Modelling and solving the Present Wrapping Problem with Constraint Programming

Cellini Lorenzo - 0000951921
lorenzo.cellini3@studio.unibo.it

November 4, 2020

This work describes a proposed solution to the Present Wrapping Problem in the framework of Contraint Programming (CP). Two models have been developed: the "basic" one addresses the easier version of the problem, where there is only one copy for each piece and rotation is not allowed; the "general" one addresses the more relaxed version of PWP problem, where rotation is allowed and there could also be repeated rectangles.

1 PROBLEM DESCRIPTION

1.1 The Present Wrapping Problem PWP

The present wrapping problem is a well known constraint optimization problem that can be described as follow: given an enclosing rectangle of area $w * h$ and a list of smaller rectangles i of given dimensions $dimension_{ij}$, where i is the rectangle index and $j \in \{1,2\}$ the $width$ or $heigth$, find a satisfying assignement for the rectangles position such that they cover the entire area. In the literature there exist several different formulation of the same problem, where what changes is the goal: e.g. it could be asked to find the configuration that minimizes the covered area or minimizes only one of the two dimension. Futhermore the rectangles used to cover the area could take different shapes, like circles or irregular polygons as well.
In this work the problem addressed is the first mentioned: the goal is to find a satisfying assignement for the given list of rectangles, in order to exactly cover the entire given enclosing area.

1.2 Data pre- and post-processing

Two python scripts have been developed in order to convert the given instances into Minizinc required format and to plot the results for faster validation. These are *dzn_parser.py* and *results_to_graph.py*. Please read README file in the same *.py* files directory to know how to run them.

1.2.1 Instances format

The given instances are *.txt* files of the form:

```
9 12
5
3 3
2 4
2 8
3 9
4 12
```

where the first line represents the width and heigth of the enclosing area, the second line states how many rectangles are at disposal and the following lines are the actual rectangles expressed as the couple $dimension_{i1} * dimension_{i2}$

1.2.2 Instances preprocessing

The python script *dzn_parser.py* generates two *.dzn* files for each instance: a file called *basic_*.dzn*, that will be used as input instance for the Minizinc basic model, and a file called *general_*.dzn* wich will be used as instance for the Minizinc general model. In the file name the symbol * is a placeholder for the original file name.

The basic *.dzn* files are of the form:

```
w = 8;
h = 8;
n = 4;
dimension = [|
3, 3|
3, 5|
5, 3|
5, 5|];
```

The general *.dzn* files are of the form:

```
w = 8;
h = 8;
n = 4;
m = 6;
rect_size = [|
3, 3|
3, 5|
5, 3|
5, 3|
3, 5|
5, 5|];
rect_offset = [|
0, 0|
0, 0|
0, 0|
0, 0|
0, 0|
0, 0|];
```

```
shape = [{1}, {2}, {3}, {4}, {5}, {6}];
shapeind = [{1}, {2, 3}, {4, 5}, {6}];
ncopy = [1, 1, 1, 1];
c = 4;
```

For a detailed explanation of the structure of the **.dzn* files see the respective modeling sections.

2 THE BASIC MODEL

2.1 Modeling

The idea behind this CP model is to map each rectangle to its bottom-left corner and use the latter as the problem variable. Because the rectangles rotation is not allowed, the only degree of freedom is the rigid translation of the rectangles. So the problem can be translated into: *find a satisfaction assignement for the rectangles bottom-left corners such that the relative rectangles cover the entire area and do not overlap.*
Under this assumption the formulation of the problem constraints is straigthforward:

- for a rectangle i to stay in the enclosing area $w * h$, its bottom left corner coordinates can take values only from $\{0..w - dimension_{i1}, 0..h - dimension_{i2}\}$;
- the non overlap constraints means that when a rectangle i is placed, the set of points in $\{x_i..x_i + dimension_{i1}, y_i..y_i + dimension_{i2}\}$ are removed from the domains of the remaining variables.

As shown in the Global constraints paragraph, sometimes it is useful to explicitly add implied, or redundant, constraints in order to improve constraint propagation and to generate a bigger domain cut in order to speed up the search phase. Such a constraint is, for example, the *cumulative* constraint that sets a capacity limit, in the number of stacked rectangles, on the width and heigth (this is obviously implied by the two previous constraints). In other words:

- the amount of heigth (width) that can be used in any position during packing is up to $h(w)$, so one can think about rectangles as tasks of duration $dimension_{i1}$ ($dimension_{i2}$) and resource consumption $dimension_{i2}$ ($dimension_{i1}$) and the cumulative constraint must hold in any solution.

In order to achieve a better model is also useful to break solution symmetries. In the basic version of this problem the only simmetries are solution isomorphisms, namely, given a solution, any 90° rotation of the packing is still a solution (like in the n-queens problem). In order to break this kind of symmetry I tried to programmatically impose the biggest rectangle position. So the constraint could be:

- setting a fixed and predetermined position (e.g. the paper origin (0,0)) for the most difficult to place rectangle, the biggest one.

In the following paragraphs a detailed explanation of data, variables and constraints is given.

2.1.1 Data

The following is an excerpt of code from the Minizinc *basic_model.mzn* program:

```
1 int: n; % Number of rectangles
2 int: w; % Paper width
```

```
3 int: h; % Paper heigth
4
5 set of int: TILES= 1..n;
6 array[TILES, 1..2] of int: dimension; % Dimensions of tiles
```

Data used at this stage are the number of rectangles n, width w and heigth h of the enclosing area (paper roll). They come directly from the *basic_*.dzn* files after instances preprocessing as shown in Data pre- and post-processing paragraph. From these, a $dimension$ array is built: it is a $n * 2$ matrix where rows are rectangles and columns are the two rectangles' dimensions.

2.1.2 Variables

As stated before, in this basic version of the problem, the only degree of freedom is the rigid translation of rectangles and so the only variables are the coordinates of rectangles' bottom-left corner. I called these variables $coords_{i,j}$.

```
1 % Variables
2 array[TILES, 1..2] of var int: coords; % Bottom left corner
    coordinates
```

2.1.3 Local constraints

In this and the following paragraph problem constraints are explained.

The first constraint is on the **domain of bottom-left corner**. Indeed, there are a lower and an upper bound on the coordinates that make any rectangle to completely stay in the enclosing area.
This constraint can be written as follow:

```
1 % Bottom left corner constraint
2 constraint forall(i in TILES)
3                   ((coords[i,1] >= 0 /\ coords[i,1] <= w -
                      dimension[i,1])
4                  /\ (coords[i,2] >= 0 /\ coords[i,2] <= h -
                    dimension[i,2]));
```

2.1.4 Global constraints

The second constraint is the **non-overlap** constraint, meanig that for any two given rectangles i and j, j can be placed on the rigth OR on the left OR below OR above the rectangle i.
This can be expressed as follows:

```
1 % Non overlap constraint
2 constraint forall(i,j in TILES where i<j)
3       (coords[i,1] +  dimension[i,1] <= coords[j,1]   \/
4        coords[j,1] +  dimension[j,1] <= coords[i,1]   \/
5        coords[i,2] +  dimension[i,2] <= coords[j,2]   \/
6        coords[j,2] +  dimension[j,2] <= coords[i,2]    );
```

meanig that given any two rectangles i and j, j can be placed on the rigth OR on the left OR below OR above the rectangle i.
While this formulation is correct, Minizinc has a built-in global constraint that handles this kind of situation called *diffn* (4.2.6.6 Packing constraint from minizinc tutorial).

Remember that a **global constraint** is a constraint that states a relation between a non-fixed number of variables. While these constraints are typically semantically redundant, because the relation they capture could be expressed as the conjunction of more variable specific constraints, they simplify the programming task, making the code easier to read, and, even more, they speed up the search phase by giving hints on the problem structure to the constraints solver.

Thus the previous constraint is substituted with the following:

```
% Non overlap global constraint
constraint diffn(coords[..,1],coords[..,2], dimension[..,1],
  dimension[..,2]);
```

It is worth to highligth that this constraint is not redundant because it substittutes the previous non-overlap constraint.

At this point the model is already able to find good solutions, but it can be improved by adding redundant constraints and breaking symmetries.

The third constraint is the **cumulative** constraint. This comes from the family of scheduling constraints (4.2.6.7. Scheduling constraints), indeed rectangles can be treated as tasks, their dimensions are the duration along the respective coordinates, the other dimension represents the resource consumption and the upper bounds are respectively the width and heigth of the paper roll.
Obviously for any packing solution, where the previous two constraints are satisfied, cumulative must also hold. Indeed this is a **redundant** constraint needed to improve constraint propagation and speed up the search phase.

```
constraint cumulative(coords[..,1], dimension[..,1],
  dimension[..,2], h); % Vertical constraint
constraint cumulative(coords[..,2], dimension[..,2],
  dimension[..,1], w); % Horizontal constraint
```

2.1.5 Symmetry breaking constraints

Running the model so far and looking at the solutions is easy to notice some symmetries. For example, for the *basic_8x8.dzn* instance we have the 12 solutions of Figure 2.1. In particular each line contains the same solution rotated by 90°, 180° and 270°.
I managed to remove these isomorphisms by imposing the position of the most difficult to place rectangle (the biggest one). The following is the constraint in the Minzinc program:

```
% Symmetry breaking constraint: set the most difficult to
  place rectangle in (0,0)
constraint coords[arg_max([dimension[i,1]*dimension[i,2] | i
  in TILES]),1] = 0;
constraint coords[arg_max([dimension[i,1]*dimension[i,2] | i
  in TILES]),2] = 0;
```

I have built a dynamic constraint that select the biggest rectangle at runtime and places its bottom-left corner in the (0,0) position. In this way, only the first solution of each line of Symmetries for the 8x8 case figure is returned.
The discussion about wheter or not include this constraint in the final model is postponed to the Conclusion section.

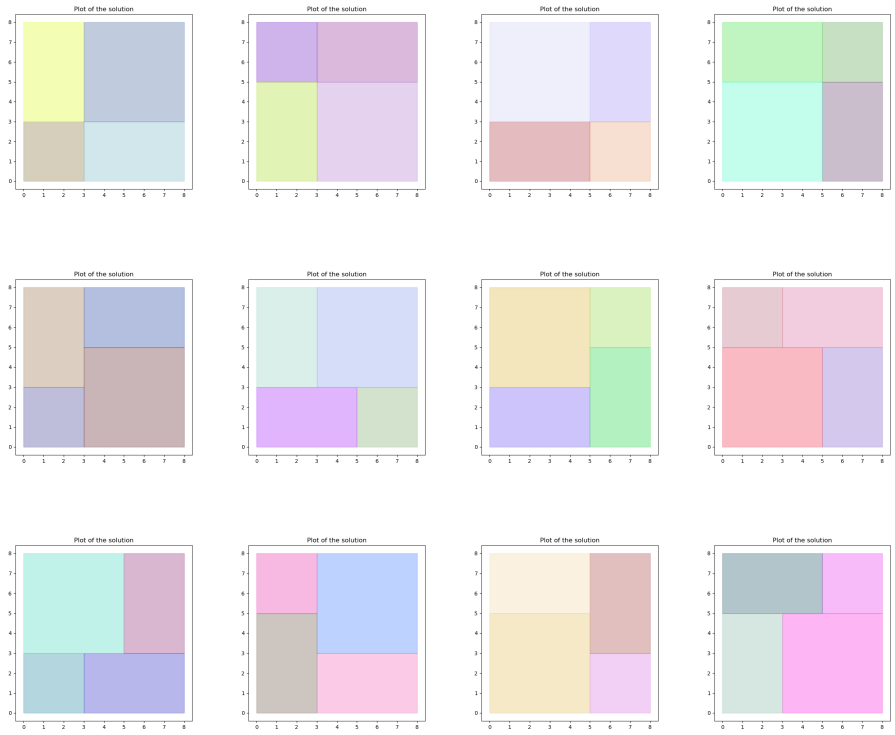

Figure 2.1: Symmetries for the 8x8 case

3 THE GENERAL MODEL

3.1 Modeling

In the general case, what changes is the possibility to have **rectangles with the same dimensions** and **rectangle rotation is now allowed**. This means that the solution space is larger than before because, now, each rectangle can be placed in two different ways and for each instance containing repeated tiles, the solution space contains all the solution given by the permutations of pieces with the same dimensions. The possibility to have repeated rectangles introduces a symmetry that needs to be broken, so this will translate in adding a new symmetry breaking constraint.
To model this case one can extend the previous basic model: indeed the bottom-left corner coordinates can still be used as variable, but another variable set is needed in order to address the rotation possibility. I chose to use $kind_i$ as the new variable. It represents the orientation of the *i-th* rectangle: indeed the problem to place a rectangle and take into account its rotation can be viewed as the problem to assign a value to its bottom-left corner coordinates plus the problem to choose wich, of the two possible values, the horizontal dimension takes (respectively vertical).

3.1.1 Data

In order to address the increased complexity of the problem, some new auxiliary variables are needed.
In the following exrcept of code, constants n, w, h and $TILES$ are the same as in the basic model. The constant m is the number of original rectangles plus their rotated instance. In particular rotation is taken into account only for rectangles and not for squares, namely rectangles

with $w_i = h_i$. In other words a new orientation is like a new rectangle.
The *shape* array is a list of set of integers that represents the list of possible shapes to handle.
The *shapeind* array, instead, maps each shape to the rectangle it derives from.
The *rect_size* and *rect_offset* arrays are respectively the dimensions and offsets of rectangles. Although in this case the rectangles offset is always 0, this array is needed, together with *rect_size* and *shape*, for the *geost_bb* constraint, as shown in the Global constraints section.
The constant c is the number of distinct rectangles and the array $ncopy$ stores the number of copies for each rectangle.

```
int: n; % Number of tiles/rectangles
int: w; % Paper width
int: h; % Paper heigth
int: m; % Number of possible shapes: original rectangles +
   their rotations

set of int: DRECT = 1..m;
set of int: ROT=1..2; % Different positioning of the same
   rectangles: original position and 90   rotated position.
set of int: TILES = 1..n;

% Following data are needed for geost_bb constraint
array[int] of set of DRECT: shape; % List of distinct shape
   that takes into account rotated rectangles
array[TILES] of set of int: shapeind; % Indices of original
   and rotated rectangles
array[DRECT, 1..2] of int: rect_size; % Rectangles dimensions
array[DRECT, 1..2] of int: rect_offset; % Rectangles offsets
   -> in this problem always 0

int: c;
set of int: NUMDISTINCTTILES = 1..c;
array[NUMDISTINCTTILES] of int: ncopy; % Number of copies
   for each rectangle
```

3.1.2 Variables

The only difference with respect the basic case is the array of variables *kind*. This variable is responsible to select the rectangle rotation among the two possible.

```
% Variables
array[TILES, 1..2] of var int: coords; % Bottom left corner
   coordinates
array[TILES] of var int: kind; % The selected rotation among
   the two allowed
```

3.1.3 Local constraints

The **bottom-left corner domain** constraint becomes:

```
% Bottom left corner constraint
constraint forall(i in TILES)((coords[i,1] >= 0
              /\ coords[i,1] <= w - rect_size[kind[i],1])
```

```
4                 /\ (coords[i,2] >= 0
5                 /\ coords[i,2] <= h - rect_size[kind[i],2]));
```

It is clear here the role of the variable $kind_i$ in selecting the dimension, so implicitly selecting the rotation.

3.1.4 Global constraints

The global constraints are again the **non overlapping constraint** and the **redundant cumulative constraint**, adjusted with the introduction of the variable $kind_i$. In particular the *diffn* constraint is substitued by the **geost_bb** constraint, a Minizinc built-in constraint that handles packing of objects of different shapes forcing them to fit in a $w * h$ bounding box. The first constraint in the following excerpt of code is needed in order to force the variable $kind_i$ to take the right values of shape.
The *cumulative constraint* stays in place with the adjustments on dimensions arguments.

```
1  % Non-overlap constraint
2  constraint forall(i in TILES)(kind[i] in shapeind[i]);
3  constraint geost_bb(2,
4                      rect_size,
5                      rect_offset,
6                      shape,
7                      coords,
8                      kind,
9                      [0,0],
10                     [w,h]);
11
12 % Constraint on the paper size
13 constraint cumulative([coords[i,1] | i in TILES],
14             [rect_size[kind[i],1] | i in TILES],
15             [rect_size[kind[i],2] | i in TILES], h);
16 constraint cumulative([coords[i,2] | i in TILES],
17             [rect_size[kind[i],2] | i in TILES],
18             [rect_size[kind[i],1] | i in TILES], w);
```

3.1.5 Symmetry breaking constraints

Again, in order to break isomorphisms it suffices to manually set the position of the biggest rectangle but letting the algorithm to choose the orientation. The symmetry breaking constraint become:

```
1  % Symmetry breaking: set the most difficult to place
     rectangle, i.e. the biggest one, in the position (0,0)
2  constraint
     coords[arg_max([rect_size[kind[i],1]*rect_size[kind[i],2]
     | i in TILES]),1] = 0;
3  constraint
     coords[arg_max([rect_size[kind[i],1]*rect_size[kind[i],2]
     | i in TILES]),2] = 0;
```

Besides the isomorphism symmetry, this time another class of symmetry arise because of the permutations of rectangles with the same dimensions. This is the kind of symmetry broken by lexicographical ordering. In order to do that the following constraint is added:

```
% Generating an array of indeces for the distinct sizes in
  dimension
array[NUMDISTINCTTILES] of int: base =[
      if i = 1 then 0
      else sum(j in 1..i-1)(ncopy[j]) endif | i in
        NUMDISTINCTTILES];

% Ordering pieces: this comes into play only for rectangles
  with the same size
constraint forall(i in NUMDISTINCTTILES)(
  forall(j in 1..ncopy[i]-1)(
    lex_greater(coords[base[i]+j,..],coords[base[i]+j+1,..])));
```

Under the assumption that the given rectangles are ordered ascending, the *base* array contains the starting index of each series of rectangle with the same dimension. For example, given the array $[3x3, 3x3, 3x3, 4x5, 4x5, 6x7]$ the corresponding $base$ array would be $[0,3,5]$. The $base$ array lets to apply the lexicogrphical ordering only to rectangles with the same dimensions.
IMPORTANT: it is important tohighlight here the **incompatibility** between the two previous constraints due to the case where the isomorphism breaking constraint select a repeated rectangle and the ordering breaking (with "lex_greater") try to put it as the last of the series. The isomorphism breaking constraint, in this formulation, can work only paired with **lex_less** constraint (because we are choosing (0,0) as the fixed position). Aware of this, in the next section performances of the general model will be evaluated changing this combination of constraints.

4 Solving the problem

4.1 Search strategies

In order to have an almost complete picture on how the basic model performs during search, several combinations of variable and value selection strategies, over a subset of the given instances have been tried.
In particular, the chosen variable selection strategies are (using Minizinc naming):

- **first_fail:** choose the variable with the smallest domain size;
- **anti_first_fail:** choose the variable with the largest domain size;
- **dom_w_deg:** choose the variable with the smallest value of domain size divided by weighted degree, which is the number of times it has been in a constraint that caused failure earlier in the search;
- **smallest:** choose the variable with the smallest value in its domain;
- **largest:** choose the variable with the largest value in its domain;
- **input_order:** choose in order from the variable array.

For each of the variable selection strategy above, the following value selection strategies have been tried:

- **indomain_min:** assign the variable its smallest domain value;
- **indomain_max:** assign the variable its largest domain value;
- **indomain_median:** assign the variable its median domain value;
- **indomain_split:** bisect the variables domain excluding the upper half first;
- **indomain_reverse_split:** bisect the variables domain excluding the lower half first.

4.1.1 Comparing different search strategies

The following tables show the search results in a time limit of 5'. The number of nodes and failures to return the first solution have been collected. I chose these, among the available statistics, because they give hints about the dimension of the search space in terms of choices (nodes) and backtracking (failures). Time-to-solve statistic has been discarded because it is too much related to the hardware characteristics. I will omit here tables for the last three variable selection strategies: the density of results is the same as in the *anti_first_fail* case. Anyway an .xlsx file containing the statistics will be attached in the project directory.
In cases where the *indomain_random* search failed, a search with **luby_restart(1000)** has been tried.
Every time a random search is performed, the seed has been set to 42.

The following list of tables contains data collected, for the first three variable selction strategies described, trying the basic model *with* and *without* the symmetry breaking constraint. This because, while in principle it is sure that symmetries breaking reduces the solution space, it is not obvious wheter it speeds the search up or not, because it adds more constraints to check.

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	0	0	1	0	0	0	-
10x10	1	3	5	1	3	1	-
17x17	**25**	44	58481	27	38	171767	-
18x18	-	-	-	-	-	-	-
19x19	3289	6726	2910569	**2618**	4364	397692	-
25x25	58977	70981	-	**41172**	47050	-	-
33x33	**2172**	3491	-	5370	2374	-	-
40x40	**7**	18	-	10	18	-	212146

Table 4.1: number of failures for **first_fail** strategy without symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	0	0	1	0	0	0	
10x10	0	1	3	0	1	1	
17x17	321	483	**194**	278	376	15930	
18x18	-	-	-	-	-	-	-
19x19	240231	410810	4178593	**185233**	319837	-	114980
25x25	6576862	6534312	-	**5333277**	5340565	-	14891599
33x33	**2172**	3491	-	5370	2374	-	-
40x40	**7**	18	1396	8	18	-	645527

Table 4.2: number of failures for **first_fail** strategy with symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	0	0	36	0	0	2	
10x10	2	11	190	16	1	135	
17x17	1323154	5165191	-	19835	**2943**	-	-
18x18	-	-	-	-	-	-	-
19x19	**122**	969	-	-	-	-	-
25x25	-	-	-	-	-	-	-
33x33	-	-	-	-	-	-	-
40x40	-	-	-	-	-	-	-

Table 4.3: number of failures for **anti_first_fail** strategy without symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	0	0	6	0	0	0	
10x10	1	1	38	1	0	13	
17x17	**152**	820	-	11640	1313	-	8661874
18x18	-	-	-	-	-	-	-
19x19	-	-	-	804331	**333703**	-	-
25x25	-	-	-	-	-	-	-
33x33	-	-	-	-	-	-	-
40x40	-	-	-	-	-	-	-

Table 4.4: number of failures for **anti_first_fail** strategy with symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	0	0	1	0	0	0	
10x10	1	3	5	1	3	1	
17x17	**25**	44	56340	27	38	17839	
18x18	17359378	25809407	-	**12862318**	17730760	-	-
19x19	3289	6726	2910386	**2618**	4364	397547	
25x25	58977	70981		**41172**	47050	-	-
33x33	**2172**	3491	-	5370	2374	-	-
40x40	**7**	18	-	10	18	-	44166

Table 4.5: number of failures for **dom_w_deg** strategy without symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	0	0	1	0	0	0	
10x10	0	1	6	0	1	1	
17x17	194	296	1268	**162**	228	4748	
18x18	-	**4738909**	-	12862400	-	-	-
19x19	3289	6726	2910016	**2618**	4364	397363	173802
25x25	58977	70981	-	**41172**	47050	-	14278829
33x33	**2172**	3491	-	5370	2374	-	8067109
40x40	**7**	18	1396	8	18	-	313662

Table 4.6: number of failures for **dom_w_deg** strategy with symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	5	5	7	6	5	5	
10x10	9	13	17	14	14	10	
17x17	63	101	116975	90	103	343549	
18x18	-	-	-	-	-	-	-
19x19	6592	13466	5821152	5278	8759	795400	-
25x25	117975	141983	-	82409	94150	-	-
33x33	4367	7005	-	10819	4811	-	-
40x40	45	66	-	95	93	-	425342

Table 4.7: number of nodes for **first_fail** strategy without symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	2	3	4	2	3	3	
10x10	4	6	10	5	6	7	
17x17	656	980	400	582	776	31873	
18x18	-	-	-	-	-	-	-
19x19	480475	821633	8357199	370504	639701	-	230313
25x25	13153744	13068644	-	10666615	10681176	-	29812270
33x33	4366	7004	-	10814	4807	-	-
40x40	44	65	2813	83	87	-	1293398

Table 4.8: number of nodes for **first_fail** strategy with symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	5	5	77	5	5	8	
10x10	12	30	386	40	11	278	
17x17	2646321	10330395	-	39698	5907	-	-
18x18	-	-	-	-	-	-	-
19x19	264	1958	-	-	-	-	-
25x25	-	-	-	-	-	-	-
33x33	-	-	-	-	-	-	-
40x40	-	-	-	-	-	-	-

Table 4.9: number of nodes for **anti_first_fail** strategy without symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	3	2	14	3	2	2	
10x10	5	7	80	5	6	30	
17x17	314	1652	-	23299	2645	-	17338703
18x18	-	-	-	-	-	-	-
19x19	-	-	-	1608692	667432	-	-
25x25	-	-	-	-	-	-	-
33x33	-	-	-	-	-	-	-
40x40	-	-	-	-	-	-	-

Table 4.10: number of nodes for **anti_first_fail** strategy with symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	5	5	7	6	5	5	
10x10	9	13	17	14	14	10	
17x17	63	101	112693	90	103	35692	
18x18	34718777	51618835	-	25724704	35461566	-	-
19x19	6592	13466	5820787	5278	8759	795110	
25x25	117975	141983	-	82409	94150	-	-
33x33	4367	7005	-	10819	4811	-	-
40x40	45	66	-	95	93	-	88589

Table 4.11: number of nodes for **dom_w_deg** strategy without symmetry breaking constraint

n	**min**	**max**	**median**	**split**	**rev_split**	**random**	**rnd+luby(1000)**
8x8	2	3	4	2	3	3	
10x10	4	6	16	5	6	7	
17x17	403	605	2549	352	478	9506	
18x18	-	9477838	-	25724847	-	-	-
19x19	6591	13465	5820045	5274	8755	794741	348022
25x25	117974	141982	-	82405	94146	-	28584862
33x33	4366	7004	-	10814	4807	-	16154513
40x40	44	65	2813	83	87	-	628672

Table 4.12: number of nodes for **dom_w_deg** strategy with symmetry breaking constraint

instance	**n**	**h_pieces**	**v_pieces**	**squares**	**num_h/num_v**	**default**	**default_symm**	**dom_w_deg**	**dom_w_deg_symm**
8x8	4	1	1	2	1	**0**	**0**	**0**	**0**
9x9	5	0	4	1	0	1	**0**	2	1
10x10	6	0	4	2	0	1	1	1	**0**
11x11	7	3	3	1	1	**5**	12	60	14
12x12	8	1	6	1	0,17	**26**	31	27	37
13x13	9	2	5	2	0,4	**2**	**2**	39	84
14x14	9	1	7	1	0,14	78	78	85	**7**
15x15	10	1	8	1	0,13	82	**69**	102	116
16x16	10	0	9	1	0	**4**	40	102	59
17x17	12	3	8	1	0,38	25	**5**	27	162
18x18	16	3	10	3	0,3	14629346	14629346	**12862318**	-
19x19	14	0	13	1	0	**791**	**791**	2618	2618
20x20	14	1	12	1	0,08	943	21444	**814**	15546
21x21	15	0	13	2	0	3864	3803	**1118**	**1118**
22x22	16	2	13	1	0,15	2656	**76**	1556	164
23x23	19	2	15	2	0,13	-	-	11052510	**11052382**
24x24	18	1	15	2	0,07	**266809**	**266809**	461872	464953
25x25	19	1	17	1	0,06	59441	59441	**41172**	**41172**
26x26	22	1	19	2	0,05	**2171466**	2171467	7701736	7737933
27x27	21	1	18	2	0,06	2008052	2008052	**1075859**	1079012
28x28	22	1	19	2	0,05	427531	427531	**242688**	**242688**
29x29	24	2	20	2	0,1	-	-	**9377805**	9483623
30x30	20	1	17	2	0,06	22	26	-	**10**
31x31	19	1	17	1	0,06	765353	26	474430	**8**
32x32	27	1	24	2	0,04	-	-	-	-
33x33	23	1	20	2	0,05	4577330	4577330	**5370**	**5370**
34x34	21	0	19	2	0	1961959	**13**	1142437	35
35x35	22	1	19	2	0,05	-	321962	-	**2**
36x36	23	3	19	1	0,16	-	**91**	-	10158
37x37	27	3	22	2	0,14	-	-	-	-
38x38	19	1	16	2	0,06	-	-	**1614162**	**1614162**
39x39	29	3	23	3	0,13	-	-	-	-
40x40	20	2	16	2	0,13	14	14	0	**8**

Table 4.13: Overview of instances features and comparison of failures for default search with and without symmetries breaking

instance	**default**	**default_symm**	**ff_min**	**ff_min_symbre**	**dwd_min**	**dwd_min_symbre**
8x8	0	0	0	0	0	0
10x10	1	1	1	0	1	0
17x17	25	**5**	25	321	25	194
18x18	**14629346**	**14629346**	-	-	17359378	-
19x19	**791**	**791**	3289	240231	3289	3289
25x25	59441	59441	**58977**	6576862	**58977**	**58977**
33x33	4577330	4577330	**2172**	**2172**	**2172**	**2172**
40x40	14	14	**7**	**7**	**7**	**7**

Table 4.14: number of failures for the best three variable selection strategies paired with indomain_min value selection strategy

instance	**fail_def**	**fail_def_symm**	**ff_split**	**ff_split_symbre**	**dwd_split**	**dwd_split_symbre**
8x8	0	0	0	0	0	0
10x10	1	1	1	0	1	0
17x17	**25**	5	27	278	27	162
18x18	14629346	14629346	-	-	**12862318**	12862400
19x19	**791**	791	2618	185233	2618	2618
25x25	59441	59441	**41172**	5333277	**41172**	**41172**
33x33	4577330	4577330	**5370**	**5370**	**5370**	**5370**

Table 4.15: number of failures for the best three variable selection strategies paired with indomain_split value selection strategy

n	**less**	**less + 00**	**greater**
8x8_symm	1	0	1
10x10_symm	65	15	0
16x16_symm	6	27	4

Table 4.16: Comparing number of failures for general model on symmetric instances for different combinations of symm. break. constraint

n	**less**	**less + 00**	**greater**
8x8_symm	5	2	4
10x10_symm	147	42	16
16x16_symm	26	68	23

Table 4.17: Comparing number of nodes for general model on symmetric instances for different combinations of symm. break. constraint

n	**less**	**less + 00**	**greater**
8x8_symm	1	1	1
10x10_symm	60336	13536	60336
16x16_symm	45984	4616	45984

Table 4.18: Comparing number of solution after 3' for general model on symmetric instances for different combinations of symm. break. constraint

5 Conclusion

The two best variable selection strategies are **first_fail** and **dom_w_deg**, most of times paired with the **indomain_min** and **indomain_split** value selection strategies. Sometimes their performaces are exactly the same, as in the 25x25, 33x33 or 40x40, without symmetry breaking constraint. An interesting fact is how *dom_w_deg* can solve the 18x18 instance, in 4 over 6 of the value selection strategies tried, while *first_fail* can't. What seems clear is that the instance difficulty is not a linear function of the "paper" area: the tables show how the 18x18 instance is way more hard to solve than , for example, the 40x40 one. In table Table 4.13 I have written down the number of vertical and horizontal rectangles and the number of squares in each instance, in order to check out wheter there was or not a correlation between these features and the instance difficulty. My idea was that the instance difficulty would raise when the ratio of vertical rectangles over horizontal ones was small. Unfortunately there is no such an obvious correlation, but could be interesting to try a regression model trying different combination of the instance feature in order to be able to predict the instance difficulty on the fly, and maybe select the model to be used accordingly.

Tables Table 4.14 and Table 4.15 show a comparison between the best two search strategies, previously found, and the Gecode default search. Furthermore, Table 4.13 shows a comparison between the two best strategies over all the provided instances. The best search strategy seems to be the **dom_w_deg + indomain_split** search, because it performs better on hard instances (e.g 18x18). Anyway, there are instances, e.g. 32x32, that after 1h of search are still with no solution.
Tha last three tables show a 3' run of the general model over the symmetric instances. The column *"less"* means that only the *lex_less* constraint is active; the column *"less + 00"* means that the *lex_less* constraint and the fixed position for the biggest rectangle are active; the column *"greater"* means that only the *lex_greater* constraint is active. The chosen search strategy is the *dom_w_deg* paired with *indomain_split*.
As can be seen, the best general model is the third one, with only the *lex_greater* constraint active. It comes that the final model does not include isomorphism breaking as pointed out in section Global constraints.
It is important to highlight here that, while in the basic model variables involved in the search process were the $coords_{i,j}$, in the general model search it is conveniente to search over the $kind_{i,j}$. because they uniquely specify the relative rectangle. Obviously it is possible to search over the $coords_{i,j}$ as in the basic case, but, after some tries, it has been discovered that this would result in a much slower search.

5.0.1 Symmetry breaking, yes or not?

The question about wheter or not include this constraint, in the basic model, has not an obvious answer. Just as a reminder: here I am talking about the constraint that fixes the biggest

rectangle bottom left corner.
In table Table 4.13 a 5' run has been tried for each instance with default search strategy, for the basic model, with and without symmetry breaking constraint.
The effect of constraining the position of the biggest rectangle in (0,0) has two different effects: on one side, for very easy instances, it slows down the search, making it to fail more times; on the other side, for hard instances, it helps the search to reach a solution (e.g. 34x34, 35x35, 36x36 instances). So, my answer to the title question is **yes**, because breaking symmetries paired with the best search strategy (dom_w_deg + indomain_split strategy) helps to solve more instances.

6 REFERENCES

- Class lectures
- "Advanced Modeling for Discrete Optimization" - Coursera course
- Uppsala University resource for MiniZinc rendering in Latex - link