{opt})^r$ in a greedy fashion assigning the production periods from the latest to the earliest. Clearly the orders should be produced as late as possible (i.e. as close as possible to their due-date) in order to minimize their individual stocking cost. Unfortunately, the capacity constraints usually prevent us from assigning every X_i to its maximum value $X_i^{\max}$. We now characterize an optimal solution of $\mathcal{P}^r$.

3.2 Conditions for optimality of $\mathcal{P}^r$

By considering $\mathcal{P}^r$, without loss of generality, we assume that all orders $i \in [1, \ldots, n]$ are such that $h_i > 0$. If this is not the case, assuming that the `gcc` constraint is bound consistent, one can produce $n_0 = |\{X_i : h_i = 0\}|$ orders in the first n_0 periods and then consider the other orders over the planning horizon $[n_0 + 1, \ldots, T]$.

Observe first that if in a valid solution of $\mathcal{P}^r$, there is a place available for production in period t and there is an order that can be assigned to t but is assigned to $t' < t$ then that solution is not optimal.

Definition 1 Denote by $assPeriod$ a valid assignment vector in which $assPeriod[i]$ is the value (period) taken by X_i. Considering a valid assignment wrt $\mathcal{P}^r$ and a period t, the boolean value $t.full$ indicates whether this period is used at maximal capacity or not: $t.full \equiv |\{X_i : assPeriod[i] = t\}| = c_t$.

Observation 1 *Consider a valid assignment assPeriod:* $assPeriod[i], \forall i \in [1, \ldots, n]$ *wrt* $\mathcal{P}^r$*. If this assignment is optimal, then*
(i) $\forall i \in [1, \ldots, n], \nexists t : (assPeriod[i] < t) \wedge (X_i^{\max} \geq t) \wedge (\neg t.full)$.

Proof Let assume that $assPeriod$ does not respect the criterion (i). This means that $\exists X_k \wedge \exists t : (assPeriod[k] < t) \wedge (X_k^{\max} \geq t) \wedge (\neg t.full)$. In this case, by moving X_k from $assPeriod[k]$ to t, we obtain a valid solution that is better than $assPeriod$. The improvement is: $(t - assPeriod[k]) \cdot h_k$. Thus the criterion (i) is a necessary condition for optimality of $\mathcal{P}^r$. □

Corollary 1 *Any optimal solution assPeriod uses the same set of periods:* $\{assPeriod[k] : \forall k\}$ *and this set is unique.*

This unique set can be obtained from right to left by considering orders decreasingly according to their $X_i^{\max}$, not assigning any order before its $X_i^{\max}$ and moving to a previous not completely filled period in case the current period is full.

On the other hand, if in a solution of $\mathcal{P}^r$ a valid permutation between two orders decreases the cost of that solution, then this latter is not optimal.

Observation 2 *Consider a valid assignment assPeriod:* $assPeriod[i]$, $\forall i \in [1, \ldots, n]$ *wrt* $\mathcal{P}^r$*. If this assignment is optimal, then*
(ii) $\nexists (X_{k_1}, X_{k_2})$*:* $(assPeriod[k_1] < assPeriod[k_2]) \wedge (h_{k_1} > h_{k_2}) \wedge$
$(X_{k_1}^{\max} \geq assPeriod[k_2])$.

Proof Let assume that $assPeriod$ does not respect the criterion (ii). That means $\exists (X_{k_1}, X_{k_2}) : (assPeriod[k_1] < assPeriod[k_2]) \wedge (h_{k_1} > h_{k_2}) \wedge (X_{k_1}^{\max} \geq assPeriod[k_2])$. In this case, by swapping the orders k_1 and k_2, we obtain a valid solution that is better than $assPeriod$. The improvement is : $(assPeriod[k_2] - assPeriod[k_1]) \cdot h_{k_1} - (assPeriod[k_2] - assPeriod[k_1]) \cdot h_{k_2} > 0$. Thus the criterion (ii) is a necessary optimality condition. □

The next proposition states that the previous two necessary conditions are also sufficient for testing optimality to problem $\mathcal{P}^r$.

Proposition 1 *Consider a valid assignment assPeriod:* $assPeriod[i], \forall i \in [1, \ldots, n]$ *wrt* $\mathcal{P}^r$*. This assignment is optimal iff*
(i) $\forall i \in [1, \ldots, n], \nexists t : (assPeriod[i] < t) \wedge (X_i^{\max} \geq t) \wedge (\neg t.full)$
(ii) $\nexists (X_{k_1}, X_{k_2})$*:* $(assPeriod[k_1] < assPeriod[k_2]) \wedge (h_{k_1} > h_{k_2}) \wedge$
$(X_{k_1}^{\max} \geq assPeriod[k_2])$.

Proof Without loss of generality, we assume that 1) All the orders have different stocking costs : $\forall (k_1, k_2) : h_{k_1} \neq h_{k_2}$. If this is not the case for two orders, we can increase the cost of one by an arbitrarily small value. 2) Unary capacity for all periods : $c_t = 1, \forall t$. The periods with zero capacity can simply be discarded and periods with capacities $c_t > 1$ can be replaced by c_t "artificial" unit periods. Of course the planning horizon changes. To reconstruct the solution of the initial problem, one can simply have a map that associates to each artificial period the corresponding period in the initial problem. 3) All the orders are sorted such that $assPeriod[i] > assPeriod[i+1]$.

We know that (i) and (ii) are necessary conditions for optimality. The objective is to prove that a solution that respects (i) and (ii) is unique and thus also optimal. From Corollary 1, we know that all optimal solutions use the same set of periods: $\{t_1, t_2, \ldots, t_n\}$ with $t_1 = \max_i\{X_i^{\max}\} > t_2 > \ldots > t_n$. Let $\mathcal{C}_1 = \{k : X_k^{\max} \geq t_1\}$ be the orders that could possibly be assigned to the first period t_1. To respect the property (ii), for the first period t_1, we must select the unique order $\text{argmax}_{k \in \mathcal{C}_1} h_k$. Now assume that periods $t_1 > t_2 > \ldots > t_i$ were successively assigned to orders $1, 2, \ldots, i$ and produced the unique partial solution that can be expanded to a solution for all the orders $1, \ldots, n$. We show that we have also a unique choice to expand the solution in period t_{i+1}. The order to select in period t_{i+1} is $\text{argmax}_{k \in \mathcal{C}_{i+1}}\{h_k\}$ with $\mathcal{C}_{i+1} = \{k : k > i \wedge X_k^{\max} \geq t_{i+1}\}$ is the set of orders that could possibly be assigned in period t_{i+1}. Indeed, selecting any other order would lead to a violation of property (ii). Hence the final complete solution obtained is unique. □

3.3 Filtering algorithm of the cost variable H

This section describes an algorithm to filter the cost variable H based on an optimal solution of $\mathcal{P}^r$. As mentioned above, a gcc bound consistent filtering is performed before any filtering from the `IDStockingCost` constraint. Algorithm 1 computes an optimal solution of $\mathcal{P}^r$ and filters the variable H. This algorithm considers orders sorted decreasingly according to their $X_i^{\max}$. A virtual sweep line decreases in period starting at $\max_i\{X_i^{\max}\}$. The sweep line (at position t) collects in a priority queue all the orders that can be possibly scheduled in that period (such that $t \leq X_i^{\max}$). Each time it is decreased new orders can possibly enter into a priority queue (loop $12-14$). The priorities in the queue are the stocking costs h_i of the orders. A large cost h_i means that this order has a higher priority to be scheduled as late as possible (since t is decreasing). The variable *availableCapacity* represents the current remaining capacity in period t. It is initialized to the capacity c_t (line 10) and decreased by one it each time an order is scheduled at t (line 19). An order is scheduled at lines $15-20$ by choosing the one with highest stocking cost from *ordersToSchedule*. The capacity and the cost are updated accordingly. The orders are scheduled at t until the capacity is reached (and then the current period is updated to the previous period with non null capacity) or the queue is empty (and then the algorithm jumps to the maximum value of the next order to produce). This process is repeated until all orders have been scheduled. Algorithm 1 has two invariants. Each of them is related to a condition of Proposition 1 to ensure that the solution returned by the algorithm respects the conditions for optimality of $\mathcal{P}^r$. At the end 1) $optPeriod[i], \forall i \in [1, \ldots, n]$ is the optimal schedule showing the period assigned to the order i; and 2) $optOrders[t], \forall t \in [1, \ldots, T]$ is the set of orders produced at period t - stored in a stack such that the order on the top is the last to be produced. We thus have at the end: $\sum_{i=1}^{n}(d_i - optPeriod[i]) \cdot h_i = (H^{opt})^r$ and $optOrders[t] = \{X_i : optPeriod[i] = t\}$, $\forall t \in [1, \ldots, T]$ is the set of orders produced in period t.

Algorithm 1 uses the priority queue *ordersToSchedule* with two primitives that have been used: 1) *ordersToSchedule.insert*(i) inserts the order i in the queue; and

2) $ordersToSchedule.delMax()$ returns an order with the highest cost and removes it from $ordersToSchedule$. Also Algorithm 1 uses $optOrders[t].push(j)$ to add the order j on the top of the stack $optOrders[t]$ for a given period t.

Proposition 2 Algorithm 1 *computes an optimal solution of* $\mathcal{P}^r$ *in* $O(n \log n)$.

Proof Algorithm 1 works as suggested in the proof of Proposition 1 and then Invariant (a) and Invariant (b) hold for each t from $\max_i\{X_i^{\max}\}$. Thus the solution returned by the algorithm 1) is feasible and 2) respects the properties (i) and (ii) of Proposition 1 and is therefore optimal.
Complexity: the loop at lines $12-14$ that increments the order index i from 1 to n ensures that the main loop of the algorithm is executed $O(n)$ times. On the other hand, each order is pushed and popped exactly once in the queue $ordersToSchedule$ in the main loop. Since $ordersToSchedule$ is a priority queue, the global complexity is $O(n \log n)$. □

Algorithm 1: Filtering of lower bound with $(H^{opt})^r$

Input: $X = [X_1, \ldots, X_n]$ such that $X_i \leq d_i$ and sorted ($X_i^{\max} \geq X_{i+1}^{\max}$)

1 gccBC.$propagate()$ // trigger the gcc bound consistent propaagator
2 $(H^{opt})^r \leftarrow 0$ // total minimum stocking cost for $\mathcal{P}^r$
3 $optPeriod \leftarrow map()$ // $optPeriod[i]$ is the period assigned to order i
  // orders placed in t sorted top-down in non increasing h_i
4 $\forall t : optOrders[t] \leftarrow stack()$
5 $ordersToSchedule \leftarrow priorityQueue()$ // priority=h_i
6 $i \leftarrow 1$
7 **while** $i \leq n$ **do**
8     $t \leftarrow X_i^{\max}$ // current period
9     $availableCapacity \leftarrow c_t$ // available capa at t
10   **repeat**
11   **while** $i \leq n \wedge X_i^{\max} = t$ **do**
12             $ordersToSchedule.insert(i)$
13             $i \leftarrow i + 1$
14   **if** $availableCapacity > 0$ **then**
15             $j \leftarrow ordersToSchedule.delMax()$ // order with highest cost
16             $optPeriod[j] \leftarrow t$
17             $optOrders[t].push(j)$
18             $availableCapacity \leftarrow availableCapacity - 1$
19             $(H^{opt})^r \leftarrow (H^{opt})^r + (d_j - t) * h_j$
20   **else**
21             $t \leftarrow previousPeriodWithNonNullCapa(t)$ // t is equal to the previous period (wrt t) with non null capacity
22             $availableCapacity \leftarrow c_t$
         // **Invariant (a):** $\forall X_i$ **such that** $optPeriod[i]$ **is defined: condition** (i) **of Proposition 1 holds**
         // **Invariant (b):** $\forall X_{k_1}, X_{k_2}$ **such that** $optPeriod[k_1]$ **and** $optPeriod[k_2]$ **are defined: condition** (ii) **of Proposition 1 holds**
23   **until** $ordersToSchedule.size > 0$
24     $fullSet.push(t)$
25 $H^{\min} \leftarrow \max(H^{\min}, (H^{opt})^r)$

The next example shows the execution of Algorithm 1 on a small instance of $\mathcal{P}^r$.

Example 1 Consider the following instance:
`IDStockingCost`$([X_1 \in [1,\ldots,4], X_2 \in [1,\ldots,5], X_3 \in [1,\ldots,4], X_4 \in [1,\ldots,5], X_5 \in [1,\ldots,8], X_6 \in [1,\ldots,8]], [d_1 = 4, d_2 = 5, d_3 = 4, d_4 = 5, d_5 = 8, d_6 = 8], [h_1 = 3, h_2 = 10, h_3 = 4, h_4 = 2, h_5 = 2, h_6 = 4], H \in [0,\ldots,34], c_1 = c_2 = c_4 = c_5 = c_6 = c_7 = c_8 = 1, c_3 = 0)$.
The main steps of the execution of Algorithm 1 are:

- $t = 8$, $ordersToSchedule = \{5, 6\}$ and $X_6 \leftarrow 8$. $(H^{opt})^r = 0$.
- $t = 7$, $ordersToSchedule = \{5\}$ and $X_5 \leftarrow 7$. $(H^{opt})^r = h_5 = 2$.
- $t = 5$, $ordersToSchedule = \{4, 2\}$ and $X_2 \leftarrow 5$. $(H^{opt})^r = 2$.
- $t = 4$, $ordersToSchedule = \{4, 1, 3\}$ and $X_3 \leftarrow 4$. $(H^{opt})^r = 2$.
- $t = 3$, $ordersToSchedule = \{4, 1\}$ $(c_3 = 0)$.
- $t = 2$, $ordersToSchedule = \{4, 1\}$ and $X_1 \leftarrow 2$. $(H^{opt})^r = 2 + 2 \cdot h_1 = 8$.
- $t = 1$, $ordersToSchedule = \{4\}$ and $X_4 \leftarrow 1$. $(H^{opt})^r = 8 + 4 \cdot h_4 = 16$.

Then $H \in [16,\ldots,34]$. Fig. 1 shows the optimal period assignments for $\mathcal{P}^r$. Period 6 (filled with light gray color) is an idle period in which there is no production while the capacity of production is not null. The maximum capacity of the machine is 1 (represented by a line) for all periods except period 3.

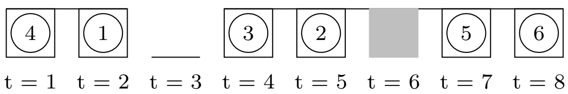

Fig. 1 An optimal assignment for $\mathcal{P}^r$

4 Pruning the decision variables X_i

The filtering of a decision variable X_i relies on an efficient computation of the marginal costs, that is the value of the optimal solution of problem $\mathcal{P}$ (resp. $\mathcal{P}^r$) if X_i is forced to take a given value. We propose an efficient algorithm to compute a lower bound on the marginal costs for values $v < optPeriod[i]$ that allows to prune $X_i^{\min}$ based on the linear evolution of these. Unfortunately we are not able to efficiently compute these costs for $v > optPeriod[i]$ because the monotonicity[3] property does not hold in this case.

To compute the marginal costs, two important notions are 1) a period that uses all its capacity in an optimal solution (called a full period) and 2) an ordered set of full periods with non null capacity in an optimal solution (called a full set). The next section formally defines a full period and a full set and gives a complete algorithm to obtain all full sets while computing the optimal cost of $\mathcal{P}^r$. Then a lower bound on the marginal costs is introduced before using it for the filtering of $X_i^{\min}$. Finally we illustrate with an example the non monotonic evolution of the marginal costs for $v > optPeriod[X_i]$ making it difficult to filter $X_i^{\max}$ efficiently.

[3] The monotonicity ensures that if we prune the upper bound of a variable to a given value, all other values greater than this value in the domain of the variable are inconsistent.

4.1 Full periods and full sets

In an optimal solution of $\mathcal{P}^r$, a period t is full ($t.full$) iff its capacity is reached: $t.full \equiv |optOrders[t]| = c_t$. Actually, an optimal solution of $\mathcal{P}^r$ is a sequence of full periods (obtained by scheduling orders as late as possible) separated by some non full periods. Let us formally define these sequences of production periods. We call them full sets. These are used to filter the decision variables in the following sections.

Definition 2 For a period t with $c_t > 0$, $minfull[t]$ is the largest period $\leq t$ such that all orders $k : X_k^{\max} \geq minfull[t]$ have $optPeriod[X_k] \geq minfull[t]$.

Definition 3 For a period t with $c_t > 0$, $maxfull[t]$ is the smallest period $\geq t$ such that all orders $k : X_k^{\max} > maxfull[t]$ have $optPeriod[X_k] > maxfull[t]$.

For the instance in Example 1, $minfull[5] = minfull[4] = minfull[3] = minfull[2] = minfull[1] = 1$ and $maxfull[5] = maxfull[4] = maxfull[3] = maxfull[2] = maxfull[1] = 5$.

Definition 4 An ordered set of periods $fs = \{M, \ldots, m\}$ (with $M > \ldots > m$) is a full set iff:

$$(\forall t \in fs \setminus \{m\} : c_t > 0 \wedge t.full) \wedge (\forall t \in fs, maxfull[t] = M \wedge minfull[t] = m).$$

We consider that $minfull[fs] = m$ and $maxfull[fs] = M$.

For the instance in Example 1, there are two full sets: $\{8, 7\}$ and $\{5, 4, 2, 1\}$.

We show that all full sets of a given optimal solution can be obtained while computing this solution (by using Algorithm 1). Algorithm 2 is a complete algorithm that computes an optimal solution of $\mathcal{P}^r$ and all full sets. The new four invariants ((a), (b), (e), and (f)) that appear in this algorithm ensure that all full sets are computed correctly. The invariants (a), (b), (e), and (f) respectively identify maxfull periods, full periods, minfull periods and full sets.

Proposition 3 Algorithm 2 *computes fullSetsStack: a stack of all full sets of an optimal solution of* $\mathcal{P}^r$.

Proof Invariant:

(a) and (e) - invariants for the maxfull and minfull periods.

Note that since `gcc` is bound consistent, $\forall X_i : c_{X_i^{\max}} > 0$. Consider the first iterations of the algorithm. At the beginning, $t_{\max} = \max_i\{X_i^{\max}\}$ is a maxfull period (by definition). Exiting the loop 11−25 means that all orders in $\{k : t_{\max} \geq X_k^{\max} \geq t\}$ (t is the current period) are already produced and the current period t is the closest period to $t_{\max}$ such that all orders in $\{k : t_{\max} \geq X_k^{\max} \geq t\}$ are produced: the current period t is then the minfull of all orders in $\{k : t_{\max} \geq X_k^{\max} \geq t\}$. Thus Invariant (a) and Invariant (e) hold for the first group of orders. The algorithm repeats the process - when it starts at line 9 with another group of orders not yet produced - until all orders are produced. We know that: $\forall i : c_{X_i^{\max}} > 0$. Then, for each group of orders i.e. each time the algorithm comes back to line 9 (resp. line 26), the current t is a maxfull (resp. minfull).

(b) At line 22, t is a full period with $c_t > 0$.
At line 22, for the current period t: $availableCapacity = 0$ and at least one order is produced before the period t. Thus $t.full$ and $c_t > 0$.

(f) $fullSet$ is a full set
This invariant holds because the invariants (a), (b) and (e) hold.

The algorithm starts from $\max_i\{X_i^{\max}\}$ and Invariant (f) holds until the last order is produced. Therefore the proposition is true. ◻

Algorithm 2: Filtering of lower bound with $(H^{opt})^r$

Input: $X = [X_1, \ldots, X_n]$ such that $X_i \leq d_i$ and sorted ($X_i^{\max} \geq X_{i+1}^{\max}$)

1 gccBC.$propagate()$ `// trigger the gcc bound consistent propaagator`
2 $(H^{opt})^r \leftarrow 0$ `// total minimum stocking cost for` $\mathcal{P}^r$
3 $optPeriod \leftarrow map()$ `//` $optPeriod[i]$ `is the period assigned to order` i
`// orders placed in` t `sorted top-down in non increasing` h_i
4 $\forall t : optOrders[t] \leftarrow stack()$
5 $ordersToSchedule \leftarrow priorityQueue()$ `// priority=`h_i
6 $fullSetsStack \leftarrow stack()$ `// stack of full sets`
7 $i \leftarrow 1$
8 **while** $i \leq n$ **do**
9     $fullSet \leftarrow stack()$
10     $t \leftarrow X_i^{\max}$ `// current period`
    `//` **Invariant (a):** t **is a maxfull period**
11     $availableCapacity \leftarrow c_t$ `// available capa at` t
12   **repeat**
13   **while** $i \leq n \wedge X_i^{\max} = t$ **do**
14             $ordersToSchedule.insert(i)$
15             $i \leftarrow i + 1$
16   **if** $availableCapacity > 0$ **then**
17             $j \leftarrow ordersToSchedule.delMax()$ `// order with highest cost`
18             $optPeriod[j] \leftarrow t$
19             $optOrders[t].push(j)$
20             $availableCapacity \leftarrow availableCapacity - 1$
21             $(H^{opt})^r \leftarrow (H^{opt})^r + (d_j - t) * h_j$
22   **else**
            `//` **Invariant (b):** t **is a full period with** $c_t > 0$
23             $fullSet.push(t)$
24             $t \leftarrow previousPeriodWithNonNullCapa(t)$
25             $availableCapacity \leftarrow c_t$
        `//` **Invariant (c):** $\forall X_i$ **such that** $optPeriod[i]$ **is defined: condition** (i) **of Proposition 1 holds**
        `//` **Invariant (d):** $\forall X_{k_1}, X_{k_2}$ **such that** $optPeriod[k_1]$ **and** $optPeriod[k_2]$ **are defined: condition** (ii) **of Proposition 1 holds**
26   **until** $ordersToSchedule.size > 0$
    `//` **Invariant (e):** t **is a minfull period**
27     $fullSet.push(t)$
    `//` **Invariant (f):** $fullSet$ **is a full set**
28     $fullSetsStack.push(fullSet)$
29 $H^{\min} \leftarrow \max(H^{\min}, (H^{opt})^r)$

4.2 A lower bound on the marginal cost

Given the value of an optimal solution of $\mathcal{P}$ (resp. $\mathcal{P}^r$), the marginal costs $m_{X_i \leftarrow v}$ (resp. $(m_{X_i \leftarrow v})^r$) is the cost increase when the variable X_i is forced to take the value $v \in D_i$. Let $H^{opt}_{X_i \leftarrow v}$ (resp. $(H^{opt}_{X_i \leftarrow v})^r$) denote the optimal cost of $\mathcal{P}$ (resp. $\mathcal{P}^r$) in a situation in which X_i is forced to take the value v in an optimal solution of $\mathcal{P}$ (resp. $\mathcal{P}^r$): that is equivalent to adding the constraint $X_i = v$ to $\mathcal{P}$ (resp. $\mathcal{P}^r$). We have $m_{X_i \leftarrow v} = H^{opt}_{X_i \leftarrow v} - H^{opt}$ and $(m_{X_i \leftarrow v})^r = (H^{opt}_{X_i \leftarrow v})^r - (H^{opt})^r$.

Note that if $H^{opt}_{X_i \leftarrow v} = H^{opt} + m_{X_i \leftarrow v} > H^{\max}$, then the resulting problem is inconsistent and v can be safely removed from the domain of X_i. Since $\mathcal{P}^r$ is a relaxed problem of $\mathcal{P}$, if $(H^{opt}_{X_i \leftarrow v})^r = (H^{opt})^r + (m_{X_i \leftarrow v})^r > H^{\max}$, then v can be removed from the domain of X_i. To filter the decision variables X_i, the idea is to find a valid lower bound for $(H^{opt}_{X_i \leftarrow v})^r$ by performing some sensitivity analysis of the optimal solution of $\mathcal{P}^r$ returned by Algorithm 1.

If X_i is forced to take a value v with $v < optPeriod[i]$, it increases $(H^{opt})^r$ by at least $(optPeriod[i] - v) \cdot h_i$ but an additional production slot in $optPeriod[i]$ becomes free in the associated optimal solution. Consequently the production of some orders can possibly be delayed and $(H^{opt})^r$ decreased. Formally,

Definition 5 Let $newoptPeriod[j], \forall j \in [1, \ldots, n] \setminus \{i\}$ denote the new optimal assignment of periods when the order i is removed from its position $optPeriod[i]$.

$gainCost[t]_i$ is the maximum cost decrease when order i scheduled in $t = optPeriod[i]$ is removed:

$$gainCost[t]_i = \sum_{j \in [1,\ldots,n] \setminus \{i\}} (newoptPeriod[j] - optPeriod[j]) \cdot h_j \geq 0$$

Of course, $newoptPeriod[j], \forall j \in [1, \ldots, n] \setminus \{i\}$ must respect the two conditions for optimality of Proposition 1. It is worth noting that, given an order k and its position in an optimal solution $t_k = optPeriod[k]$, $gainCost[t_k]_k$ can be strictly greater than 0 only if t_k is a full period with $c_{t_k} > 0$. Otherwise $gainCost[t_k]_k = 0$. Actually, a period t is not full means that there is at least one free place in t for production. Since these places are not used in the initial optimal assignment then they will not be used if another place in t is freed (see condition (i) of Proposition 1).

Example 2 Consider the instance of Example 1 and its optimal solution represented in Fig. 1:

- period 8: if the order 6 is removed from its optimal period 8, then $newoptPeriod[5] = 8$ (Fig. 2) and $newoptPeriod[j] = optPeriod[j], \forall j \notin \{5, 6\}$. $gainCost[8]_6 = h_5 = 2$;
- period 7: $newoptPeriod[j] = optPeriod[j], \forall j \neq 5$. $gainCost[7]_5 = 0$;
- period 5: if the order 2 is removed from its optimal period 5, then $newoptPeriod[4] = 5$ (Fig. 3) and $newoptPeriod[3] = optPeriod[3]$, $newoptPeriod[1] = optPeriod[1]$ because $d_1, d_3 < 5$. $gainCost[5]_2 = (5 - 1) * h_4 = 8$;
- period 4: $gainCost[4]_3 = 2 \cdot h_1 + h_4 = 8$ (Fig. 4);
- period 2: $gainCost[2]_1 = h_4$ (Fig. 5);

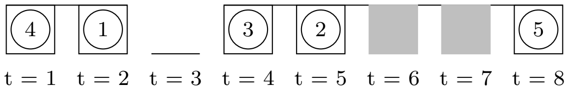

Fig. 2 An optimal assignment for $\mathcal{P}^r$ without X_6

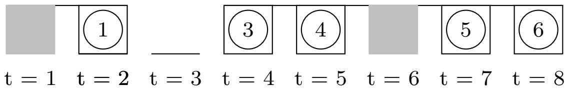

Fig. 3 An optimal assignment for $\mathcal{P}^r$ without X_2

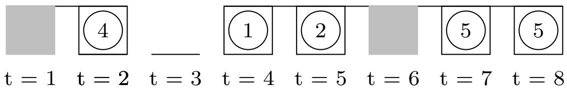

Fig. 4 An optimal assignment for $\mathcal{P}^r$ without X_3

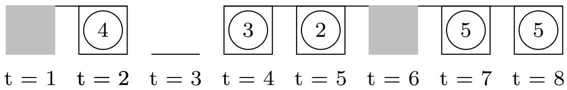

Fig. 5 An optimal assignment for $\mathcal{P}^r$ without X_1

– period 1: $gainCost[1]_4 = 0$.

Intuitively, one can say that, if a place is freed in a full period t, only orders k that have $optPeriod[k] \leq t$ in the full set of t will eventually move. More precisely, for each full period t, let $left[t]$ be the set of orders such that: $left[t] = \{X_i : optPeriod[i] \in [minfull[t], \dots, t[\}$.

Proposition 4 *If a full period t is no longer full due to the removal of order i (with $optPeriod[i] = t$) from t, then only orders in $k \in left[t]$ can have $newoptPeriod[k] \neq optPeriod[k]$. All other orders j have the same optimal period $newoptPeriod[j] = optPeriod[j]$.*

Proof We run Algorithm 2 again with order i removed:

1. all orders k with $optPeriod[k] > maxfull[t]$ will conserve their respective optimal periods because $X_i^{\max} < optPeriod[k], \forall k$ and then i is not taken into account (in the queue $ordersToSchedule$) for optimal assignment of periods $> maxfull[t]$;
2. all orders k with $t \leq optPeriod[k] \leq maxfull[t]$ will conserve their respective optimal periods. Actually, from Proposition 1, we know that for a given order k, $X_i^{\max} < optPeriod[k]$ or $h_k \geq h_i$. In these two cases, the presence/absence of X_i does not change the decision taken for orders k with their optimal periods in $[t, \dots, maxfull[t]]$;
3. all orders k with $optPeriod[k] < minfull[t]$ will conserve their respective optimal periods because X_i is not taken into account (in the queue $ordersToSchedule$) for optimal assignment of periods for all these orders. □

Observation 3 *For a period t, $gainCost[t]_i$ does not depend on the order i in $optOrders[t]$ that is removed. Thus we can simplify the notation from $gainCost[t]_i$ to $gainCost[t]$: $gainCost[t] = gainCost[t]_i, \forall X_i \in optOrders[t]$.*

Corollary 2 *For a full period t with $c_t > 0$:*

$$gainCost[t] = \sum_{j:X_j \in left[t]} (newoptPeriod[j] - optPeriod[j]) \cdot h_j$$

Observe that only orders k in $left[t]$ that have $X_k^{\max} \geq t$ can replace the order removed from t in the new optimal assignment. For each full period t, let $candidate[t]$ denote the set of orders $\in left[t]$ that can jump to the freed place in t when an order is removed from t. Formally, for a full period t, $candidate[t] = \{i \in left[t] : X_i^{\max} \geq t\}$. Let s_t denote the order that will replace the removed order in t: $s_t \in candidate[t] \wedge newoptPeriod[s_t] = t$. For a given full period t with $c_t > 0$, $gainCost[t]$ depends on the order s_t and also depends on $gainCost[optPeriod[s_t]]$ since there is recursively another freed place created when s_t jumps to t.

We want to identify the order s_t that will take the freed place in t when an order is removed from t. This order must have the highest "potential *gainCost*" for period t among all other order in $candidate[t]$. More formally,

Definition 6 Let $(gainCost[t])^k$ be the potential $gainCost[t]$ by assuming that it is the order $k \in candidate[t]$ that takes the freed place in t when an order is removed from t:

$$(gainCost[t])^k = (t - optPeriod[k]) \cdot h_k + gainCost[optPeriod[k]]$$

The objective is to find the order $s_t \in candidate[t]$ with the following property: $(gainCost[t])^{s_t} \geq (gainCost[t])^k, \forall k \in candidate[t]$ and then $gainCost[t] = (gainCost[t])^{s_t}$.

For each full period t, let $toSelect[t]$ denote the set of orders in $candidate[t]$ that have the highest stocking cost: $toSelect[t] = \arg\max_{k \in candidate[t]} h_k$.

Proposition 5 *For a full period t with $candidate[t] \neq \emptyset$: $s_t \in toSelect[t]$.*

Proof Consider a full period t such that $candidate[t] \neq \emptyset$ (and then $toSelect[t] \neq \emptyset$). If $s_t \notin toSelect[t]$, then $\exists k \in toSelect[t]$ such that $X_k^{\max} \geq t \wedge h_k > h_{s_t}$. This is not possible in an optimal solution (see condition (ii) of Proposition 1). □

Now we know that $s_t \in toSelect[t]$, but which one exactly must we select? The next proposition states that whatever order s in $toSelect[t]$ is chosen, we can compute $gainCost[t]$ from s. That means that all orders in $toSelect[t]$ have the same potential *gainCost*.

Proposition 6 $\forall s \in toSelect[t], (gainCost[t])^s = gainCost[t]$.

Proof If $|toSelect[t]| = 1$, then the proposition is true. Now we assume that $|toSelect[t]| > 1$. Consider two orders X_{k_1} and X_{k_2} in $toSelect[t]$. From Proposition 1, only null capacity periods can appear between $optPeriod[X_{k_1}]$ and $optPeriod[X_{k_2}]$ because $k_1, k_2 \in \arg\max_{k \in candidate[t]} h_k$ (and $X_k^{\max} \geq t$). Since all orders $X_k \in toSelect[t]$ have the same stocking cost and $X_k^{\max} \geq t$, any pair of orders k_1 and k_2 in $toSelect[t]$ can swap their respective *optPeriod* without affecting the feasibility of the solution and the optimal cost. Thus the proposition is true. □

We can summarize the computation of *gainCost* for each period in a full set.

Corollary 3 *Consider a full set* $\{M, \ldots, m\}$ *with* $m = minfull[t]$ *and* $M = maxfull[t]$, $\forall t \in \{M, \ldots, m\}$*:* $gainCost[m] = 0$ *and for all full periods* $t \in \{M, \ldots, m\} \setminus \{m\}$ *from* m *to* M*:*
$gainCost[t] = (t - optPeriod[s]) \cdot h_s + gainCost[optPeriod[s]]$ *with* $s \in toSelect[t]$.

By assuming that $gainCost[t], \forall t$ is known, the next proposition gives a lower bound on $(H^{opt}_{X_i \leftarrow v})^r$.

Proposition 7

$$(H^{opt}_{X_i \leftarrow v})^r \geq (H^{opt})^r + (optPeriod[i] - v) \cdot h_i - gainCost[optPeriod[i]]$$

Proof The cost $gainCost[optPeriod[i]]$ is the maximum decrease in cost when an order is removed from $optPeriod[i]$. We know that the cost $(optPeriod[i] - v) \cdot h_i$ is a lower bound on the increased cost when the order X_i is forced to take the value v because the capacity restriction can be violated for the period v. Thus $(H^{opt})^r + (optPeriod[i] - v) \cdot h_i - gainCost[optPeriod[i]]$ is a lower bound on $(H^{opt}_{X_i \leftarrow v})^r$. □

In the following, we use this lower bound to filter the decision variable.

4.3 Pruning $X_i^{\min}$

From the lower bound on $(H^{opt}_{X_i \leftarrow v})^r$ (Proposition 7), we have the following filtering rule for variables $X_i, \forall i \in [1, \ldots, n]$.

Corollary 4 $\forall i \in [1, \ldots, n]$,

$$X_i^{\min} \geq optPeriod[i] - \lfloor \frac{H^{\max} - (H^{opt})^r + gainCost[optPeriod[i]]}{h_i} \rfloor$$

Proof We know that v can be removed from the domain of X_i if $(H^{opt}_{X_i \leftarrow v})^r > H^{\max}$ and $(H^{opt}_{X_i \leftarrow v})^r \geq (H^{opt})^r + (optPeriod[i] - v) \cdot h_i - gainCost[optPeriod[i]]$. The highest integer value v^* that respects the condition $(H^{opt})^r + (optPeriod[i] - v) \cdot h_i - gainCost[optPeriod[i]] \leq H^{\max}$ is
$v^* = optPeriod[i] - \lfloor \frac{H^{\max} - (H^{opt})^r + gainCost[optPeriod[i]]}{h_i} \rfloor$. □

Algorithm 3 computes $gainCost[t]$ for all full periods in $[1, \ldots, T]$ in chronogical order and filters the n decision variables. It uses the stack *orderToSelect* that, after processing, contains an order in $toSelect[t]$ on top. At each step, the algorithm treats each full set (loop $5 - 15$) from their respective minfull periods thanks to *fullSetsStack* computed in Algorithm 1. For a given full set, the algorithm pops each full period (in chronological order) and computes its $gainCost[t]$ until the current full set is empty ; in this case it takes the next full set in *fullSetsStack*. Now let us focus on how $gainCost[t]$ is computed for each full period t. For a given full period t, the algorithm puts all orders in $left[t]$ into the stack *orderToSelect* (lines $14 - 15$) in chronological order. Thus for a given period t, for each pair of orders k_1 (with $X_{k_1}^{\max} \geq t$) and k_2 (with $X_{k_2}^{\max} \geq t$) in *orderToSelect*: if k_1 is above k_2, then $h_{k_1} \geq h_{k_2}$. The algorithm can safely remove orders k with $X_k^{\max} < t$ from the top of the stack (lines $7 - 8$) since these orders $k \notin candidate[t'], \forall t' \geq t$. After this operation, if the stack *orderToSelect* is not empty ($orderToSelect.isNotEmpty =$

True), the order on top is an order in *toSelect*[t] (Invariant (b) - see the proof of Proposition 8) and can be used to compute *gainCost*[t] based on Corollary 3 (lines $12-13$). Note that for a period t if the stack is empty, then *toSelect*[t] $= \emptyset$ (ie *candidate*[t] $= \emptyset$) and *gainCost*[t] $= 0$. Algorithm 3 has three invariants that ensure that *gainCost*[t] for all full period t are well computed. At the end, this algorithm filters each variable X_i based on the lower bound from Corollary 4 (lines $16-18$).

Algorithm 3 uses the classical primitives of stack such as 1) *isNotEmpty* (resp. *isEmpty*) that returns *True* (resp. *False*) if the stack is not empty and *False* (resp. *True*) otherwise; 2) *first* that returns the element on the top of the stack; and 3) *pop* that returns the element on the top of the stack and removes it from the stack.

Proposition 8 Algorithm 3 *computes gainCost*[t] *for all* $O(n)$ *full periods in linear time* $O(n)$.

Proof Invariants:

(a) After line 6, $\forall t' \in [minfull[t], \ldots, t[$ with $c_{t'} > 0$: *gainCost*[t'] is defined.
For a given full set fs, the algorithm computes the different *gainCost* of periods inside fs in increasing value of t from its $minfull$. Thus Invariant (a) holds.

(c) After line 15, $\forall k_1, k_2 \in \{X_k \in X : k \in orderToSelect \wedge X_k^{\max} \geq t\}$: if k_1 is above k_2 in *orderToSelect*, then $h_{k_1} \geq h_{k_2}$.
From Proposition 1, we know that $\forall k_1, k_2$ such that $optPeriod[k_1] < optPeriod[k_2]$ we have $h_{k_1} \leq h_{k_2}$ or $((h_{k_1} > h_{k_2}) \wedge (X_{k_1}^{\max} < optPeriod[k_2]))$. The algorithm pushes orders into *orderToSelect* from $minfull[t]$ to t. If we are on period $t' = optPeriod[k_2]$, then all orders k_1 pushed before are such that $(h_{k_1} \leq h_{k_2})$ or $((h_{k_1} > h_{k_2}) \wedge (X_{k_1}^{\max} < t'))$. Thus Invariant (c) holds.

(b) After line 8, $orderToSelect.first \in toSelect[t]$.
For a period t, the loop $14-15$ ensures that all orders X_i in $left[t]$ are pushed once in the stack. The loop $7-8$ removes orders that are on the top of stack such that $\{k : X_k^{\max} < t\}$. That operation ensures that the order s on the top of the stack can jump to the period t (i.e. $X_s^{\max} \geq t$). Since this order is on the top and Invariant (c) holds for the previous period processed, it has the highest stocking cost wrt $\{k : k \in orderToSelect \wedge X_k^{\max} \geq t\}$ and then $s \in toSelect[t]$. Thus Invariant (b) holds.
Based on Invariant (a) and Invariant (b), the algorithm computes *gainCost*[t] for each full period from Corollary 3.

Complexity: there are at most n full periods and then the main loop of the algorithm (lines $3-15$) is executed $O(n)$ times. Inside this loop, the loop $14-15$ that adds orders of the current period to *orderToSelect* is in $O(c)$ with $c = \max\{c_t, t \in fullSetsStack\}$. On the other hand, the orders removed from the stack *orderToSelect* by the loop at lines $7-8$ will never come back into the stack and then the complexity associated is globally in $O(n)$. Hence, the global complexity of the computation *gainCost*[t] for all full periods is $O(n)$. □

We give below an example of the execution of Algorithm 3.

Example 3 Let us run Algorithm 3 on the instance of Example 1. There are two full sets: $fullSetsStack = \{\{8, 7\}, \{5, 4, 2, 1\}\}$

1. $fullSet = \{5, 4, 2, 1\}$, $orderToSelect \leftarrow \{\}$.

Algorithm 3: Filtering of n date variables - $O(n)$

```
   Input: optPeriod, optOrders and fullSetsStack (computed in Algorithm 1)
1  gainCost ← map() // gainCost(t)=cost won if an order is removed in t
2  orderToSelect ← stack() // items that could jump on current period
3  while fullSetsStack.notEmpty do
4      fullSet ← fullSetsStack.pop
5      while fullSet.isNotEmpty do
6          t ← fullSet.pop
           // Invariant (a): ∀t' ∈ [minfull[t], ..., t[ with c_{t'} > 0: gainCost[t'] is
              defined
7          while orderToSelect.isNotEmpty ∧ (X^max_{orderToSelect.top} < t) do
8              orderToSelect.pop
           // Invariant (b): orderToSelect.first ∈ toSelect[t]
9          if orderToSelect.isEmpty then
10             gainCost[t] ← 0
11         else
12             s ← orderToSelect.first
13             gainCost[t] ← gainCost[optPeriod[s]] + (t − optPeriod[s]) · h_s
14         while optOrders[t].isNotEmpty do
15             orderToSelect.push(optOrders[t].pop)
           // Invariant (c): ∀k1, k2 ∈ {k ∈ X : k ∈ orderToSelect ∧ X^max_k ≥ t}: if k1 is
              above k2 in orderToSelect, then h_{k1} ≥ h_{k2}
16 for each order X_i do
17     v ← optPeriod[i] − ⌊(H^max + gainCost[optPeriod[i]] − (H^opt)^r) / h_i⌋
18     X^min_i ← max{X^min, v}
```

- $t = 1$, $gainCost[1] = 0$ and $orderToSelect \leftarrow \{4\}$,
- $t = 2$, $s = 4$, $gainCost[2] = gainCost[1] + (2-1) \cdot h_4 = 2$ and $orderToSelect \leftarrow \{4, 1\}$,
- $t = 4$, $s = 1$, $gainCost[4] = gainCost[2] + (4-2) \cdot h_1 = 8$ and $orderToSelect \leftarrow \{4, 1, 3\}$,
- $t = 5$, after line 8 $orderToSelect \leftarrow \{4\}$, $s = 4$, $gainCost[5] = gainCost[1] + (5-1) \cdot h_4 = 8$ and $orderToSelect \leftarrow \{4, 2\}$.

2. $fullSet = \{8, 7\}$, $orderToSelect \leftarrow \{\}$.
   - $t = 7$, $gainCost[7] = 0$ and $orderToSelect \leftarrow \{5\}$,
   - $t = 8$, $s = X_5$, $gainCost[8] = gainCost[7] + (8-7) \cdot h_5 = 2$ and $orderToSelect \leftarrow \{5, 6\}$.

Now the filtering is achieved for each order. Consider the order X_2: $v = optPeriod[2] - \lfloor \frac{H^{\max} + gainCost[optPeriod[2]] - (H^{opt})^r}{h_i} \rfloor = 5 - \lfloor \frac{34+8-16}{10} \rfloor = 3$ and $X_2^{\min} = 3$. Since $c_3 = 0$, $X_2^{\min} = 4$ thanks to the gcc constraint.

4.4 Strengthening the filtering

During the search some orders are fixed by branching decisions or during the filtering in the fix-point calculation. The lower bound $(H^{opt})^r$ can be strengthened by preventing these fixed orders from moving. This strengthening requires very little modification to our algorithm. First the fixed orders are filtered out such that they are not considered by Algorithm 1 and Algorithm 3. A reversible [4] integer maintains the contributions of those to the objective. This value is denoted H^{fixed}. Also when an order is fixed in a period t, the corresponding capacity c_t - also a reversible integer - is decreased by one. The strengthened bound is then $(H^{opt})^r + H^{fixed}$. This bound is also used for the filtering of the X_i's.

4.5 Pruning $X_i^{\max}$

Algorithm 3 uses a lower bound on the marginal cost $m_{X_i \leftarrow v}$ when the variable X_i is forced to take a value v such that $v < optPeriod[i]$ to filter $X_i^{\min}$. One could compute $m_{X_i \leftarrow v}$ for $v \in [optPerio[i] + 1, \ldots, X_i^{\max}]$ and then filter the decision variable accordingly as illustrated in the next example.

Example 4 Consider the following instance. `IDStockingCost`$([X_1 \in [1, \ldots, 4], X_2 \in [1, \ldots, 3], X_3 \in [1, \ldots, 3], X_4 \in [1, \ldots, 4]], [d_1 = 4, d_2 = 3, d_3 = 3, d_4 = 4], [h_1 = 1, h_2 = 10, h_3 = 20, h_4 = 2], H \in [0, \ldots, 20], c_1 = c_2 = c_3 = c_4 = 1)$. Fig. 6 shows the optimal solution wrt $\mathcal{P}^r$. The cost of this solution is $H^{opt} = h_2 + 3 * h_1 = 13$ and then the domain of the variable H can be updated to $[13, \ldots, 20]$.

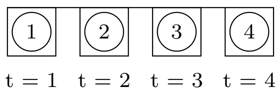

Fig. 6 The optimal assignment for $\mathcal{P}^r$ of Example 4

If the variable X_1 is forced to take the value 2, the order 2 must be delayed and the optimal cost will increase by $m_{X_1 \leftarrow 2} = h_2 - h_1 = 9$. Thus $(H^{opt}_{X_1 \leftarrow 2})^r = 13 + 9 = 22$ and the value 2 can be removed from the domain of X_1 since $(H^{opt}_{X_1 \leftarrow 2})^r > H^{\max}$. However the evolution of $m_{X_1 \leftarrow v}$ with $v > optPeriod[i]$ is not monotone and prevents us to directly update $X_1^{\max}$ to 2. In this example, $m_{X_1 \leftarrow 3} = h_2 + h_1 - 2 * h_0 = 28$ (see the corresponding solution in Fig. 7) and $m_{X_1 \leftarrow 4} = 3 * h_3 - 3 * h_0 = 3$ (see the corresponding solution in Fig. 8). The value 3 should be removed from the domain of X_1 but not the value 4.

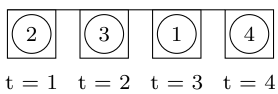

Fig. 7 The optimal assignment corresponding to $(H^{opt}_{X_1 \leftarrow 3})^r$

To filter $X_i^{\max}$ based on $m_{X_i \leftarrow v}$ for $v > optSlot[i]$, one should test (explicitely or implicitely) the different values from $X_i^{\max}$ to $optSlot[i]$ for each variable X_i.

[4] A reversible variable is a variable that can restore its domain when backtracks occur during the search.

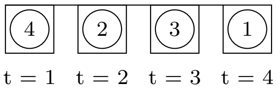

Fig. 8 The optimal assignment corresponding to $(H^{opt}_{X_1 \leftarrow 4})^r$

Since this work focusses on the scalability of the filtering, prefer not to increase the complexity of our algorithm to $O(n^2)$, the complexity of a naive approach that would test each value one by one. Finding an efficient algorithm that can efficiently update $X_i^{\max}, \forall i$ in less than $O(n^2)$ is an open research question. In this work we simply rely on the decomposed model to filter $X_i^{\max}, \forall i$.

4.6 Consistency property

We make a relaxation to filter the cost variable H in order to have a fast filtering algorithm. Moreover, to filter the decision variable, we use a lower bound on the marginal cost when a variable is forced to take a value $v < \mathit{optSlot}[i]$ and do not consider the case in which $v > \mathit{optSlot}[i]$. Hence the filtering obtained is weaker than bound consistency but offers a good computational tradeoff as shown in the next section. It is worth noting that this kind of partial filtering (such as that based on linear programming reduced costs) is difficult to characterize but often provides a relatively good filtering.

5 Experimental Results

This section describes the experiments we performed on a variant of capacitated lot-sizing called the Pigment Sequencing Problem [12].

5.1 The problem description

We consider the multiple item capacitated lot-sizing problem with sequence-dependent changeover costs. There is a single machine with capacity limited to one unit per period. There are item-dependent stocking costs and sequence-dependent changeover costs: 1) the total stocking cost of an order is proportional to its stocking cost and the number of periods between its due date and the production period; 2) the changeover cost is induced when passing from the production of one item another. More precisely, consider n orders (from $m \leq n$ different items[5]) that have to be scheduled over a discrete time horizon of T periods on a machine that can produce one unit per period. Each order $p \in [1, \ldots, n]$ has a due date d_p and a stocking (storage) cost $h_{\mathcal{I}(p)} \geq 0$ (in which $\mathcal{I}(p) \in [1, \ldots, m]$ is the corresponding item of order p). There is a changeover cost $q^{i,j} \geq 0$ between each pair of items (i, j) with $q^{i,i} = 0, \forall i \in [1, \ldots, m]$. Let $\mathit{successor}(p)$ be the order produced just after producing the order p. One wants to associate to each order p a production period $\mathit{date}(p) \in [1, \ldots, T]$ such that each order is produced on or

[5] item: order type.

before its due date ($date(p) \leq d_p, \forall p$), the capacity of the production is respected ($|\{p \mid date(p) = t\}| \leq 1, \forall t \in [1, \ldots, T]$), and the total stocking costs and changeover costs ($\sum_p (d_p - date(p)) \cdot h_{\mathcal{I}(p)} + \sum_p q^{\mathcal{I}(p),\mathcal{I}(successor(p))}$) are minimized. A small instance of the PSP is described next.

Example 5 Two types of orders (1 and 2) must be produced over the planning horizon $[1, \ldots, 5]$: $m = 2$, $T = 5$ and $c_t = 1, \forall t \in [1, \ldots, T]$. The stocking costs are respectively $h_1 = 5$ and $h_2 = 2$ for each item. The demands for item 1 are $d^1_{t \in [1,\ldots,5]} = [0, 1, 0, 1, 0]$ and for the second item are $d^2_{t \in [1,\ldots,5]} = [0, 0, 1, 0, 1]$. Thus the number of orders is $n = 4$, two for each item. The changeover costs are: $q^{1,2} = 10$, $q^{2,1} = 5$ and $q^{1,1} = q^{2,2} = 0$. The solution $S_1 = [1, 2, 0, 1, 2]$ (represented in Fig. 9) is a feasible solution. This means that the item 1 will be produced in

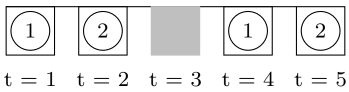

Fig. 9 A feasible solution of the PSP instance of Example 5

periods 1 and 4 while the item 2 will be produced in periods 2 and 5. Period 3 is an idle period[6]. The cost associated to S_1 is $C_{s_1} = h_1 + h_2 + q^{1,2} + q^{2,1} + q^{1,2} = 32$. The optimal solution for this problem is $S_2 = [2, 1, 0, 1, 2]$ (represented in Fig. 10) with cost $C_{s_2} = 19$.

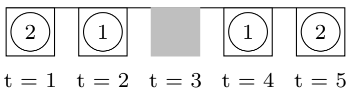

Fig. 10 An optimal solution of the PSP instance of Example 5

To the best of our knowledge, the state-of-the-art of exact method for the PSP is an Integer Programming formulation strengthened by some particular valid inequalities. We refer to [12] for details concerning this formulation.

5.2 The CP model

The model used is a variant of that described in [7]. Each order is uniquely identified. The decision variables are $date[p] \in [1, \ldots, T]$, $\forall p \in [1, \ldots, n]$. For the order p, $date[p]$ is the period for production of the order p. Note that $date[p]$ must repect its $dueDate[p]$: $date[p] \leq dueDate[p]$. Let $objStorage$ denote the total stocking cost: $objStorage = \sum_p (dueDate(p) - date(p)) \cdot h_p$ with $h_p = h_i$ is the stocking cost of the order p for an item i. The changeover costs are computed as in [7] using a successor based model and the *circuit* [11] constraint. The changeover costs

[6] idle period: period in which there is no production.

are aggregated into the variable $objChangeOver$. The total stocking cost variable $objChangeOver$ is computed using the constraint introduced in this paper:

$$\texttt{IDStockingCost}(date, dueDate, [h_1, \ldots, h_n], objStorage, [c_1, \ldots, c_T])$$

with $c_t = 1, \forall t \in [1, \ldots, T]$. The overall objective to minimize is: $objStorage + objChangeover$.

5.3 Methodology and experimental settings

In our experiment we study the gains obtained in terms of filtering and speed when replacing the `IDStockingCost` constraint by its decomposition or using the alternative `minimumAssignment` formulations. The implementations and tests have been realized within the OscaR open source solver [16]. All our source-code for the models, the global constraints and the instances are available at [?].

All experiments were conducted on a 2.4 GHz Intel core i5 processor using OS X 10.11. The evaluation of our global constraint uses the methodology that is described in [18]. The search tree with a baseline model is recorded and then the gains are computed by replaying the search tree with stronger alternative filtering. This allows us to use dynamic search heuristics without interfering with the filtering. In particular we use the conflict ordering search (COS) [5] that performs well on the problem. The baseline model (called *Basic*) is obtained by decomposing `IDStockingCost` as:

- `allDifferent(date)` using a forward checking filtering,
- $\sum_p (dueDate(p) - date(p)) \cdot h_p \leq objStorage$

The search tree recorded are obtained with an exploration limit of 60 seconds using the *Basic* model.

5.4 Comparison on small instances

As first experiment, we consider 100 small random instances limited to 20 periods, 20 orders and 5 items. We measure the gains over the *Basic* model using as filtering for `IDStockingCost`:

1. *IDS* : our filtering algorithms for `IDStockingCost`.
2. *MinAss*: the `minimumAssignment` constraint with linear programming (LP) reduced costs based filtering + the `allDifferent` constraint with bound consistency filtering. Actually, after some experiments, the `minimumAssignment` constraint is much more efficient when it is together with the `allDifferent` constraint (bound consistency filtering).
3. $MinAss_2$: the `minimumAssignment` constraint with exact reduced costs based filtering [3] + `allDifferent` constraint with bound consistency filtering.

Table 1 shows the arithmetic average of the number of nodes and the time required for *Basic*, *MinAss*, $MinAss_2$ and *IDS* respectively. Table 1 also shows the geometric average gain factor (wrt *Basic*) for each propagator. Not surprisingly, $MinAss_2$ prunes the search trees the most, but this improved filtering does not compensate for the time needed for the exact reduced costs. It is still at least 4 times (on average) slower than *MinAss* and *IDS*.

These results suggest that on small instances *MinAss* offers the best trade-off wrt filtering/time. Notice that *IDS* is competitive with *MinAss* in terms of computation time.

	IDS		*MinAss*		*MinAss*$_2$		*Basic*	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
Average (Av.)	30.1 10^4	15.7	26.2 10^4	**13.2**	**24.9** 10^4	51.7	130 10^4	51.4
Av. gain factor	5.0	4.0	6.2	**4.9**	**6.7**	1.0	1.0	1.0

Table 1 Results on instances with $T = 20$: *IDS*, *MinAss*, *MinAss*$_2$ and *Basic*

5.5 Comparison on large instances

The previous results showed that *MinAss* and *IDS* are competitive filtering for the `IDStockingCost` constraint on small instances. We now scale up the instance sizes to 500 periods (with the number of orders $n \in [490, \ldots, 500]$) and 10 different items. Again we consider 100 random instances. Table 2 gives the average values for the number of nodes and computation time when replaying the search trees, plus the geometric average gain over the *Basic* approach. Clearly, the reported values suggest that *IDS* performs best, in particular wrt the computation time. Fig. 11 shows the performance profiles (for *IDS*, *MinAss* and *Basic*) wrt the number of nodes visited and the time needed to complete the search respectively. For a given propagator, the performance profile provides a cumulative distribution of its performance wrt the best propagator on each instance. For a point (x, y) on the performance profile, the value $(1 - y)$ gives the percentage of instances for which the given propagator was at least x times worse than the best propagator. Then from Fig. 11, we can see that:

- wrt nodes: for ≈ 80% of instances, *IDS* provides the best filtering;
- wrt time: *IDS* requires the least time for all instances. Note that *IDS* is at least 4 times as fast as *MinAss* for ≈ 60% of instances.

	IDS		*MinAss*		*Basic*	
	Nodes	Time	Nodes	Time	Nodes	Time
Average (Av.)	**6.68** 10^4	**5.8**	8.30 10^4	25.8	73.5 10^4	52.8
Av. gain factor	**12.7**	**10.0**	11.1	2.3	1.0	1.0

Table 2 Results on instances with $T = 500$: *IDS*, *MinAss* and *Basic*

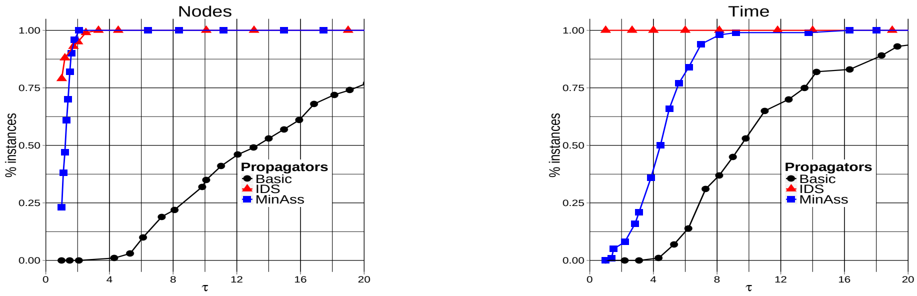

Fig. 11 Performance profiles: *IDS*, *MinAss* and *Basic*

5.6 *IDS* vs `StockingCost`

The `IDStockingCost` constraint generalizes the `StockingCost` constraint that we introduced in [7]. We now compare the performance of *IDS* with `StockingCost` on instances with equal stocking costs. We reuse the previous 100 instances generated with 500 demands and time periods, but using the same stocking cost for all the items.

As can be observed in Table 3, both `StockingCost` and *IDS* outperform *MinAss*. *MinAss* is at least 8 times slower (on average) than *IDS* and `StockingCost`. Note that, as established in [7], `StockingCost` offers a bound consistent filtering and is thus as expected the best propagator in this setting. However, the average values reported in Table 3 show that *IDS* is competitive wrt `StockingCost`. This is confirmed by the performance profiles presented in Fig. 12:

- wrt nodes: for ≈ 80% of instances, `StockingCost` is not more than 1.1 times better than *IDS*;
- wrt time: for ≈ 80% of instances, *IDS* has the best time. However, on ≈ 5% of instances, it takes more than twice as long as `StockingCost`.

	`StockingCost`		*IDS*		*MinAss*		*Basic*	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
Average (Av.)	**8.81** 10^4	7.4	9.36 10^4	**7.2**	15.6 10^4	62.3	76.9 10^4	52.3
Av. gain factor	**11.4**	**8.8**	10.0	8.3	6.1	1.0	1.0	1.0

Table 3 Results on instances with $T = 500$: `StockingCost`, *IDS*, *MinAss* and *Basic*

6 Conclusion

We have introduced the `IDStockingCost` constraint to handle the stocking cost aspect of some Capacitated Lot Sizing problems using Constraint Programming. This constraint takes into account item independent stocking and production capacity that may vary over time. We have proposed a scalable filtering algorithm for

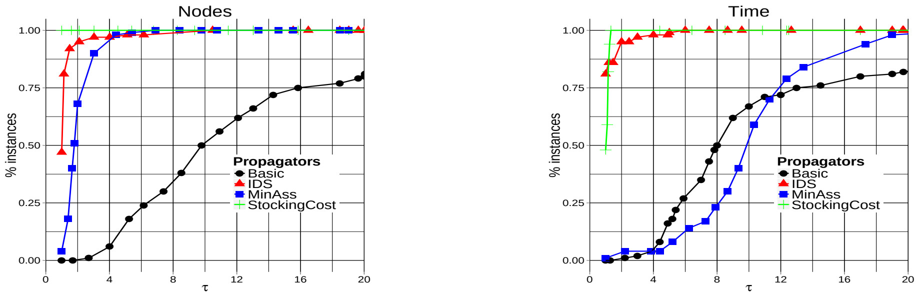

Fig. 12 Performance profiles: `StockingCost`, *IDS*, *MinAss* and *Basic*

this constraint in $O(n \log n)$. Our experimentation on a variant of the capacitated lot-sizing problem shows that the filtering algorithm proposed: 1) scales well wrt a CP formulation based on the minimum assignment problem, and 2) can be used instead of the `StockingCost` constraint [7] even when the stocking costs are the same for all items.

The filtering described in this paper is based on a lower bound on the marginal cost increase when one is forced to produce an order earlier than its optimal period. An interesting direction for future work is to compute efficiently the exact marginal cost and also consider the case when a variable is forced to take a value greater than its optimal period. Also, in this paper, we have focussed only on the filtering of the stocking costs that may arise in a CLSP. It would be interesting to propose some global constraints to efficiently filter the other costs of such problems (production costs, set up costs, changeover costs, etc.). In particular, an efficient filtering algorithm for the changeover cost part of the Pigment Sequencing Problem would certainly improve the performance of CP on this problem. On the other hand, this paper does not include customized heuristics for this problem. Research on search aspects should be conducted in order to compare the CP approach with the specialized approaches on these problems. For instance, one could use LNS [**?**] to drive the search quickly toward good solutions or develop dedicated heuristics.

References

1. Armentano, V.A., Franca, P.M., de Toledo, F.M.B.: A network flow model for the capacitated lot-sizing problem. Omega **27**, 275–84 (1999)
2. Barany, I., Roy, T.J.V., Wolsey, L.A.: Strong formulations for multi-item capacitated lot sizing. Management Science **30**, 1255–61 (1984)
3. Belvaux, G., Wolsey, L.A.: Modelling practical lot-sizing problems as mixed integer programs. Management Science **47**, 724–738 (2001)
4. Drexl, A., Kimms, A.: Lot sizing and scheduling - survey and extensions. European Journal of Operational Research pp. 221–235 (1997)
5. Ducomman, S., Cambazard, H., Penz, B.: Alternative filtering for the weighted circuit constraint: Comparing lower bounds for the tsp and solving tsptw. In: Thirtieth AAAI Conference on Artificial Intelligence (2016)
6. Focacci, F., Lodi, A., Milano, M.: Cost-based domain filtering. In: Principles and Practice of Constraint Programming–CP'99, pp. 189–203. Springer (1999)
7. Gay, S., Hartert, R., Lecoutre, C., Schaus, P.: Conflict ordering search for scheduling problems. In: Principles and Practice of Constraint Programming - CP 2015, pp. 144–148. Springer (2015)

8. German, G., Cambazard, H., Gayon, J.P., Penz, B.: Une contrainte globale pour le lot sizing. In: Journée Francophone de la Programation par Contraintes - JFPC 2015, pp. 118–127 (2015)
9. Ghomi, S.M.T.F., Hashemin, S.S.: An analytical method for single level-constrained resources production problem with constant set-up cost. Iranian Journal of Science and Technology **26**(B1), 69–82 (2001)
10. Gicquel, C.: Mip models and exact methods for the discrete lot-sizing and scheduling problem with sequence-dependent changeover costs and times. Ph.D. thesis, Ecole centrale Paris, France (2008)
11. Harris, F.W.: How many parts to make at once. Factory, The magazine of management **10**(2), 135–136 (1913)
12. Houndji, V.R., Schaus, P., Wolsey, L.: Cp4pp: Constraint programming for production planning. `https://bitbucket.org/ratheilesse/cp4pp`
13. Houndji, V.R., Schaus, P., Wolsey, L., Deville, Y.: The stockingcost constraint. In: Principles and Practice of Constraint Programming–CP 2014, pp. 382–397. Springer (2014)
14. Jans, R., Degraeve, Z.: Modeling industrial lot sizing problems: A review. International Journal of Production Research (2006)
15. Karimi, B., Ghomi, S.M.T.F., Wilson, J.: The capacitated lot sizing problem: a review of models. Omega, The international Journal of Management Science pp. 365–378 (2003)
16. Leung, J.M.Y., Magnanti, T.L., Vachani, R.: Facets and algorithms for capacitated lot sizing. Mathematical Programming **45**, 331–359 (1989)
17. López-Ortiz, A., Quimper, C.G., Tromp, J., van Beek, P.: A fast and simple algorithm for bounds consistency of the alldifferent constraint. In: International Joint Conference on Artificial Intelligence – IJCAI03 (2003)
18. Pesant, G., Gendreau, M., Potvin, J.Y., Rousseau, J.M.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transportation Science **32**(1), 12–29 (1998)
19. Pochet, Y., Wolsey, L.: Production Planning by Mixed Integer Programming. Springer (2005)
20. Quimper, C.G., Van Beek, P., López-Ortiz, A., Golynski, A., Sadjad, S.B.: An efficient bounds consistency algorithm for the global cardinality constraint. In: Principles and Practice of Constraint Programming–CP 2003, pp. 600–614. Springer (2003)
21. Régin, J.C.: Generalized arc consistency for global cardinality constraint. In: Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1, pp. 209–215. AAAI Press (1996)
22. Régin, J.C.: Cost-based arc consistency for global cardinality constraints. Constraints **7**(3-4), 387–405 (2002)
23. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: International Conference on Principles and Practice of Constraint Programming, pp. 417–431. Springer (1998)
24. Team, O.: Oscar: Scala in or. `https://bitbucket.org/oscarlib/oscar` (2012)
25. Ullah, H., Parveen, S.: A literature review on inventory lot sizing problems. Global Journal of Researches in Engineering **10**, 21–36 (2010)
26. Van Cauwelaert, S., Lombardi, M., Schaus, P.: Understanding the potential of propagators. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CPAIOR 2015, pp. 427–436. Springer (2015)