Present Wrapping Problem

Constraint Programming (CP)

Pierpasquale Colagrande

October, 2020

1 Introduction

Given a wrapping paper roll of a certain width and height and a list of pieces to cut, with their relative widths and heights, the Present Wrapping Problem (PWP), also know as Two-dimensional Orthogonal Packing Problem (2OPP), consists in finding a way to cut all the pieces from the paper roll, without excluding none.

The aim of this model is to solve the Present Wrapping Problem using Constraint Programming with MiniZinc language. The problem was solved by building a MiniZinc model as a .mzn file which was then used in a Python program using the MiniZinc Python Interface.

1.1 Specifications

The problem was modeled using input instances, two models and output files. All of these components are managed by a CLI-based Python program that is shared between all the solving methods, which allows the user to choose the instance(s) to solve, the solver (CP or SMT) and the model (general or standard) to use. The program saves the solution on a specific output file and then it also allows to plot the solution visually.

1.1.1 Input

An input instance is a .txt file where:

- the first line contains the **width** and **height** of the paper roll
- the second line contains the number **n** of the pieces to cut
- the following n lines contain the **widths** and **heights** of the n pieces to cut

1.1.2 Models

There are two models:

- **standard**: basic model that allows pieces to be cutted from the paper roll without the possibility to rotate them and without any optimization for pieces of the same size
- **general**: improved model that allows pieces to be cutted from the paper roll also by rotating them and also optimizing pieces of the same size

1.1.3 Output

The solution is saved to an output .txt file where:

- the first line contains the **width** and **height** of the paper roll
- the second line contains the number **n** of the pieces to cut
- the following n lines contain the **widths** and **heights**, along with the **x and y coordinates** of the bottom-left corners of the n pieces to cut; for the general model (the model with rotation), these n lines also contain a boolean string representing the rotation of each piece

The solution is, in fact, represented by the coordinate on the paper roll of the bottom-left corner of the piece to cut.

2 Constraint Programming

This section will cover and explain the parameters, variables and constraints used in the two MiniZinc models, along with the modifications done to also implement the features of the general model. The constraints will be described both with and without the use of MiniZinc global constraints.

2.1 Parameters

Several useful parameters have been defined:

- two integers x and y representing **the x and y axes** of the paper roll
- a set of integer values *AXES* representing the possible **axes** on which the problem is defined
- an integer value *roll_width* representing the **paper roll width**
- an integer value *roll_height* representing the **paper roll height**
- an integer value *n_pieces* representing the **number of pieces** to cut
- a set of integer values *PIECES* representing the *n_pieces* **pieces of paper** to cut

- a matrix of integer values *pieces_dimensions* with as many rows as *PIECES* and as many columns as *AXES* representing the **dimensions of each piece of paper** over the two axes x and y defined above
- an array of integer values *lower_bounds* with as many elements as *AXES* representing the **lower bounds** of the possible coordinates over the two axes x and y defined above
- an array of integer values *upper_bounds* with as many elements as *AXES* representing the **upper bounds** of the possible coordinates over the two axes x and y defined above

It is important to specify that some of these parameters were only defined to make the mode more readable and understandable.

2.2 Variables

The problem has been modeled using a matrix of integer decision variables *pieces_corners* which represents the coordinates of the bottom-left corner of all the pieces of paper over the two axes x and y. **Figure 1** represents the encoding of the bottom-left coordinates of the pieces and their relative visualization on the paper roll.

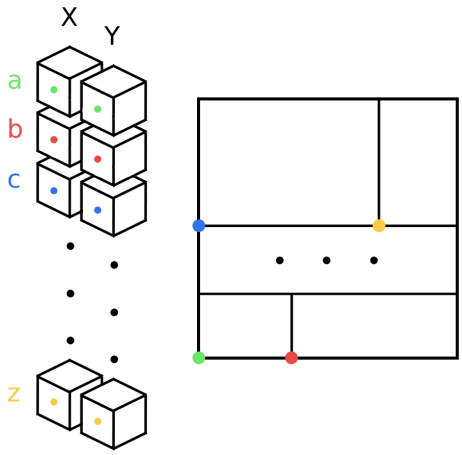

Figure 1: Decision variables modeling and their visual representation on the paper roll

The rows of the matrix are the *PIECES* while the columns are the *AXES*. The first column represents all the **x** coordinates of the bottom-left corner of the pieces of paper, while the second column represents the **y** coordinates. The matrix has as many rows as the number of pieces of paper to cut.

2.3 Main constraints

Before defining the main constraint, we need to do some observations:

1. the pieces must be cutted **inside** the limits of the paper roll, so the position of the bottom-left corner must stay inside the coordinates described by the paper roll limits, which have been defined with *lower_bounds* and *upper_bounds* parameters; this property is defined by the following constraint:

$$\forall i \in PIECES$$

$$pieces\_corners[i,\ x] \geq lower\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ x] \leq upper\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \geq lower\_bounds[y]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \leq upper\_bounds[y]$$

2. it is possible to calculate the coordinates occupied by a piece of paper by adding to the y coordinate of the bottom-left corner its height and by adding to the x coordinate its width; these coordinates **must not exceed** the paper roll limits; this property is defined by the following constraint:

$$\forall i \in PIECES$$

$$pieces\_corners[i,\ x] + pieces\_dimensions[i,\ x] \leq upper\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ y] + pieces\_dimensions[i,\ y] \leq upper\_bounds[y]$$

3. the pieces of paper must also **not overlap**, which means that the coordinates occupied by a piece of paper must not be occupied by other pieces of paper

There is, however, a better way to define the property 1. and 2. using only one constraint:

$$\forall i \in PIECES$$

$$pieces\_corners[i,\ x] \geq lower\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ x] \leq upper\_bounds[x] - pieces\_dimensions[i,\ x]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \geq lower\_bounds[y]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \leq upper\_bounds[y] - pieces\_dimensions[i,\ y]$$

This constraint represents the fact that each piece of paper must be cutted inside the boundaries defined by the paper roll dimensions. In this constraint, we also need to consider the piece of paper dimensions in order to consider also the fact that the coordinates occupied by the piece of paper must not exceed the limits of the paper roll.

MiniZinc provides a global constraint, ***forall***, which constraints all elements in the specified array to respect the specified constraint. In MiniZinc, this constraint can so be written as:

```
constraint forall(i in PIECES)(
pieces_corners[i, x] >= lower_bounds[x] /\
pieces_corners[i, x] <= upper_bounds[x] - pieces_dimensions[i, x]
  /\
pieces_corners[i, y] >= lower_bounds[y] /\
pieces_corners[i, y] <= upper_bounds[y] - pieces_dimensions[i, y]);
```

The third property can instead be defined as the following constraint:

$$\forall i, j \in PIECES,\ i < j$$

$$pieces\_corners[i,\ x] + pieces\_dimensions[i,\ x] \leq pieces\_corners[j,\ x]$$

$$\vee$$

$$pieces\_corners[i,\ y] + pieces\_dimensions[i,\ y] \leq pieces\_corners[j,\ y]$$

$$\vee$$

$$pieces\_corners[j,\ x] + pieces\_dimensions[j,\ x] \leq pieces\_corners[i,\ x]$$

$$\vee$$

$$pieces\_corners[j,\ y] + pieces\_dimensions[j,\ y] \leq pieces\_corners[i,\ y]$$

This constraint represent the fact that, for each pair of pieces of paper, the coordinates occupied by the first must not overlap the coordinates occupied by the second and viceversa.

For this case, MiniZinc provides a global constraint too, ***diffn_k***, which does the job of checking the non-overlap properties. In MiniZinc, this constraint can so be written as:

```
constraint diffn_k(pieces_corners, pieces_dimensions);
```

2.4 Implied cumulative constraints

If we draw a vertical line and sum the vertical sides of the traversed pieces, the sum can be **at most equal to the height** of the paper roll. The same property must hold if we draw an horizontal line, which means that the sum of the horizontal sides of the traversed pieces can be **at most equal to the width** of the paper roll. To define this property through a constraint, we need two new parameters:

- a set of integer values *X_COORDINATES* representing the coordinates along the x axis (the columns of the paper roll)
- a set of integer values *Y_COORDINATES* representing the coordinates along the y axis (the rows of the paper roll)

The rows and columns of the paper roll are the stripes of the paper roll obtained by placing an imaginary grid over the paper roll which has as many rows as the height of the paper roll and as many columns has the width of the paper roll. These two parameters are used to check this property on each row and column, which basically means we are representing each "vertical and horizontal lines to draw". The possible coordinates over the two axes x and y basically represent these rows and columns.

This property can be expressed with the following constraint:

$$\forall y\_coord \in Y\_COORDINATES$$

$$\sum_{i \in PIECES} width(i,\ y\_coord) \leq roll\_width$$

where

$$width(i,\ y\_coord) = \begin{cases} pieces\_dimensions[i,\ x] & \text{if } y\_coord \geq pieces\_corners[i,\ y] \wedge \\ & y\_coord < pieces\_corners[i,\ y] \\ & + pieces\_dimensions[i,\ y] \\ 0 & \text{otherwise} \end{cases}$$

$$\bigwedge$$

$$\forall x\_coord \in X\_COORDINATES$$

$$\sum_{i \in PIECES} height(i,\ x\_coord) \leq roll\_height$$

where

$$height(i,\ x\_coord) = \begin{cases} pieces\_dimensions[i,\ y] & \text{if } x\_coord \geq pieces\_corners[i,\ x] \wedge \\ & x\_coord < pieces\_corners[i,\ x] \\ & + pieces\_dimensions[i,\ x]) \\ 0 & \text{otherwise} \end{cases}$$

What this constraint is doing is checking, for each row of the paper roll, if the sum of the widths of the pieces on that row is at most the paper roll width. So, basically, for each row of the paper roll, it checks if each piece is placed on that row (by checking the coordinates occupied by each single piece). If a piece is on that row, the width of that piece is used for the sum, otherwise, its width is not used in the sum. This sum is calculated for each row of the paper roll and it must be at most the paper roll width for each row of the paper roll.

This operation is done also for the columns of the paper roll by checking if each piece is placed on that column and by summing their heights if they are, as described before. The sum over each column must be at most the paper roll height. These two conditions must be both satisfied.

Fortunately, MiniZinc has a global constraint, ***cumulative*** which does this operation. So, in MiniZinc, we don't need to define such a complex constraint

and also we don't need to define the two parameters described above. This constraint in MiniZinc can be written as:

```
constraint cumulative(pieces_corners[.., x],
pieces_dimensions[.., x], pieces_dimensions[.., y],
roll_height) /\
cumulative(pieces_corners[.., y],
pieces_dimensions[.., y], pieces_dimensions[.., x],
roll_width);
```

However, after some testing, we noticed that it was necessary a modification. We replaced these constraint defined with MiniZinc's *cumulative* global constraint with the implied constraints described above with a little modification: the sum of the widths/heights of the pieces in each column/row of the paper roll must be **exactly equal** to the paper roll height/width, not at most equal. By doing this, we are removing the possibility to have a white space at the edges of the paper roll or between the pieces. This is a problem in case we have input instances where the solution can be found without using the full paper roll, meaning that we can have space between pieces or at the edges of the roll, but in this case none of our input instances presented this situation.

The two modified constraints are

$$\forall y\_coord \in Y\_COORDINATES$$

$$\sum_{i \in PIECES} width(i,\ y\_coord) = roll\_width$$

where

$$width(i,\ y\_coord) = \begin{cases} pieces\_dimensions[i,\ x] & \text{if } y\_coord \geq pieces\_corners[i,\ y] \wedge \\ & y\_coord < pieces\_corners[i,\ y] \\ & + pieces\_dimensions[i,\ y] \\ 0 & \text{otherwise} \end{cases}$$

$$\bigwedge$$

$$\forall x\_coord \in X\_COORDINATES$$

$$\sum_{i \in PIECES} height(i,\ x\_coord) = roll\_height$$

where

$$height(i,\ x\_coord) = \begin{cases} pieces\_dimensions[i,\ y] & \text{if } x\_coord \geq pieces\_corners[i,\ x] \wedge \\ & x\_coord < pieces\_corners[i,\ x] \\ & + pieces\_dimensions[i,\ x]) \\ 0 & \text{otherwise} \end{cases}$$

which in MiniZinc were described as

```
constraint forall (x_coord in X_COORDINATES) (
                sum(i in PIECES)
                (if x_coord >= pieces_corners[i, x] /\ x_coord <
                  pieces_corners[i, x] + pieces_dimensions[i, x]
                  then pieces_dimensions[i, y] else 0 endif) =
                  roll_height) /\
           forall (y_coord in Y_COORDINATES) (
                sum(i in PIECES)
                (if y_coord >= pieces_corners[i, y] /\ y_coord <
                  pieces_corners[i, y] + pieces_dimensions[i, y]
                  then pieces_dimensions[i, x] else 0 endif) =
                  roll_width);
```

2.5 General model

This section will cover the necessary modifications to add the possibility to rotate pieces and to manage and optimize pieces of the same size.

2.5.1 Rotation

We must consider that a piece could be rotated in 4 different ways:

- 0° rotation, where width and height are not swapped
- 90° rotation, where width and height are swapped
- 180° rotation, where width and height are not swapped
- 270° rotation, where width and height are swapped

Since the pieces of paper have regular shapes (they are rectangles or squares), a piece with a 0° or a 180° rotation will have the same shape in both cases, with the same width and height, as well as a piece with a 90° or a 270° rotation. We can so encode only two cases:

- the piece is **not rotated**, so width and height are **not swapped**
- the piece is **rotated**, so width and height are **swapped**

This rotation property has been modeled using another array of boolean decision variables called *pieces_rotation*. The array has as many elements as *PIECES*. This array of boolean decision variables describes if a present is rotated or not.

In order to encode this property, we need to define a new model by modifying some properties of the standard model. In fact, if a piece is rotated, its width and height are swapped, so the constraints of the standard model must be modified in order to use the correct widths and heights for all the pieces, whether they are rotated or not.

In particular, a function which returns the correct dimension of the specified piece along the specified axis based on the piece rotation has been defined:

$$get\_dimension(i,\ axis) = \begin{cases} pieces\_dimensions[i,\ y] & \text{if } axis = x \land pieces\_rotation[i] \\ pieces\_dimensions[i,\ x] & \text{if } axis = x \land \neg pieces\_rotation[i] \\ pieces\_dimensions[i,\ x] & \text{if } axis = y \land pieces\_rotation[i] \\ pieces\_dimensions[i,\ y] & \text{if } axis = y \land \neg pieces\_rotation[i] \end{cases}$$

This function, in MiniZinc, is defined as:

```
function var int: get_dimension(int: i, int: axis) =
  if axis == x then
    if pieces_rotation[i] then pieces_dimensions[i, y] else
      pieces_dimensions[i, x] endif
  else
    if pieces_rotation[i] then pieces_dimensions[i, x] else
      pieces_dimensions[i, y] endif
  endif;
```

Also, we need to modify the previously defined constraints: in fact, we need to use the function defined above to get the correct dimensions for the piece when it is used inside a constraint.

The constraint defining the paper limits is modified to:

$$\forall i \in PIECES$$

$$pieces\_corners[i,\ x] \geq lower\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ x] \leq upper\_bounds[x] - get\_dimension(i,\ x)$$

$$\wedge$$

$$pieces\_corners[i,\ y] \geq lower\_bounds[y]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \leq upper\_bounds[y] - get\_dimension(i,\ y))$$

In MiniZinc, this is written as:

```
constraint forall(i in PIECES)(
pieces_corners[i, x] >= lower_bounds[x] /\
pieces_corners[i, x] <= upper_bounds[x] - get_dimension(i, x) /\
pieces_corners[i, y] >= lower_bounds[y] /\
pieces_corners[i, y] <= upper_bounds[y] - get_dimension(i, y));
```

Then, we also need to modify the non-overlap constraint:

$$\forall i, j \in PIECES,\ i < j$$

$$pieces\_corners[i,\ x] + get\_dimension(i,\ x) \leq pieces\_corners[j,\ x]$$

$$\vee$$

$$pieces\_corners[i,\ y] + get\_dimension(i,\ y) \leq pieces\_corners[j,\ y]$$

$$\vee$$

$$pieces\_corners[j,\ x] + get\_dimension(j,\ x) \leq pieces\_corners[i,\ x]$$

$$\vee$$

$$pieces\_corners[j,\ y] + get\_dimension(j,\ y) \leq pieces\_corners[i,\ y]$$

In MiniZinc, this is modified by using MiniZinc's global constraint ***diffn***, which differs from *diffn_k* because it requires to specify the corners and the dimensions of the pieces of paper over both the two axes separately, which allows us to

specify the correc dimensions based on the rotation of pieces. The constraint in MiniZinc is then the following:

```
constraint diffn(pieces_corners[.., x],
pieces_corners[.., y],
[get_dimension(i, x) | i in PIECES],
[get_dimension(i, y) | i in PIECES]);
```

The cumulative implied constraint are modified too:

$$\forall y\_coord \in Y\_COORDINATES$$

$$\sum_{i \in PIECES} width(i,\ y\_coord) = roll\_width$$

where

$$width(i,\ y\_coord) = \begin{cases} get\_dimension(i,\ x) & \text{if } y\_coord \geq pieces\_corners[i,\ y] \wedge \\ & y\_coord < pieces\_corners[i,\ y] \\ & + get\_dimension(i,\ y) \\ 0 & \text{otherwise} \end{cases}$$

$$\bigwedge$$

$$\forall x\_coord \in X\_COORDINATES$$

$$\sum_{i \in PIECES} height(i,\ x\_coord) = roll\_height$$

where

$$height(i,\ x\_coord) = \begin{cases} get\_dimension(i,\ y) & \text{if } x\_coord \geq pieces\_corners[i,\ x] \wedge \\ & x\_coord < pieces\_corners[i,\ x] \\ & + get\_dimension(i,\ x)) \\ 0 & \text{otherwise} \end{cases}$$

In MiniZinc, the constraint becomes the following:

```
constraint forall (x_coord in X_COORDINATES) (
              sum(i in PIECES)
              (if x_coord >= pieces_corners[i, x] /\ x_coord <
                pieces_corners[i, x] + get_dimension(i, x) then
                get_dimension(i, y) else 0 endif) = roll_height) /\
```

```
        forall (y_coord in Y_COORDINATES) (
               sum(i in PIECES)
               (if y_coord >= pieces_corners[i, y] /\ y_coord <
                 pieces_corners[i, y] + get_dimension(i, y) then
                 get_dimension(i, x) else 0 endif) = roll_width);
```

2.5.2 Pieces with same dimension

If an input instance has multiple pieces of the same dimension, there is no need to consider the solutions where the positions of these equal pieces are swapped. In fact, these solutions will be all equivalent, the only difference will be that pieces with the same dimension will be interchanged, but since they have the same dimension, the solution will look the same.

To do this, it is sufficient to lexicographically order pieces of the same dimension. The constraint is then

$$\forall_{\substack{i,j \in PIECES, \\ i<j \ \wedge \\ pieces\_dimensions[i,\ x]=pieces\_dimensions[j,\ x] \ \wedge \\ pieces\_dimensions[i,\ y]=pieces\_dimensions[j,\ y]}}$$

$$[pieces\_corners[i,\ axis] \mid \text{for } axis \in AXES]$$

$$<$$

$$[pieces\_corners[j,\ axis] \mid \text{for } axis \in AXES]$$

where $<$ is a lexicographical comparison.

MiniZinc has a global constraint which performs the lexicographical comparison between two specified arrays. This constraint is ***lex_less***, where "lex_less" stands for "lexicographically smaller".

```
constraint forall(i, j in PIECES where i < j /\
pieces_dimensions[i, ..] == pieces_dimensions[j, ..])
(lex_less(pieces_corners[i, ..], pieces_corners[j, ..]));
```

This constraint works if the pieces can or cannot be rotated, but if we can rotate pieces, we can do another optimization on the pieces of the same dimension. We need to modify this constraint by considering two pieces being of the same dimension also if one of the two pieces has the same dimension of the other, but width and height are inverted. We need this because if a piece which has the same dimensions of another, but its width and height are swapped with respect to the other piece, is rotated, then it will have exactly the same dimension and shape of the other piece, and for this reason these two pieces need to be ordered as well.

The constraint described before is modified to

$$\forall_{\substack{i,j \in PIECES, \\ i<j \ \wedge \\ (pieces\_dimensions[i,\ x]=pieces\_dimensions[j,\ x] \wedge \\ pieces\_dimensions[i,\ y]=pieces\_dimensions[j,\ y]) \\ \vee \\ (pieces\_dimensions[i,\ x]=pieces\_dimensions[j,\ y] \wedge \\ pieces\_dimensions[i,\ y]=pieces\_dimensions[j,\ x])}}$$

$$[pieces\_corners[i,\ axis] \mid \text{for } axis \in AXES]$$

$$<$$

$$[pieces\_corners[j,\ axis] \mid \text{for } axis \in AXES]$$

where $<$ is a lexicographical comparison between two arrays. In MiniZinc, this is written as the following:

```
constraint forall(i, j in PIECES where i < j /\
((pieces_dimensions[i, x] == pieces_dimensions[j, y] /\
  pieces_dimensions[i, y] == pieces_dimensions[j, x]) \/
(pieces_dimensions[i, x] == pieces_dimensions[j, x] /\
  pieces_dimensions[i, y] == pieces_dimensions[j, y])))
(lex_less(pieces_corners[i, ..], pieces_corners[j, ..]));
```

3 Optimization

In this section, the constraints and techniques used to optimize the search of the solution will be described. Also, a comparison of the performance of the model before and after applying these techniques will be showed.

3.1 Basic optimization techniques

As written in the MiniZinc tutorial, to optimize a CP model, some basic optimization techniques are:

- use a small number of variables
- use a small domain sizes of variables
- use a succinct, or direct, definition of the constraints of the model
- use global constraints as much as possible

Both general and standard models have been coded following these guidelines.

3.2 Optimization constraints

Optimizing a model means reducing the number of possible solutions by the use of constraints, which also means pruning the search tree and reducing the search space. This allows to solve more instances because, with a reduced search space, hardest instances will require less time to be solved. In order to do so, it is possible to add constraint which are specifically designed for this purpose.

For the standard model, there's no optimization to do.

3.2.1 General model

For the general model, an optimization constraint can be added in order to optimize it, which deals with managing the rotation of square pieces.

If a piece is square (its width and height are the same) there is no need to consider also solutions where the square pieces are rotated. So, it is possible to add a constraint which excludes these solutions by forcing square pieces to not be rotated:

$$\forall i \in PIECES,\ pieces\_dimensions[i,\ x] = pieces\_dimensions[i,\ y]$$

$$\neg pieces\_rotation[i]$$

In MiniZinc, this becomes the following:

```
constraint forall(i in PIECES where pieces_dimensions[i, x] ==
  pieces_dimensions[i, y])
(not pieces_rotation[i]);
```

The constraint specified in **Section 2.5.2** is also an optimization constraint since it manages pieces with the same dimension by ordering them, as explained in that section. In particular, it is a symmetry breaking constraint.

3.3 Search strategy

MiniZinc allows the use of search annotations to optimize the model by trying different search strategies to find the solution. Search annotations are useful to change the behaviour of the model in searching the solution and inexploring the search space, in order to find other search strategies that allow to solve a higher number of instances. The number of total solutions is not reduced by these annotations, so the search tree is not pruned by changing strategy. That can be done using special constraints which are discussed in **Section 3.2**. The standard model was executed multiple times over all the instances using different search strategies: *input_order*, *first_fail*, *dom_w_deg* and using no strategy. The comparison was made using *indomain_min* and *indomain_split* alternatively as value choice annotations. **Table 1** represents a comparison between the different tries executed with different search strategies. The row in bold text represents the best found search strategy. The comparison is done using as metrics the average solving time, the average number of failures and the number of unsolved instances and also each try was done with a time limit of 5 minutes. Since the

corner variables are represented by a 2D matrix, it is necessary to "linearize" the search. For this reason, two MiniZinc's *int_search* search annotations, with the same search strategy and the same value choice annotation, were used in the search of the best strategy, one for the x coordinates and the other for y coordinates of the corners. These two annotations were used in combination with MiniZinc's *seq_search* in order to perform the search sequentially.

The best performing strategy was *dom_w_deg* with *indomain_min* as value choice annotation. Also, as written in MiniZinc Tutorial:

> "Some variable selection strategies make use of information gathered from earlier search and hence will give different behaviour, for example *dom_w_deg*."

For this reason, *dom_w_deg, indomain_min* was chosen. **Figure 2** represents a comparison between the model with the best found strategy and the model with no strategy. It is possible to see how using search strategies improved the efficiency of the search process.

Search strategy	Time	Failures	Unsolved
input_order, indomain_min	0:02:41.955	964584	16
input_order, indomain_split	0:02:35.747	891385	15
first_fail, indomain_min	0:01:25.960	479432	9
first_fail, indomain_split	0:01:26.247	451353	8
dom_w_deg, indomain_min	**0:00:44.242**	**205520**	**4**
dom_w_deg, indomain_split	0:00:51.478	259749	4

Table 1: Performances of the model with different search strategies

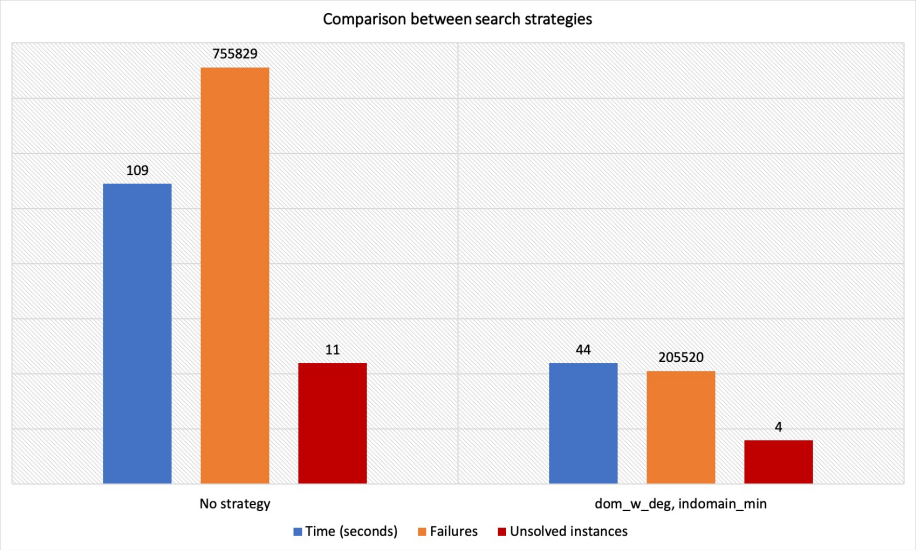

Figure 2: Performances of model with and without search strategies

Restart annotation	Time	Failures	Restarts	Unsolved
restart_linear(500)	0:00:26.996	92823	10	1
restart_linear(1000)	0:00:31.499	0:00:31.499	8	2
restart_linear(1500)	0:00:26.862	90945	5	2
restart_linear(2000)	0:00:28.096	105299	5	0
restart_geometric(1.5, 500)	0:00:34.915	135297	5	2
restart_geometric(1.5, 1000)	0:00:27.195	103132	4	2
restart_geometric(1.5, 1500)	0:00:29.408	119453	4	2
restart_geometric(1.5, 2000)	0:00:36.236	119858	3	3
restart_geometric(2, 500)	0:00:35.909	134511	4	2
restart_geometric(2, 1000)	0:00:30.943	109202	3	2
restart_geometric(2, 1500)	0:00:29.686	97901	3	3
restart_geometric(2, 2000)	0:00:22.500	86381	3	1
restart_luby(500)	0:00:36.356	141159	77	2
restart_luby(1000)	**0:00:22.315**	**99251**	**33**	**0**
restart_luby(1500)	0:00:31.537	117682	26	1
restart_luby(2000)	0:00:27.898	101267	18	1

Table 2: Performance of the model with different restart annotations, using *dom_w_deg* strategy

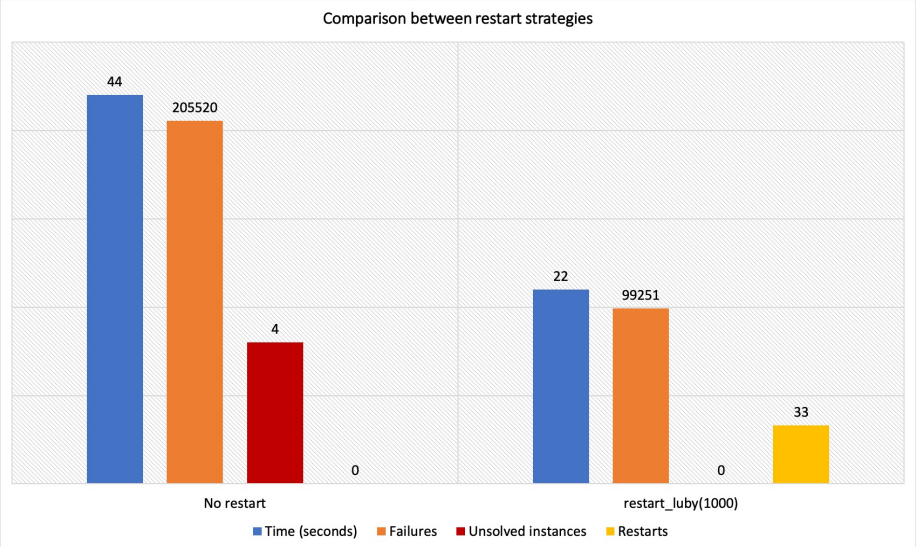

Figure 3: Performances of model with and without restart strategies

3.4 Restart

MiniZinc allows the use of restart annotations to control the restart behaviour to try solving more instances in a shorter time by restarting the search with a specific criteria. Different restarts have been tried. In particular, *restart_linear*, *restart_geometric* and *restart_luby* with different parameters were tried. **Table 2** describes a comparison of the different restart strategies tried, in a time interval of 5 minutes, using as metrics the average solving time, the average number of failures, the average number of restarts and the number of unsolved instances.

It is possible to see that the best restart strategy was *restart_luby(1000)*, which allowed to solve all the instances within 5 minutes with a low average time. **Table 2** represents a comparison between the different search strategies, using average time, average number of failures, average number of restarts and number of unsolved instances as metric. **Figure 3** represents a comparison between the model with and without the best restart strategy.

4 Conclusions

To solve hardest instances, as said in **Section 2.4**, a modified cumulative implied constraint was used. Choosing the right implied cumulative constraint was a difficult challenge: in fact, as said in section **Section 2.4**, these modified constraint will cause the model to fail in case an input instance has pieces which can be cutted from the paper roll without consume it all. This is not a problem, however, since all our input instances make use of all the paper roll, so this won't be a problem in our case.

Coming up with this model was not easy and required a lot of different experimentations, using different optimization constraints (like symmetry breaking), different search strategies etc. which did not prove to be useful in the end and, in some cases, were actually slowing or making the model worse.