

Six Ways of Integrating Symmetries within Non-Overlapping Constraints

M. Ågren¹, N. Beldiceanu², M. Carlsson¹, M. Sbihi², C. Truchet³, and S. Zampelli²

¹ SICS, P.O. Box 1263, SE-164 29 Kista, Sweden

{Magnus.Agren, Mats.Carlsson}@sics.se

² École des Mines de Nantes, LINA UMR CNRS 6241, FR-44307 Nantes, France

{Nicolas.Beldiceanu, Mohamed.Sbihi, Stephane.Zampelli}@emn.fr

³ Université de Nantes, LINA UMR CNRS 6241, FR-44322 Nantes, France

Charlotte.Truchet@univ-nantes.fr

Abstract. This paper introduces six ways for handling a *chain of lexicographic ordering (lex-chain)* constraint between the origins of identical orthotopes (e.g., rectangles, boxes, hyper-rectangles) subject to the fact that they should not pairwise overlap. While the first two ways deal with the integration of a *lex-chain* constraint within a generic geometric constraint kernel, the four latter ways deal with the conjunction of a *lex-chain* constraint and a *non-overlapping* or a *cumulative* constraint. Experiments on academic two and three dimensional placement problems as well as on industrial problems show the benefit of such a strong integration of symmetry breaking constraints and non-overlapping ones.

1 Introduction

Symmetry constraints among identical objects are ubiquitous in industrial placement problems that involve packing a restricted number of types of orthotopes (generalized rectangles) subject to *non-overlapping* constraints.

In this context, an orthotope corresponds to the generalization of a rectangle in the k -dimensional case. An *orthotope* is defined by the coordinates of its smallest corner and by its potential orientations. An *orientation* is defined by k integers that give the size of the orthotope in the different dimensions. Two orthotopes are said to be *identical* if and only if their respective orientation sizes form identical multisets. In the rest of this paper, we assume that we pack each orthotope in such a way that its borders are parallel to the boundaries of the placement space.

In the context of Operations Research, breaking symmetries has been handled by characterizing and taking advantage of equivalence and dominance relations between patterns of fixed objects [1]. In the context of Constraint Programming, a natural way to break symmetries is to enforce a lexicographic ordering on the origin coordinates of identical orthotopes. This can be directly done by using a *lex-chain* constraint such as the one introduced in [2]. Even if this drastically reduces the number of solutions, it does not allow much pruning and/or speedup when we are looking for one single solution. This stems from the fact that symmetry is handled independently from non-overlapping. The question addressed by this paper is how to directly integrate a *lex-chain* constraint within a *non-overlapping* constraint and how it pays off in practice.

Section 2 recalls the context of this work, namely the generic geometric constraint kernel and its core algorithm, a multi-dimensional sweep algorithm, which performs filtering introduced in [3]. Since this algorithm will be used in the rest of the paper, Section 2 also recalls the principle of the filtering algorithm behind a *lex-chain* constraint. Section 3 describes two ways of directly handling symmetries in the multi-dimensional sweep algorithm, while Section 4 shows how to derive bounds on the coordinates of an orthotope from the interaction of symmetries and *non-overlapping* constraints. Since the *cumulative* constraint is a necessary condition for the *non-overlapping* constraint [4], Section 5 shows how to directly integrate symmetries within two well known filtering algorithms attached to the *cumulative* constraint. Section 6 evaluates the different proposed methods both on academic and industrial benchmarks and Section 7 concludes the paper.

2 Context

This work is in the context of the global constraint $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ introduced in [3], which handles the location in space of k -dimensional orthotopes \mathcal{O} ($k \in \mathbb{N}^+$), each of which taking an orientation among a set of possible orientations \mathcal{S} , subject to geometrical constraints \mathcal{C} .¹ Each possible orientation from \mathcal{S} is defined as a box in a k -dimensional space with the given sizes. More precisely, a *possible orientation* $s \in \mathcal{S}$ is an entity defined by its orientation id $s.sid$, and sizes $s.l[d]$ (where $s.l[d] > 0$ and $0 \leq d < k$). All attributes of a possible orientation are integer values. Each object $o \in \mathcal{O}$ is an entity defined by its unique object id $o.oid$ (an integer), possible orientation id $o.sid$ (an integer for *monomorphic* objects, which have a fixed orientation, or a domain variable² for *polymorphic* objects, which have alternative orientations), and origin $o.x[d]$, $0 \leq d < k$ (integers, or domain variables).

Since the most common geometrical constraint is the *non-overlapping* constraint between orthotopes, this paper focuses on breaking symmetries in this context (i.e., each shape is defined by one single box). For this purpose, we impose a *lex-chain* constraint on the origins of identical orthotopes. Given two vectors, x and y of k variables, $\langle x_0, x_1, \dots, x_{k-1} \rangle \leq_{lex} \langle y_0, y_1, \dots, y_{k-1} \rangle$ if and only if $k = 0 \vee (x_0 < y_0) \vee (x_0 = y_0 \wedge \langle x_1, \dots, x_{k-1} \rangle \leq_{lex} \langle y_1, \dots, y_{k-1} \rangle)$. Unless stated otherwise, the constraint is imposed wrt. the k dimensions $0, 1, \dots, k-1$. The original filtering algorithm of the *lex-chain* constraint described in [2] is a two phase algorithm. In a first phase, it computes for each vector of the chain feasible lexicographic lower and upper bounds. In a second phase, a specific algorithm [5] filters the components of each vector of the chain according to the fact that it has to be located between two fixed vectors.

3 Integrating Symmetries within the Sweep Kernel

This section first recalls the principle of the sweep point algorithm attached to *geost*. It then indicates how to modify it in order to take advantage of the fact that we have

¹ In the context of this paper we have simplified the presentation of *geost*.

² A *domain variable* v is a variable ranging over finite set of integers denoted by $\text{dom}(v)$; \underline{v} and \overline{v} denote respectively the minimum and maximum possible values of v .

a restricted number of types of orthotopes. Without loss of generality, it assumes that we have one *non-overlapping* constraint over all orthotopes of *geost* and one *lex-chain* constraint for each set of identical orthotopes.

3.1 Description of the Original Sweep Algorithm

The use of sweep algorithms in constraint filtering algorithms was introduced in [6] and applied to the non-overlapping 2D rectangles constraints. Let a *forbidden region* f be an orthotope of values for $o.x$ that would falsify the *geost* constraint, represented as a fixed lower bound vector $f.min$ and a fixed upper bound vector $f.max$. Algorithm 1, $\text{PruneMin}(o, d, k)$, searches for the first point c , by lexicographic order wrt. dimensions $d, (d+1) \bmod k, \dots, (d-1) \bmod k$, that is inside the domain of $o.x$ but not inside any forbidden region. If such a c exists, the algorithm sets $o.x[d]$ to $c[d]$, otherwise it fails. Two state vectors are maintained: the *sweep point* c , which holds a candidate value for $o.x$, and the *jump vector* n , which records knowledge about encountered forbidden regions.

The algorithm starts its recursive traversal of the placement space at point $c = o.x$ with $n = \overline{o.x} + 1$ and could in principle explore all points of the domains of $o.x$, one by one, in increasing lexicographic order wrt. dimensions $d, (d+1) \bmod k, \dots, (d-1) \bmod k$, until the first desired point is found. To make the search efficient, it skips points that are known to be inside some forbidden region. This knowledge is encoded in n , which is updated for every new f (see line 5) recording the fact that new candidate points can be found beyond that value. Whenever we skip to the next candidate point, we reset the elements of n that were used to their original values (see lines 6–15).

3.2 Enhancing the Original Sweep Kernel wrt. Identical Shapes

In the context of multiple occurrences of identical orthotopes, we can enhance the sweep algorithm attached to *geost* by trying to reuse the information computed so far from one orthotope to another orthotope. For this purpose we introduce the notion of *domination* of an orthotope by another orthotope.

Given a $\text{geost}(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ constraint where \mathcal{C} consists of one *non-overlapping* constraint between all orthotopes of \mathcal{O} and a *lex-chain* constraint between each set of identical orthotopes, an orthotope $o_j \in \mathcal{O}$ is *dominated* by another orthotope $o_i \in \mathcal{O}$ if and only if the following conditions hold:

1. $\text{dom}(o_j.x[p]) \subseteq \text{dom}(o_i.x[p]), \forall p \in [0, k-1]$,
2. $\text{dom}(o_j.sid) \subseteq \text{dom}(o_i.sid)$,
3. the origin of o_j should be lexicographically greater than or equal to the origin of o_i .

Now, for one invocation of the sweep algorithm, which performs a recursive traversal of the placement space, we can make the following observation. If an orthotope o_j is dominated by another orthotope o_i and if we have already called the sweep algorithm for updating the minimum value of $o_i.x[p]$ ($p \in [0, k-1]$), we can take advantage of the information obtained while computing the minimum of $o_i.x[p]$. Let c_{ip} and n_{ip} respectively denote the final values of vectors c and n after running $\text{PruneMin}(o_i, p, k)$. Note

```

PROCEDURE PruneMin( $o, d, k$ ) : bool
1:  $c \leftarrow \underline{o.x}$  // initial position of the point
2:  $n \leftarrow \overline{o.x} + 1$  // upper limits+1 in the different dimensions
3:  $f \leftarrow \text{GetFR}(o, c, k)$  // check if  $c$  is infeasible
4: while  $f \neq \perp$  do
5:    $n \leftarrow \min(n, f.\text{max} + 1)$  // maintain  $n$  as min of u.b. of forbidden regions
6:   for  $j \leftarrow k - 1$  downto 0 do
7:      $j' \leftarrow (j + d) \bmod k$  // least significant dimension first
8:      $c[j'] \leftarrow n[j']$  // use  $n[j']$  to jump
9:      $n[j'] \leftarrow \overline{o.x}[j'] + 1$  // reset  $n[j']$  to max
10:    if  $c[j'] \leq \overline{o.x}[j']$  then
11:      goto next // candidate point found
12:    else
13:       $c[j'] \leftarrow \underline{o.x}[j']$  // exhausted a dimension, reset  $c[j']$ 
14:    end if
15:  end for
16:  return false // no next candidate point
17:  next:  $f \leftarrow \text{GetFR}(o, c, k)$  // check again if  $c$  is infeasible
18: end while
19:  $\underline{o.x}[d] \leftarrow c[d]$  // adjust earliest start in dim.  $d$ 
20: return true

```

Algorithm 1: Adjusting the lower bound $\underline{o.x}[d]$. $\text{GetFR}(o, c, k)$ scans a list of forbidden regions, starting at the latest encountered one, returns \perp if c is in the domain of $\underline{o.x}$ and not inside any forbidden region f , and $f \neq \perp$ otherwise.

that while computing the minimum of $\underline{o_j.x}[p]$, instead of starting the recursive traversal of the placement space from $c = \underline{o_j.x}$ with $n = \overline{o_j.x} + 1$, we can start from the position c_{ip} and with the jump vector n_{ip} . By using this observation, we decrease the number of jumps needed for filtering the bounds of the coordinates of n identical orthotopes from $k \cdot n^2$ to $k \cdot n$. Finally note that for one invocation of the sweep algorithm, forbidden regions for the origins of identical orthotopes need only be computed once. This observation is valid even if we don't have any lexicographic ordering constraints and is crucial for scalability in the context of identical orthotopes.

3.3 Integrating a Chain of Lexicographic Ordering Constraint within the Sweep Kernel

The main interest of the sweep algorithm attached to *geost* is to aggregate the set of forbidden points coming from different geometric constraints. In our context, these are the *non-overlapping* and *lex-chain* constraints. As a concrete example, consider the following problem:

Example 1. We have to place within a placement space of size 6×5 three squares s_1, s_2, s_3 of size 2×2 so that their respective origin coordinates $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ are lexicographically ordered in increasing order. Moreover, assume that the first and third squares are fixed so that $(x_1, y_1) = (2, 3)$ and $(x_3, y_3) = (5, 2)$, and that $(x_2, y_2) \in ([1, 5], [1, 4])$. If we don't consider together the *non-overlapping* and the *lex-chain* constraint we can only restrict the domain

of x_2 to interval $[2, 5]$. But, as shown in Figure 1, if we aggregate the forbidden points coming from the *lex-chain* and *non-overlapping* constraints, we can further restrict the domain of x_2 to interval $[3, 4]$. \square

So the question is how to generate forbidden regions for a *lex-chain* constraint of the form $\langle l_0, l_1, \dots, l_{k-1} \rangle \leq_{\text{lex}} \langle x_0, x_1, \dots, x_{k-1} \rangle \leq_{\text{lex}} \langle u_0, u_1, \dots, u_{k-1} \rangle$ where l_i , x_i and u_i respectively correspond to integers, domain variables and integers.³ Let us first illustrate what forbidden regions we want to obtain in the context of Example 1.

Continuation of Example 1. Consider the constraint $\langle 2, 4 \rangle \leq_{\text{lex}} \langle x_2, y_2 \rangle \leq_{\text{lex}} \langle 5, 1 \rangle$. We can associate to this *lex-chain* constraint the following forbidden regions; see the crosses in Part (B) of Figure 1:

- Since $x_2 < 2$ is not possible, we have $f. \min = [1, 1], f. \max = [1, 5]$ (column 1);
- Since $x_2 = 2 \wedge y_2 < 4$ is not possible, we have $f. \min = [2, 1], f. \max = [2, 3]$ (column 2);
- Since $x_2 > 5$ is not possible, we have $f. \min = [6, 1], f. \max = [6, 5]$ (column 6);
- Since $x_2 = 5 \wedge y_2 > 1$ is not possible, we have $f. \min = [5, 2], f. \max = [5, 5]$ (column 5).

\square

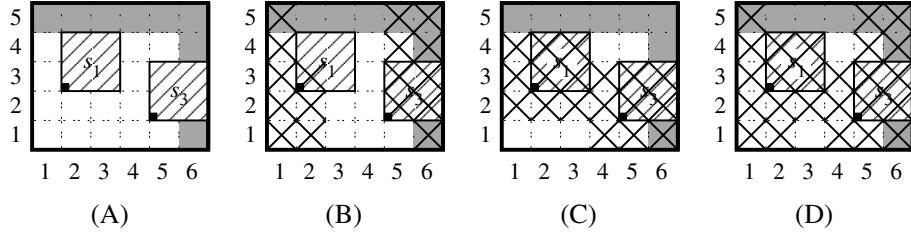


Fig. 1. (A) The two fixed squares s_1 and s_3 (gray squares are not possible for the origin of s_2 since it has to be included within the placement space depicted by a thick line); (B) Forbidden points (a cross) wrt. the *lex-chain* constraint; (C) Forbidden points (a cross) wrt. the *non-overlapping* constraint; (D) Aggregating all forbidden points: (3, 1) and (4, 4) are the only feasible points for the origin of s_2 , which leads to restricting x_2 to interval $[3, 4]$.

We show in Algorithm 2 how to generate such forbidden regions in a systematic way. As in Example 1, lines 1–6 generate for the lower bound constraint a forbidden region according to the fact that the most significant components x_0, x_1, \dots, x_{i-1} of vector x are respectively fixed to l_0, l_1, \dots, l_{i-1} ($i \in [0, k - 1]$). Similarly, lines 7–12 generate k forbidden regions wrt. the upper bound u .

³ As mentioned in Section 2, propagating a *lex-chain* constraint leads to generating such sub-problems.

```

PROCEDURE LexBetweenGenForbiddenReg( $k, x, l, u$ ) :  $f[0..2 \cdot k - 1]$ 
1:      // GENERATE FORBIDDEN REGIONS WITH RESPECT TO LOWER BOUND  $l$ 
2:  for  $i \leftarrow 0$  to  $k - 1$  do
3:     $\forall j \in [0, i) : f[i].\text{min}[j] \leftarrow l_j; f[i].\text{max}[j] \leftarrow l_j$ 
4:     $f[i].\text{min}[i] \leftarrow \underline{x_i}; f[i].\text{max}[i] \leftarrow l_i - 1$ 
5:     $\forall j \in [i + 1, k) : f[i].\text{min}[j] \leftarrow \underline{x_j}; f[i].\text{max}[j] \leftarrow \overline{x_j}$ 
6:  end for
7:      // GENERATE FORBIDDEN REGIONS WITH RESPECT TO UPPER BOUND  $u$ 
8:  for  $i \leftarrow 0$  to  $k - 1$  do
9:     $\forall j \in [0, i) : f[k + i].\text{min}[j] \leftarrow u_j; f[k + i].\text{max}[j] \leftarrow u_j$ 
10:    $f[k + i].\text{min}[i] \leftarrow u_i + 1; f[k + i].\text{max}[i] \leftarrow \overline{x_i}$ 
11:    $\forall j \in [i + 1, k) : f[k + i].\text{min}[j] \leftarrow \underline{x_j}; f[k + i].\text{max}[j] \leftarrow \overline{x_j}$ 
12: end for
13: return  $f$ 

```

Algorithm 2: Generates the $2 \cdot k$ forbidden regions wrt. variables x_0, x_1, \dots, x_{k-1} associated with the constraint $\langle l_0, l_1, \dots, l_{k-1} \rangle \leq_{\text{lex}} \langle x_0, x_1, \dots, x_{k-1} \rangle \leq_{\text{lex}} \langle u_0, u_1, \dots, u_{k-1} \rangle$.

4 Integrating Symmetries within the Non-Overlapping Constraint

We just saw how to aggregate forbidden regions coming from a *lex-chain* and a set of *non-overlapping* constraints. This section shows how to combine these two types of constraints more intimately in order to perform more deduction.

4.1 Deriving Bounds from the Interaction of the Chain of Lexicographic Ordering and Non-Overlapping Constraints: the Monomorphic Case

We first consider the case of n orthotopes $\{o_j \mid 0 \leq j < n\}$ corresponding to a given fixed orientation s subject to *non-overlapping* as well as *lex-chain*.⁴ In this context, we provide a lower bound low_j and an upper bound up_j for the origin of each orthotope, wrt. both constraints. Let $S[i]$ denote the size of the placement space in dimension i ($0 \leq i < k$). Furthermore, let us denote by $O[0..k - 1]$ and $P[0..k - 1]$ the points respectively defined by $O[i] = \min(\underline{o_0.x[i]}, \underline{o_1.x[i]}, \dots, \underline{o_{n-1}.x[i]})$ and by $P[i] = \max(\overline{o_0.x[i]}, \overline{o_1.x[i]}, \dots, \overline{o_{n-1}.x[i]}) + s.l[i]$. We have $low_j \leq_{\text{lex}} o_j.x \leq_{\text{lex}} up_j$, $0 \leq j < n$, where:

$$low_j[i] = O[i] + \left\lfloor \frac{j \bmod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor} \right\rfloor \cdot s.l[i] \quad (0 \leq i < k) \quad (1)$$

$$up_j[i] = P[i] - \left\lfloor \frac{(n - 1 - j) \bmod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor} \right\rfloor \cdot s.l[i] - s.l[i] \quad (0 \leq i < k) \quad (2)$$

⁴ In practice, this occurs in placement problems involving *several* occurrences of a given orthotope with the *same fixed orientation*.

The intuition behind formula (1)⁵ in order to find the lower bound of the j^{th} object in dimension i is:

- First, fill complete slices wrt. dimensions $i, i + 1, \dots, k - 1$ (such a complete slice involves $\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor$ objects),
- Then, with the remaining objects to place (i.e., $j \bmod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor)$ objects), compute the number of complete slices wrt. dimensions $i + 1, i + 2, \dots, k - 1$ (i.e., $\left\lfloor \frac{j \bmod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor} \right\rfloor$ slices) and multiply this number by the length of a slice (i.e., $s.l[i]$).

4.2 Deriving Bounds from the Interaction of the Chain of Lexicographic Ordering and Non-Overlapping Constraints: the Polymorphic Case

We now consider the case of n identical orthotopes $\{o_j \mid 0 \leq j < n\}$, again subject to *non-overlapping* as well as *lex-chain*. In this context, we provide three incomparable lower and upper bounds for the origin of each object. The first bound is based on the bound previously introduced. It simply consists in reducing the box sizes to their smallest values. For the second and third bounds, instead of reducing the sizes of a box to its smallest size, we decompose a box into n_ℓ smaller identical boxes that all have the same size ℓ in the different dimensions.⁶ Assume that we want to find the lower bound for box o_j . The idea is to saturate the placement space with $n_\ell \cdot (j + 1)$ boxes by considering the least significant dimension first and by starting at the lower left corner of the placement space. Then we subtract from the last end corner the different sizes of o_j in decreasing order (i.e., for the most significant dimension we subtract the largest size).⁷ In the context of an upper bound, the idea is to saturate the placement space with $n_\ell \cdot (n - j) - 1$ small boxes by considering the least significant dimension first and by starting at the upper right corner of the placement space. Then we subtract ℓ from the last end corner of the $(n_\ell \cdot (n - j))^{th}$ smallest box. Based on the preceding formulas we obtain the following bounds. Without loss of generality, we assume that $s.l$ are sorted in decreasing order. A box can be decomposed into $n_\ell = \prod_{d=0}^{k-1} \lfloor \frac{s.l[d]}{\ell} \rfloor$ cubes of size ℓ with possibly some loss. We have⁸

$$low_j[i] = O[i] + \left\lfloor \frac{((j + 1) \cdot n_\ell - 1) \bmod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{\ell} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{\ell} \rfloor} \right\rfloor \cdot \ell + \ell - s.l[i] \quad (3)$$

$$up_j[i] = P[i] - \left\lfloor \frac{((n - j) \cdot n_\ell - 1) \bmod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{\ell} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{\ell} \rfloor} \right\rfloor \cdot \ell - \ell \quad (4)$$

⁵ Formula (2) is obtained in a similar way. The proof is available in [7].

⁶ ℓ takes its value between 1 and the smallest size of the box we consider (i.e., $1 \leq \ell \leq \min\{s.l[i] \mid 0 \leq i < k\}$).

⁷ In Figures 2 and 3, diagonal lines depict this subtraction.

⁸ The proof is available in [7].

In practice it is not clear which value of ℓ provides the best bound. Therefore, we currently restrict ourselves to the values $s.minl$ and $\gcd(s.l[0], s.l[1], \dots, s.l[k-1])$. The bounds obtained with these two values are incomparable. Figures 2 and 3 respectively illustrate this second bound for placing a set of 5 rectangles for which the orientation sizes form the multiset $\{\{3, 4\}\}$ within a big rectangle of size 10×9 with $\ell = \min(4, 3)$ and $\ell = \gcd(4, 3)$.

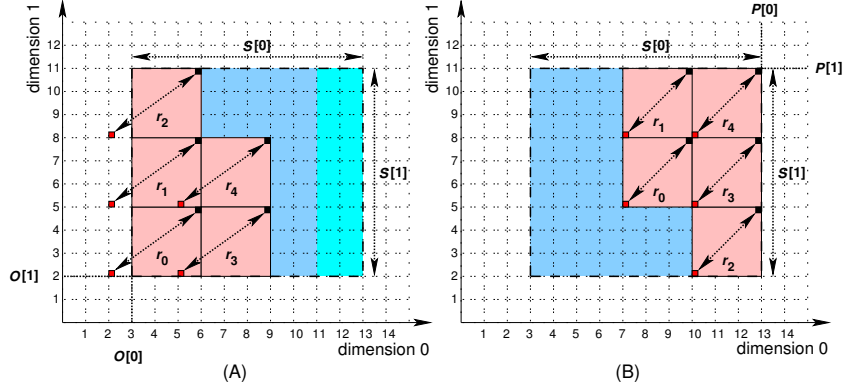


Fig. 2. Computing the lower (A) and upper (B) bounds of a set of rectangles for the second bound with $\ell = \min(4, 3)$ for the polymorphic case.

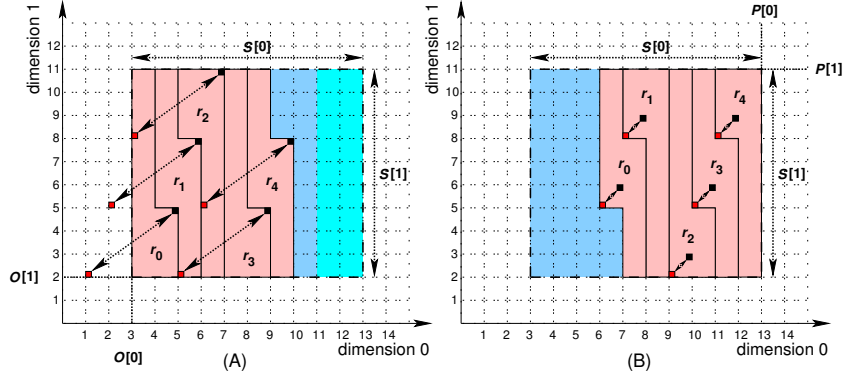


Fig. 3. Computing the lower (A) and upper (B) bounds of a set of rectangles for the second bound with $\ell = \gcd(4, 3)$ for the polymorphic case.

5 Integrating Symmetries within the Cumulative Constraint

We have already shown how to combine a *lex-chain* and a *non-overlapping* constraint. But, in the context of a *non-overlapping* constraint, the *cumulative* constraint is a well known necessary condition [4]. This section shows how to directly integrate the fact that we have a *lex-chain* constraint within two well known filtering algorithms of the *cumulative* constraint: filtering wrt. the *compulsory part profile* [8] and filtering wrt. *task intervals* [9].

5.1 Handling Symmetries in the Context of the Compulsory Part Profile

Let us first recall the notion of compulsory part profile, which will be used throughout this section. In the context of the *cumulative* constraint, the *compulsory part* of a task t corresponds to the intersection of all feasible schedules of t . As the domain of the start of task t gets more and more restricted the compulsory part of t will increase until becoming a schedule of task t . The compulsory part of a task t can be directly computed by making the intersection between the earliest start and the latest end of task t . The *compulsory part profile* associated with the tasks \mathcal{T} of a *cumulative* constraint is the cumulated profile of all compulsory parts of tasks of \mathcal{T} .

In the context of *non-overlapping* constraints, many search strategies [10] try to first fix the coordinates of all objects in a given dimension d before fixing all the coordinates in the other dimensions.⁹ But now, if we don't take care of the interaction between the *cumulative* and *lex-chain* constraints, we can have a huge compulsory part profile which will be totally ignored by the *lex-chain* constraint. The following illustrative example will make things clear.

Example 2. Assume that we have to place 8 squares of size 2×2 within the bounding box $[0, 9] \times [0, 3]$ (i.e., in the context of *cumulative*, 0 and $9 + 1$ respectively correspond to the earliest start and the latest end, while 4 is the resource limit). In addition, assume that the compulsory part profile in the most significant (wrt. \leq_{lex}) dimension of the placement space corresponds to the following 3 consecutive intervals $[0, 3]$, $[4, 5]$ and $[6, 9]$ of respective heights 0, 2 and 0.¹⁰ If there is no interaction between this *cumulative* constraint and the lexicographic ordering constraint that states that the eight 2×2 squares should be lexicographically ordered, then we get the following domain reductions: The earliest start of the first two squares of the lexicographic ordering is 0, the earliest start of the third and fourth squares is 2, the earliest start of the fifth and sixth squares is 4, and the earliest start of the last two squares is 6. This is obviously an underestimation since, because of the compulsory part profile of the *cumulative* constraint, we can start at most one single square at instant 4. \square

In the context of a *cumulative* constraint, we now show how to estimate the earliest start in the most significant dimension (msd) of each orthotope of a *lex-chain* constraint according to an existing compulsory part profile.¹¹ To each orthotope o corresponds a

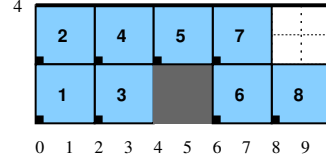
⁹ In the benchmarks presented in Section 6, this is the case e.g. for the heuristic used for the monomorphic Partridge problem.

¹⁰ The compulsory part corresponding to interval $[4, 5]$ does not correspond to the 8 squares to place, for it comes from another fixed object.

¹¹ The same idea can be used for estimating the latest end in the msd.

task t for which the origin, the duration and the height are respectively the coordinate of o in the msd, the size of o in the msd, and the product of the sizes of o in the dimensions different from the msd. Now, the idea is to simply consider the orthotopes in increasing lexicographic order and to find out for each corresponding task its earliest possible start on the msd. The following condition is checked for testing whether a start is feasible or not: When added to the cumulative profile, the maximum height should not exceed the resource limit.¹²

By reconsidering Example 2, this idea is illustrated on the right hand side, estimating the minimum value of the coordinates in the msd of eight squares of size 2. The squares are successively placed at their earliest possible start according to the compulsory part profile. Consequently, the minimum values of the coordinates in the most significant dimension of squares 1, 2, ..., 8 equal respectively 0, 0, 2, 2, 4, 6, 6 and 8 (and not to 0, 0, 2, 2, 4, 4, 6 and 6 as before).



5.2 Handling Symmetries in the Context of Task Intervals

In the context of the *cumulative* constraint, *task interval methods* prevent the overuse as well as the underuse of intervals derived from the earliest start and the latest end of the tasks to schedule. This section focuses on the problem of pruning the origin of the tasks of the *cumulative* constraint so that we don't lose too much space within a given fixed interval according to the fact that we have an ordering on the origin of identical tasks.¹³ For this purpose, consider the set of all identical tasks T of duration d and height h , an interval $[\inf, \sup)$ and the height *gap* of free space on top of the interval, and the slack σ of the interval (i.e., the maximum allowed unused space of the interval). For a given set of tasks S , let $overlap(S)$ denote the sum of the maximum overlap of the tasks in S . To find out whether or not $t \in T$ must intersect $[\inf, \sup)$, the task intervals pruning rule makes the test:

$$(\sup - \inf) \cdot gap - (overlap(T) - overlap(\{t\})) > \sigma \quad (5)$$

If this test succeeds, we know that t must overlap the free space of $[\inf, \sup)$ to some extent. Specifically, t must then overlap the free space of $[\inf, \sup)$ at least by

$$(\sup - \inf) \cdot gap - (overlap(T) - overlap(\{t\})) - \sigma$$

which means that t must intersect in time $[\inf, \sup)$ at least by:

$$\left\lceil \frac{(\sup - \inf) \cdot gap - (overlap(T) - overlap(\{t\})) - \sigma}{d} \right\rceil$$

¹² The resource limit equals the product of the sizes of the placement space in the dimensions different from the msd.

¹³ Such an ordering exists for the *cumulative* constraint associated with the msd of the lexicographic ordering constraint.

This can be strengthened in the presence of symmetries. Assume a partial order \preceq over the start times of the tasks T implied by a *lex-chain* constraint. Assume moreover that $t_i \neq t_j \in T$ are tasks such that $t_i \preceq t_j$. Then the positionings of t_i and t_j wrt. interval $[\inf, \sup)$ are in fact not independent:

- if t_j is assumed to end strictly before the interval $[\inf, \sup)$, then t_i must also be assumed to end strictly before $[\inf, \sup)$; and
- if t_i is assumed to start strictly after the interval $[\inf, \sup)$, then t_j must also be assumed to start strictly after $[\inf, \sup)$.

Considering now the chain $t_1 \preceq \dots \preceq t_n$ and assuming that t is the i^{th} task t_i of this chain, we split the pruning rule above into two cases: the first case corresponding to the tasks t_1, \dots, t_{i-1} not succeeding t_i ; and the second case corresponding to the tasks t_{i+1}, \dots, t_n not preceding t_i .

For the first case, since each of the tasks t_1, \dots, t_{i-1} must not succeed t_i , assuming that t_i ends before $[\inf, \sup)$ implies that the tasks t_1, \dots, t_{i-1} must also end before $[\inf, \sup)$. Hence, the test (5) can be strengthened to:

$$(\sup - \inf) \cdot \text{gap} - (\text{overlap}(T) - \text{overlap}(\{t_1, \dots, t_i\})) > \sigma \quad (6)$$

If this test succeeds, we know that all the tasks t_1, \dots, t_i must overlap the free space of $[\inf, \sup)$ at least by:

$$(\sup - \inf) \cdot \text{gap} - (\text{overlap}(T) - \text{overlap}(\{t_1, \dots, t_i\})) - \sigma \quad (7)$$

Now, since we wish to prune t_i , this must be translated into how far into $[\inf, \sup)$ we must force t_i so that the remaining tasks may overlap the free space of $[\inf, \sup)$ enough. This can be calculated in two steps as follows:

- STEP 1: Calculate the largest number d_{fill} of columns of maximum height and width d , covering part of but not more than the free space of $[\inf, \sup)$.
- STEP 2: Calculate the smallest number $unit_{fill}$ of columns of maximum height and width 1, covering the remaining free space of $[\inf, \sup)$.

We use $tofill$ to denote the value (7). STEP 1 can be calculated by:

$$\alpha \leftarrow \min \left(\left\lfloor \frac{gap}{h} \right\rfloor, i \right) \quad [\text{largest number of stacked tasks}]$$

$$\beta \leftarrow \left\lfloor \frac{tofill}{\alpha \cdot h} \right\rfloor \quad [\text{largest number of unit-size columns}]$$

$$d_{fill} \leftarrow \left\lfloor \frac{\beta}{d} \right\rfloor \quad [\text{largest number of d-size columns}]$$

Given this, the remaining free space of $[\inf, \sup)$ is:

$$rest_{fill} = tofill - d_{fill} \cdot \alpha \cdot d \cdot h$$

When $rest_{fill} > 0$, STEP 2 can then be calculated by:

$\gamma \leftarrow \min(i - dfill \cdot \alpha, \alpha)$ [largest number of stacked tasks still available]

$unitfill \leftarrow \left\lceil \frac{restfill}{h \cdot \gamma} \right\rceil$ [smallest number of unit-size columns]

Now, given the values $dfill$ and $unitfill$, to overlap the free space of $[\inf, \sup)$ by at least the value (7), the start time of t_i must be at least $\inf + (dfill - 1) \cdot t_i.d + unitfill$. An example of this method can be found in [7].

6 Performance Evaluation

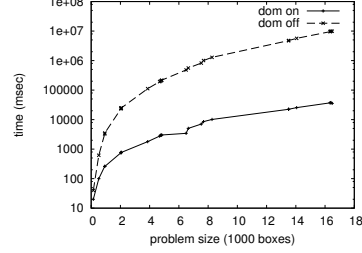
All the new filtering methods described in this paper were integrated into our *geost* kernel [3] in order to strengthen the sweep-based filtering for non-overlapping constraints. The experiments were run in SICStus Prolog 4 compiled with gcc version 4.1.0 on a 3GHz Pentium IV with 1MB of cache.

We ran two benchmarks, *Scale* and *KLS*, seeking to evaluate the performance gain of domination in *greedy* execution mode, where the constraint tries to assign all variables in a single run, and simply fails if it cannot. Note that this greedy mode fits well inside a tree search based procedure: at every node of the search tree, a greedy step can be attempted in order to solve the problem in one shot, and if it fails, a normal propagation and branching step can be done. Three benchmarks, *Conway*, *Partridge* and *Pallet* were run in normal propagation mode, under tree search. The symmetry that stems from multiple pieces of the same shape is broken by imposing a lexicographic order on their origins. The purpose here was to compare the performance of treating these lexicographic ordering constraints inside *non-overlapping* and *cumulative* as opposed to posting them separately. Since this is not a paper on heuristics, the exact models and search procedures are probably of little interest, and are only given in the corresponding code of the benchmarks in Appendix B of [7]. We now describe the five benchmarks and the results, which are shown in Table 1.

Scale. As in [3], we constructed a set of loosely constrained placement problems (i.e., 20% spare space), generating one set of random problem instances of $m \in \{2^{10}, 2^{11}, \dots, 2^{22}\}$ 2D items involving $t \in \{1, 16, 256, 1024\}$ distinct shapes. The results indicate that domination brings the time complexity down from roughly $O(m^2)$ to virtually $O(m)$. The results also show that the speedup gained by domination goes down as the number of distinct shapes goes up. In the larger instances, the total number of items vastly outnumbers the number of distinct shapes. With domination, we could now pack 2^{22} 2D items of 1024 distinct shapes (over 8 million domain variables) in four CPU minutes, an improvement by more than two orders of magnitude over [3].

KLS. To evaluate the greedy mode in a setting involving side-constraints in addition to non-overlapping, we studied the problem of packing a given number of 3D items

into containers, with the objective to minimize the number of containers required. The containers all have the same size and weight capacity, whereas the items come in 59 different shapes and weights. The items cannot overlap and must be fully inside some container. The total weight of the items inside a given container must not exceed the weight capacity. Also, some items must be placed on the container floor, whereas other items cannot be placed underneath any other item. The whole problem can be modeled as a single 6D *geost* constraint. We ran 25 instances of different size; see the figure on the right hand side. The largest instance, with 16486 items, was solved in 35 seconds with domination and 1284 seconds without.



Conway. The problem consists in placing 6 pieces of shape $4 \times 2 \times 1$, 6 pieces of shape $3 \times 2 \times 2$ and 5 unit cubes within a $5 \times 5 \times 5$ cube. All pieces can be rotated freely.

Partridge. The problem consists in tiling a square of size $\frac{n \cdot (n+1)}{2}$ by 1 square of size 1, 2 squares of size 2, ..., n squares of size n . It was initially proposed by R. Wainwright.¹⁴ We tried the instances $n = 8, \dots, n = 12$. Note that, to our best knowledge, this is the first reported solution for $n = 12$. We also tried a polymorphic variant of the problem: tile a rectangle of size 21×63 by 1 rectangle of size 1×3 , 2 rectangles of size 2×6 , ..., 6 rectangles of size 6×18 , where all rectangles can be rotated.

Pallet. The problem consists in placing a given number of identical, non-overlapping, rectangular pieces of a given size onto a rectangular pallet, also of a given size. We selected several instances from D. Lobato's data sets¹⁵ and ran two variants of each instance: (i) a polymorphic variant, with 90 degrees rotation allowed, and (ii) a monomorphic variant with the number of horizontal vs. vertical pieces fixed.

Evaluation of methods. We ran the last three benchmarks in versions where only one given method at a time was switched on. For reasons of space we cannot present the full results; instead, we summarize the findings. First, we found that for monomorphic benchmarks, integration of symmetries into *cumulative* was more effective than integration into *non-overlapping*, but for polymorphic benchmarks, it was the other way around. Finally, integration into *non-overlapping* had the highest runtime overhead among the methods. Integration into task intervals was the least effective among the methods, but had a very low overhead.

7 Conclusion

For the first time, symmetry breaking has been fully integrated into the filtering algorithms of global constraints. This was done in two contexts:

¹⁴ See <http://mathpuzzle.com/partridge.html>.

¹⁵ See <http://lagrange.ime.usp.br/~lobato/packing/>.

- (a) Real-life placement problems tend to involve many more objects to place than distinct shapes. They can be too large to solve solely with constructive search. The ability to perform a greedy assignment, possibly with a limited amount of search, staying inside a constraint programming framework, can be crucial to solving such problems. By using the fact that many objects are of the same shape, we showed that the complexity of such a greedy assignment in the context of a sweep algorithm can go down from $O(n^2)$ to virtually $O(n)$ for n objects.
- (b) We identified and exploited four ways of handling symmetry breaking *lex-chain* constraints inside a *non-overlapping* or *cumulative* constraint. Our results show that the tight integration saves search effort but not necessarily CPU time: slowdown up to 2 times, but also sometimes speedup up to 2.5 times, was observed.

Finally, we found the first reported solution to **partridge(12,1)**.

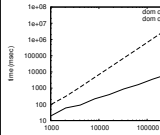
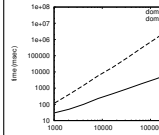
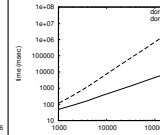
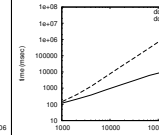
Acknowledgements

This research was conducted under European Union Sixth Framework Programme Contract FP6-034691 “Net-WMS”. In this context thanks to A. Aggoun from KLS OPTIM (<http://www.klsoptim.com/>) for providing relevant industrial benchmarks.

References

1. G. Scheithauer. Equivalence and dominance for problems of optimal packing of rectangles. *Ricerca Operativa*, 27(83):3–34, 1998.
2. M. Carlsson and N. Beldiceanu. Arc-consistency for a *chain of lexicographic ordering* constraints. Technical Report T2002-18, Swedish Institute of Computer Science, 2002.
3. N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k -dimensional objects. In C. Bessière, editor, *Principles and Practice of Constraint Programming (CP'2007)*, volume 4741 of *LNCS*, pages 180–194. Springer-Verlag, 2007.
4. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.
5. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-08, Swedish Institute of Computer Science, 2005. See the *lex_between* constraint at http://www.emn.fr/x-info/sdemasse/gccat/Clex_between.html.
6. N. Beldiceanu and M. Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraints. In T. Walsh, editor, *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 377–391. Springer-Verlag, 2001.
7. M. Ågren, N. Beldiceanu, M. Carlsson, M. Sbihi, C. Truchet, and S. Zampelli. Six ways of integrating symmetries within non-overlapping constraints. SICS Technical Report T2009:01, Swedish Institute of Computer Science, 2009.
8. A. Lahrachi. Scheduling: the notions of hump, compulsory parts and their use in cumulative problems. *C.R. Acad. Sci., Paris*, 294:209–211, February 1982.
9. Y. Caseau and F. Laburthe. Cumulative scheduling with task intervals. In *Joint International Conference and Symposium on Logic Programming (JICSLP'96)*. MIT Press, 1996.
10. H. Simonis and B. O’Sullivan. Search strategies for rectangle packing. In P. J. Stuckey, editor, *Principles and Practice of Constraint Programming (CP'2008)*, volume 5202 of *LNCS*, pages 52–66. Springer-Verlag, 2008.

m	$t = 1$		$t = 16$		$t = 256$		$t = 1024$	
	dom on	dom off	dom on	dom off	dom on	dom off	dom on	dom off
1024	20	100	30	120	50	120	120	150
2048	60	310	50	410	90	370	210	400
4096	90	1160	100	1480	170	1270	380	1320
8192	220	4640	230	5780	360	5030	780	5170
16384	400	18060	450	19010	710	19990	1550	20270
32768	890	71210	910	73230	1410	77340	3050	77200
65536	1650	279480	1880	300540	2920	296650	6100	299510
131072	3590	1118410	3760	1177900	5910	1188740	10280	1186030
262144	7020	4488510	7980	4812300	12020	4758390	25280	4746410
524288	17100	22671540	18000	23210070	29210	23553550	58910	23512450

			
---	---	--	---

		backtracks		runtime	
		lex in	lex out	lex in	lex out
conway(5,5,5)		6658	10192	11890	12850
partridge(8,1)		565	853	6400	3460
partridge(9,1)		27714	63429	347100	367050
partridge(10,1)		683643	1265284	15160080	9154320
partridge(11,1)		80832	189797	2009150	1964130
partridge(12,1)		790109	1676827	37850240	24203920
partridge(6,3)		7122	20459	13680	29610

	monomorphic				polymorphic			
	backtracks		runtime		backtracks		runtime	
	lex in	lex out	lex in	lex out	lex in	lex out	lex in	lex out
pallet(26,19,5,2,49,30)	0	0	130	110	8	8	180	90
pallet(28,17,5,2,47,25)	184	325	570	320	398	433	660	360
pallet(29,20,4,3,48,28)	664	1419	1890	1300	9767	14457	22500	14870
pallet(30,17,4,3,42,18)	778	1580	2380	1290	19807	28015	28190	20130
pallet(30,19,7,2,40,24)	74	115	190	140	19	81	150	90
pallet(31,19,7,2,41,24)	20544	73695	34190	57840	728743	932846	666010	506730
pallet(32,17,7,2,38,20)	491	850	630	660	159	172	310	140
pallet(33,17,7,2,39,20)	8129	26644	13300	26030	390539	567304	366320	286930
pallet(33,19,7,2,44,30)	3556	34778	9690	23450	789894	1460451	689080	743530
pallet(33,22,5,3,48,24)	41	54	220	160	65	73	290	140
pallet(34,17,5,3,38,24)	0	268	90	170	425	900	390	380
pallet(36,34,7,4,43,25)	14030	28855	25830	16800	33874	41648	66520	42220
pallet(37,19,7,2,49,33)	96	136	240	160	113	215	260	170
pallet(38,26,5,4,49,29)	6141	12830	14880	10910	39486	52787	75450	46530

Table 1. Top: Scale for 2D items, with domination on and off. **Center:** Results for *Conway* and *Partridge*. **Bottom left:** An instance $\text{pallet}(x, y, a, b, n, h)$ denotes the task of packing h pieces of shape $a \times b$ and $n - h$ pieces of shape $b \times a$ into a placement space of shape $x \times y$. **Bottom right:** Polymorphic variants of the same instances, where the parameter h has been left free. *lex-chain* constraints are treated inside *geost* in columns marked **lex in** and posted separately in columns marked **lex out**. All runtimes (ms) and backtrack counts are for finding the first solution.