Net-WMS

SPECIFIC TARGETED RESEARCH OR INNOVATION PROJECT

Networked Businesses

D4.1 – A catalogue of generic placement constraint in a WMS

Due date of deliverable: 31-08-2007

Actual submission date: 17-10-2007

Start date of project: 1 September 2006 Duration: 36 months

Organisation name of lead contractor for this deliverable: EMN

Project co-funded by the European Commission within the Sixth Framework Programme (2006-2009)

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	**Net-WMS**
Project Full Name:	Towards integrating Virtual Reality and optimisation techniques in a new generation of **Net**worked businesses in **W**arehouse **M**anagement **S**ystems under constraints
Document id:	D4.1
Document name:	A catalogue of generic placement constraints in a WMS
Document type (PU, INT, RE, CO)	PU
Version:	1
Submission date:	17-10-2007
Authors: **Organisation:** **Email:**	Nicolas Beldiceanu (EMN) Nicolas.Beldiceanu@emn.fr

Document type PU = public, INT = internal, RE = restricted, CO = confidential

ABSTRACT:

This deliverable is a description regarding the work on the catalogue of generic placement constraints in a WMS.

KEYWORD LIST:

Net-WMS, WMS, constraint, catalogue.

MODIFICATION CONTROL			
Version	Date	Status	Author
1	10-17-2007	Final	Nicolas Beldiceanu (EMN)

Deliverable manager

- Nicolas Beldiceanu (ARMINES)

List of Contributors

- Nicolas Beldiceanu (ARMINES)
- Mats Carlsson (SICS)
- Julien Martin (INRIA)
- Abder, Aggoun, KLS

List of Evaluators

- François Fages (INRIA)
- Abder, Aggoun (KLS OPTIM)
- Philippe Rohou (ERCIM)

1 Table of Contents

2 Summary

Within the existing catalogue of global constraints (see http://www.emn.fr/x-info/sdemasse/gccat/index.html) we have identified the already existing global constraints that are relevant within the NetWMS project. These constraints correspond to geometrical constraints that can be used for expressing non-overlapping constraints (like *diffn*), necessary conditions for non-overlapping (*cumulative*, *cumulatives*), symmetry breaking constraints (*lex_chain_less*), load balancing constraints (*cumulative_two_d*) or placement constraints (*place_in_pyramid*).

Based on collaboration with partners of the projects (KLS, INRIA, SICS) we have introduced within the catalogue of global constraints a restricted set of very general packing constraints like the *diffst* and the *visible* constraints. We have illustrated the possible uses of these two global constraints regarding different type of packing problems as well as regarding pallet loading and pick up delivery problems. In addition to these generic constraints we have introduced positioning constraints like *contains_sboxes*, *coveredby_sboxes*, *covers_sboxes*, *disjoint_sboxes*, *inside_sboxes*, *meet_sboxes*, *non_overlap_sboxes* and *overlap_sboxes*.

For further information look at the corresponding entries of the global constraint catalogue.
The relevant constraints are added in an annex of this document. A published paper at CP2007 about the main non-overlapping constraint is also given as a second annex of this document.

4.64 contains_sboxes

DESCRIPTION **LINKS**

Origin Geometry, derived from [227]

Constraint contains_sboxes(K, DIMS, OBJECTS, SBOXES)

Synonym(s) contains.

Type(s)
VARIABLES : collection(v−dvar)
INTEGERS : collection(v−int)
NATURALS : collection(v−int)

Argument(s)

K	:	int
DIMS	:	sint
OBJECTS	:	collection(oid−int, sid−int, x − VARIABLES)
SBOXES	:	collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)
required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if, for each pair of objects (O_i, O_j), $i < j$, O_i contains O_j with respect to a set of dimensions depicted by DIMS. O_i and O_j are objects that take a shape among a set of shapes. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a K-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id sid, shift offset t, and sizes l. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier oid, shape id sid and origin x.
An object O_i *contains* an object O_j with respect to a set of dimensions depicted by DIMS if and only if, for all shifted boxes s_j associated with O_j, there exists a shifted box s_i of O_i such that s_i contains s_j. A shifted box s_i *contains* a shifted box s_j if and only if, for all dimensions $d \in$ DIMS, (1) the start of s_i in dimension d is strictly less than the start of s_j in dimension d and (2)the end of s_j in dimension d is strictly less than the end of s_i in dimension d.

Example

$$\left\langle \begin{array}{l} 2, \{0,1\}, \\ \left\langle \begin{array}{lll} \texttt{oid}-1 & \texttt{sid}-1 & \texttt{x}-\langle 1,1\rangle, \\ \texttt{oid}-2 & \texttt{sid}-2 & \texttt{x}-\langle 2,2\rangle, \\ \texttt{oid}-3 & \texttt{sid}-3 & \texttt{x}-\langle 3,3\rangle \end{array} \right\rangle, \\ \left\langle \begin{array}{lll} \texttt{sid}-1 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 5,5\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,3\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 1,1\rangle \end{array} \right\rangle \end{array} \right\rangle$$

Figure 4.133 shows the objects of the example. Since O_1 contains both O_2 and O_3, and since O_2 contains O_3, the `contains_sboxes` constraint holds.

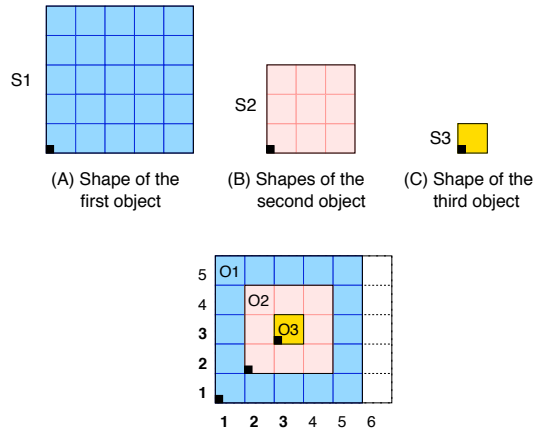


Figure 4.133: The three objects of the example

Remark One of the eight relations of the *Region Connection Calculus* [227]. The constraint `contains_sboxes` is a restriction of the original relation since it requires that each shifted box of an object is contained by one shifted box of the other object.

See also `coveredby_sboxes`, `covers_sboxes`, `disjoint_sboxes`, `equal_sboxes`, `inside_sboxes`, `meet_sboxes`, `non_overlap_sboxes`, `overlap_sboxes`.

Key words **constraint type:** predefined constraint.

geometry: geometrical constraint, rcc8.

4.68 coveredby_sboxes

DESCRIPTION **LINKS**

Origin Geometry, derived from [227]

Constraint coveredby_sboxes(K, DIMS, OBJECTS, SBOXES)

Synonym(s) coveredby.

Type(s)

VARIABLES	:	collection(v−dvar)
INTEGERS	:	collection(v−int)
NATURALS	:	collection(v−int)

Argument(s)

K	:	int
DIMS	:	sint
OBJECTS	:	collection(oid−int, sid−int, x − VARIABLES)
SBOXES	:	collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)

required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if, for each pair of objects (O_i, O_j), $i < j$, O_i is covered by O_j with respect to a set of dimensions depicted by `DIMS`. O_i and O_j are objects that take a shape among a set of shapes. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a `K`-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id `sid`, shift offset `t`, and sizes `l`. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier `oid`, shape id `sid` and origin `x`.
An object O_i *is covered by* an object O_j with respect to a set of dimensions depicted by `DIMS` if and only if, for all shifted box s_i of O_i, there exists a shifted box s_j of O_j such that:

- For all dimensions $d \in$ `DIMS`, (1) the start of s_j in dimension d is less than or equal to the start of s_i in dimension d, and (2) the end of s_i in dimension d is less than or equal to the end of s_j in dimension d.
- There exists a dimension d where, (1) the start of s_j in dimension d coincide with the start of s_i in dimension d, or (2) the end of s_j in dimension d coincide with the end of s_i in dimension d.

Example

$$\left(\begin{array}{l} 2, \{0,1\}, \\ \left\langle \begin{array}{llll} \texttt{oid}-1 & \texttt{sid}-4 & \texttt{x}-\langle 1,1\rangle, \\ \texttt{oid}-2 & \texttt{sid}-2 & \texttt{x}-\langle 2,2\rangle, \\ \texttt{oid}-3 & \texttt{sid}-1 & \texttt{x}-\langle 2,3\rangle \end{array} \right\rangle, \\ \left\langle \begin{array}{lll} \texttt{sid}-1 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,3\rangle, \\ \texttt{sid}-1 & \texttt{t}-\langle 3,0\rangle & \texttt{l}-\langle 2,2\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 2,2\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 2,0\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 2,2\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 2,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-4 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 1,1\rangle \end{array} \right\rangle \end{array}\right)$$

Figure 4.142 shows the objects of the example. Since O_1 is covered by both O_2 and O_3, and since O_2 is covered by O_3, the `coveredby_sboxes` constraint holds.

Remark

One of the eight relations of the *Region Connection Calculus* [227]. The constraint `coveredby_sboxes` is a restriction of the original relation since it requires that each shifted box of an object is covered by one shifted box of the other object.

See also

`contains_sboxes`, `covers_sboxes`, `disjoint_sboxes`, `equal_sboxes`, `inside_sboxes`, `meet_sboxes`, `non_overlap_sboxes`, `overlap_sboxes`.

Key words

constraint type: predefined constraint.

geometry: geometrical constraint, rcc8.

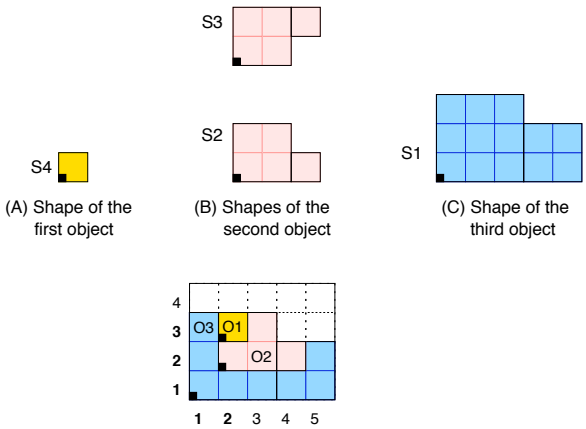


Figure 4.142: The three objects of the example

4.69 covers_sboxes

DESCRIPTION **LINKS**

Origin Geometry, derived from [227]

Constraint covers_sboxes(K, DIMS, OBJECTS, SBOXES)

Synonym(s) covers.

Type(s)

VARIABLES	:	collection(v−dvar)
INTEGERS	:	collection(v−int)
NATURALS	:	collection(v−int)

Argument(s)

K	:	int
DIMS	:	sint
OBJECTS	:	collection(oid−int, sid−int, x − VARIABLES)
SBOXES	:	collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)

required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if, for each pair of objects (O_i, O_j), $i < j$, O_i covers O_j with respect to a set of dimensions depicted by DIMS. O_i and O_j are objects that take a shape among a set of shapes. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a K-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id sid, shift offset t, and sizes l. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier oid, shape id sid and origin x.

An object O_i *covers* an object O_j with respect to a set of dimensions depicted by DIMS if and only if, for all shifted box s_j of O_j, there exists a shifted box s_i of O_i such that:

- For all dimensions $d \in$ DIMS, (1) the start of s_i in dimension d is less than or equal to the start of s_j in dimension d, and (2) the end of s_j in dimension d is less than or equal to the end of s_i in dimension d.
- There exists a dimension d where, (1) the start of s_i in dimension d coincide with the start of s_j in dimension d, or (2) the end of s_i in dimension d coincide with the end of s_j in dimension d.

Example

$$\left(\begin{array}{l} 2, \{0,1\}, \\ \left\langle \begin{array}{llll} \mathtt{oid}-1 & \mathtt{sid}-1 & \mathtt{x}-\langle 1,1\rangle, \\ \mathtt{oid}-2 & \mathtt{sid}-2 & \mathtt{x}-\langle 2,2\rangle, \\ \mathtt{oid}-3 & \mathtt{sid}-4 & \mathtt{x}-\langle 2,3\rangle \end{array} \right\rangle, \\ \left\langle \begin{array}{lll} \mathtt{sid}-1 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 3,3\rangle, \\ \mathtt{sid}-1 & \mathtt{t}-\langle 3,0\rangle & \mathtt{l}-\langle 2,2\rangle, \\ \mathtt{sid}-2 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 2,2\rangle, \\ \mathtt{sid}-2 & \mathtt{t}-\langle 2,0\rangle & \mathtt{l}-\langle 1,1\rangle, \\ \mathtt{sid}-3 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 2,2\rangle, \\ \mathtt{sid}-3 & \mathtt{t}-\langle 2,1\rangle & \mathtt{l}-\langle 1,1\rangle, \\ \mathtt{sid}-4 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 1,1\rangle \end{array} \right\rangle \end{array}\right)$$

Figure 4.143 shows the objects of the example. Since O_1 covers both O_2 and O_3, and since O_2 covers O_3, the covers_sboxes constraint holds.

Remark One of the eight relations of the *Region Connection Calculus* [227]. The constraint covers_sboxes is a relaxation of the original relation since it requires that each shifted box of an object is covered by one shifted box of the other object.

See also contains_sboxes, coveredby_sboxes, disjoint_sboxes, equal_sboxes, inside_sboxes, meet_sboxes, non_overlap_sboxes, overlap_sboxes.

Key words **constraint type:** predefined constraint.

geometry: geometrical constraint, rcc8.

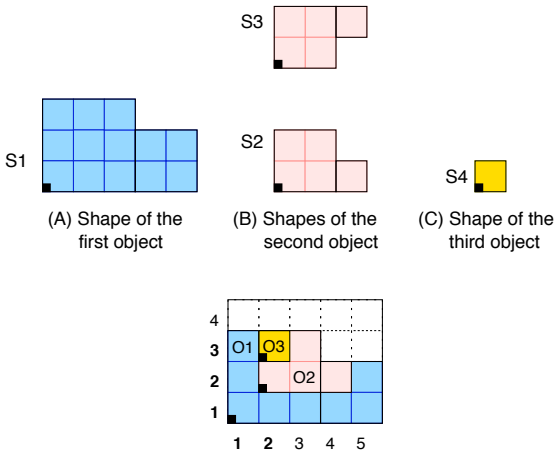


Figure 4.143: The three objects of the example

4.74 cumulative_two_d

DESCRIPTION **LINKS**

Origin Inspired by `cumulative` and `diffn`.

Constraint `cumulative_two_d(RECTANGLES, LIMIT)`

Argument(s)

RECTANGLES	:	collection	(start1−dvar, size1−dvar, last1−dvar, start2−dvar, size2−dvar, last2−dvar, height−dvar)
LIMIT	:	int	

Restriction(s)

`require_at_least(2, RECTANGLES, [start1, size1, last1])`
`require_at_least(2, RECTANGLES, [start2, size2, last2])`
`required(RECTANGLES, height)`
RECTANGLES.size1 ≥ 0
RECTANGLES.size2 ≥ 0
RECTANGLES.height ≥ 0
LIMIT ≥ 0

Purpose Consider a set $\mathcal{R}$ of rectangles described by the `RECTANGLES` collection. Enforces that at each point of the plane, the cumulated height of the set of rectangles that overlap that point, does not exceed a given limit.

Example

$$\left(\left\langle \begin{array}{lllllll} \texttt{start1}-1 & \texttt{size1}-4 & \texttt{last1}-4 & \texttt{start2}-3 & \texttt{size2}-3 & \texttt{last2}-5 & \texttt{height}-4, \\ \texttt{start1}-3 & \texttt{size1}-2 & \texttt{last1}-4 & \texttt{start2}-1 & \texttt{size2}-2 & \texttt{last2}-2 & \texttt{height}-2, \\ \texttt{start1}-1 & \texttt{size1}-2 & \texttt{last1}-2 & \texttt{start2}-1 & \texttt{size2}-2 & \texttt{last2}-2 & \texttt{height}-3, \\ \texttt{start1}-4 & \texttt{size1}-1 & \texttt{last1}-4 & \texttt{start2}-1 & \texttt{size2}-1 & \texttt{last2}-1 & \texttt{height}-1 \end{array} \right\rangle, 4 \right)$$

Part (A) of Figure 4.154 shows the 4 parallelepipeds of height 4, 2, 3 and 1 associated with the items of the `RECTANGLES` collection (parallelepipeds since each rectangle has also a height). Part (B) gives the corresponding cumulated 2-dimensional profile, where each number is the cumulated height of all the rectangles that contain the corresponding region. The `cumulative_two_d` constraint holds since the heighest peak of the cumulated 2-dimensional profile does not exceed the upper limit 4 imposed by the last argument of the `cumulative_two_d` constraint.

Usage The `cumulative_two_d` constraint is a necessary condition for the `diffn` constraint in 3 dimensions (i.e., the placement of parallelepipeds in such a way that they do not pairwise overlap and that each parallelepiped has his sides parallel to the sides of the placement space).

Algorithm A first natural way to handle this constraint would be to accumulate the compulsory part [171] of the different rectangles in a quadtree [250]. To each leave of the quadtree we associate the cumulated height of the rectangles containing the corresponding region.

See also `cumulative`, `diffn`, `bin_packing`.

Key words **characteristic of a constraint:** derived collection.
constraint type: predefined constraint.
filtering: quadtree, compulsory part.
geometry: geometrical constraint.

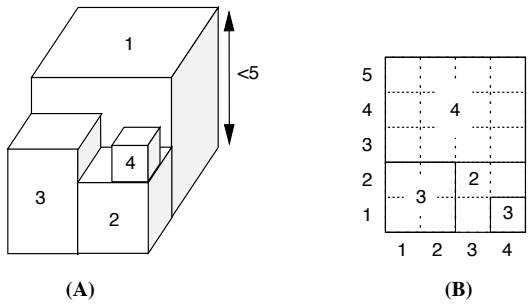


Figure 4.154: Two representations of a 2-dimensional cumulated profile

4.71 cumulative

DESCRIPTION **LINKS** **GRAPH** **AUTOMATON**

Origin [1]

Constraint cumulative(TASKS, LIMIT)

Argument(s)

TASKS	:	collection(origin−dvar, duration−dvar, end−dvar, height−dvar)
LIMIT	:	int

Restriction(s)

require_at_least(2, TASKS, [origin, duration, end])
required(TASKS, height)
TASKS.duration ≥ 0
TASKS.origin $\leq$ TASKS.end
TASKS.height ≥ 0
LIMIT ≥ 0

Purpose

Cumulative scheduling constraint or scheduling under resource constraints. Consider a set $\mathcal{T}$ of tasks described by the TASKS collection. The cumulative constraint enforces that at each point in time, the cumulated height of the set of tasks that overlap that point, does not exceed a given limit. It also imposes for each task of $\mathcal{T}$ the constraint origin + duration = end.

Example

$$\left\langle \begin{array}{cccc} \texttt{origin}-1 & \texttt{duration}-3 & \texttt{end}-4 & \texttt{height}-1, \\ \texttt{origin}-2 & \texttt{duration}-9 & \texttt{end}-11 & \texttt{height}-2, \\ \texttt{origin}-3 & \texttt{duration}-10 & \texttt{end}-13 & \texttt{height}-1, \\ \texttt{origin}-6 & \texttt{duration}-6 & \texttt{end}-12 & \texttt{height}-1, \\ \texttt{origin}-7 & \texttt{duration}-2 & \texttt{end}-9 & \texttt{height}-3 \end{array} \right\rangle, 8$$

Figure 4.146 shows the cumulated profile associated with the example. To each task of the cumulative constraint corresponds a set of rectangles coloured with the same colour: the sum of the lengths of the rectangles corresponds to the duration of the task, while the height of the rectangles (i.e., all the rectangles associated with a task have the same height) corresponds to the resource consumption of the task. The cumulative constraint holds since at each point in time we don't have a cumulated resource consumption strictly greater than the upper limit 8 enforced by the last argument of the cumulative constraint.

Algorithm [171, 100, 69, 182]. Within the context of linear programming, the reference [149] provides a relaxation of the cumulative constraint.

A necessary condition for the cumulative constraint is obtained by stating a disjunctive constraint on a subset of tasks $\mathcal{T}$ such that, for each pair of tasks of $\mathcal{T}$, the sum of the two corresponding minimum heights is strictly greater than LIMIT. This can be done by applying the following procedure:

- Let h be the smallest minimum height strictly greater than $\lfloor \frac{\texttt{LIMIT}}{2} \rfloor$ of the tasks of the cumulative constraint. If no such task exists then the procedure is stopped without stating any disjunctive constraint.

- Let $\mathcal{T}_h$ denotes the set of tasks of the `cumulative` constraint for which the minimum height is greater than or equal to h. By construction, the tasks of $\mathcal{T}_h$ cannot overlap. But we can eventually add one more task as shown by the next step.
- When it exists, we can add one task that does not belong to $\mathcal{T}_h$ and such that its minimum height is strictly greater than `LIMIT` $- h$. Again, by construction, this task cannot overlap all the tasks of $\mathcal{T}_h$.

When the tasks are involved in several `cumulative` constraints more sophisticated methods are available for extracting `disjunctive` constraints [10, 9].

See also `disjunctive`, `diffn`, `bin_packing`, `cumulative_product`, `coloured_cumulative`, `cumulative_two_d`, `coloured_cumulatives`, `cumulatives`, `cumulative_with_level_of_priority`, `cumulative_convex`, `calendar`.

Key words **characteristic of a constraint:** core, automaton, automaton with array of counters.

complexity: sequencing with release times and deadlines.

constraint type: scheduling constraint, resource constraint, temporal constraint.

filtering: linear programming, compulsory part.

problems: producer-consumer.

puzzles: squared squares.

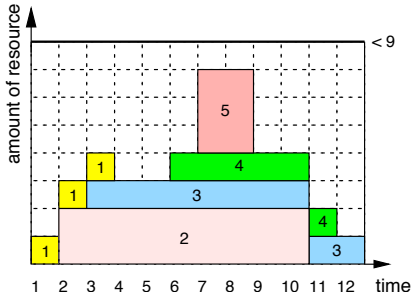


Figure 4.146: Resource consumption profile

Arc input(s) TASKS

Arc generator $SELF \mapsto$ collection(tasks)

Arc arity 1

Arc constraint(s) tasks.origin + tasks.duration = tasks.end

Graph property(ies) $\mathbf{NARC} = |$TASKS$|$

Arc input(s) TASKS TASKS

Arc generator $PRODUCT \mapsto$ collection(tasks1, tasks2)

Arc arity 2

Arc constraint(s)
- tasks1.duration > 0
- tasks2.origin $\leq$ tasks1.origin
- tasks1.origin $<$ tasks2.end

Graph class
- ACYCLIC
- BIPARTITE
- NO_LOOP

Sets

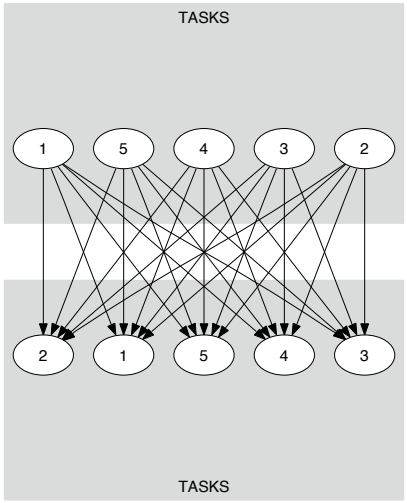
$$\text{SUCC} \mapsto \left[\begin{array}{l} \text{source}, \\ \text{variables} - \text{col}\left(\begin{array}{l} \text{VARIABLES}-\text{collection}(\text{var}-\text{dvar}), \\ [\text{item}(\text{var} - \text{TASKS.height})] \end{array} \right) \end{array} \right]$$

Constraint(s) on sets sum_ctr(variables, $\leq$, LIMIT)

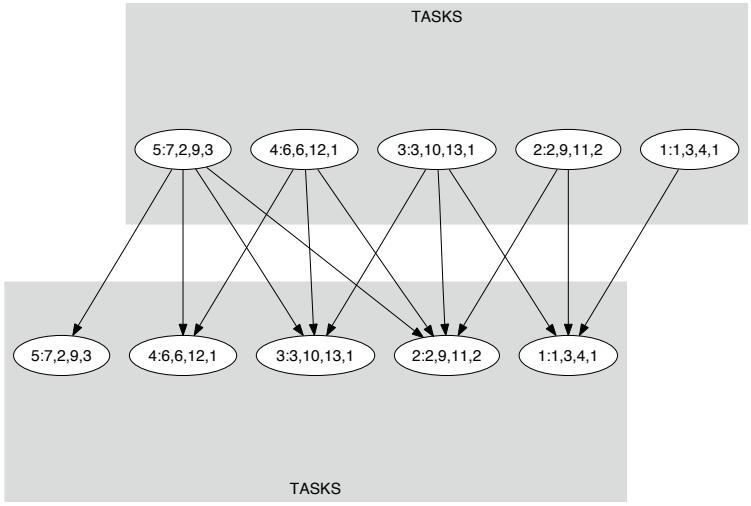
Graph model The first graph constraint enforces for each task the link between its origin, its duration and its end. The second graph constraint makes sure, for each time point t corresponding to the start of a task, that the cumulated heights of the tasks that overlap t does not exceed the limit of the resource.

Parts (A) and (B) of Figure 4.147 respectively show the initial and final graph associated with the second graph constraint of the **Example** slot. On the one hand, each source vertex of the final graph can be interpreted as a time point. On the other hand the successors of a source vertex correspond to those tasks that overlap that time point. The cumulative constraint holds since for each successor set $\mathcal{S}$ of the final graph the sum of the heights of the tasks in $\mathcal{S}$ does not exceed the limit LIMIT $= 8$.

Signature Since TASKS is the maximum number of vertices of the final graph of the first graph constraint we can rewrite $\mathbf{NARC} = |$TASKS$|$ to $\mathbf{NARC} \geq |$TASKS$|$. This leads to simplify $\underline{\overline{\mathbf{NARC}}}$ to $\overline{\mathbf{NARC}}$.



(A)



(B)

Figure 4.147: Initial and final graph of the `cumulative` constraint

Automaton

Figure 4.148 depicts the automaton associated with the `cumulative` constraint. To each item of the collection `TASKS` corresponds a signature variable $\mathtt{S}_i$ that is equal to 1.

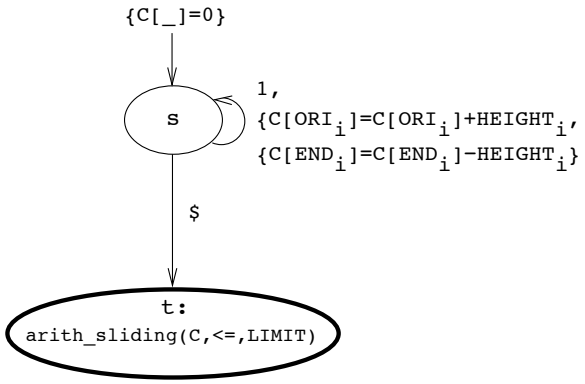


Figure 4.148: Automaton of the `cumulative` constraint

4.89 diffn

DESCRIPTION **LINKS** **GRAPH**

Origin [27]

Constraint diffn(ORTHOTOPES)

Synonym(s) disjoint1, disjoint2.

Type(s) ORTHOTOPE : collection(ori−dvar, siz−dvar, end−dvar)

Argument(s) ORTHOTOPES : collection(orth − ORTHOTOPE)

Restriction(s)
|ORTHOTOPE| > 0
require_at_least(2, ORTHOTOPE, [ori, siz, end])
ORTHOTOPE.siz ≥ 0
ORTHOTOPE.ori $\leq$ ORTHOTOPE.end
required(ORTHOTOPES, orth)
same_size(ORTHOTOPES, orth)

Purpose Generalised multi-dimensional non-overlapping constraint: Holds if, for each pair of orthotopes (O_1, O_2), O_1 and O_2 do not overlap. Two orthotopes do not overlap if there exists at least one dimension where their projections do not overlap.

Example
$$\left\langle \begin{array}{l} \texttt{orth} - \langle \texttt{ori} - 2\ \texttt{siz} - 2\ \texttt{end} - 4, \texttt{ori} - 1\ \texttt{siz} - 3\ \texttt{end} - 4\rangle, \\ \texttt{orth} - \langle \texttt{ori} - 4\ \texttt{siz} - 4\ \texttt{end} - 8, \texttt{ori} - 3\ \texttt{siz} - 3\ \texttt{end} - 6\rangle, \\ \texttt{orth} - \langle \texttt{ori} - 9\ \texttt{siz} - 2\ \texttt{end} - 11, \texttt{ori} - 4\ \texttt{siz} - 3\ \texttt{end} - 7\rangle \end{array} \right\rangle$$

Figure 4.183 represents the respective position of the three rectangles of the example. The co-ordinates of the leftmost lowest corner of each rectangle are stressed in bold. The diffn constraint holds since the three rectangles do not overlap.

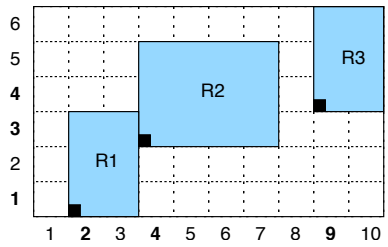


Figure 4.183: The three rectangles of the example

Usage The diffn constraint occurs in placement and scheduling problems. It was for instance used for scheduling problems where one has to both assign each non-preemptive task to a resource and fix its origin so that two tasks, which are assigned to the same resource, do not overlap. A practical application from the area of the design of memory-dominated embedded systems [272] can be found in [273]. Together with arithmetic and cumulative

constraints, the `diffn` constraint was used in [271] for packing more complex shapes such as angles. Figure 4.184 illustrates the angle packing problem on an instance involving 10 angles taken from [271].

One other packing problem attributed to S. Golomb is to find the smallest square that can contain the set of consecutive squares from 1×1 up to $n \times n$ so that these squares do not overlap each other. A program using the `diffn` constraint was used to construct such a table for $n \in \{1, 2, \ldots, 25, 27, 29, 30\}$ in [20]. Algorithms and lower bounds for solving the same problem can also be respectively found in [165] and in [60]. In that paper, Richard E. Korf also considers the problem of finding the minimum-area rectangle that can contain the set of consecutive squares from 1×1 up to $n \times n$.

Remark

When we have segments (respectively rectangles) the `diffn` constraint is referenced under the name `disjoint1` (respectively `disjoint2`) in SICStus Prolog [67].

It was shown in [275, page 137] that, finding out whether a non-overlapping constraint between a set of rectangles has a solution or not is NP-hard. This was achieved by reduction from sequencing with release times and deadlines.

Algorithm

Checking whether a `diffn` constraint for which all variables are fixed is satisfied or not is related to the Klee's measure problem: given a collection of axis-aligned multi-dimensional boxes, how quickly can one compute the volume of their union. Then the `diffn` constraint holds if the volume of the union is equal to the sum of the volumes of the different boxes.

A first possible method for filtering is to use constructive disjunction. The idea is to try out each alternative of a disjunction (e.g., given two orthotopes o_1 and o_2 that should not overlap, we successively assume for each dimension that o_1 finishes before o_2, and that o_2 finishes before o_1) and to remove values that were pruned in all alternatives. For the two-dimensional case of `diffn` a second possible solution used in [247] is to represent explicitly the two-dimensional domain of the origin of each rectangle by a quadtree [250] and to accumulate all forbidden regions within this data structure. As for conventional domain variables, a failure occurs when a two-dimensional domain get empty. A third possible filtering algorithm based on sweep is described in [22].

The thesis of J. Nelissen [199] considers the case where all rectangles have the same size and can be rotated from 90 degrees (i.e., the pallet loading problem.). For the n-dimensional case of `diffn` a filtering algorithm handling the fact that two objects do not overlap is given in [30].

Extensions of the non-overlapping constraint to polygons and to more complex shapes are respectively described in [30] and in [243]. Specialised propagation algorithms for the squared squares problem [59] (based on the fact that no waste is permitted) are given in [123] and in [122].

The `cumulative` constraint can be used as a necessary condition for the `diffn` constraint. Figure 4.186 illustrates this point for the two-dimensional case. A first (respectively second) `cumulative` constraint is obtained by forgetting the y-co-ordinate (respectively the x-co-ordinate) of the origin of each rectangle occurring in a `diffn` constraint. Parts (B) and (C) respectively depict the cumulated profiles associated with the projection of the rectangles depicted by part (A) on the x and y axes. The `cumulative` constraint is a necessary but not sufficient condition for the two-dimensional case of the `diffn` constraint. Figure 4.187 illustrates this point on an example taken from [52] where one has to place the 8 rectangles R1, R2, R3, R4, R5, R6, R7, R8 of respective size 5×2, 8×2, 6×1,

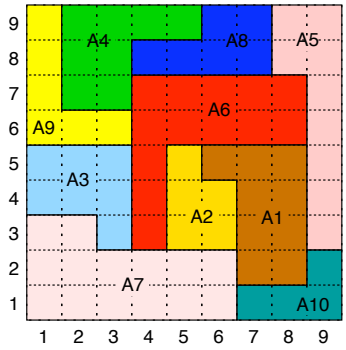


Figure 4.184: A solution for the angle packing problem of items A1 $= [2, 4, 3, 1]$, A2 $= [2, 2, 1, 3]$, A3 $= [1, 3, 3, 2]$, A4 $= [2, 1, 4, 3]$, A5 $= [1, 7, 2, 2]$, A6 $= [1, 2, 5, 5]$, A7 $= [6, 2, 2, 3]$, A8 $= [4, 2, 2, 1]$, A9 $= [3, 1, 1, 4]$, A10 $= [3, 2, 1, 1]$.

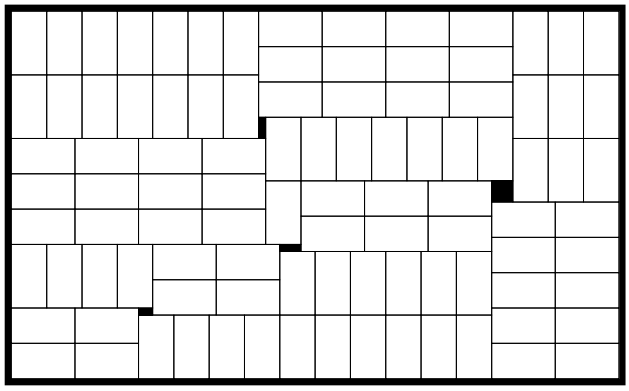


Figure 4.185: A hard instance from [199, page 165]: A solution for packing 99 rectangles of size 5×9 into a rectangle of size 86×52

5×1, 2×1, 3×1, 2×2 and 1×2 in a big rectangle of size 12×4. As shown by Figure 4.187 there is a cumulative solution where R8 is splitted in two parts but M. Hujter proves in [151] that their is no solution where no rectangle is splitted.

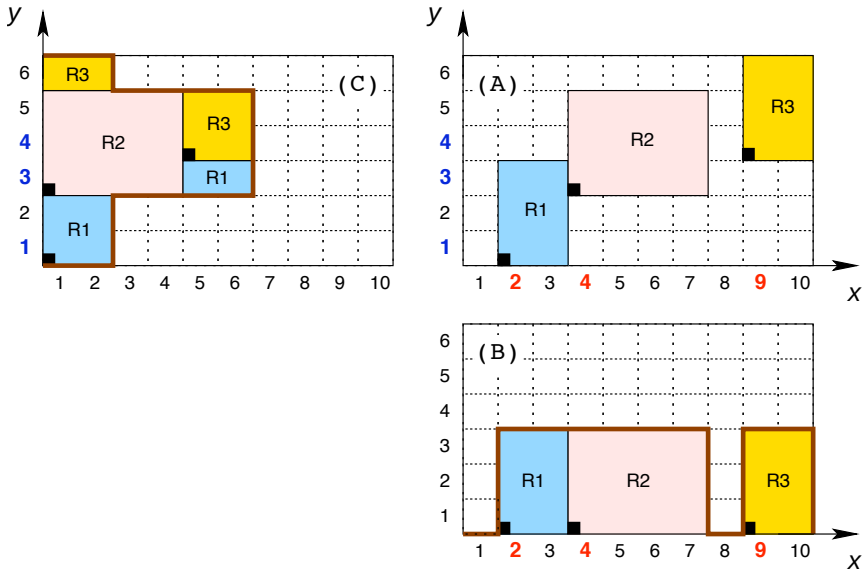


Figure 4.186: Looking from the perspective of the `cumulative` constraint in a two-dimensional rectangles placement problem

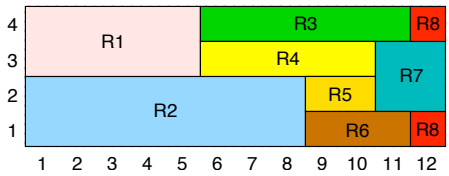


Figure 4.187: Illustrating the necessary but not sufficient placement condition

In the context of n parallelepipeds that have to be packed [127, 179] within a box of sizes $X \times Y \times Z$ one can proceed as follows for stating three `cumulative` constraints. The i^{th} ($i \in [1, n]$) parallelepiped is described by the following attributes:

- ox_i, oy_i, oz_i ($i \in [1, n]$) the co-ordinates of its origin on the x, y and z-axes.
- sx_i, sy_i, sz_i ($i \in [1, n]$) its sizes on the x, y and z-axes.
- px_i, py_i, pz_i ($i \in [1, n]$) the surfaces of its projections on the planes yz, xz, and xy respectively equal to $sy_i sz_i$, $sx_i sz_i$, and $sx_i sy_i$.
- v_i its volume (equal to $sx_i sy_i sz_i$).

For the placement of n parallelepipeds we get the following necessary conditions that respectively correspond to three `cumulative` constraints on the planes yz, xz, and xy:

$$\begin{cases} \forall i \in [1, X] : \sum_{j|ox_j \leq i \leq ox_j + sx_j - 1} px_j \leq YZ \\ \forall i \in [1, Y] : \sum_{j|oy_j \leq i \leq oy_j + sy_j - 1} py_j \leq XZ \\ \forall i \in [1, Z] : \sum_{j|oz_j \leq i \leq oz_j + sz_j - 1} pz_j \leq XY \end{cases}$$

Used in `diffn_column`, `diffn_include`, `place_in_pyramid`.

See also `diffst`, `cumulative`, `orth_link_ori_siz_end`, `two_orth_do_not_overlap`, `calendar`.

Key words

characteristic of a constraint: core.

complexity: sequencing with release times and deadlines.

constraint type: decomposition.

filtering: Klee measure problem, sweep, quadtree, compulsory part, constructive disjunction.

geometry: geometrical constraint, orthotope, polygon, non-overlapping.

problems: pallet loading.

puzzles: squared squares.

Arc input(s)	ORTHOTOPES
Arc generator	$SELF \mapsto$ collection(orthotopes)
Arc arity	1
Arc constraint(s)	orth_link_ori_siz_end(orthotopes.orth)
Graph property(ies)	**NARC**= |ORTHOTOPES|

Arc input(s)	ORTHOTOPES
Arc generator	$CLIQUE(\neq) \mapsto$ collection(orthotopes1, orthotopes2)
Arc arity	2
Arc constraint(s)	two_orth_do_not_overlap(orthotopes1.orth, orthotopes2.orth)
Graph property(ies)	**NARC**= |ORTHOTOPES| $*$ |ORTHOTOPES| $-$ |ORTHOTOPES|

Graph model The diffn constraint is expressed by using two graph constraints:

- The first graph constraint enforces for each dimension and for each orthotope the link between the corresponding ori, siz and end attributes.
- The second graph constraint imposes each pair of distinct orthotopes to not overlap.

Parts (A) and (B) of Figure 4.188 respectively show the initial and final graph associated with the second graph constraint of the **Example** slot. Since we use the **NARC** graph property, the arcs of the final graph are stressed in bold.

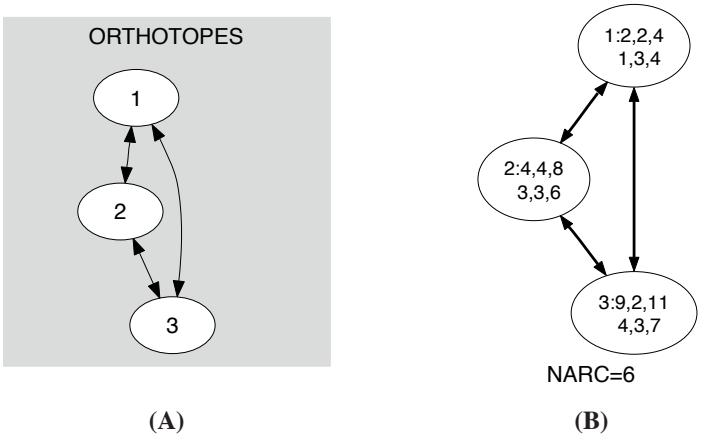


Figure 4.188: Initial and final graph of the diffn constraint

Signature

Since |ORTHOTOPES| is the maximum number of vertices of the final graph of the first graph constraint we can rewrite $\mathbf{NARC} =$ |ORTHOTOPES| to $\mathbf{NARC} \geq$ |ORTHOTOPES|. This leads to simplify $\underline{\overline{\mathbf{NARC}}}$ to $\overline{\mathbf{NARC}}$.

Since we use the $CLIQUE(\neq)$ arc generator on the ORTHOTOPES collection, |ORTHOTOPES| $\cdot$ |ORTHOTOPES| $-$ |ORTHOTOPES| is the maximum number of vertices of the final graph of the second graph constraint. Therefore we can rewrite $\mathbf{NARC} =$ |ORTHOTOPES| $\cdot$ |ORTHOTOPES| $-$ |ORTHOTOPES| to $\mathbf{NARC} \geq$ |ORTHOTOPES| $\cdot$ |ORTHOTOPES| $-$ |ORTHOTOPES|. Again, this leads to simplify $\underline{\overline{\mathbf{NARC}}}$ to $\overline{\mathbf{NARC}}$.

4.92 diffst

DESCRIPTION **LINKS**

Origin Generalisation of diffn.

Constraint diffst(K, DIMS, OBJECTS, SBOXES)

Type(s)

VARIABLES : collection(v−dvar)
INTEGERS : collection(v−int)
NATURALS : collection(v−int)

Argument(s)

K	:	int
DIMS	:	sint
OBJECTS	:	collection(oid−int, sid−dvar, x − VARIABLES, start−dvar, duration−dvar, end−dvar)
SBOXES	:	collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)

required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
require_at_least(2, OBJECTS, [start, duration, end])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
OBJECTS.duration ≥ 0
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if (1) the difference between the end in time and the start in time of each object is equal to its duration in time, and if (2) for each pair of objects (O_i, O_j), $i < j$, O_i and O_j do not overlap with respect to a set of dimensions depicted by `DIMS` as well as to the time axis. O_i and O_j are objects that take a shape among a set of shapes. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a `K`-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id `sid`, shift offset `t`, and sizes `l`. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier `oid`, shape id `sid` and origin `x`.

An object O_i *does not overlap* an object O_j with respect to a set of dimensions depicted by `DIMS` as well as to the time axis if and only if:

- The start in time of O_i is greater than or equal to the end in time of O_j.
- The start in time of O_j is greater than or equal to the end in time of O_i.
- For all shifted box s_i associated with O_i and for all shifted box s_j associated with O_j there exists a dimension $d \in$ `DIMS` such that the start of s_i in dimension d is greater than or equal to the end of s_j in dimension d, or the start of s_j in dimension d is greater than or equal to the end of s_i in dimension d.

Example

$$\left(\begin{array}{l} 2, \{0,1\}, \\ \left\langle \begin{array}{llllll} \texttt{oid}-1 & \texttt{sid}-1 & \texttt{x}-\langle 1,2\rangle & \texttt{start}-0 & \texttt{duration}-1 & \texttt{end}-1, \\ \texttt{oid}-2 & \texttt{sid}-5 & \texttt{x}-\langle 2,1\rangle & \texttt{start}-0 & \texttt{duration}-1 & \texttt{end}-1, \\ \texttt{oid}-3 & \texttt{sid}-8 & \texttt{x}-\langle 4,1\rangle & \texttt{start}-0 & \texttt{duration}-1 & \texttt{end}-1 \end{array} \right\rangle, \\ \begin{array}{lll} \texttt{sid}-1 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 2,1\rangle, \\ \texttt{sid}-1 & \texttt{t}-\langle 0,1\rangle & \texttt{l}-\langle 1,2\rangle, \\ \texttt{sid}-1 & \texttt{t}-\langle 1,2\rangle & \texttt{l}-\langle 3,1\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,1\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,1\rangle & \texttt{l}-\langle 1,3\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 2,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 2,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 1,1\rangle & \texttt{l}-\langle 1,2\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle -2,2\rangle & \texttt{l}-\langle 3,1\rangle, \\ \texttt{sid}-4 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,1\rangle, \\ \texttt{sid}-4 & \texttt{t}-\langle 0,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-4 & \texttt{t}-\langle 2,1\rangle & \texttt{l}-\langle 1,3\rangle, \\ \texttt{sid}-5 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 2,1\rangle, \\ \texttt{sid}-5 & \texttt{t}-\langle 1,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-5 & \texttt{t}-\langle 0,2\rangle & \texttt{l}-\langle 2,1\rangle, \\ \texttt{sid}-6 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,1\rangle, \\ \texttt{sid}-6 & \texttt{t}-\langle 0,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-6 & \texttt{t}-\langle 2,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-7 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,2\rangle, \\ \texttt{sid}-8 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 2,3\rangle \end{array} \end{array}\right)$$

Parts (A), (B) and (C) of Figure 4.191 respectively represent the potential shapes associated with the three objects of the example. Part (D) shows the position of the three objects of the example, where the first, second and third objects were respectively assigned shapes 1, 5 and 8. The coordinates of the leftmost lowest corner of each object are stressed in bold. The `diffst` constraint holds since the three objects do not overlap: even if the

time intervals associated with each object overlap (i.e., they are in fact identical), their corresponding shapes do not overlap (i.e., see part (D) if Figure 4.191).

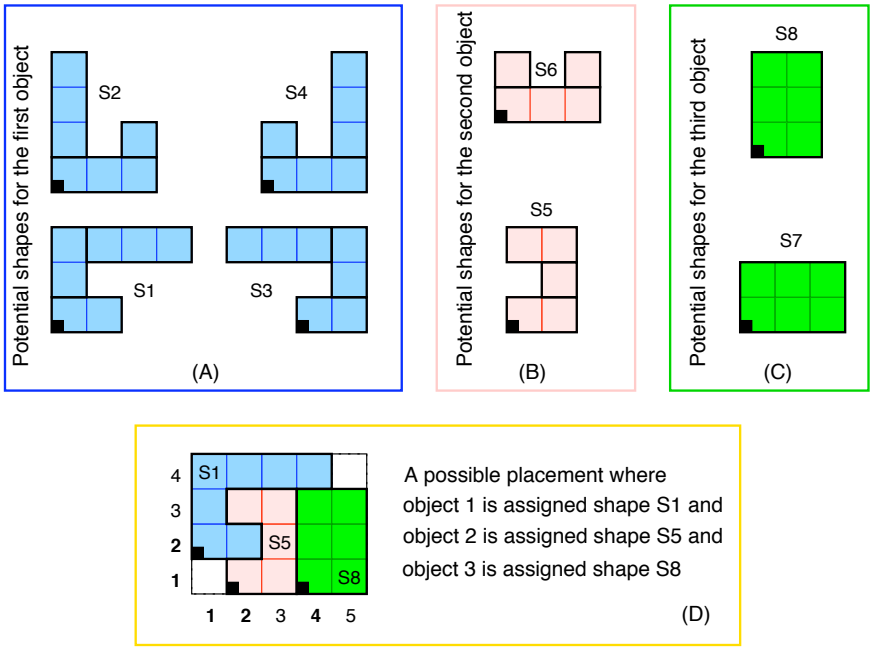


Figure 4.191: The three objects of the example

Usage

The `diffst` constraint allows to model directly a large number of placement problems. Figure 4.192 sketches ten typical use of the `diffst` constraint:

- The first case (A) corresponds to a non-overlapping constraint among three segments.
- The second, third and fourth cases (B,C,D) correspond to a non-overlapping constraint between rectangles where (B) and (C) are special cases where the sizes of all rectangles in the second dimension are equal to 1; this can be interpreted as a *machine assignment problem* where each rectangle corresponds to a non-pre-emptive task that has to be placed in time and assigned to a specific machine so that no two tasks assigned to the same machine overlap in time. In Part (B) the duration of each task is fixed, while in Part (C) the duration depends on the machine to which the task is actually assigned. This dependence can be expressed by the `element` constraint, which specifies the dependence between the shape variable and the assignment variable of each task.
- The fifth case (E) corresponds to a non-overlapping constraint between rectangles where each rectangle can have two orientations. This is achieved by associating with each rectangle two shapes of respective sizes $l \cdot h$ and $h \cdot l$. Since their orientation is not initially fixed, an `element_lesseq` constraint can be used for enforcing the three rectangles to be included within the bounding box defined by the origin's coordinates $1, 1$ and sizes $8, 3$.

- The sixth case (F) corresponds to a non-overlapping constraint between more complex objects where each object is described by a given set of rectangles.
- The seventh case (G) describes a rectangle placement problem where one has to first assign each rectangle to a strip so that all rectangles that are assigned to the same strip do not overlap.
- The eighth case (H) corresponds to a non-overlapping constraint between parallelepipeds.
- The ninth case (I) can be interpreted as a non-overlapping constraint between parallelepipeds that are assigned to the same container. The first dimension corresponds to the identifier of the container, while the next three dimensions are associated with the position of a parallelepiped inside a container.
- Finally the tenth case (J) describes a rectangle placement problem over three consecutive time-slots: rectangles assigned to the same time-slot should not overlap in time. We initially start with the three rectangles 1, 2 and 3. Rectangle 3 is no more present at instant 2 (the arrow $\downarrow$ within rectangle 3 at time 1 indicates that rectangle 3 will disappear at the next time-point), while rectangle 4 appears at instant 2 (the arrow $\uparrow$ within rectangle 4 at time 2 denotes the fact that the rectangle 4 appears at instant 2). Finally rectangle 2 disappears at instant 3 and is replaced by rectangle 5.

Algorithm A sweep-based filtering algorithm for this constraint is described in [25]. Unlike previous sweep filtering algorithms which move a line for finding a feasible position for the origin of an object, this algorithm performs a recursive traversal of the multidimensional placement space. It explores all points of the domain of the origin of the object under focus, one by one, in increasing lexicographic order, until a point is found that is not infeasible for any non-overlapping constraints. To make the search efficient, instead of moving each time to the successor point, the search is arranged so that it skips points that are known to be infeasible for some non-overlapping constraint.

See also `visible`, `non_overlap_sboxes`, `diffn`.

Key words **constraint type:** decomposition.

filtering: sweep.

geometry: geometrical constraint, non-overlapping.

puzzles: squared squares.

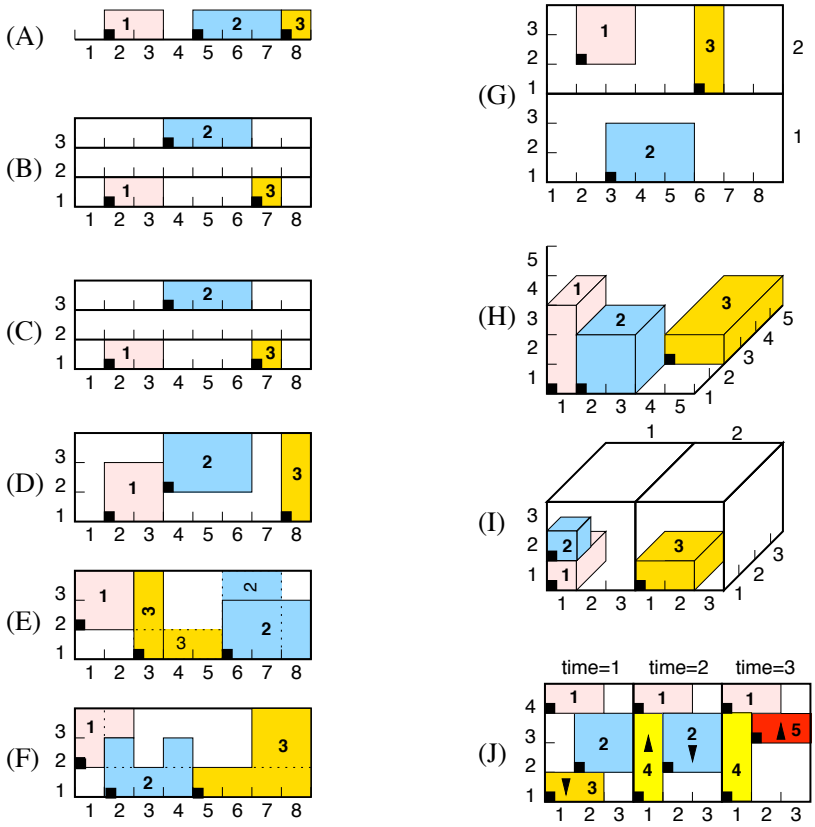


Figure 4.192: Ten typical examples of use of the `diffst` constraint (ground instances)

4.96 disjoint_sboxes

DESCRIPTION **LINKS**

Origin Geometry, derived from [227]

Constraint disjoint_sboxes(K, DIMS, OBJECTS, SBOXES)

Synonym(s) disjoint.

Type(s)
VARIABLES : collection(v−dvar)
INTEGERS : collection(v−int)
NATURALS : collection(v−int)

Argument(s)
K : int
DIMS : sint
OBJECTS : collection(oid−int, sid−int, x − VARIABLES)
SBOXES : collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)
required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if, for each pair of objects (O_i, O_j), $i \neq j$, O_i and O_j are disjoint with respect to a set of dimensions depicted by DIMS. O_i and O_j are objects that take a shape among a set of shapes. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a K-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id sid, shift offset t, and sizes l. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier oid, shape id sid and origin x.
Two objects O_i and object O_j are *disjoint* with respect to a set of dimensions depicted by DIMS if and only if for all shifted box s_i associated with O_i and for all shifted box s_j associated with O_j there exists at least one dimension $d \in$ DIMS such that (1) the origin of s_i in dimension d is strictly greater than the end of s_j in dimension d, or (2) the origin of s_j in dimension d is strictly greater than the end of s_i in dimension d.

Example

$$\left(\begin{array}{l} 2, \{0,1\}, \\ \left\langle \begin{array}{lll} \texttt{oid}-1 & \texttt{sid}-1 & \texttt{x}-\langle 1,1\rangle, \\ \texttt{oid}-2 & \texttt{sid}-2 & \texttt{x}-\langle 4,1\rangle, \\ \texttt{oid}-3 & \texttt{sid}-4 & \texttt{x}-\langle 2,4\rangle \end{array} \right\rangle, \\ \left\langle \begin{array}{ll} \texttt{sid}-1 & \texttt{t}-\langle 0,0\rangle \quad \texttt{l}-\langle 1,2\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,0\rangle \quad \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 1,0\rangle \quad \texttt{l}-\langle 1,3\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,2\rangle \quad \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,0\rangle \quad \texttt{l}-\langle 3,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,1\rangle \quad \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 2,1\rangle \quad \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-4 & \texttt{t}-\langle 0,0\rangle \quad \texttt{l}-\langle 1,1\rangle \end{array} \right\rangle \end{array}\right)$$

Figure 4.198 shows the objects of the example. Since these objects are pairwise disjoint the `disjoint_sboxes` constraint holds.

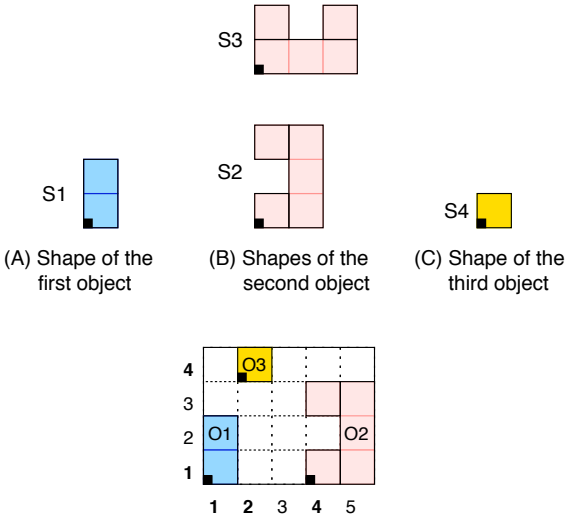


Figure 4.198: The three mutually disjoint objects of the example

Remark One of the eight relations of the *Region Connection Calculus* [227].

See also `contains_sboxes`, `coveredby_sboxes`, `covers_sboxes`, `equal_sboxes`, `inside_sboxes`, `meet_sboxes`, `non_overlap_sboxes`, `overlap_sboxes`.

Key words **constraint type:** predefined constraint.

geometry: geometrical constraint, rcc8.

4.143 inside_sboxes

DESCRIPTION **LINKS**

Origin Geometry, derived from [227]

Constraint inside_sboxes(K, DIMS, OBJECTS, SBOXES)

Synonym(s) inside.

Type(s)

VARIABLES	:	collection(v−dvar)
INTEGERS	:	collection(v−int)
NATURALS	:	collection(v−int)

Argument(s)

K	:	int
DIMS	:	sint
OBJECTS	:	collection(oid−int, sid−int, x − VARIABLES)
SBOXES	:	collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)

required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if, for each pair of objects (O_i, O_j), $i < j$, O_i is inside O_j with respect to a set of dimensions depicted by DIMS. O_i and O_j are objects that take a shape among a set of shapes. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a K-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id sid, shift offset t, and sizes l. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier oid, shape id sid and origin x.
An object O_i *is inside* an object O_j with respect to a set of dimensions depicted by DIMS if and only if, for all shifted boxes s_i associated with O_i, there exists a shifted box s_j of O_j such that s_j is inside s_i. A shifted box s_j *is inside* a shifted box s_i if and only if, for all dimensions $d \in$ DIMS, (1) the start of s_j in dimension d is strictly less than the start of s_i in dimension d, and (2) the end of s_i in dimension d is strictly less than the end of s_j in dimension d.

Example

$$\left\langle \begin{array}{l} 2, \{0,1\}, \\ \left\langle \begin{array}{lll} \texttt{oid} - 1 & \texttt{sid} - 1 & \texttt{x} - \langle 1,1\rangle, \\ \texttt{oid} - 2 & \texttt{sid} - 2 & \texttt{x} - \langle 2,2\rangle, \\ \texttt{oid} - 3 & \texttt{sid} - 3 & \texttt{x} - \langle 3,3\rangle \end{array} \right\rangle, \\ \left\langle \begin{array}{lll} \texttt{sid} - 1 & \texttt{t} - \langle 0,0\rangle & \texttt{l} - \langle 1,1\rangle, \\ \texttt{sid} - 2 & \texttt{t} - \langle 0,0\rangle & \texttt{l} - \langle 3,3\rangle, \\ \texttt{sid} - 3 & \texttt{t} - \langle 0,0\rangle & \texttt{l} - \langle 5,5\rangle \end{array} \right\rangle \end{array} \right\rangle$$

Figure 4.299 shows the objects of the example. Since O_1 is inside O_2 and O_3, and since O_2 is also inside O_3, the `inside_sboxes` constraint holds.

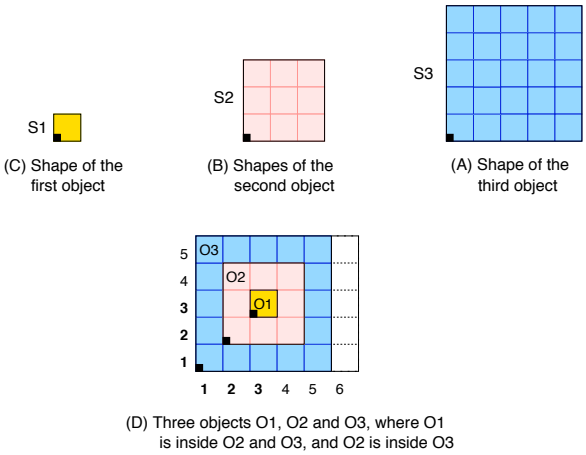


Figure 4.299: The three objects of the example

Remark One of the eight relations of the *Region Connection Calculus* [227]. The constraint `inside_sboxes` is a restriction of the original relation since it requires that each box of an object is contained by one box of the other object.

See also `contains_sboxes`, `coveredby_sboxes`, `covers_sboxes`, `disjoint_sboxes`, `equal_sboxes`, `meet_sboxes`, `non_overlap_sboxes`, `overlap_sboxes`.

Key words **constraint type:** predefined constraint.

geometry: geometrical constraint, rcc8.

4.167 lex_chain_less

DESCRIPTION **LINKS** **GRAPH**

Origin [65]

Constraint lex_chain_less(VECTORS)

Usual name lex_chain

Type(s) VECTOR : collection(var−dvar)

Argument(s) VECTORS : collection(vec − VECTOR)

Restriction(s) required(VECTOR, var)
required(VECTORS, vec)
same_size(VECTORS, vec)

Purpose For each pair of consecutive vectors $\texttt{VECTOR}_i$ and $\texttt{VECTOR}_{i+1}$ of the VECTORS collection we have that $\texttt{VECTOR}_i$ is lexicographically strictly less than $\texttt{VECTOR}_{i+1}$. Given two vectors, $\vec{X}$ and $\vec{Y}$ of n components, $\langle X_0, \ldots, X_{n-1}\rangle$ and $\langle Y_0, \ldots, Y_{n-1}\rangle$, $\vec{X}$ is *lexicographically strictly less than* $\vec{Y}$ if and only if $X_0 < Y_0$ or $X_0 = Y_0$ and $\langle X_1, \ldots, X_{n-1}\rangle$ is lexicographically strictly less than $\langle Y_1, \ldots, Y_{n-1}\rangle$.

Example
$$\left\langle \begin{array}{l} \texttt{vec} - \langle 5,2,3,9\rangle, \\ \texttt{vec} - \langle 5,2,6,2\rangle, \\ \texttt{vec} - \langle 5,2,6,3\rangle \end{array} \right\rangle$$

The lex_chain_less constraint holds since:

- The first vector $\langle 5,2,3,9\rangle$ of the VECTORS collection is lexicographically strictly less than the second vector $\langle 5,2,6,2\rangle$ of the VECTORS collection.
- The second vector $\langle 5,2,6,2\rangle$ of the VECTORS collection is lexicographically strictly less than the third vector $\langle 5,2,6,3\rangle$ of the VECTORS collection.

Usage This constraint was motivated for breaking symmetry: more precisely when one wants to lexicographically order the consecutive columns of a matrix of decision variables. A further motivation is that using a set of lexicographic ordering constraints between two vectors does usually not allows to come up with a complete pruning.

Algorithm A filtering algorithm achieving arc-consistency for a chain of lexicographical constraints is presented in [65].

See also lex_between, lex_chain_lesseq, lex_less, lex_lesseq, lex_greater, lex_greatereq.

Key words **characteristic of a constraint:** vector.

constraint type: decomposition, order constraint.

filtering: arc-consistency.

symmetry: symmetry, matrix symmetry, lexicographic order.

Arc input(s)	VECTORS
Arc generator	$PATH \mapsto$ collection(vectors1, vectors2)
Arc arity	2
Arc constraint(s)	lex_less(vectors1.vec, vectors2.vec)
Graph property(ies)	**NARC**= |VECTORS| − 1

Graph model Parts (A) and (B) of Figure 4.335 respectively show the initial and final graph associated with the **Example** slot. Since we use the **NARC** graph property, the arcs of the final graph are stressed in bold. The lex_chain_less constraint holds since all the arc constraints of the initial graph are satisfied.

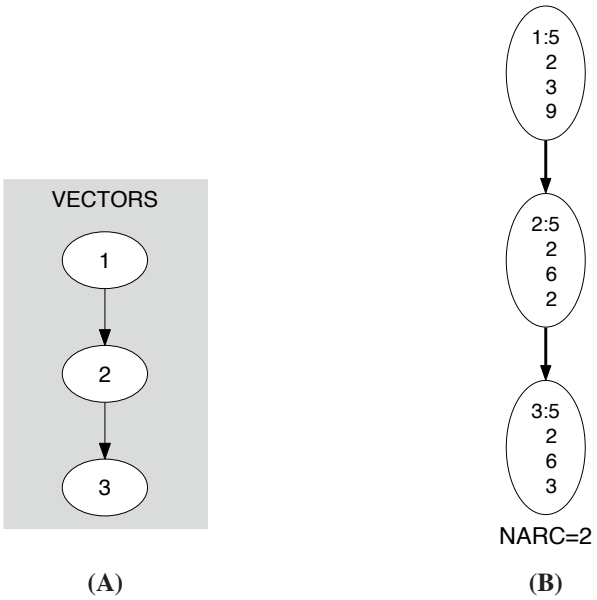


Figure 4.335: Initial and final graph of the lex_chain_less constraint

Signature Since we use the *PATH* arc generator on the VECTORS collection the number of arcs of the initial graph is equal to |VECTORS| − 1. For this reason we can rewrite **NARC** = |VECTORS| − 1 to **NARC** ≥ |VECTORS| − 1 and simplify $\underline{\mathbf{NARC}}$ to $\overline{\mathbf{NARC}}$.

4.185 meet_sboxes

DESCRIPTION **LINKS**

Origin Geometry, derived from [227]

Constraint meet_sboxes(K, DIMS, OBJECTS, SBOXES)

Synonym(s) meet.

Type(s)

VARIABLES	:	collection(v−dvar)
INTEGERS	:	collection(v−int)
NATURALS	:	collection(v−int)

Argument(s)

K	:	int
DIMS	:	sint
OBJECTS	:	collection(oid−int, sid−int, x − VARIABLES)
SBOXES	:	collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)

required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if, for each pair of objects (O_i, O_j), $i \neq j$, O_i and O_j meet with respect to a set of dimensions depicted by DIMS. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a K-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id sid, shift offset t, and sizes l. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier oid, shape id sid and origin x.
Two objects O_i and object O_j *meet* with respect to a set of dimensions depicted by DIMS if and only if the two following conditions hold:

- For all shifted box s_i associated with O_i and for all shifted box s_j associated with O_j there exists a dimension $d \in$ DIMS such that (1) the start of s_i in dimension d is greater than or equal to the end of s_j in dimension d, or (2) the start of s_j in dimension d is greater than or equal to the end of s_i in dimension d (i.e., there is no overlap between the shifted box of O_i and the shifted box of O_j).
- There exists a shifted box s_i of O_i and there exists a shifted box s_j of O_j such that for all dimensions d (1) the end of s_i in dimension d is greater than or equal to the start of s_j in dimension d, and (2) the end of s_j in dimension d is greater than or equal to the start of s_i in dimension d (i.e., at least two shifted box of O_i and O_j are in contact).

Example

$$\left(\begin{array}{l} 1, \{0,1\}, \\ \left\langle \begin{array}{lll} \texttt{oid}-1 & \texttt{sid}-1 & \texttt{x}-\langle 3,2\rangle, \\ \texttt{oid}-2 & \texttt{sid}-2 & \texttt{x}-\langle 4,1\rangle, \\ \texttt{oid}-3 & \texttt{sid}-4 & \texttt{x}-\langle 3,4\rangle \end{array} \right\rangle, \\ \left\langle \begin{array}{lll} \texttt{sid}-1 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 1,2\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 1,0\rangle & \texttt{l}-\langle 1,3\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,2\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 2,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-4 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 1,1\rangle \end{array} \right\rangle \end{array}\right)$$

Figure 4.366 shows the objects of the example. Since all the pairs of objects meet the meet_sboxes constraint holds.

Remark One of the eight relations of the *Region Connection Calculus* [227].

See also contains_sboxes, coveredby_sboxes, covers_sboxes, disjoint_sboxes, equal_sboxes, inside_sboxes, non_overlap_sboxes, overlap_sboxes.

Key words **constraint type:** predefined constraint.

geometry: geometrical constraint, rcc8.

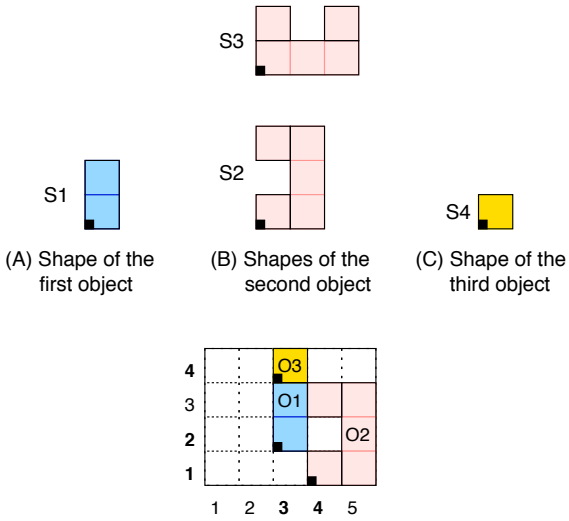


Figure 4.366: The three objects of the example

4.204 non_overlap_sboxes

DESCRIPTION **LINKS**

Origin Geometry, derived from [25]

Constraint non_overlap_sboxes(K, DIMS, OBJECTS, SBOXES)

Synonym(s) non_overlap, non_overlapping.

Type(s)

VARIABLES	:	collection(v−dvar)
INTEGERS	:	collection(v−int)
NATURALS	:	collection(v−int)

Argument(s)

K	:	int
DIMS	:	sint
OBJECTS	:	collection(oid−int, sid−int, x − VARIABLES)
SBOXES	:	collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)

required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if, for each pair of objects $(O_i, O_j), i < j$, O_i and O_j do not overlap with respect to a set of dimensions depicted by DIMS. O_i and O_j are objects that take a shape among a set of shapes. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a K-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id sid, shift offset t, and sizes l. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier oid, shape id sid and origin x.
An object O_i *does not overlap* an object O_j with respect to a set of dimensions depicted by DIMS if and only if, for all shifted box s_i associated with O_i and for all shifted box s_j associated with O_j, there exists a dimension $d \in$ DIMS such that the start of s_i in dimension d is greater than or equal to the end of s_j in dimension d, or the start of s_j in dimension d is greater than or equal to the end of s_i in dimension d.

Example

$$\left(\begin{array}{l} 1, \{0,1\}, \\ \left\langle \begin{array}{lll} \texttt{oid}-1 & \texttt{sid}-1 & \texttt{x}-\langle 4,1\rangle, \\ \texttt{oid}-2 & \texttt{sid}-2 & \texttt{x}-\langle 2,2\rangle, \\ \texttt{oid}-3 & \texttt{sid}-4 & \texttt{x}-\langle 5,4\rangle \end{array} \right\rangle, \\ \left\langle \begin{array}{lll} \texttt{sid}-1 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-1 & \texttt{t}-\langle 1,0\rangle & \texttt{l}-\langle 1,3\rangle, \\ \texttt{sid}-1 & \texttt{t}-\langle 0,2\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,1\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 2,1\rangle & \texttt{l}-\langle 1,1\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 1,2\rangle, \\ \texttt{sid}-4 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 1,1\rangle \end{array} \right\rangle \end{array}\right)$$

Figure 4.399 shows the objects of the example. Since O_1 and O_2 do not overlap, since O_1 and O_3 do not overlap, and since O_2 and O_3 also do not overlap, the `non_overlap_sboxes` constraint holds.

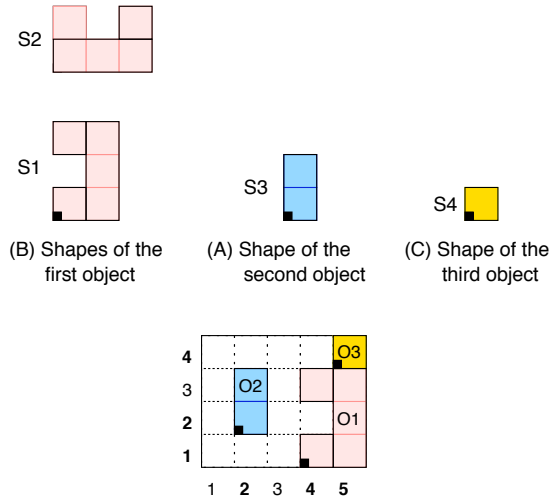


Figure 4.399: The three objects of the example

See also `diffst`, `contains_sboxes`, `coveredby_sboxes`, `covers_sboxes`, `diffn`, `disjoint_sboxes`, `equal_sboxes`, `inside_sboxes`, `meet_sboxes`, `overlap_sboxes`.

Key words **constraint type:** predefined constraint.

geometry: geometrical constraint, non-overlapping.

4.226 overlap_sboxes

DESCRIPTION **LINKS**

Origin Geometry, derived from [227]

Constraint overlap_sboxes(K, DIMS, OBJECTS, SBOXES)

Synonym(s) overlap.

Type(s)
VARIABLES : collection(v−dvar)
INTEGERS : collection(v−int)
NATURALS : collection(v−int)

Argument(s)

K	:	int
DIMS	:	sint
OBJECTS	:	collection(oid−int, sid−int, x − VARIABLES)
SBOXES	:	collection(sid−int, t − INTEGERS, l − NATURALS)

Restriction(s)
required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if, for each pair of objects (O_i, O_j), $i < j$, O_i overlaps O_j with respect to a set of dimensions depicted by DIMS. O_i and O_j are objects that take a shape among a set of shapes. Each *shape* is defined as a finite set of shifted boxes, where each shifted box is described by a box in a K-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* is an entity defined by its shape id sid, shift offset t, and sizes l. Then, a shape is defined as the union of shifted boxes sharing the same shape id. An *object* is an entity defined by its unique object identifier oid, shape id sid and origin x.
An object O_i *overlaps* an object O_j with respect to a set of dimensions depicted by DIMS if and only if, there exists a shifted box s_i associated with O_i and there exists a shifted box s_j associated with O_j, such that (1) there exists a dimension $d \in$ DIMS where the end of O_i in dimension d is strictly greater than the start of O_j in dimension d, and (2) the end of O_j in dimension d is strictly greater than the start of O_i in dimension d.

Example

$$\left\langle \begin{array}{l} 2, \{0,1\}, \\ \left\langle \begin{array}{lll} \texttt{oid}-1 & \texttt{sid}-1 & \texttt{x}-\langle 1,1\rangle, \\ \texttt{oid}-2 & \texttt{sid}-2 & \texttt{x}-\langle 3,2\rangle, \\ \texttt{oid}-3 & \texttt{sid}-3 & \texttt{x}-\langle 2,4\rangle \end{array} \right\rangle, \\ \left\langle \begin{array}{lll} \texttt{sid}-1 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 4,5\rangle, \\ \texttt{sid}-2 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 3,3\rangle, \\ \texttt{sid}-3 & \texttt{t}-\langle 0,0\rangle & \texttt{l}-\langle 2,1\rangle \end{array} \right\rangle \end{array} \right\rangle$$

Figure 4.431 shows the objects of the example. Since O_1 overlaps both O_2 and O_3, and since O_2 overlaps O_3, the `overlap_sboxes` constraint holds.

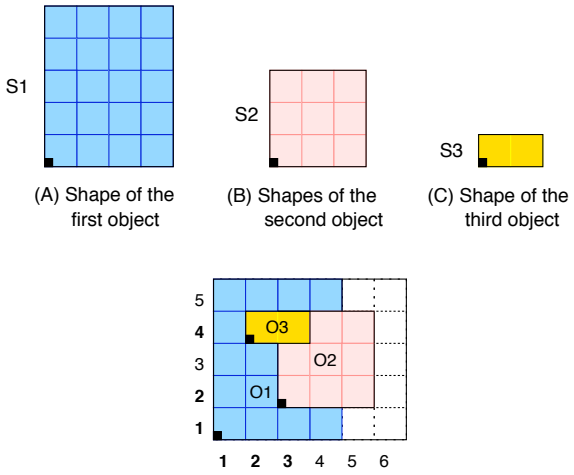


Figure 4.431: The three objects of the example

Remark One of the eight relations of the *Region Connection Calculus* [227].

See also `coveredby_sboxes`, `covers_sboxes`, `contains_sboxes`, `disjoint_sboxes`, `equal_sboxes`, `inside_sboxes`, `meet_sboxes`, `non_overlap_sboxes`.

Key words **constraint type:** predefined constraint.

geometry: geometrical constraint, rcc8.

4.232 place_in_pyramid

DESCRIPTION **LINKS** **GRAPH**

Origin N. Beldiceanu

Constraint place_in_pyramid(ORTHOTOPES, VERTICAL_DIM)

Type(s) ORTHOTOPE : collection(ori−dvar, siz−dvar, end−dvar)

Argument(s)

ORTHOTOPES	:	collection(orth − ORTHOTOPE)
VERTICAL_DIM	:	int

Restriction(s)

|ORTHOTOPE| > 0
require_at_least(2, ORTHOTOPE, [ori, siz, end])
ORTHOTOPE.siz ≥ 0
ORTHOTOPE.ori ≤ ORTHOTOPE.end
required(ORTHOTOPES, orth)
same_size(ORTHOTOPES, orth)
VERTICAL_DIM ≥ 1
diffn(ORTHOTOPES)

Purpose For each pair of orthotopes (O_1, O_2) of the collection ORTHOTOPES, O_1 and O_2 do not overlap (two orthotopes do not overlap if there exists at least one dimension where their projections do not overlap). In addition, each orthotope of the collection ORTHOTOPES should be supported by one other orthotope or by the ground. The vertical dimension is given by the parameter VERTICAL_DIM.

Example

$$\left(\left\langle \begin{array}{l} \texttt{orth} - \langle \texttt{ori} - 1\ \texttt{siz} - 3\ \texttt{end} - 4, \texttt{ori} - 1\ \texttt{siz} - 2\ \texttt{end} - 3\rangle, \\ \texttt{orth} - \langle \texttt{ori} - 1\ \texttt{siz} - 2\ \texttt{end} - 3, \texttt{ori} - 3\ \texttt{siz} - 3\ \texttt{end} - 6\rangle, \\ \texttt{orth} - \langle \texttt{ori} - 5\ \texttt{siz} - 6\ \texttt{end} - 11, \texttt{ori} - 1\ \texttt{siz} - 2\ \texttt{end} - 3\rangle, \\ \texttt{orth} - \langle \texttt{ori} - 5\ \texttt{siz} - 2\ \texttt{end} - 7, \texttt{ori} - 3\ \texttt{siz} - 2\ \texttt{end} - 5\rangle, \\ \texttt{orth} - \langle \texttt{ori} - 8\ \texttt{siz} - 3\ \texttt{end} - 11, \texttt{ori} - 3\ \texttt{siz} - 2\ \texttt{end} - 5\rangle, \\ \texttt{orth} - \langle \texttt{ori} - 8\ \texttt{siz} - 2\ \texttt{end} - 10, \texttt{ori} - 5\ \texttt{siz} - 2\ \texttt{end} - 7\rangle \end{array} \right\rangle, 2\right)$$

Figure 4.438 depicts the placement associated with the example, where the i^{th} item of the ORTHOTOPES collection is represented by the rectangle Ri. The place_in_pyramid constraint holds since the rectangles do not overlap and since rectangles R1, R2, R3, R4, R5, and R6 are respectively supported by the ground, R1, the ground, R3, R3, and R5.

Usage The diffn constraint is not enough if one wants to produce a placement where no orthotope floats in the air. This constraint is usually handled with a heuristic during the enumeration phase.

See also orth_on_top_of_orth, orth_on_the_ground.

Key words **geometry:** geometrical constraint, non-overlapping, orthotope.

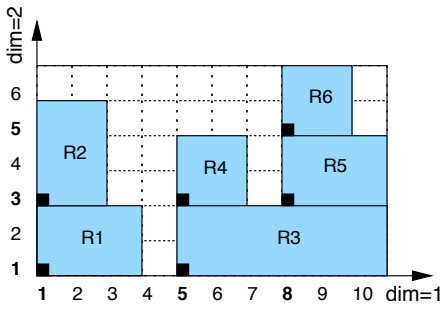


Figure 4.438: Solution corresponding to the example

Arc input(s) ORTHOTOPES

Arc generator $CLIQUE \mapsto$ collection(orthotopes1, orthotopes2)

Arc arity 2

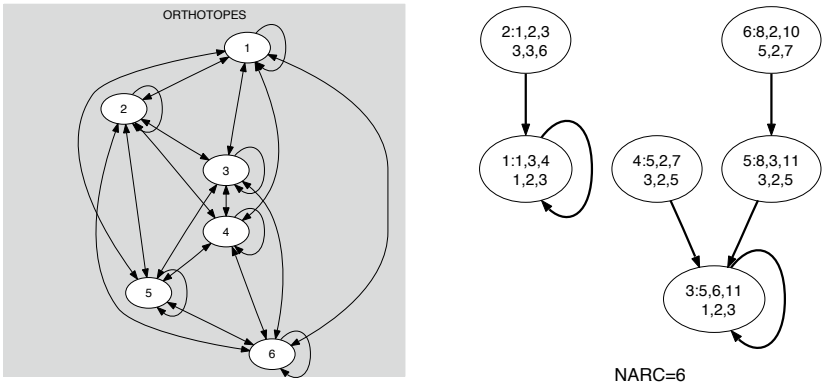
Arc constraint(s)
$$\bigvee \left(\begin{array}{l} \wedge \left(\begin{array}{l} \texttt{orthotopes1.key} = \texttt{orthotopes2.key}, \\ \texttt{orth\_on\_the\_ground}(\texttt{orthotopes1.orth}, \texttt{VERTICAL\_DIM}) \end{array} \right), \\ \wedge \left(\begin{array}{l} \texttt{orthotopes1.key} \neq \texttt{orthotopes2.key}, \\ \texttt{orth\_on\_top\_of\_orth}(\texttt{orthotopes1.orth}, \texttt{orthotopes2.orth}, \texttt{VERTICAL\_DIM}) \end{array} \right) \end{array} \right)$$

Graph property(ies) **NARC**= |ORTHOTOPES|

Graph model The arc constraint of the graph constraint enforces one of the following conditions:

- If the arc connects the same orthotope O then the ground directly supports O,
- Otherwise, if we have an arc from an orthotope O_1 to a distinct orthotope O_2, the condition is: O_1 is on top of O_2 (i.e., in all dimensions, except dimension VERTICAL_DIM, the projection of O_1 is included in the projection of O_2, while in dimension VERTICAL_DIM the projection of O_1 is located after the projection of O_2).

Parts (A) and (B) of Figure 4.439 respectively show the initial and final graph associated with the **Example** slot. Since we use the **NARC** graph property, the arcs of the final graph are stressed in bold.



(A) **(B)**

Figure 4.439: Initial and final graph of the place_in_pyramid constraint

4.306 visible

DESCRIPTION **LINKS**

Origin Extension of *accessibility* parameter of diffn.

Constraint visible(K, DIMS, FROM, OBJECTS, SBOXES)

Type(s)

VARIABLES : collection(v−dvar)
INTEGERS : collection(v−int)
NATURALS : collection(v−int)
DIMDIR : collection(dim−int, dir−int)

Argument(s)

K : int
DIMS : sint
FROM : DIMDIR
OBJECTS : collection(oid−int, sid−dvar, x − VARIABLES, start−dvar, duration−dvar, end−dvar)
SBOXES : collection(sid−int, t − INTEGERS, l − NATURALS, f − DIMDIR)

Restriction(s)

required(VARIABLES, v)
|VARIABLES| = K
required(INTEGERS, v)
|INTEGERS| = K
required(NATURALS, v)
|NATURALS| = K
NATURALS.v > 0
required(DIMDIR, [dim, dir])
|DIMDIR| > 0
|DIMDIR| ≤ K + K
distinct(DIMDIR, [])
DIMDIR.dim ≥ 0
DIMDIR.dim < K
DIMDIR.dir ≥ 0
DIMDIR.dir ≤ 1
K ≥ 0
DIMS ≥ 0
DIMS < K
required(OBJECTS, [oid, sid, x])
require_at_least(2, OBJECTS, [start, duration, end])
OBJECTS.oid ≥ 1
OBJECTS.oid ≤ |OBJECTS|
OBJECTS.sid ≥ 1
OBJECTS.sid ≤ |SBOXES|
OBJECTS.duration ≥ 0
required(SBOXES, [sid, t, l])
SBOXES.sid ≥ 1
SBOXES.sid ≤ |SBOXES|

Purpose

Holds if and only if:

1. The difference between the end in time and the start in time of each object is equal to its duration in time.
2. Given a collection of potential observations places FROM, where each observation place is specified by a *dimension* (i.e., an integer between 0 and $k-1$) and by a *direction* (i.e., an integer between 0 and 1), and given for each shifted box of SBOXES a set of visible faces, enforce that *at least one visible face of each shifted box associated with an object $o \in$ OBJECTS should be entirely visible from at least one observation place of* FROM *at time* o.start *as well as at time* o.end -1 *are transparent*. This notion is defined in a more formal way in the **Remark** slot.

Example

$$\left(\begin{array}{l} 2,\{0,1\}, \\ \langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \left\langle\begin{array}{llllll} \mathtt{oid}-1 & \mathtt{sid}-1 & \mathtt{x}-\langle 1,2\rangle & \mathtt{start}-8 & \mathtt{duration}-8 & \mathtt{end}-16, \\ \mathtt{oid}-2 & \mathtt{sid}-2 & \mathtt{x}-\langle 4,2\rangle & \mathtt{start}-1 & \mathtt{duration}-15 & \mathtt{end}-16 \end{array}\right\rangle, \\ \left\langle\begin{array}{lll} \mathtt{sid}-1 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 1,2\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \mathtt{sid}-2 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 2,3\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle \end{array}\right\rangle \end{array}\right)$$

$$\left(\begin{array}{l} 2,\{0,1\}, \\ \langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \left\langle\begin{array}{llllll} \mathtt{oid}-1 & \mathtt{sid}-1 & \mathtt{x}-\langle 1,2\rangle & \mathtt{start}-1 & \mathtt{duration}-8 & \mathtt{end}-9, \\ \mathtt{oid}-2 & \mathtt{sid}-2 & \mathtt{x}-\langle 4,2\rangle & \mathtt{start}-1 & \mathtt{duration}-15 & \mathtt{end}-16 \end{array}\right\rangle, \\ \left\langle\begin{array}{lll} \mathtt{sid}-1 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 1,2\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \mathtt{sid}-2 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 2,3\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle \end{array}\right\rangle \end{array}\right)$$

$$\left(\begin{array}{l} 2,\{0,1\}, \\ \langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \left\langle\begin{array}{llllll} \mathtt{oid}-1 & \mathtt{sid}-1 & \mathtt{x}-\langle 1,1\rangle & \mathtt{start}-1 & \mathtt{duration}-15 & \mathtt{end}-16, \\ \mathtt{oid}-2 & \mathtt{sid}-2 & \mathtt{x}-\langle 2,2\rangle & \mathtt{start}-6 & \mathtt{duration}-6 & \mathtt{end}-12 \end{array}\right\rangle, \\ \left\langle\begin{array}{lll} \mathtt{sid}-1 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 1,2\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \mathtt{sid}-2 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 2,3\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle \end{array}\right\rangle \end{array}\right)$$

$$\left(\begin{array}{l} 2,\{0,1\}, \\ \langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \left\langle\begin{array}{llllll} \mathtt{oid}-1 & \mathtt{sid}-1 & \mathtt{x}-\langle 4,1\rangle & \mathtt{start}-1 & \mathtt{duration}-8 & \mathtt{end}-9, \\ \mathtt{oid}-2 & \mathtt{sid}-2 & \mathtt{x}-\langle 1,2\rangle & \mathtt{start}-1 & \mathtt{duration}-15 & \mathtt{end}-16 \end{array}\right\rangle, \\ \left\langle\begin{array}{lll} \mathtt{sid}-1 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 1,2\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \mathtt{sid}-2 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 2,3\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle \end{array}\right\rangle \end{array}\right)$$

$$\left(\begin{array}{l} 2,\{0\}, \\ \langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \left\langle\begin{array}{llllll} \mathtt{oid}-1 & \mathtt{sid}-1 & \mathtt{x}-\langle 2,1\rangle & \mathtt{start}-1 & \mathtt{duration}-8 & \mathtt{end}-9, \\ \mathtt{oid}-2 & \mathtt{sid}-2 & \mathtt{x}-\langle 4,3\rangle & \mathtt{start}-1 & \mathtt{duration}-15 & \mathtt{end}-16 \end{array}\right\rangle, \\ \left\langle\begin{array}{lll} \mathtt{sid}-1 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 1,2\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle, \\ \mathtt{sid}-2 & \mathtt{t}-\langle 0,0\rangle & \mathtt{l}-\langle 2,2\rangle\ \ \mathtt{f}-\langle \mathtt{dim}-0\ \mathtt{dir}-1\rangle \end{array}\right\rangle \end{array}\right)$$

The five previous examples correspond respectively to parts (I), (II), (III) and (IV) of Figure 4.564 and to Figure 4.565. Before explaining these five examples Figure 4.563 first illustrates the notion of *observations places* and of *visible faces*.

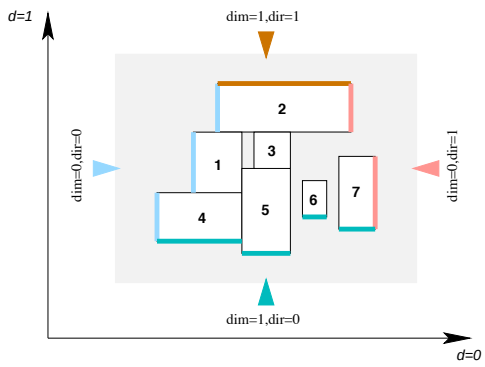


Figure 4.563: Entirely visible faces (depicted by a thick line) of rectangles 1, 2, 3, 4, 5, 6 and 7 from the four observation places $\langle dim = 0, dir = 1\rangle$, $\langle dim = 0, dir = 0\rangle$, $\langle dim = 1, dir = 1\rangle$ and $\langle dim = 1, dir = 0\rangle$ (depicted by an arrow)

We first need to introduce a number of definitions in order to illustrate the notion of *visibility*.

Definition 1. *Consider two distinct objects o and o' of the* `visible` *constraint (i.e., $o, o' \in iobjects$) as well as an observation place defined by the pair $\langle \texttt{dim}, \texttt{dir} \rangle \in$ FROM. The object o* is masked by *the object o' according to the observation place $\langle \texttt{dim}, \texttt{dir} \rangle$ if there exist two shifted boxes s and s' respectively associated with o and o' such that conditions* **A**, **B**, **C**, **D** *and* **E** *all hold:*

- *(***A***)* $o.\texttt{duration} > 0 \wedge o'.\texttt{duration} > 0 \wedge o.\texttt{end} > o'.\texttt{start} \wedge o'.\texttt{end} > o.\texttt{start}$ *(i.e., the time intervals associated with o and o' intersect).*
- *(***B***) Discarding dimension* `dim`*, s and s' intersect in all dimensions specified by* `DIMS` *(i.e., objects o and o' are in vis-à-vis).*
- *(***C***) If* $\texttt{dir} = 0$
  then $o.x[\texttt{dim}] + s.t[\texttt{dim}] \geq o'.x[\texttt{dim}] + s'.t[\texttt{dim}] + s'.l[\texttt{dim}]$
  else $o'.x[\texttt{dim}] + s'.t[\texttt{dim}] \geq o.x[\texttt{dim}] + s.t[\texttt{dim}] + s.l[\texttt{dim}]$ *(i.e., in dimension* `dim`*, o and o' are ordered in the wrong way according to direction* `dir`*).*
- *(***D***)* $o.\texttt{start} > o'.\texttt{start} \vee o.\texttt{end} < o'.\texttt{end}$ *(i.e., instants* $o.\texttt{start}$ *or* $o.\texttt{end}$ *are located within interval* $[o'.\texttt{start}, o'.\texttt{end}]$*; we consider also condition* **A***.).*
- *(***E***) The observation place $\langle \texttt{dim}, \texttt{dir} \rangle$ occurs within the list of visible faces associated with the face attribute* `f` *of the shifted box s (i.e., the pair $\langle \texttt{dim}, \texttt{dir} \rangle$ is a potentially visible face of o).*

Definition 2. *Consider an object o of the collection* `OBJECTS` *as well as a possible observation place defined by the pair $\langle \texttt{dim}, \texttt{dir} \rangle$. The object o* is masked according to the observation place $\langle \texttt{dim}, \texttt{dir} \rangle$ *if and only if at least one of the following conditions holds:*

- *No shifted box associated with o has the pair $\langle \texttt{dim}, \texttt{dir} \rangle$ as one of its potentially visible face.*
- *The object o is masked according to the possible observation place $\langle \texttt{dim}, \texttt{dir} \rangle$ by another object o'.*

Figures 4.564 and 4.565 respectively illustrate Definition 1 in the context of an observation place (depicted by a triangle) equal to the pair $\langle \texttt{dim} = 0, \texttt{dir} = 1 \rangle$. Observe that, in the context of Figure 4.565, as the `DIMS` parameter of the `visible` constraint only mentions dimension 0 (and not dimension 1), one object may be masked by another object even if the two objects do not intersect in any dimension: i.e., only their respective ordering in the dimension $\texttt{dim} = 0$ as well as their positions in time matter.

Definition 3. *Consider an object o of the collection* `OBJECTS` *as well as a possible observation place defined by the pair $\langle \texttt{dim}, \texttt{dir} \rangle$. The object o* is masked according to the observation place $\langle \texttt{dim}, \texttt{dir} \rangle$ *if and only if at least one of the following conditions holds:*

- *No shifted box associated with o has the pair $\langle \texttt{dim}, \texttt{dir} \rangle$ as one of its potentially visible face.*
- *The object o is masked according to the possible observation place $\langle \texttt{dim}, \texttt{dir} \rangle$ by another object o'.*

Definition 4. *An object of the collection* `OBJECTS` *constraint is* masked according to a set of possible observation places `FROM` *if it is masked according to each observation place of* `FROM`.

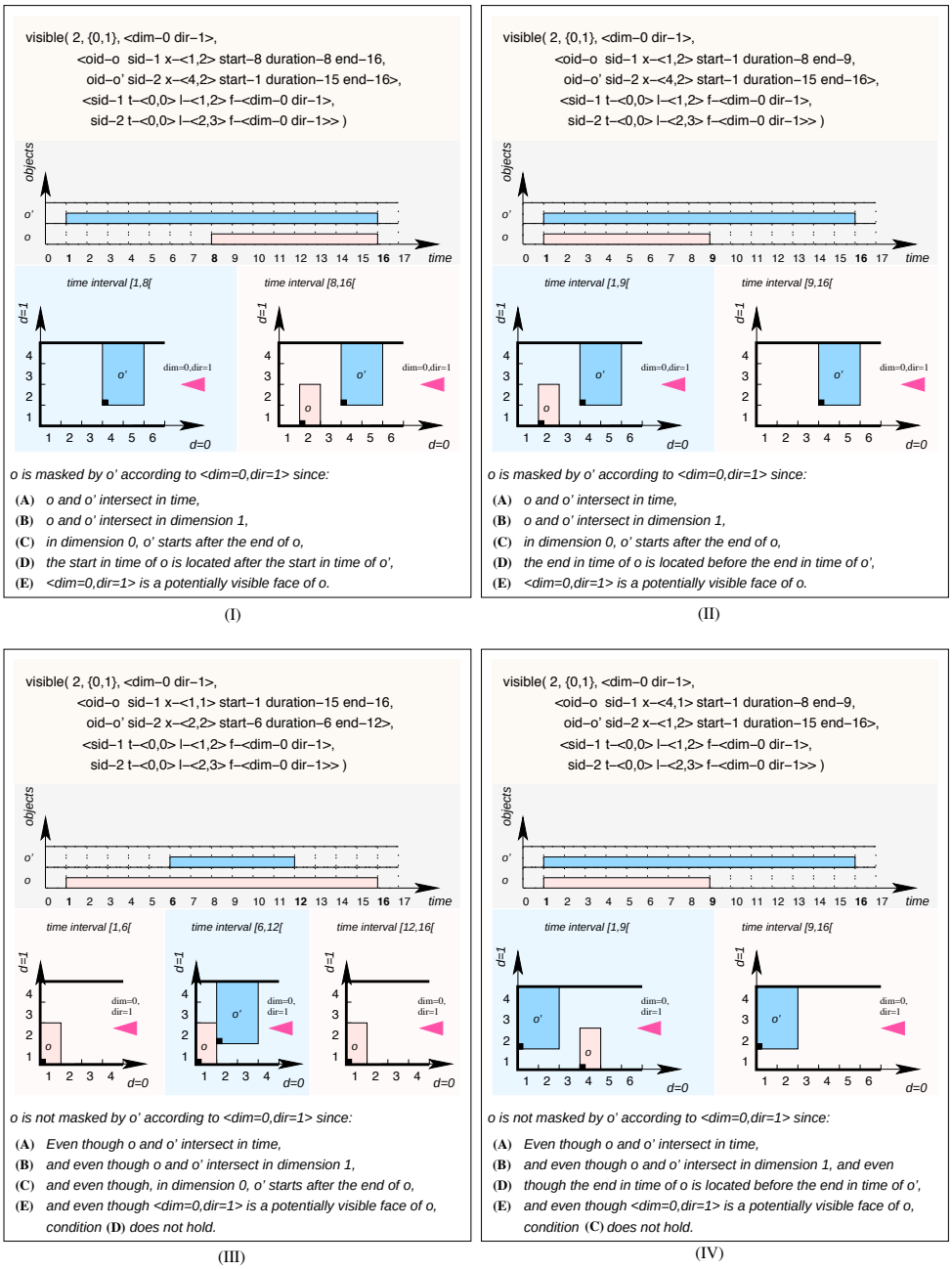


Figure 4.564: Illustration of Definition 1: (I,II) the case where an object o is masked by an object o' according to dimensions $\{0, 1\}$ and to the observation place $\langle \texttt{dim} = 0, \texttt{dir} = 1 \rangle$ because **(A)** o and o' intersect in time, **(B)** o and o' intersect in dimension 1, **(C)** o and o' are not well ordered according to the observation place, **(D)** there exists an instant where o' if present (but not o) and **(E)** $\langle \texttt{dim} = 0, \texttt{dir} = 1 \rangle$ is a potentially visible face of o; (III,IV) the case where an object o is not masked by an object o' according to the observation place $\langle \texttt{dim} = 0, \texttt{dir} = 1 \rangle$.

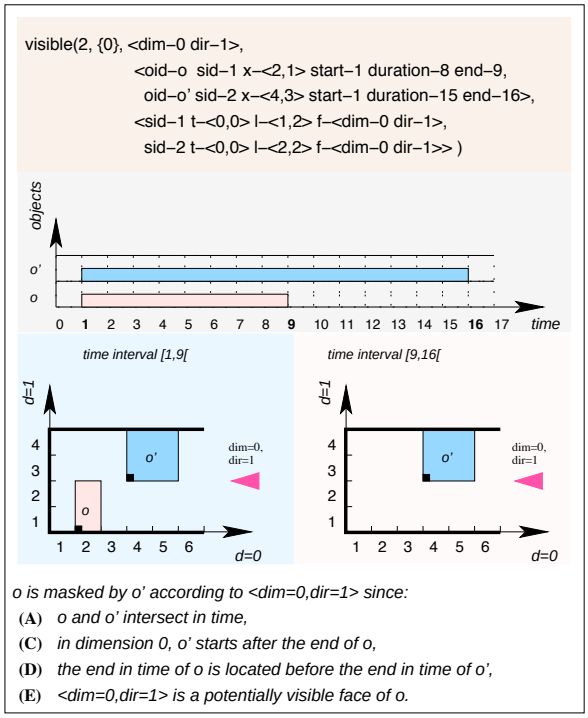


Figure 4.565: Illustration of Definition 1: the case where an object o is masked by an object o' according to dimension 0 and to the observation place $\langle \texttt{dim} = 0, \texttt{dir} = 1 \rangle$ because: **(A)** o and o' intersect in time, **(C)** o and o' are not well ordered according to the observation place and **(D)** there exists an instant where o' if present (but not o) and **(E)** $\langle \texttt{dim} = 0, \texttt{dir} = 1 \rangle$ is a potentially visible face of o.

We are now in position to define the `visible` constraint.

Definition 5. *Given a* `visible(K, DIMS, FROM, OBJECTS, SBOXES)` *constraint, the* `visible` *constraint holds if none of the objects of* `OBJECTS` *is masked according to the dimensions of* `DIMS` *and to the set of possible observation places defined by* `FROM`.

Usage

We now give several typical concrete uses of the `visible` constraint, which all mention the `diffst` as well as the `visible` constraints:

- Figure 4.566 corresponds to a *ship loading problem* where containers are piled within a ship by a crane each time the ship visits a given harbour. In this context we have first to express the fact that *a container can only be placed on top of an already placed container* and second, that *a container can only be taken away if no container is placed on top of it*. These two conditions are expressed by one single `visible` constraint for which the `DIMS` parameter mentions all three dimensions of the placement space and the `FROM` parameter mentions the pair $\langle \texttt{dim} = 2, \texttt{dir} = 1 \rangle$ as its unique observation place. In addition we also use a `diffst` constraint for expressing non-overlapping.

visible(3, {0,1,2}, <dim−2 dir−1>,
<oid−1 sid−1 x−<1,1,1> start−0 duration−17 end−17,
oid−2 sid−1 x−<1,1,3> start−0 duration−8 end−8 ,
oid−3 sid−1 x−<4,1,1> start−0 duration−8 end−8 ,
oid−4 sid−1 x−<1,1,3> start−8 duration−9 end−17,
oid−5 sid−1 x−<4,1,1> start−8 duration−16 end−24,
oid−6 sid−1 x−<1,1,1> start−17 duration−7 end−24>
<sid−1 t−<0,0,0> l−<2,4,2> f−<dim−2 dir−1>>)

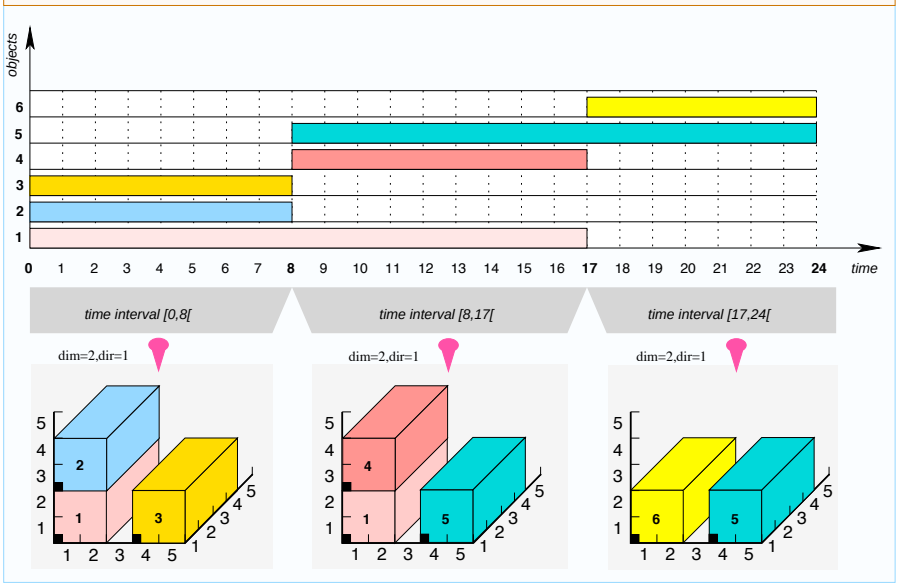


Figure 4.566: Illustration of the ship loading problem

- Figure 4.567 corresponds to a *container loading/unloading problem* in the context of a pick-up delivery problem where the loading/unloading takes place with respect to the front door of the container. Beside the `diffst` constraint used for expressing non-overlapping, we use two distinct `visible` constraints:

- The first `visible` constraint takes care of the location of the front door of the container (each object o has to be loaded/unloaded without moving around any other object, i.e., objects that are in the vis-à-vis of o according to the front door of the container). This is expressed by one single `visible` constraint for which the `DIMS` parameter mentions all three dimensions of the placement space and the `FROM` parameter mentions the pair $\langle \texttt{dim} = 1, \texttt{dir} = 0 \rangle$ as its unique observation place.
- The second `visible` constraint takes care of the *gravity dimension* (i.e., each object that has to be loaded should not be put under another object, and reciprocally each object that has to be unloaded should not be located under another object). This is expressed by the same `visible` constraint that was used for the ship loading problem, i.e., a `visible` constraint for which the `DIMS` parameter mentions all three dimensions of the placement space and the `FROM` parameter mentions the pair $\langle \texttt{dim} = 2, \texttt{dir} = 1 \rangle$ as its unique observation place.

- Figure 4.568 corresponds to a *pallet loading problem* where one has to place six objects on a pallet. Each object corresponds to a parallelepiped that has a bar code on one of its four sides (i.e., the sides that are different from the top and the bottom of the parallelepiped). If, for some reason, an object has no bar code then we simply remove it from the objects that will be passed to the `visible` constraint: this is for instance the case of the sixth object. In this context the constraint to enforce (beside the non-overlapping constraint between the parallelepipeds that are assigned to a same pallet) is the fact that the bar code of each object should be visible (i.e., visible from one of the four sides of the pallet). This is expressed by the `visible` constraint given in Part (F) of Figure 4.568.

Remark The `visible` constraint is a generalisation of the `accessibility` constraint initially introduced in the context of the `diffn` constraint.

See also `diffst`, `diffn`, `non_overlap_sboxes`.

Key words **constraint type:** decomposition.

filtering: sweep.

geometry: geometrical constraint.

```
visible(3, {0,1,2}, <dim−1 dir−0>,
          <oid−1 sid−1 x−<1,2,3> start−0   duration−8   end−8,
           oid−2 sid−2 x−<1,3,3> start−0   duration−8   end−8,
           oid−3 sid−3 x−<1,1,1> start−0   duration−17 end−17,
           oid−4 sid−4 x−<4,1,1> start−0   duration−17 end−17,
           oid−5 sid−5 x−<1,2,3> start−8   duration−9   end−17,
           oid−6 sid−6 x−<3,1,1> start−8   duration−12 end−24,
           oid−7 sid−3 x−<1,1,1> start−17 duration−7   end−24>,
          <sid−1 t−<0,0,0> l−<2,1,1> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−2 t−<0,0,0> l−<2,2,2> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−3 t−<0,0,0> l−<2,4,2> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−4 t−<0,0,0> l−<2,4,1> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−5 t−<0,0,0> l−<2,3,1> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−6 t−<0,0,0> l−<1,2,2> f−<dim−1 dir−0, dim−2 dir−1>> )

visible(3, {0,1,2}, <dim−2 dir−1>,
          <oid−1 sid−1 x−<1,2,3> start−0   duration−8   end−8,
           oid−2 sid−2 x−<1,3,3> start−0   duration−8   end−8,
           oid−3 sid−3 x−<1,1,1> start−0   duration−17 end−17,
           oid−4 sid−4 x−<4,1,1> start−0   duration−17 end−17,
           oid−5 sid−5 x−<1,2,3> start−8   duration−9   end−17,
           oid−6 sid−6 x−<3,1,1> start−8   duration−12 end−24,
           oid−7 sid−3 x−<1,1,1> start−17 duration−7   end−24>,
          <sid−1 t−<0,0,0> l−<2,1,1> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−2 t−<0,0,0> l−<2,2,2> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−3 t−<0,0,0> l−<2,4,2> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−4 t−<0,0,0> l−<2,4,1> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−5 t−<0,0,0> l−<2,3,1> f−<dim−1 dir−0, dim−2 dir−1>,
           sid−6 t−<0,0,0> l−<1,2,2> f−<dim−1 dir−0, dim−2 dir−1>> )
```

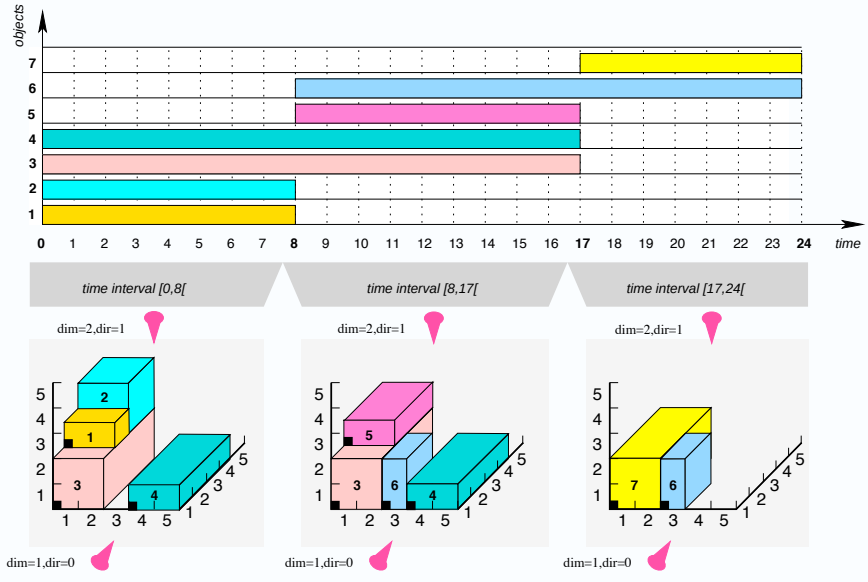


Figure 4.567: Illustration of the pick-up delivery problem

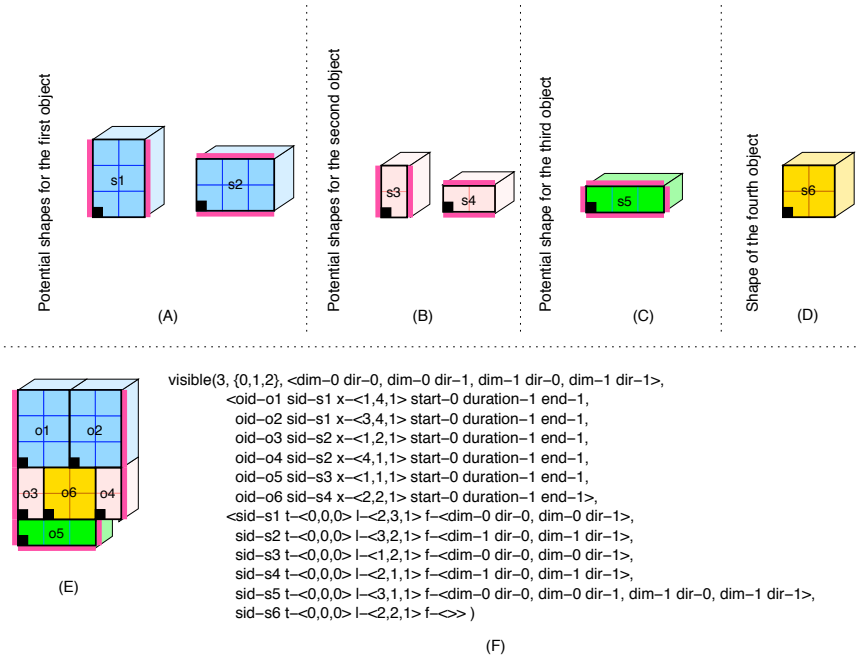


Figure 4.568: Illustration of the pallet loading problem

A Generic Geometrical Constraint Kernel in Space and Time for Handling Polymorphic k-Dimensional Objects

N. Beldiceanu[1], M. Carlsson[2], E. Poder[1], R. Sadek[1], and C. Truchet[3]

[1] École des Mines de Nantes, LINA FRE CNRS 2729, FR-44307 Nantes, France
{Nicolas.Beldiceanu,Emmanuel.Poder,Rida.Sadek}@emn.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se
[3] Université de Nantes, LINA FRE CNRS 2729, FR-44322 Nantes, France
Charlotte.Truchet@univ-nantes.fr

Abstract. This paper introduces a geometrical constraint kernel for handling the location in space and time of polymorphic k-dimensional objects subject to various geometrical and time constraints. The constraint kernel is generic in the sense that one of its parameters is a set of constraints on subsets of the objects. These constraints are handled globally by the kernel.
We first illustrate how to model several placement problems with the constraint kernel. We then explain how new constraints can be introduced and plugged into the kernel. Based on these interfaces, we develop a generic k-dimensional lexicographic sweep algorithm for filtering the attributes of an object (i.e., its shape and the coordinates of its origin as well as its start, duration and end in time) according to all constraints where the object occurs. Experiments involving up to hundreds of thousands of objects and 1 million integer variables are provided in 2, 3 and 4 dimensions, both for simple shapes (i.e., rectangles, parallelepipeds) and for more complex shapes.

1 Introduction and Presentation of the Kernel

This paper introduces a constraint kernel $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ for handling in a generic way a variety of geometrical constraints $\mathcal{C}$ in space and time between polymorphic k-dimensional objects $\mathcal{O}$ ($k \in \mathbb{N}^+$), where each object takes a shape among a set of shapes described by $\mathcal{S}$ during a given time interval and at a given position in space. This line of research can be seen as a continuation and generalisation of previous work on non-overlapping parallelepipeds [1–4].

Each shape is defined as a finite set of shifted boxes, where each shifted box is described by a box in a k-dimensional space at a given offset (from the origin of the shape) with given sizes. More precisely, a *shifted box* $s = \mathrm{sbox}(sid, t[], l[]) \in \mathcal{S}$ is an entity defined by its shape id $s.sid$, shift offset $s.t[d]$, $0 \leq d < k$, and sizes $s.l[d]$ ($s.l[d] > 0, 0 \leq d < k$). All attributes of a shifted box are integer values. Then, a *shape* is defined as the union of shifted boxes sharing the same shape id. Each *object* $o = \mathrm{object}(id, sid, x[], start, duration, end) \in \mathcal{O}$ is an entity defined by its unique object id $o.id$, shape id $o.sid$, origin $o.x[d]$, $0 \leq d < k$, start in time $o.start$, duration in time

$o.duration$ ($o.duration \geq 0$) and end in time $o.end$.[1] All these attributes correspond to domain variables.[2] Typical constraints from the list of constraints $\mathcal{C}$ can express, for instance, the fact that a given subset of objects from $\mathcal{O}$ do not pairwise overlap or that they are all included within a given bounding box. Constraints always have two first arguments $\mathcal{A}_i$ and $\mathcal{O}_i$ (followed by possibly some additional arguments) which resp. specify:

- A list of distinct dimensions (integers in $[0, k-1]$) that the constraint considers.
- A list of identifiers of the objects to which the constraint applies.

Example 1. Assume we have a 3D placement problem (i.e., $k = 3$) involving a set of parallelepipeds $\mathcal{P}$ and one subset $\mathcal{P}'$ of $\mathcal{P}$, where we want to express the fact that (1) no parallelepipeds of $\mathcal{P}$ should overlap, and (2) no parallelepipeds of $\mathcal{P}'$ should be piled. Constraints (1) and (2) resp. correspond to *non-overlapping*$([0,1,2], \mathcal{P})$ and to *non-overlapping*$([0,1], \mathcal{P}')$. Within the first *non-overlapping* constraint, the argument $[0,1,2]$ expresses the fact that we consider a non-overlapping constraint according to dimensions 0, 1 and 2 (i.e., given any pair of parallelepipeds p' and p'' of $\mathcal{P}$ there should exist at least one dimension d ($d \in \{0,1,2\}$) where the projections of p' and p'' on d do not overlap). Similarly, the argument $[0,1]$ of the second non-overlapping constraint expresses the fact that, given any pair of parallelepipeds p' and p'' of $\mathcal{P}'$, there should exist at least one dimension d ($d \in \{0,1\}$) where p' and p'' do not overlap).

$geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ is defined in the following way: given a constraint $ctr_i(\mathcal{A}_i, \mathcal{O}_i)$ from the list of constraints $\mathcal{C}$ between a subset of objects $\mathcal{O}_i \subseteq \mathcal{O}$ according to the attributes $\mathcal{A}_i$, let $\mathcal{MC}_i$ denote the sets of cliques stemming from the objects of $\mathcal{O}_i$ that all overlap in time.[3] The constraints of $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ hold if and only if $\forall ctr_i \in \mathcal{C}$, $\forall \mathcal{O}_{\mathcal{MC}_i} \in \mathcal{MC}_i : ctr_i(\mathcal{A}_i, \mathcal{O}_{\mathcal{MC}_i})$ holds.

Example 2. Fig. 1 presents a typical example of a dynamic 2D placement problem where one has to place four objects, in time and within a given box, so that objects that overlap in time do not overlap. Parts (A), (B), (C) and (D) resp. represent the potential shapes associated with the four objects to place, where the origin of each object is represented by a black square ■. Part (E) shows the position of the four objects of the example as the time varies, where the first, second, third and fourth objects were resp. assigned shapes S_1, S_5, S_8 and S_9:

- During the first time interval $[2, 9]$ we have only object O_1 at position $(1, 2)$.
- Then, at instant 10 objects O_2 and O_3 both appear. Their origins are resp. placed at positions $(2, 1)$ and $(4, 1)$.
- At instant 14 object O_1 disappears and is replaced by object O_4. The origin of O_4 is fixed at position $(1, 1)$. Finally, at instant 22 all three objects O_2, O_3 and O_4 disappear.

The corresponding arguments are:

[1] The time dimension is treated specially since the $duration$ attribute may not be fixed, which is not the case for the sizes of a shifted box. Also, the geometrical constraints only apply on objects that intersect in time.

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

[3] In fact, these cliques (of an interval graph) are only used for defining the declarative semantics of $geost$'s constraints.

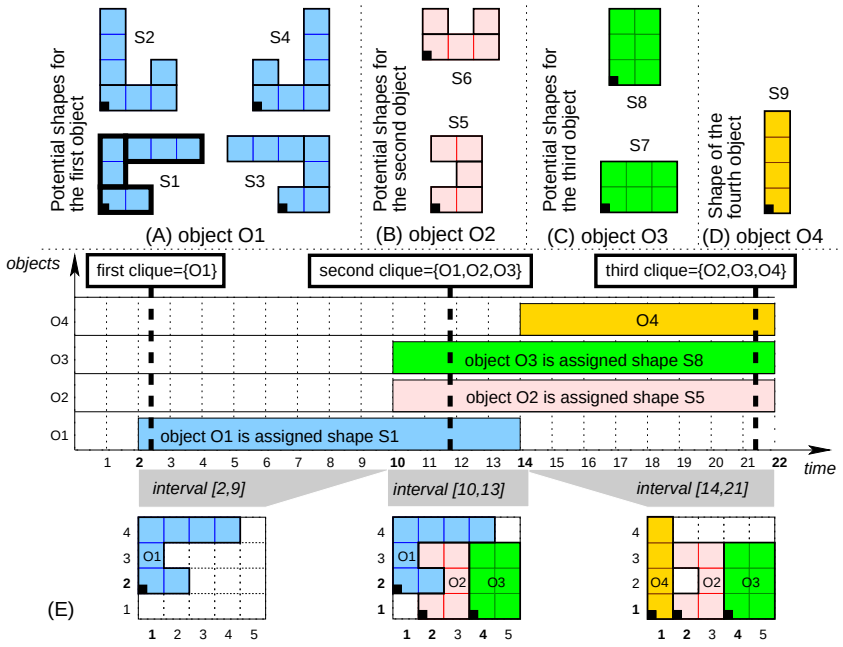


Fig. 1. Example with 4 objects, 9 shapes, one *non-overlapping* and one *included* constraints

```
01 geost(2,
02       [object(1,1,[1,2], 2,12,14), object(2,5,[2,1],10,12,22),
03        object(3,8,[4,1],10,12,22), object(4,9,[1,1],14, 8,22)],
04       [sbox(1,[0,0],[2,1]), sbox(1,[0,1],[1,2]), sbox(1,[1,2],[3,1]),
05        sbox(2,[0,0],[3,1]), sbox(2,[0,1],[1,3]), sbox(2,[2,1],[1,1]),
06        sbox(3,[0,0],[2,1]), sbox(3,[1,1],[1,2]), sbox(3,[2,2],[3,1]),
07        sbox(4,[0,0],[3,1]), sbox(4,[0,1],[1,1]), sbox(4,[2,1],[1,3]),
08        sbox(5,[0,0],[2,1]), sbox(5,[1,1],[1,1]), sbox(5,[0,2],[2,1]),
09        sbox(6,[0,0],[3,1]), sbox(6,[0,1],[1,1]), sbox(6,[2,1],[1,1]),
10        sbox(7,[0,0],[3,2]), sbox(8,[0,0],[2,3]), sbox(9,[0,0],[1,4])],
11       [non-overlapping([0,1],[1,2,3,4]),included([0,1],[1,2,3,4],[1,1],[5,4])])
```

Its first argument 2 is the number of dimensions of the placement space we consider. Its second and third arguments resp. describe the four objects and the shifted boxes of the nine shapes we have. For instance, the 3 boxes of shape S_1 (depicted by 3 thick rectangles in Part (A) of Fig. 1) respectively correspond to the 3 boxes declared at line 04 of the example. Finally, its last argument gives the list of geometrical constraints imposed by $geost$: the first constraint expresses a non-overlapping constraint between the four objects, while the second constraint imposes the four objects to be located within the box containing all points (x, y) such that $1 \leq x \leq 1+5-1$ and $1 \leq y \leq 1+4-1$. The constraints of $geost$ hold since the four objects do not simultaneously overlap in time and in space and since they are completely included within the previous box (i.e., see Part (E) of Fig. 1).

Within the scope of $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$, this paper presents a filtering algorithm that prunes the domain of each attribute of every object $o = \text{object}(id, sid, x[], start, duration, end) \in \mathcal{O}$. All values found infeasible are deleted from the shape attribute sid; for the other attributes (i.e., the origin $x[]$, the start, the duration and the end), the minimum and maximum are adjusted.

The paper is organised as follows. Section 2 provides an overview of placement problems that can be modelled with the constraints currently available in $geost$. Section 3 presents the overall architecture of the geometrical kernel. It explains how to define geometrical constraints in terms of a programming interface by the geometrical kernel. Section 4 focusses on the main contribution of this paper: a multi-dimensional lexicographic sweep algorithm used for filtering the attributes of an object of $geost$. Section 5 evaluates the scalability of the $geost$ kernel as well as its ability to deal with a variety of specific placement problems. Before we conclude, Section 6 compares $geost$ with related work and suggests future directions.

2 Modelling Problems with $geost$

As illustrated by Fig. 2 in the context of *non-overlapping*, $geost$ allows to model directly a large number of placement problems:

- Case (A) corresponds to a non-overlapping constraint among three segments.
- The second and third cases (B,C) correspond to a non-overlapping constraint between rectangles where (B) is a special case where the sizes of all rectangles in the second dimension are equal to 1; this can be interpreted as a *machine assignment problem*.
- Case (D) corresponds to a non-overlapping constraint between rectangles where each rectangle can have two orientations. This is achieved by associating with each rectangle two shapes of respective sizes $l \times h$ and $h \times l$. Since their orientation is not initially fixed, the $included$ constraint enforces the three rectangles to be included within the bounding box defined by the origin's coordinates $1, 1$ and sizes $8, 3$.
- Case (E) corresponds to a non-overlapping constraint between more complex objects where each object is described by a given set of rectangles.
- Case (F) describes a placement problem where one has to first assign each rectangle to a strip so that all rectangles that are assigned to the same strip do not overlap.
- Case (G) corresponds to a non-overlapping constraint between parallelepipeds.
- Case (H) can be interpreted as a non-overlapping constraint between parallelepipeds that are assigned to the same container. The first dimension corresponds to the identifier of the container, while the next three dimensions are associated with the position of a parallelepiped inside a container.
- Case (I) describes a rectangle placement problem over three consecutive time-slots: rectangles assigned to the same time-slot should not overlap in time. We initially start with the three rectangles 1, 2 and 3. Rectangle 3 is no longer present at instant 2 (the triangle ▼ within rectangle 3 at time 1 indicates that rectangle 3 will disappear at the next time-point), while rectangle 4 appears at instant 2 (the triangle ▲ within rectangle 4 at time 2 denotes the fact that the rectangle 4 appears at instant 2). Finally, rectangle 2 disappears at instant 3 and is replaced by rectangle 5.

3 Standard Representation of Geometrical Constraints

The key idea for handling multiple geometrical constraints in a common kernel is the following. For each type of geometrical constraint found in $\mathcal{C}$ (also called *exter-*

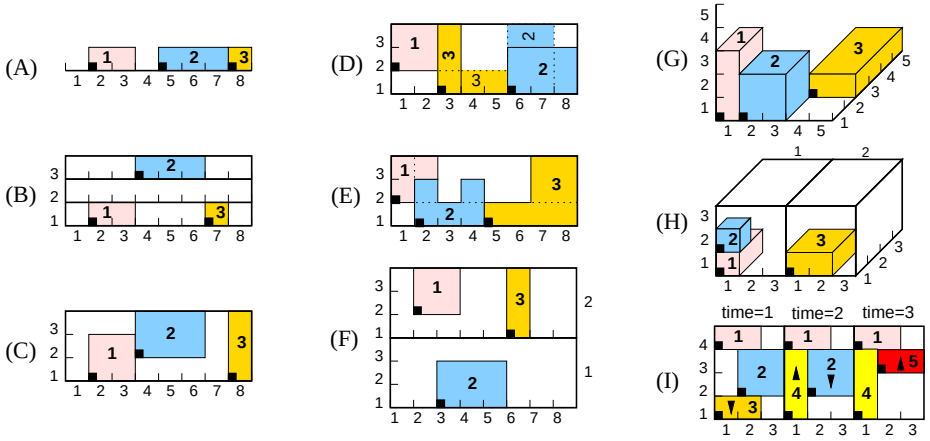


Fig. 2. Nine typical examples of use of *geost*

nal constraints), one has to provide a service that computes necessary conditions (also called *internal* constraints) for a given object and shape. Given an external geometrical constraint $ectr_i(\mathcal{A}_i, \mathcal{O}_i)$ $(\mathcal{A}_i \subseteq \{0, 1, \ldots, k-1\}, \mathcal{O}_i \subseteq \mathcal{O})$, one of its object $o \in \mathcal{O}_i$ and one potential shape s of o, such a necessary condition generated by $ectr_i$, o and s is a unary[4] constraint $ictr(o.x)$ such that: $o.sid = s \wedge ectr_i(\mathcal{A}_i, \mathcal{O}_i) \Rightarrow ictr(o.x)$. Now, the key to being able to globally treat such necessary conditions in the kernel is to give them a uniform representation. We have chosen the following one:

– A constraint $\mathrm{outbox}(t, l)$ on $o.x$ holds iff $o.x$ is located outside the shifted box defined by its origins point $t[d]$, $0 \leq d < k$, and sizes $l[d]$, $0 \leq d < k$ (i.e., $\exists d \in [0, k-1] \mid o.x[d] < t[d] \vee o.x[d] > t[d] + l[d] - 1$).

Thus, an outbox corresponds to a box-shaped set of points that are infeasible for $o.x$. The purpose of the introduction of outboxes is to have a common representation for the kernel, suitable for the k-dimensional lexicographic sweep algorithm presented in the next section, which considers all the outboxes, for a selected object and shape, in one run.

Consequently, for each type of external geometrical constraint, found in $\mathcal{C}$ a service $\mathrm{GenOutboxes}(ectr_i, o, s) : (ictrs)$, responsible for generating outboxes, must be provided. This service is assumed to generate outboxes that intersect the domains of the origin coordinates of o. Also, if all attributes mentioned by $ectr_i$ belonging to objects other than o are fixed, those outboxes are assumed to be necessary and sufficient conditions, lest the kernel accept false solutions.

Example of External Geometrical Constraints. We now illustrate some external geometrical constraints that are currently available within the constraint kernel. As we saw in the introduction, an external constraint always has at least two arguments that resp.

[4] Unary, since it involves the k coordinates of a *single* object.

correspond to a list of distinct dimensions and to a list of object identifiers to which the constraint apply.

The $included$ **and** *non-overlapping* **external constraints.** The $included(\mathcal{A}_i, \mathcal{O}_i, t, l)$ and the *non-overlapping*$(\mathcal{A}_i, \mathcal{O}_i)$ external constraints take as input a list of distinct dimensions $\mathcal{A}_i$ in $\{0, 1, \ldots, k-1\}$ and a list $\mathcal{O}_i$ of distinct object identifiers of $geost$. In addition, the $included$ constraint considers a shifted box defined by its origin point $t[d]$, $0 \leq d < k$, and size $l[d]$, $0 \leq d < k$.

The $included$ constraint enforces for each object o (with $o.id \in \mathcal{O}_i$) and for any corresponding shifted box s (with $o.sid = s.sid$) the condition $\forall d \in \mathcal{A}_i \mid t[d] \leq o.x[d] + s.t[d] \wedge o.x[d] + s.t[d] + s.l[d] - 1 \leq t[d] + l[d] - 1$ (i.e., s is included within the shifted box attribute defined by the parameters t and l of the $included$ constraint). Depending on which shape of an object we actually consider, the $included$ constraint can be translated to $2k$ outbox constraints.

The *non-overlapping* constraint enforces the following condition: given two distinct objects o and o' (with $o.id, o'.id \in \mathcal{O}_i$) that overlap in time, no shifted box s (with $o.sid = s.sid$) should overlap any shifted box s' (with $o'.sid = s'.sid$); i.e. it should hold that $\exists d \in \mathcal{A}_i \mid o.x[d] + s.t[d] + s.l[d] \leq o'.x[d] + s'.t[d] \vee o'.x[d] + s'.t[d] + s'.l[d] \leq o.x[d] + s.t[d]$ (i.e., there exists a dimension where they do not intersect). While focussing on an object o we can easily generate an outbox constraint for each object o' that should not overlap o by reusing the results of [2].

4 The Geometrical Kernel: a Generic k-dimensional Lexicographic Sweep Algorithm

In this section, we first present the sweep algorithm used for filtering the coordinates of the origin of an object o of $geost$ when each object has one single shape. We initially assume that time is treated exactly like the space dimensions, i.e. that the $o.x$ array is extended by one element. Toward the end of this section, we explain in detail how to treat the time attributes of an object. We also assume for now that the shape attribute is fixed, and explain later how to handle multiple potential shapes for an object (i.e., polymorphism). We now introduce some notation used throughout this section.

Notation. Assume v and w are vectors of scalars of k components. Then $v \leftarrow w$ denotes the element-wise assignment of w to v, $w + d$ (resp. $w - d$) denotes the element-wise addition of d (resp. $-d$) to w. Given a scalar d, $0 \leq d \leq k-1$, $\mathrm{rot}(v, d, k)$ denotes the vector $(v[d], v[(d+1) \mod k], \ldots, v[(d-1) \mod k])$. That is, in the rotated vector, $v[d]$ is the most significant element, which is what we need when running the sweep algorithm on dimension d.

The Sweep Algorithm. This algorithm first considers all outboxes $\mathcal{IC}_o$ derived from $\mathcal{C}$ where object o actually appears, and then performs a recursive traversal of the placement space for each coordinate and direction (i.e., $\min$ or $\max$). Without loss of generality, assume we want to adjust the minimum value of the d^{th} coordinate $o.x[d]$,

$0 \leq d < k$, of the origin of o. The algorithm starts its recursive traversal of the placement space at point $c = \mathrm{rot}(\underline{o.x}, d, k)$ and could in principle explore all points of the domains of $o.x$, one by one, in increasing lexicographic order, until a point is found that is not inside any outbox, in which case $c[0]$ is the computed new minimum value. To make the search efficient, instead of moving each time to the successor point, we arrange the search so that it skips points that are known to be inside some outbox.[5]

Thus, we compute the lexicographically smallest point c' such that:

1. c' is lexicographically greater than or equal to c,
2. every element of c' is in the domain of the corresponding element of $o.x$,
3. c' is not inside any outbox of $\mathcal{IC}_o$.

If no such c' exists, the constraint fails. Otherwise, the minimum value of $o.x[d]$ is adjusted to $c'[0]$. As we saw, the sweep algorithm moves in increasing lexicographic order a point c from its lexicographically smallest potential feasible position to its lexicographically largest potential feasible position through all potential points. The algorithm uses the following data structures:

– The current position c of the sweep.
– A vector $n[0..k-1]$ that records knowledge about already encountered sets of infeasible points while moving c from its first potential feasible position. The vector n is always element-wise greater than c and maintained as follows. Let $\mathrm{inf}, \mathrm{sup}$ denote the vectors $\mathrm{inf} = \mathrm{rot}(\underline{o.x}, d, k)$ and $\mathrm{sup} = \mathrm{rot}(\overline{o.x} + 1, d, k)$:
  - Initially, $n = \mathrm{sup}$.
  - Whenever an outbox f containing c is found, n is updated by taking the element-wise minimal value of n and the upper boundary of $\mathrm{rot}(f, d, k)$, indicating 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 the corresponding values of sup.

  The following invariant holds for the vector n, and is used when advancing c to the next candidate point. Let i be the smallest j such that $n[j+1] = \mathrm{sup}[j+1] \wedge \cdots \wedge n[k-1] = \mathrm{sup}[k-1]$ and suppose c is known to be in some outbox. Then, the next point, lexicographically greater than c and not yet known to be in any outbox, is $(c[0], \ldots, c[i-1], n[i], \mathrm{inf}[i+1], \ldots, \mathrm{inf}[k-1])$.

Algorithm 1 implement this idea. The algorithm prunes the bounds of each coordinate of every object wrt. its relevant outboxes, iterating to fix-point.

Efficiency. The main inefficiency in this sweep algorithm lies in searching the set of outboxes (line 4 of $\mathrm{PruneMin}$). In order to make this search more efficient, we can make the sweep algorithm more sophisticated by the following modifications to $\mathrm{PruneMin}$:

– We extend the state of the algorithm by an *event point series*, ordered in lexicographically increasing order. These events correspond to the lexicographically smallest (insert events) and largest (delete events) relevant infeasible point associated with each outbox $ictr_o \in \mathcal{IC}_o$. They are sorted in lexicographically increasing order, and we maintain a pointer into the series in sync with point c.

[5] Potential holes in the domains are reflected in outboxes.

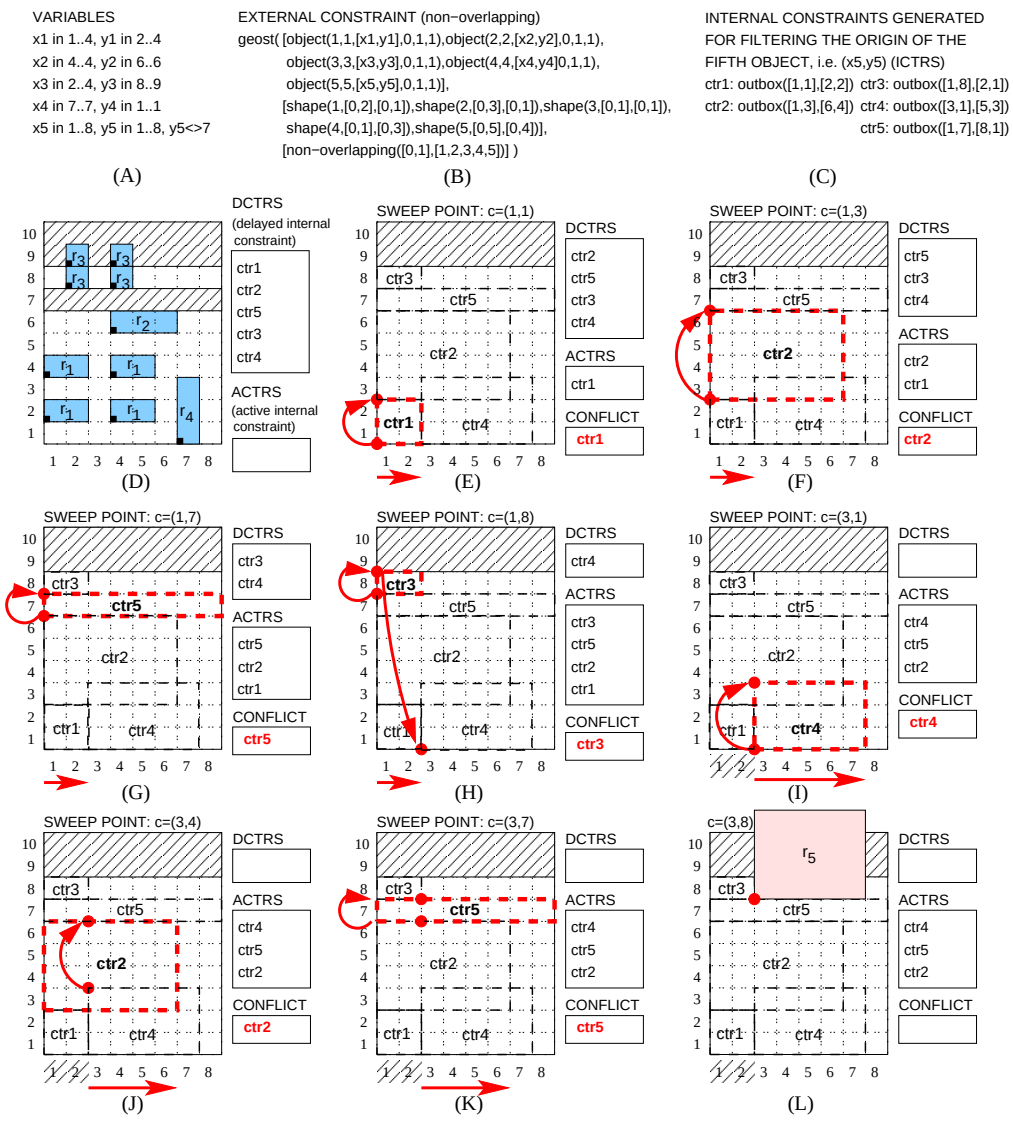


Fig. 3. Illustration of the lexicographic sweep algorithm for adjusting the minimum value of the abscissa of rectangle r_5

```
PROCEDURE FilterCtrs(k, O, S, C) : bool
 1: nonfix ← true                                              // fixpoint not yet reached
 2: while nonfix do
 3:    nonfix ← false                                          // assumes no filtering will be done
 4:    for all o ∈ O do
 5:       I ← ⋃_{e∈C} GenOutboxes(e, o, o.sid)                 // build the set of outboxes on o
 6:       I ← I ∪ ⋃_{0≤d<k} possible outboxes corresponding to holes in o.x[d]
 7:       for d ← 0 to k − 1 do
 8:          if ¬PruneMin(o, d, k, I) ∨ ¬PruneMax(o, d, k, I) then
 9:             return false                                   // no feasible origin
10:          else if o.x was pruned then
11:             nonfix ← true                                // fixpoint not yet reached
12:          end if
13:       end for
14:    end for
15: end while
16: return true                                               // feasible origin
PROCEDURE PruneMin(o, d, k, I) : bool
 1: b ← true                                                  // b = true while we have not failed
 2: c ← o.x                                                   // initial position of the point
 3: n ← o.x + 1                                               // upper limits+1 in the different dimensions
 4: while b ∧ ∃f ∈ I | c ∈ f do
 5:    n ← min(n, f.t + f.l)                                  // update vector n according to an outbox f containing c
 6:    b ← false                                              // no new point to jump to yet
 7:    for j ← k − 1 downto 0 do
 8:       j' ← (j + d) mod k                                  // rotation wrt. d, k
 9:       c[j'] ← n[j']                                       // use vector n to jump
10:       n[j'] ← o.x[j'] + 1                                 // reset component of n to maximum value
11:       if c[j'] ≤ o.x[j'] then
12:          b ← true                                         // jump target found
13:          j ← 0                                            // exit for loop
14:       else
15:          c[j'] ← o.x[j']                                  // reset component of c, for exhausted a dimension
16:       end if
17:    end for
18: end while
19: if b then
20:    o.x[d] ← max(o.x[d], c[d])
21: end if
22: return b
```

Algorithm 1: FilterCtrs is the main filtering algorithm associated with $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$, where k, $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{C}$ resp. correspond to the number of dimensions, to the objects, to the shapes and to the external geometrical constraints. PruneMin adjusts the lower bound of the d^{th} coordinate of the origin of object o where I is the set of outboxes associated with object o (since PruneMax is similar to PruneMin it is omitted). The given fixpoint loop is an over-simplification. The implementation maintains a set of objects that need filtering. Whenever an object o is pruned, all non-fixed objects connected to o by an external constraint are added to this set. When the set becomes empty, the fixpoint is reached.

– We maintain the set of *active outboxes*, corresponding to all outboxes $ictr_o \in \mathcal{IC}_o$ such that c is between its lexicographically smallest and largest infeasible points. This set is initially empty.
– When c is initialized in line 2 as well as when c is incremented in lines 7-17, the relevant events up to point c from the event point series are processed, and the corresponding outboxes are added to or deleted from the set of active outboxes.
– In line 4, only the active outboxes are considered.

Example 3. Fig. 3 illustrates the k-dimensional lexicographic sweep algorithm in the context of $k = 2$. Parts (A) and (B) provide the variables of the problem (i.e., the abscissa and ordinate of each rectangle r_1, r_2, r_3, r_4 and r_5) as well as the non-overlapping constraint between the five previous rectangles. On Part (D) we have represented the extreme possible feasible positions of each rectangle (i.e., rectangles r_1 to r_4): for instance the leftmost lower corner of rectangle r_1 can only be fixed at positions $(1, 2)$, $(1, 3)$, $(1, 4)$, $(2, 2)$, $(2, 3)$, $(2, 4)$, $(3, 2)$, $(3, 3)$, $(3, 4)$, $(4, 2)$, $(4, 3)$ and $(4, 4)$. Parts (C) to (L) of Fig. 3 detail the different steps of the algorithm for adjusting the minimum value of the abscissa of rectangle r_5. Part (C) provides the outboxes associated with the fact that we want to prune the coordinates of r_5: constraints ctr_1, ctr_2, ctr_3 and ctr_4 resp. correspond to the fact that rectangle r_5 should not overlap rectangles r_1, r_2, r_3 and r_4, while constraint ctr_5 represents the fact that the ordinate of r_5 should be different from 7. Part (D) represents the initialisation phase of the algorithm where we have all five outboxes with their respective lexicographically smallest infeasible point (i.e., $(1, 1)$ for ctr_1, $(1, 3)$ for ctr_2, $(1, 7)$ for ctr_5, $(1, 8)$ for ctr_3 and $(3, 1)$ for ctr_4). Part (E) represents the first step of the sweep algorithm where we start the traversal of the placement space at point $c = (1, 1)$. We first transfer to the list of active outboxes all outboxes for which the first lexicographically smallest infeasible point is lexicographically greater than or equal to the current position of the sweep $c = (1, 1)$ (i.e., constraint $ctr_1 = \mathrm{outbox}([1, 1], [2, 2])$). We then search through the list of active constraints (represented on the figure by a box with the legend ACTRS on top of it) the first constraint for which $c = (1, 1)$ is infeasible. In fact, since ctr_1 is infeasible (represented on the figure by a box with the legend CONFLICT on top of it) we compute the vector $f = (3, 3)$ that tells how to get the next potentially feasible point in the different dimensions. Consequently the sweep moves to the next position $(1, 3)$ (see Part (F)) and the process is repeated until we finally find a feasible point for all outboxes (i.e., point $(3, 8)$ in Part (L)). Note that, when the lexicographically largest infeasible point associated with an active outbox is lexicographically less than the current position of the sweep, we remove that constraint from the list of active outboxes. This is for instance the case in Part (I), where we remove constraint ctr_3 from the list of active outboxes (since its lexicographically largest infeasible point $(2, 8)$ is lexicographically less than the position of the sweep $c = (3, 1)$).

Complexity. Rather than analysing the complexity of the $geost$ kernel for a fixed k, which depends both on the type of each external constraint (i.e., the complexity of a given external constraint for generating all its corresponding outboxes as well as their number), we rather focus on $\mathrm{PruneMin}$ for adjusting the minimum value of the d^{th} coordinate of the origin of an object. Assuming that the maximum number of outboxes is equal to n we give an upper bound on the maximum number of jumps of $\mathrm{PruneMin}$ (i.e., the maximum number of times the sweep is moved).

First note that we always jump to an upper border (+1) of an outbox (i.e., see line 5 of $\mathrm{PruneMin}$) or that we reset some coordinates of the sweep to its minimum value (i.e., see line 15 of $\mathrm{PruneMin}$). Consequently, all coordinates of the sweep are always equal to an upper border (+1) of some outboxes or to a minimum possible value. Since we

want to evaluate the maximum number of jumps, let us assume that for every dimension d ($0 \leq d < k$) the upper limits of all the n outboxes are distinct. Having this in mind we can construct a maximum of $(n+1)^k$ points. Even if we found a systematic construction where this number of jumps is reached, the performance evaluation of Section 5 indicates that we can handle a reasonable number of objects for $k = 2, 3, 4$.

From a memory consumption point of view, the algorithm only records the coordinates of the sweep from one invocation to the next, in order not to restart the search from scratch (i.e., $2k$ points for each object).

Handling Time. Given an object $o \in \mathcal{O}$ of $geost$, the sweep algorithm that we have introduced in the previous section can be easily adapted to handle the start in time $o.start$, duration in time $o.duration$ and end in time $o.end$. Beside maintaining bound consistency for the constraint $o.end = o.start + o.duration$, we add an extra *time* dimension to the geometric coordinates of object o. Roughly, this new time coordinate corresponds to $o.start$ resp. $o.end$ depending on whether we are adjusting the minimum or maximum.

Handling Polymorphism. In order to handle the fact that objects can have several potential shapes we modify the previous algorithm in the following way. For adjusting the minimum value of the coordinate of the origin of an object that has more than one shape we call the sweep algorithm for each potential shape of the object (i.e., for each value of its shape variable). Then we take the smallest minimum value obtained (i.e., we use constructive disjunction) and prune the shape variable of an object if we did not find any feasible point for a given potential shape of that object.

Other Internal Constraints. The standard representation of geometrical constraints given in Section 3 is an over-simplification. For some constraints, e.g. distance constraints, outboxes are not a suitable representation, as the set of forbidden coordinates cannot be covered by a small number of boxes. Therefore, the constraint kernel internally handles other representations of necessary conditions, with an appropriate internal API. For details, see the technical report [5].

5 Performance Evaluation

We evaluate the implementation[6] of the $geost$ kernel from three perspectives:

Wanting to measure the speed and the scalability of the sweep algorithm for finding a first solution on loosely constrained placement problems (i.e., 20% spare space), we generated one set of random problem instances of m k-dimensional boxes for $k \in \{2, 3, 4\}$ involving $t \in \{1, 16, 256, 1024\}$ distinct types of boxes, and for $m \in \{1024, 2048, \ldots, 262144\}$. The results for $k = 2$ are shown in Fig. 4 (top left) and indicates that the approach is sensible to the number of distinct types of boxes. It can typically pack 1024 2D, 3D and 4D distinct boxes in at most 200 msec. The longest

[6] The experiments were run in SICStus Prolog 4 compiled with gcc -02 version 4.0.2 on a 3GHz Pentium IV with 1MB of cache.

time, 13694 seconds (close to 4 hours), was obtained for packing 262144 4D parallelepipeds (over 1 million domain variables) with a memory consumption of 351MB.

Wanting to get an idea of the performance of the $geost$ kernel on very tight placement problems (i.e., 0% spare space), we considered the *perfect squared squares problem* [1, 6] as well as the *3D pentominoes problem* [7]:

- A *perfect squared square of order* n is a square that can be tiled with n smaller squares where each of the smaller squares has a different integer size. We used the data available (i.e., the size of the small squares to pack) from the catalogue [8] and tested the corresponding 207 instances. The labelling strategy is roughly to repeat the following, first for the x dimension, then for the y dimension:
  1. Find the smallest position where some square can be placed.
  2. Find a square to place in that position.
- *Pentominoes* are pieces made of 5 connected unit cubes laid on a plane surface. Their shapes look like the 12 letters F, I, L, P, N, T, U, V, W, X, Y and Z. We considered the problem of finding the different ways of putting 12 distinct shapes that can be reflected and rotated in a box having a volume of 60 unit cubes. Our labelling strategy is roughly to repeat the following:
  1. Find a slot in the space that has not yet been filled by some piece.
  2. Find a piece that can fill that slot.

Fig. 4 (top right) and Table 1 respectively report, for the squared squares and the pentominoes problems, the time and number of backtracks for exploring all the search space[7] without breaking any symmetry. For the squared squares problems the maximum time of 1585 seconds was spent on problem 48; on the other hand, 148 problems were completely solved within 60 seconds. For the 3D pentomino packing instances, performance results for comparison can be found in [7]. However, they stop the search when the first 100 solutions have been found, so the results are only partly comparable.

Finally, wanting to compare the $geost$ kernel with a recent exact state of the art method for the 2D orthogonal packing problem [4], we reused the benchmarks proposed by Clautiaux et al. [9]. This is a feasibility problem which consists in determining whether a set of rectangles that cannot be rotated, can be packed or not into a rectangle of fixed size. In these instances the discrepancy between the sum of the areas of the rectangles to pack and the area of the big rectangle vary from 0% to 20%. We have 41 instances involving between 10 and 23 rectangles. Moreover, from these 41 instances, 26 instances are not feasible. In order to break symmetries between multiple rectangles of the same shape we added lexicographic ordering constraints. All x coordinates were labelled followed by all y coordinates, by decreasing rectangle size. Values were tried by increasing value. Fig. 4 (bottom) compares our results with the ones reported in [4]. Note that the sequence order for the curves differs, since the instances of each curve are ordered by increasing y value. We solved all instances and are comparable with [4], although 8 instances are much easier for [4] and 10 instances are much easier for us.

Note that for the last three problems (i.e., Squared Squares, Pentominoes and 2D orthogonal packing) extra filtering algorithms mostly based on cumulative relaxation were integrated within our kernel. Since this paper focusses on the constraint kernel and because of space limitations these methods were not detailed.

[7] Finding all solutions and proving that there is no other solution.

6 Related Work and Future Directions

The rectangles packing problem has been studied by Clautiaux et al. [9, 4] where scheduling-based reasoning is used [1]. The use of sweep algorithms in constraint filtering algorithms was introduced in [3] and applied to the non-overlapping 2D rectangles constraints. This paper generalizes and extends that work in several ways.

- The 2D sweep is generalized to a lexicographic sweep, independent of the number of dimensions.
- The notion of forbidden regions for non-overlapping rectangles is generalized to necessary conditions for general geometric constraints.

The idea of generating necessary conditions is reminiscent of indexicals [10], a.k.a. projection constraints [11]. An indexical for a constraint $c(x_1, \ldots, x_n)$ computes a unary constraint on a single variable x_i, i.e. a set S of values such that $c \Rightarrow x_i \in S$, in reaction to domain changes in $x_1, \ldots, x_n$. The constraint kernel then immediately enforces $x_i \in S$. Our kernel generalizes this in two ways:

- We compute necessary conditions in the form of k-dimensional forbidden regions.
- We treat all such forbidden regions, for a selected object and shape, in one run of the sweep algorithm. Projecting a single forbidden region on one coordinate often does not yield any pruning, whereas considering the union of forbidden regions is much more effective.

Dal Palù et al. in [12] proposed a constraint solver specialized for 3D discrete domains. Their solver was targeted to the study of problems in molecular, chemical and crystal structures. Our work, however, remains in the setting of mainstream finite domain constraint systems, whereas our kernel internally handles k-dimensional objects.

Even though the $geost$ kernel has been designed over discrete domains, it could rather easily be extended to continuous domains with the coordinates of the objects approximated by the floating-point numbers $\mathbb{F}$. Since switching from $\mathbb{N}$ to $\mathbb{F}$ may cause rounding errors at this level, the sweep algorithm needs to handle these rounding errors when moving the sweep out of an outbox constraint. If the projections of the forbidden regions on all dimensions are intervals of real bounds we can proceed as follows. On continuous domains, an outbox will have an very thin strip at the border where the feasibility of the corresponding internal constraint is unknown. The region inside this strip is strictly forbidden, and outside, the constraints is certainly satisfied. The outbox must be computed including this strip, by taking lower and upper approximations of the region's coordinates. In that case, the solutions are guaranteed to be valid, but the solver may not be complete, because it may (rarely) happen that the real forbidden region allows positions that are forbidden by its approximation.

This research was conducted under the European Union project "Net-WMS", a major task of which is to study packing problems in warehouse management. In this context, our constraint kernel is a step towards being able to capture a large set of packing rules in a constraint programming setting. Future work involves extending our set of external geometric constraints to include such packing rules.

7 Conclusion

The main contribution of this paper is a geometrical constraint kernel for handling the location in space and time of polymorphic k-dimensional objects subject to various geometrical and time constraints. The constraint kernel is generic in the sense that one of its parameters is a set of constraints on subsets of the objects. These constraints are handled globally by the kernel.

We have presented a sweep algorithm for filtering the attributes of the objects. Thank to its architecture, new geometric constraints can be plugged into this sweep algorithm without modifying it. The strong point of this sweep algorithm is that it considers all the geometrical constraints for a selected object and shape in one run. As a first result, more deduction can be performed by combining sets of forbidden points coming from multiple geometrical constraints. Secondly, it can handle within one single constraint problems involving up several tens of thousands of objects without memory consumption problems, which is often a weak point for constraint programming environment. We have also shown that we could handle tight 2D or 3D placement problems, which were traditionally solved by specific approaches.

Acknowledgements

This research was conducted under European Union Sixth Framework Programme Contract FP6-034691 “Net-WMS”.

configuration	backtracks (1st)	time (1st)	backtracks (all)	time (all)	solutions
$20 \times 3 \times 1$	1434	1840	47381	49740	8
$15 \times 4 \times 1$	290	560	888060	939060	1472
$12 \times 5 \times 1$	1594	1850	3994455	4112870	4040
$10 \times 6 \times 1$	111	260	9688985	10726810	9356
$10 \times 3 \times 2$	1267	2370	1203511	1778980	96
$6 \times 5 \times 2$	157	730	n/a	n/a	n/a
$5 \times 4 \times 3$	3567	14930	n/a	n/a	n/a

Table 1. Performance evaluation. 3D pentomino packing instances. Time in milliseconds. “n/a” corresponds to a quantity that was not available with a time-out of several hours.

References

1. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.
2. N. Beldiceanu, Q. Guo, and S. Thiel. Non-overlapping constraints between convex polytopes. In T. Walsh, editor, *Proc. CP'2001*, volume 2239 of *LNCS*, pages 392–407. Springer-Verlag, 2001.
3. N. Beldiceanu and M. Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraints. In T. Walsh, editor, *Proc. CP'2001*, volume 2239 of *LNCS*, pages 377–391. Springer-Verlag, 2001.

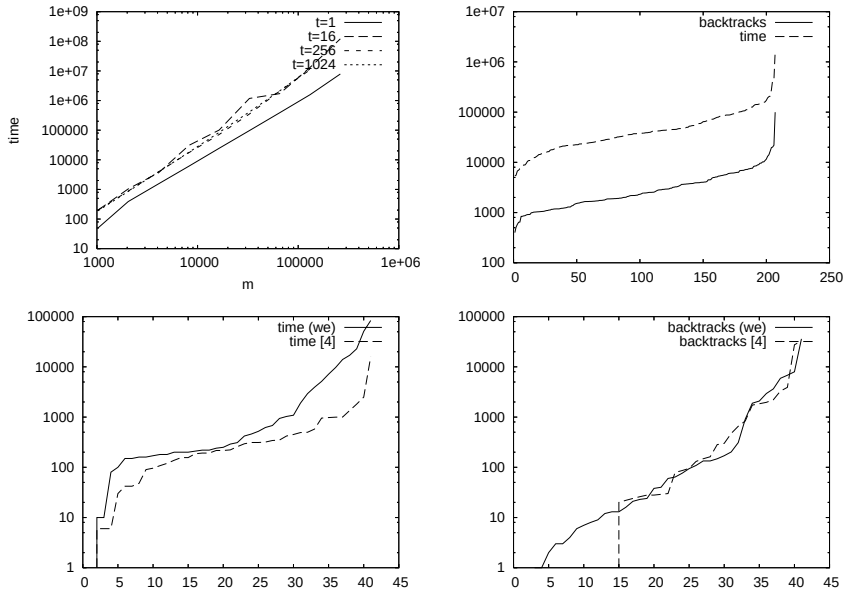


Fig. 4. Performance evaluation. Top left: scalability, $t \in \{1, 16, 256, 1024\}$. Top right: Perfect Squared Squares, runtime and backtracks. Bottom: 2D Orthogonal Packing, runtime (left) and backtracks (right). Time in milliseconds. In each curve, the instances are ordered by increasing y value.

4. F. Clautiaux, A. Jouglet, J. Carlier, and A. Moukrim. A new constraint programming approach for the orthogonal packing problem. *Computers and Operation Research*, to appear.
5. 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. SICS Technical Report T2007:08, Swedish Institute of Computer Science, 2007.
6. P. Van Hentenryck. Scheduling and packing in the constraint language cc(FD). In M. Zweben and M. Fox, editors, *Intelligent Scheduling*. Morgan Kaufmann Publishers, 1994.
7. A. Colmerauer and B. Gilleta. Solving the three-dimensional pentamino puzzle. Technical report, Laboratoire d'Informatique de Marseille, 1999. http://www.lim.univ-mrs.fr/ colmer/ArchivesPublications/Giletta/misc99.pdf.
8. C. J. Bouwkamp and A. J. W. Duijvestijn. Catalogue of simple perfect squared squares of orders 21 through 25. Technical Report EUT Report 92-WSK-03, Eindhoven University of Technology, The Netherlands, November 1992.
9. F. Clautiaux, J. Carlier, and A. Moukrim. A new exact method for the two-dimensional orthogonal packing problem. *European Journal of Operational Research*, to appear.
10. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Manuscript, 1991.
11. Gregory Sidebottom. *A Language for Optimizing Constraint Propagation*. PhD thesis, Simon Fraser University, 1993.
12. Alessandro Dal Pal'u, Agostino Dovier, and Enrico Pontelli. A new constraint solver for 3D lattices and its application to the protein folding problem. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005*, volume 3835 of *LNCS*, pages 48–63. Springer-Verlag, 2005.