The StockingCost Constraint

Vinasétan Ratheil Houndji, Pierre Schaus, Laurence Wolsey and Yves Deville

Université catholique de Louvain, Louvain-la-Neuve, Belgium
{`vinasetan.houndji, pierre.schaus, laurence.wolsey, yves.deville`}`@uclouvain.be`

Abstract. Many production planning problems call for the minimization of stocking/storage costs. This paper introduces a new global constraint `StockingCost`$([X_1, \ldots, X_n], [d_1, \ldots, d_n], H, c)$ that holds when each item X_i is produced on or before its due date d_i, the capacity c of the machine is respected, and H is an upper bound on the stocking cost. We propose a linear time algorithm to achieve bound consistency on the `StockingCost` constraint. On a version of the Discrete Lot Sizing Problem, we demonstrate experimentally the pruning and time efficiency of our algorithm compared to other state-of-the-art approaches.

Keywords: Production Planning, Discrete Lot Sizing, Constraint Programming, Global Constraint

1 Introduction

Production planning problems, such as Lot Sizing and Scheduling Problems, require one to determine a minimum cost production schedule to satisfy the demands for single or multiple items without exceeding machine capacities while satisfying demands. Reviews of those problems and the corresponding Mixed Integer Programming (MIP) formulations are presented in [8,2,5,12]. In many Lot Sizing and Scheduling problems, in particular when the planning horizon is discrete and finite, there are stocking costs to minimize. These costs depend on the time spent between the production of an item and its delivery (due date).

To handle such Lot Sizing Problems in Constraint Programming, we propose an efficient bound consistency filtering algorithm for the `StockingCost`$([X_1, \ldots, X_n], [d_1, \ldots, d_n], H, c)$ constraint that requires each item X_i to be produced on or before its due date d_i and the capacity c of the machine to be respected.

First, we define the `StockingCost` constraint and how one can achieve pruning with the state-of-the-art approaches. After, we present some algorithms to achieve Bound Consistency for the total stocking costs H and for the items $X_i, i \in [1..n]$. Then, we propose a complete $O(n)$ filtering algorithm to achieve Bound Consistency for all variables. Finally, we present some experimental results on a Lot Sizing Problem and conclude.

2 The StockingCost Constraint

The StockingCost constraint has the following form:

$$\texttt{StockingCost}([X_1, \ldots, X_n], [d_1, \ldots, d_n], H, c)$$

where

- the variable X_i is the date of production of item i on the machine,
- the integer d_i is the due-date for item i,
- the integer c is the maximum number of items the machine can produce during one time slot (capacity),
- if an item is produced before its due date, then it must be stocked. The variable H is an upper bound on the total number of slots all the items are need in stock.

The StockingCost constraint holds when each item is produced before its due date ($X_i \leq d_i$), the capacity of the machine is respected (i.e. no more than c variables X_i have the same value), and H is an upper bound on the total stocking cost ($\sum_i (d_i - X_i) \leq H$).

Definition 1. *Each variable has a finite domain. We denote by* $X_i^{\min}$ *and* $H^{\min}$ *(resp.* $X_i^{\max}$ *and* $H^{\max}$*) the minimal (resp. maximal) value in the domain of variable* X_i *and* H*. We also denote* $t_{\max} = (\max_i(X_i^{\max}) - \min_i(X_i^{\min}))$.

Note that `StockingCost` can be viewed as a soft-constraint [13] that would impose every item to be produced exactly at the deadline. The stocking cost variable H is the violation of these deadlines. We use inequality instead of equality (as in van-Hoeve's definition of soft-constraints [13]) because `StockingCost` is an optimization constraint with H typically minimized. Observe that it differs from a standard inequality constraint mainly because H^{max} (the value representing the current best solution) will change during the search for a solution [13]. In particular, it implies that we can filter the domains of variables X_i with respect to $H^{\max}$, and potentially increase $H^{\min}$ with respect to X_i's.

The objective of a filtering algorithm is to remove values that do not participate in any solution of the constraint. In this paper, we are interested in achieving bound-consistency for the `StockingCost` constraint. This consistency level generally offers a good trade-off between speed and filtering power. In a bound consistent constraint, every variable bound (maximum or minimum) occurs in a solution of the constraint. More formally, the bound-consistency definitions for the `StockingCost` constraint are:

Definition 2. *Given a domain* $\mathcal{D}$ *of variables* X_i *and* H*, the constraint* $\texttt{StockingCost}([X_1, \ldots, X_n], [d_1, \ldots, d_n], H, c)$ *is* bound consistent *with respect to* $\mathcal{D}$ *iff*

- **BC($X_i^{\min}$)** $(1 \leq i \leq n)$ *Let* $x_i = X_i^{\min}$*; there exist* $x_j \in [X_j^{\min}..X_j^{\max}]$ $(1 \leq j \leq n, i \neq j)$ *and* $h = H^{\max}$ *such that* $\texttt{StockingCost}([x_1, \ldots, x_n], [d_1, \ldots, d_n], h, c)$ *holds*

- **BC($X_i^{\max}$)** $(1 \le i \le n)$ *Let* $x_i = X_i^{\max}$*; there exist* $x_j \in [X_j^{\min}..X_j^{\max}]$ $(1 \le j \le n, i \ne j)$ *and* $h = H^{\max}$ *such that* ***StockingCost***$([x_1, \dots, x_n], [d_1, \dots, d_n], h, c)$ *holds*
- **BC($H^{\min}$)** *Let* $h = H^{\min}$*; there exist* $x_i \in [X_i^{\min}..X_i^{\max}](1 \le i \le n)$ *such that* ***StockingCost***$([x_1, \dots, x_n], [d_1, \dots, d_n], h, c)$ *holds*

Decomposing the constraint

It is classical to decompose a global constraint into a conjunction of simpler constraints, and applying the filtering algorithms available on the simpler constraints. This raises two questions. First, does the filtering on the decomposition achieves bound consistency? Second, if it achieves the same filtering, what is the complexity of this filtering?

A first decomposition of the constraint `StockingCost`$([x_1, \dots, x_n], [d_1, \dots, d_n], h, c)$ is the following:

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

$$\sum_i (X_i = t) \le c, \forall t \tag{2}$$

$$\sum_i (d_i - X_i) \le H \tag{3}$$

Assuming that the filtering algorithms for each of the separate constraints achieve bound consistency, the above decomposition does not achieve bound consistency of the `StockingCost` constraint, as illustrated in the following example.

Example 1. Consider the following instance `StockingCost`$([X_1 \in [1..2], X_2 \in [1..2]], [d_1 = 2, d_2 = 2], H \in [0..2], c = 1)$. The naive decomposition is not able to increase the lower bound on H because the computation of H gives $(2 - X_1) + (2 - X_2) = [0..1] + [0..1] = [0..2]$. The problem is that it implicitly assumes that both items can be placed at the due date but this is not possible because of the capacity 1 of the machine. The lower bound of H should be set to 1. It corresponds to one item produced in period 1 and the other in period 2.

Other decompositions can be proposed to improve the filtering of the naive decomposition.

A first improvement is to use the global cardinality constraint (gcc) to model the capacity requirement of the machine imposing that no value should occur more than c times. The *gcc* constraint can efficiently replace t_{max} constraints of equation 2 in the basic decomposition. Bound consistency on the gcc constraint can be obtained in $O(n)$ plus the time for sorting the n variables [9]. However, together with equation 3, they do not achieve bound consistency of $StockingCost$ constraint.

A second possible improvement is to use a cost based global cardinality constraint (cost-gcc) [10]. In the $cost - gcc$, the cost of the arc (X_i, v) is equal to $+\infty$ if $v > d_i$ and $d_i - v$ otherwise. The $cost - gcc$ provides more pruning than equations 2 and 3 in the basic decomposition. Enforcing arc-consistency for cost-gcc requires a time complexity of $O(n \cdot S(m, n+d, \gamma))$ to check consistency where n is the number of variables, d is the size of the domains, m is the number arcs and $S(m, n+d, \gamma)$ is the complexity of the search for shortest paths from a node to every node in a graph with m arcs and $n+d$ nodes with a maximal cost γ [10]. For the `StockingCost`, there can be up to $n \cdot t_{\max}$ arcs. Hence[1] the final complexity to obtain arc-consistency[2] on the cost-gcc used to model `StockingCost` can be up to $O(t_{\max}^3)$. To the best of our knowledge the arc-consistent cost-gcc constraint has never been implemented in a solver. Note that for $c = 1$, one can use a minimum assignment constraint with a filtering based on reduced costs [4]. The consistency check for this constraint is achieved in $O(t_{\max}^3)$ (time complexity needed to solve a minimum assignment problem with the Hungarian algorithm). The advantage of the minimum assignment is that a minimum cost assignment can be recomputed in $O((t_{\max})^2)$ for one value removal. It is not possible to clearly characterize the filtering level achieved for the minimum assignment constraint based on reduced-costs.

Without loss of generality, in the rest of paper, we assume that $X_i^{\max} \leq d_i, \forall i$.

3 Pruning the cost variable

Given an assignment/solution $\bar{X}$ on variables $X = [X_1, \ldots, X_n]$, we denote by $H(\bar{X})$ the value $\sum_i (d_i - \bar{X}_i)$.

Observation 1 *For two assignments $\bar{X}$ and $\hat{X}$ satisfying $|\{X_i : X_i = t\}| \leq c$, if the sorted sequences of values in these solutions are the same, then $H(\bar{X}) = H(\hat{X})$.*

Let $\mathcal{P}$ denote the problem of computing the optimal lower-bound for H:

$$\begin{aligned} &H^{opt}(\mathcal{P}) = \min \sum_i (d_i - X_i) \ s.t. \\ &X_i^{\min} \leq X_i \leq X_i^{\max}, \forall i \\ &|\{X_i : X_i = t\}| \leq c, \forall t \end{aligned}$$

Algorithm 1 computes the optimal value $H^{opt}(\mathcal{P})$ in $O(n \cdot log(n))$ and detects infeasibility if the problem not feasible. This algorithm greedily schedules the productions from the latest to the first one. A current time line t is decreased and at each step, all the items such that $X_i^{\max} = t$ are stored into a priority queue (heap) to be scheduled next. Note that each item is added/removed exactly once in the heap and the heap is popped at each iteration (line 11). The items with largest $X_i^{\min}$ must be scheduled first until no more items can be scheduled in time t or the maximum capacity c is reached.

[1] using Fibonacci heap to implement Dijkstra algorithm for shortest path computation
[2] without considering incremental aspects.

Algorithm 1: Filtering of lower bound on H - **BC**($H^{\min}$)

```
Input: $X = [X_1, \dots, X_n]$ such that $X_i \leq d_i$ and sorted ($X_i^{\max} > X_{i+1}^{\max}$)
1  $H^{opt} \leftarrow 0$
   // total minimum stocking cost
2  $t \leftarrow X_1^{\max}$
   // current time slot
3  $slack \leftarrow c$
   // current slack at this time slot
4  $i \leftarrow 1$
5  $heap \leftarrow \{\}$
   // priority queue sorting items in decreasing $X_i^{\min}$
6  while $i \leq n$ do
7      while $i \leq n \wedge X_i^{\max} = t$ do
8          $heap \leftarrow heap \cup \{i\}$
9          $i \leftarrow i + 1$
10     while $heap.size > 0$ do
           // we virtually produce unit $j$ in $t$
11         $j \leftarrow heap.popFirst$
12         $slack \leftarrow slack - 1$
13         $H^{opt} \leftarrow H^{opt} + (d_j - t)$
14         if $t < X_i^{\min}$ then
15             the constraint is not feasible
16         if $slack = 0$ then
17             $t \leftarrow t - 1$
18             while $i \leq n \wedge X_i^{\max} = t$ do
19                 $heap \leftarrow heap \cup \{i\}$
20                 $i \leftarrow i + 1$
21             $slack \leftarrow c$
22     if $i \leq n$ then
23         $t \leftarrow X_i^{\max}$
24 $H^{\min} \leftarrow \max(H^{\min}, H^{opt})$
```

Let $\mathcal{P}^r$ denote the same problem with relaxed lower bounds of X_i:

$$H^{opt}(\mathcal{P}^r) = \min \sum_i (d_i - X_i) \ s.t.$$
$$X_i \leq X_i^{\max}, \forall i$$
$$|\{X_i : X_i = t\}| \leq c, \forall t$$

Observation 2 *If problem $\mathcal{P}$ is feasible (i.e. the gcc constraint is feasible), then $H^{opt}(\mathcal{P}) = H^{opt}(\mathcal{P}^r)$.*

Proof. If we use a simple queue instead of a priority queue in Algorithm 1, one may virtually assign items to times $t < X_i^{\min}$ and the feasibility test is not valid anymore, but the algorithm terminates with the same ordered sequence of time slots used in the final solution. By Observation 1, the objective values of optimal solutions are the same. The complexity of the algorithm without priority queue is $O(n)$ instead of $O(n \cdot log(n))$. □

The greedy Algorithm 1 is able to compute the best lower bound $H^{opt}(\mathcal{P}^r)$ (in the following we drop problem argument since optimal values are the same) and filters the lower bound of H if possible.

4 Pruning the item variable

From now on, since we assume the *gcc* constraint is already bound-consistent and thus feasible, only the cost argument may cause a filtering of lower-bounds $X_i^{\min}$. Therefore, in the rest of the article, we implicitly assumed relaxed domains $[-\infty..X_i^{\max}] \leq d_i$.

Definition 3. *Let $H_{X_i \leftarrow v}^{opt}$ denote the optimal lower bound in a situation where X_i is forced to take the value $v \leq X_i^{\max}$.*

Clearly, v must be removed from the domain of X_i if $H_{X_i \leftarrow v}^{opt} > H^{\max}$. An interesting question is: What is the minimum value v for X_i such that $H_{X_i \leftarrow v}^{opt} = H^{opt}$?

Definition 4. *Let v_i^{opt} denote the minimum value such that $H_{X_i \leftarrow v}^{opt} = H^{opt}$. We have $v_i^{opt} = \min\{v \leq X_i^{max} \ : \ H_{X_i \leftarrow v}^{opt} = H^{opt}\}$.*

The following observation gives a lower bound on the evolution on H^{opt} when a variable X_i is forced to take a value $v < v_i^{opt}$.

Observation 3 *For $v < v_i^{opt}$, we have $H_{X_i \leftarrow v}^{opt} \geq H^{opt} + (v_i^{opt} - v)$*

After the propagation of $H^{\min}$, one may still have some slack between the upper and the lower bound $H^{\max} - H^{\min}$. Since v_i^{opt} is the minimum value such that $H_{X_i \leftarrow v}^{opt} = H^{opt}$, we can use the lower bound of Observation 3 to filter X_i as follows:

$$X_i^{\min} \leftarrow \max(X_i^{\min}, v_i^{opt} - (H^{\max} - H^{\min}))$$

In the following we show that the lower-bound of Observation 3 can be improved and that we can actually predict the exact evolution of $H_{X_i \leftarrow v}^{opt}$ for an arbitrary value $v < v_i^{opt}$. A valuable information to this end is the number of items scheduled at a given time slot t in an optimal solution:

Definition 5. *In an optimal solution $\bar{X}$ (i.e. $H(\bar{X}) = H^{opt}$), let*

$$count[t] = |\{i \ : \ \bar{X}_i = t\}|.$$

Algorithm 2 computes $v_i^{opt}, \forall i$ and $count[t], \forall t$ in linear time $O(t_{\max})$. The first step of the algorithm is to initialize $count[t]$ as the number of variables with upper bound equal to t. This can be done in linear time assuming the time horizon of size $(\max_i\{X_i^{\max}\} - \min_i\{X_i^{\min}\})$ is in $O(n)$. We can initialize an array *count* of the size of the horizon and increment the entry $count[X_i^{\max}]$ of the array in $O(1)$ for each variable X_i.

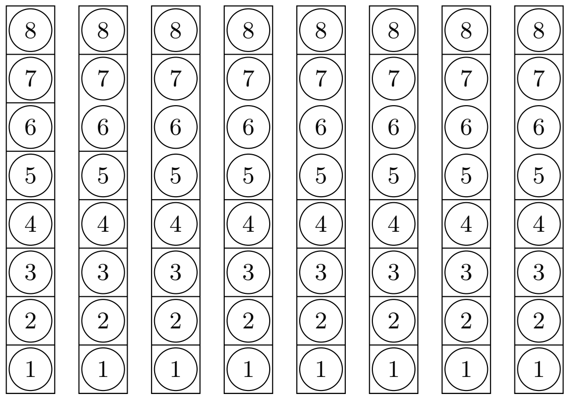
The idea of the Algorithm 2 is to use a Disjoint-Set T (also called union-find) data structure [1] making it possible to have efficient operations for $T.Union(S_1, S_2)$, grouping two disjoint sets into a same set, and $T.Find(v)$ returning a "representative" of the set containing v. It is easy to extend a disjoint-set data structure with operations $T.min(v)/T.max(v)$ returning the minimum/-maximum value of the set containing value v. As detailed in the invariant of the algorithm, time slots are grouped into a set S such that if $X_i^{\max} \in S$ then $v_i^{opt} = \min S$.

Algorithm 2: Compute v_i^{opt} for all i

```
1  Initialize count as an array such that $count[t] = |\{X_i : X_i^{\max} = t\}|$
2  Create a disjoint set data structure $T$ with the integers
   $t \in [\min_i\{X_i^{\min}\}, \max_i\{X_i^{\max}\}]$
3  $t \leftarrow \max_i\{X_i^{\max}\}$
4  repeat
5      while $count[t] > c$ do
6          $count[t-1] \leftarrow count[t-1] + count[t] - c$
7          $count[t] \leftarrow c$
8          $T.Union(t-1, t)$
       // invariant: $v_i^{opt} = t, \forall i \in \{i \ : \ t \leq X_i^{\max} \leq T.max(T.find(t))\}$
9      $t \leftarrow t - 1$
10 until $t \leq \min_i\{X_i^{\min}\}$
11 // if $count[\min_i\{X_i^{\min}] > c$ then the constraint is infeasible
12 $\forall i : v_i^{opt} = T.min(T.find(X_i^{\max}))$
```

Example 2. Consider the following instance StockingCost($[X_1 \in [1..3], X_2 \in [1..6], X_3 \in [1..7], X_4 \in [1..7], X_5 \in [1..8]], [d_1 = 3, d_2 = 6, d_3 = 7, d_4 = 7, d_5 = 8], H \in [0..4], c = 1$). At the beginning of the algorithm, $count = [0,0,1,0,0,1,2,1]$ and $T = \{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\}\}$. After the loop, $count = [0,0,1,0,1,1,1,1]$ and $T = \{\{1\},\{2\},\{3\},\{4\},\{5,6,7\},\{8\}\}$. Thus $v_{X_1}^{opt} = 3$, $v_{X_2}^{opt} = v_{X_3}^{opt} = v_{X_4}^{opt} = 5$ and $v_{X_5}^{opt} = 8$. Next figure shows the different steps of the computation of T. Note that from step 3 to step 8, there is no change since for $t \in [1..6]$, $count[t]$ always ≤ 1 in the loop.

Observation 3 gives a lower bound on the evolution of the optimal stocking cost when assigning variable X_i to v. Unfortunately, this lower bound is not



step 1 step 2 step 3 step 4 step 5 step 6 step 7 step 8
t = 8 t = 7 t = 6 t = 5 t = 4 t = 3 t = 2 t = 1

optimal. One can be convinced easily for instance that with $c = 1$, if $v < v_i^{opt}$ is assigned X_i, it virtually imposes to move to left at least all variables X_j such that $\{X_j : X_j^{max} = v\}$. This suggests for c=1, the following improved lower bound for $H^{opt}_{X_i \leftarrow v}$:

$$H^{opt}_{X_i \leftarrow v} \geq H^{opt} + (v_i^{opt} - v) + |\{X_j : X_j^{max} = v\}| \tag{4}$$

Next example illustrates that this lower bound is still not optimal. It is not sufficient just only consider the set $\{X_j : X_j^{max} = v\}$ since more variables could be impacted.

Example 3. Consider the following instance $\texttt{StockingCost}([X_1 \in [1..5], X_2 \in [1..4], X_3 \in [1..4]], [d_1 = 5, d_2 = 4, d_3 = 4], H \in [0..10], c = 1)$ with $H^{opt} = 1$ and $v^{opt}_{X_1} = 5$. For $v = 4$, $H^{opt}_{X_1 \leftarrow 4} \geq H^{opt} + (v^{opt}_{X_1} - v) + |\{X_j : X_j^{max} = v\}| = 1 + (5 - 4) + 2 = 4$. Here, $H^{opt}_{X_1 \leftarrow 4}$ is really 4. For $v = 3$, $H^{opt}_{X_1 \leftarrow 3} \geq H^{opt} + (v^{opt}_{X_1} - v) + |\{X_j : X_j^{max} = v\}| = 1 + (5 - 3) + 0 = 3$ but here $H^{opt}_{X_1 \leftarrow 3} = 4$.

Definition 6. *A slot t is full if it is using all its capacity $count[t] = c$.*

Observation 4 *There is at most $\lfloor \frac{n}{c} \rfloor$ full time slots.*

Definition 7. *$minfull[t]$ is largest time slot $\leq t$ which is not full. More exactly $minfull[t] = \max\{t' \leq t : count[t'] < c\}$.*

Definition 8. *$maxfull[t]$ is the smallest time slot $\geq t$ which is not full. More exactly $maxfull[t] = \min\{t' \geq t : count[t'] < c\}$.*

Next observation gives the exact evolution of $H^{opt}_{X_i \leftarrow v}$ that will allow the BC filtering of $X_i^{\min}$.

Observation 5 $H^{opt}_{X_i \leftarrow v} = H^{opt} + (v_i^{opt} - v) + (v - minfull[v]),\ \forall v < v_i^{opt}$

To understand the previous observation one can realize that the number of variables affected (that would need to be shifted by one to the left) by assigning $X_i \leftarrow v$ is equivalent to the impact caused by insertion of an artificial item with the domain $[-\infty..v]$. So, the exact impact of $X_i \leftarrow v$ is the number of variables affected by the move plus $(v - v^{opt}_{X_i})$. The Algorithm 3 computes $minfull[t], maxfull[t], \forall t$. The time complexity is thus $O(t_{\max})$.

Algorithm 3: Computation of $minfull[t], maxfull[t] \forall t$

```
1  Create a disjoint set data structure $F$ with the integers
   $t \in [\min_i\{X_i^{\min}\} - 1, \max_i\{X_i^{\max}\}]$
2  $t \leftarrow \max_i\{X_i^{\max}\}$
3  repeat
4      if $count[t] = c$ then
5          $F.Union(t-1, t)$
6      $t \leftarrow t - 1$
7  until $t < \min_i\{X_i^{\min}\}$
8  $\forall t : minfull(t) = F.min(F.find(t))$
9  $\forall t : |F.find(t)| > 1 : maxfull(t) = F.max(F.find(t)) + 1$
10 $\forall t : |F.find(t)| = 1 : maxfull(t) = F.max(F.find(t))$
```

Observation 6 $H^{opt}_{X_i \leftarrow t} \geq H^{opt}_{X_i \leftarrow t'} \forall t < t' \leq v^{opt}_i$

Observation 7 *If a slot t is full ($count[t] = c$) then $\forall i$:*

$$H^{opt}_{X_i \leftarrow t} = H^{opt}_{X_i \leftarrow t'},\ \forall t' \in [minfull[t]..maxfull[t])\ \textit{such that}\ t' < v^{opt}_i$$

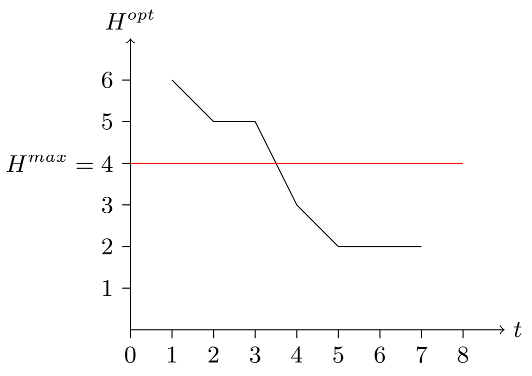
Proof. Suppose that a slot t is full. We know that $\forall t' \in [minfull[t]..maxfull[t])$, $minfull[t'] = minfull[t]$. Thus, $\forall t' \in [minfull[t]..maxfull[t])$ *such that* $t' < v^{opt}_i$, $H^{opt}_{X_i \leftarrow t'} = H^{opt} + (v^{opt}_i - t') + (t' - minfull[t']) = H^{opt} + v^{opt}_i - minfull[t] = H^{opt} + (v^{opt}_i - t) + (t - minfull[t]) = H^{opt}_{X_i \leftarrow t}$. □

The above observation is very important because if the new minimum for X_i falls on a full time slot, we can increase the lower bound further. The bound consistent filtering rule is given in Algorithm 4.

Algorithm 4: Bound Consistent Filtering of $X_i^{\min}$ - **BC**$(X_i^{\min})$

```
1  $newmin \leftarrow v^{opt}_i - (H^{\max} - H^{\min})$
2  if $count[newmin] = c$ then
3      $newmin \leftarrow \min\{v^{opt}_i, maxfull[newmin]\}$
4  $X_i^{\min} \leftarrow \max(X_i^{\min}, newmin))$
```

Example 4. Considering the following instance $\texttt{StockingCost}([X_1 \in [1..3], X_2 \in [1..6], X_3 \in [1..7], X_4 \in [1..7], X_5 \in [1..8]], [d_1 = 3, d_2 = 6, d_3 = 7, d_4 = 7, d_5 = 8], H \in [0..4], c = 1)$. We know that $v_{X_1}^{opt} = 3$, $v_{X_2}^{opt} = v_{X_3}^{opt} = v_{X_4}^{opt} = 5$, $v_{X_5}^{opt} = 8$ and $count = [0,0,1,0,1,1,1,1]$. After running the algorithm 1 we have $H^{opt} = 2$ and thus $H \in [2..4]$. Algorithm 3 gives $F = \{\{0\},\{1\},\{2,3\},\{4,5,6,7,8\}\}$, $minfull = [1,2,2,4,4,4,4,4]$ and $maxfull = [1,2,4,4,9,9,9,9]$. Algorithm 4 gives for:
- X_1 : $newmin = 3 - 2 = 1$. $count[1] = 0$ and $X_1^{min} = \max\{1,1\} = 1$;
- X_2, X_3, X_4 : $newmin = 5 - 2 = 3$. $count[3] = 1$, $newmin = \min\{4,5\} = 4$ and $X_{j\in\{2,3,4\}}^{min} = \max\{1,4\} = 4$. Next figure shows the evolution of $H_{X_3 \leftarrow t}^{opt}$. Note that for $t \in [1..3]$, $H_{X_3 \leftarrow t}^{opt} > H^{max} = 4$.



- X_5 : $newmin = 8 - 2 = 6$. $count[6] = 1$, $newmin = \min\{8,9\} = 8$ and $X_5^{min} = \max\{1,8\} = 8$.

Thus $X_1 \in [1..3]$, $X_2 \in [4..6]$, $X_3 \in [4..7]$, $X_4 \in [4..7]$ and $X_5 \in \{8\}$.

5 A complete filtering algorithm in $O(n)$

The Algorithms 2 and 3 for computing $v_i^{opt}, \forall i$ and $maxfull(t), \forall t$ presented so far have a complexity of $O(t_{\max})$. Although for some problems $t_{\max} \approx n$, in practice it can be larger than n if there is some sparsity on the deadlines. Algorithm 5 describes a complete self-contained version of the filtering for `StockingCost` running in $O(n)$ given a sorted version of the variables. This algorithm keeps tracks of the items in the same set (same v^{opt}) by maintaining two indexes j, k with the following properties:

- After line 10, items in $\{j, \ldots, i\}$ are the *open items* ($X_i : X_i^{\max} \geq t$) that still need to be placed into some slots in an optimal solution.
- After line 10, all the items in $\{k, \ldots, i\}$ have the same v^{opt}. This value v^{opt} is only known when all the current remaining open items can be placed into the current slot. That is when the condition at line 13 is true.

Variable u keeps track of the $maxfull(t)$ potential value with $maxfull(t)$ implemented as a map with constant time insertion. Only time slots t with $maxfull(t) > t$ are added to the map. Each time a full slot t is discovered (at lines 19 and 26), one entry is added to the map. By observation 4 the number of entries added into the map is at most n.

Lines 29 to 34 are just applying the filtering rules from Algorithm 4.

Implementation Details Although the Algorithm 5 is in $O(n)$, it requires the variables to be sorted. Since the filtering algorithms are called multiple times during the search process and only a few number of variables are modified between each call, simple sorting algorithms such as insertion or bubble sort are generally more efficient than classical sorting algorithms $O(n \cdot log(n))$.

The *map* can be a simple Hashmap but a simple implementation with two arrays of size $t_{\max}$ and a *magic number* incremented at each call can be used to avoid computing hash functions and the map object creation/initialization at each call to the algorithm. One array contains the value for each key index in the map, and the other array contains magic numbers containing the value of the magic number at the insertion. An entry is present only if the value at corresponding index in the magic array is equal to the current magic number. Incrementing the magic number thus amounts at emptying the map in $O(1)$. The cost $O(t_{\max})$ at the map creation has to be paid only once an is thus amortized.

6 Experimental results

Experiments were conducted on instances $MI - DLS - CC - SC$ (Multi Item - Discrete Lot Sizing - Constant Capacity - Setup Cost) problems described in [8].

Description of the $MI - DLS - CC - SC$ problem

The Discrete Lot Sizing problem considered here is a multi-item, single machine problem with capacity of production limited to one per period. There are storage costs and sequence-dependent changeover costs, respecting the triangle inequality. Each order consisting of one unit of a particular item has a due date and must be produced at latest by its due date. The stocking (inventory) cost of an order is proportional to the number of periods between the due date and the production period. The changeover cost $q^{i,j}$ is induced when passing from the production of item i to another one j with $q^{i,i} = 0\ \forall i$. Backlogging is not allowed. The objective is to assign a production period for each order respecting its due date and the machine capacity constraint so as to minimize the sum of stocking costs and changeover costs.

Next example shows a tiny instance of the problem.

Example 5. Consider the problem with the following input data: number of items type $nbItems = 2$; number of periods $nbPeriods = 5$; stocking cost $h = 2$;

Algorithm 5: Complete filtering algorithm in $O(n)$

```
Input:
X = [X_1, ..., X_n, X_{n+1}] such that X_i ≤ d_i and sorted (X_i^max > X_{i+1}^max)
X_{n+1}^max = −∞ // artificial variable
1  $H^{opt} \leftarrow 0$
2  $t \leftarrow X_1^{\max}$
3  $i \leftarrow 1$
4  $j \leftarrow 1$ // open items $\{j, \ldots, i\}$ must be placed in some slots
5  $k \leftarrow 1$ // items $\{k, \ldots, i\}$ have same $v^{opt}$
6  $u \leftarrow t + 1$
7  $maxfull \leftarrow map()$ // a map from int to int
8  while $i \le n \vee j < i$ do
9      while $i \le n \wedge X_i^{\max} = t$ do
10         $i \leftarrow i + 1$
       // place at most $c$ items into slot $t$
11     for $i' \in [j.. \min(i-1, j+c-1)]$ do
12         $H^{opt} \leftarrow H^{opt} + (d_{i'} - t)$
13     if $i - j \le c$ then // all the open items can be placed in $t$
14         $full \leftarrow i - j = c$ // true if $t$ is fill up completely
15         $v_l^{opt} \leftarrow t, \forall l \in [k..i)$
16         $j \leftarrow i$
17         $k \leftarrow i$
18         if $full$ then
               // invariant $\forall t' \in [t..u-1], count[t] = c$
19             $maxfull(t) \leftarrow u$
20             if $X_i^{\max} < t - 1$ then
21                 $u \leftarrow X_i^{\max} + 1$
22         else
23             $u \leftarrow X_i^{\max} + 1$
24         $t \leftarrow X_i^{\max}$
25     else // all open items can not be placed in $t$
           // invariant $\forall t' \in [t..u-1], count[t] = c$
26         $maxfull(t) \leftarrow u$
27         $j \leftarrow j + c$ // place $c$ items into slot $t$
28         $t \leftarrow t - 1$
29 $H^{\min} \leftarrow \max(H^{\min}, H^{opt})$
30 for $i \in [1..n]$ do
31     $newmin \leftarrow v_i^{opt} - (H^{\max} - H^{\min})$
32     if $maxfull(t).hasKey(newmin)$ then
33         $newmin \leftarrow \min\{v_i^{opt}, maxfull(newmin)\}$
34     $X_i^{\min} \leftarrow \max(X_i^{\min}, newmin))$
```

demand times for items of type 1 $d^1_{t\in\{1,\dots,5\}} = (0, 1, 0, 0, 1)$ and for items of type 2 $d^2_{t\in\{1,\dots,5\}} = (1, 0, 0, 0, 1)$; $q^{1,2} = 5$, $q^{2,1} = 3$. A feasible solution of this problem is $productionPlan = (2, 1, 2, 0, 1)$ which means that item 2 will be produced in period 1; item 1 in period 2; item 2 in period 3 and item 1 in period 5. Note that there is no production in period 4, it is an idle period. The cost associated to this solution is $q^{2,1}+q^{1,2}+q^{2,1}+2*h = 15$ but it is not the optimal cost. The optimal solution is $productionPlan = (2, 1, 0, 1, 2)$ with the cost $q^{2,1} + q^{1,2} + h = 10$.

A Constraint Programming Model

We uniquely identify each order. The aim is to associate to each of these orders a period that respects the due date of the order. Let $date(p) \in [1..nbPeriods]$, $\forall p \in [1..nbDemands]$, represents the period in which the order p is satisfied. This corresponds to period in which the order p is produced/satisfied. Let $dueDate(p)$ be the deadline for order p, that is the period in which p is due.

If $objStorage$ is an upper bound on the total number of periods in which orders have to be held in stock, the stocking part can be modeled by the constraint:

$$\texttt{StockingCost}(date, dueDate, objStorage, 1)$$

Observation 8 *There is no difference between two orders of the same item except for their due dates. Therefore given a feasible production schedule, if it is possible to swap the production periods of two orders involving the same item same item ($date(p_1), date(p_2)$ such that $item(p_1) = item(p_2)$), we obtain an identical solution with the same stocking cost..*

Based on observation 8, we remove such symmetries by adding precedence constraints on $date$ variables involving by the same item:

$$date(p_1) < date(p_2), \forall(p_1, p_2) \in [1..nbDemands] \times [1..nbDemands] \; such \; that$$

$$dueDate(p_1) < dueDate(p_2) \wedge item(p_1) = item(p_2)$$

Now, the second part of the objective $objChangeover$ concerning changeover costs has to be introduced in the model. This part is similar to a *successor* CP model for the Traveling Salesman Problem (TSP) in which the cities to be visited represent the orders and the distances between them are the corresponding changeover costs. Let $successor(p)$, $\forall p \in [1..nbDemands]$, define the order produced on the machine immediately after producing order p. We additionally create a dummy order $nbDemands + 1$ to be produced after all the other orders. In the first step, a Hamiltonian circuit successor variable is imposed. This is achieved by using the classical *circuit* [7] constraint on successor variables for dynamic subtour filtering. The *date* and *successor* variables are linked with the element constraint by imposing that the production date of p is before the production date of its successors:

$$\forall p \in [1..nbDemands] : date(p) < date(successor(p))$$

As announced, the artificial production is scheduled at the end:

$$date(nbDemands + 1) = nbPeriods + 1$$

Note that as with $date$ variables, some symmetries can be broken. For two nodes $n_1, n_2 \in [1..nbDemands]$ such that $dueDate(n_1) < dueDate(n_2)$ and $item(n_1) = item(n_2)$, we force that n_1 cannot be the successor of n_2 with $successor(n_2) \neq n_1$. Finally, a $minAsssignment$ constraint [3] is used on the $successor$ variables and the changeover part of the objective $objChangeover$.

The objective to minimize is simply the sum of stocking costs and changeover costs : $(objStorage * h) + objChangeover$, where h is the unit stocking cost.

Experimental results

By assuming that the basic filtering $date(p) \leq dueDate(p)$,$\forall p \in [1..nbDemands]$ is imposed a priori, we compare the performance of the filtering algorithm due to `StockingCost`$(date, deadline, objStorage, 1)$ constraint with that achieved by the following three sets of constraints:

- the basic decomposition :

$$\sum_p (date(p) = t) \leq 1, \forall t \in [1..nbPeriods]$$
$$\sum_p (dueDate(p) - date(p)) \leq objStorage$$

- t_{max} constraints of the previous decomposition are replaced by the global bound consistency constraint $allDifferentBC$ [6], that is the special case of gcc constraint when the capacity $c = 1$:

$$allDifferentBC(date)$$
$$\sum_p (dueDate(p) - date(p)) \leq objStorage$$

- the global constraint $minAssignment$ [3] on $date$ and $objStorage$ variables is added to the previous decomposition.

The `StockingCost` filtering algorithm and the $MI - DLS - CC - SC$ model have been implemented in the OscaR open-source solver [11]. They will be available in OscaR from release 1.1.0. As search heuristic, we used a classical static binary search on $date$ and $successor$ variables in order to reduce the impact of the search on model comparisons. Table 1 shows the results for some randomly generated instances of $MI - DLS - CC - SC$ [3]. We present, for each group of constraints, the number of nodes visited and the time (in seconds) used to complete the search.

[3] Instances available at `http://becool.info.ucl.ac.be/resources/discrete-lot-sizing-problem`

Instance	StockingCost		Minassignment		AllDifferent		Basic decomp	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
1(15_5_13)	**0.36** 10^6	**26**	0.41 10^6	36	1.25 10^6	87	1.25 10^6	99
2(15_5_14)	**0.98** 10^6	**79**	1.26 10^6	112	3.16 10^6	255	3.17 10^6	283
3(15_8_13)	**1.10** 10^6	**64**	2.34 10^6	156	8.05 10^6	515	8.07 10^6	625
4(15_10_12)	**0.22** 10^6	**12**	0.72 10^6	32	8.02 10^6	385	8.10 10^6	478
5(15_10_14)	**0.32** 10^6	**16**	1.41 10^6	79	18.7 10^6	1350	18.8 10^6	1552
6(20_5_17)	**1.14** 10^6	**135**	1.40 10^6	213	3.33 10^6	353	4.07 10^6	548
7(20_10_18)	**6.90** 10^6	**534**	8.02 10^6	805	9.68 10^6	906	—	—
8(20_10_19)	**1.32** 10^6	**95**	1.34 10^6	120	9.68 10^6	616	—	—
9(30_5_12)	**2.87** 10^6	**124**	3.00 10^6	223	3.00 10^6	127	3.00 10^6	188
10(30_10_11)	**5.51** 10^6	**244**	6.68 10^6	530	7.73 10^6	342	7.73 10^6	494
11(30_10_16)	**2.41** 10^6	**156**	4.64 10^6	439	—	—	—	—
12(100_10_11)	**1.49** 10^6	**60**	1.50 10^6	271	2.30 10^6	110	2.30 10^6	153
13(100_10_18)	**0.11** 10^6	**10**	0.15 10^6	63	0.36 10^6	23	2.96 10^6	331
14(100_15_17)	**2.79** 10^6	**143**	6.51 10^6	1132	22.2 10^6	1305	22.3 10^6	1712
15(200_15_22)	**19.3** 10^6	**854**	—	—	24.6 10^6	1187	24.6 10^6	2024

Table 1. Results for 15 MI-DLS-CC-SC instances. The format of instance is the following: $InstanceNumber(nbPeriods\_nbItems\_nbDemands)$. "—" means that the model did not complete the search after 3600 seconds.

These results suggest that our `StockingCost` version offers a stronger and faster filtering than other decompositions. In particular, the last four instances suggest that the time complexity of our filtering algorithm scales better than the *minAssignment* decomposition when the number of time slots increases. This is not surprising since filtering algorithm for `StockingCost` is in $O(nbDemands)$ and not a function of the size of horizon as is the case for the *minAssignment* decomposition.

7 Conclusion

In this paper, we have introduced a new global constraint `StockingCost` to handle the stocking aspect of Lot Sizing Problems when using Constraint Programming. We have described an advanced filtering algorithm achieving bound consistency with a time complexity linear in the number of variables. The experimental results show the pruning and time efficiency of the `StockingCost` constraint on a version of the Discrete Lot Sizing Problem compared to various decompositions of the constraint.

References

1. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms (Second ed.)*, chapter 21: Data structures for Disjoint

Sets. MIT press Cambridge, 2001.

2. A Drexl and A Kimms. Lot sizing and scheduling - survey and extensions. *European Journal of Operational Research*, pages 221–235, 1997.
3. Filippo Focacci, Andrea Lodi, and Michela Milano. Cost-based domain filtering. In *Principles and Practice of Constraint Programming–CP'99*, pages 189–203. Springer, 1999.
4. Filippo Focacci, Andrea Lodi, Michela Milano, and Daniele Vigo. Solving tsp through the integration of or and cp techniques. *Electronic notes in discrete mathematics*, 1:13–25, 1999.
5. Raf Jans and Zeger Degraeve. Modeling industrial lot sizing problems: A review. *International Journal of Production Research*, 2006.
6. Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *International Joint Conference on Artificial Intelligence – IJCAI03*, 2003.
7. Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.
8. Yves Pochet and Laurence Wolsey. *Production Planning by Mixed Integer Programming*. Springer, 2005.
9. Claude-Guy Quimper, Peter Van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Principles and Practice of Constraint Programming–CP 2003*, pages 600–614. Springer, 2003.
10. Jean-Charles Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3-4):387–405, 2002.
11. OscaR Team. Oscar: Scala in or. `https://bitbucket.org/oscarlib/oscar`, 2014.
12. Hafiz Ullah and Sultana Parveen. A literature review on inventory lot sizing problems. *Global Journal of Researches in Engineering*, 10:21–36, 2010.
13. Willem-Jan van Hoeve. Over-constrained problems. In *Hybrid Optimization*, pages 191–225. Springer, 2011.