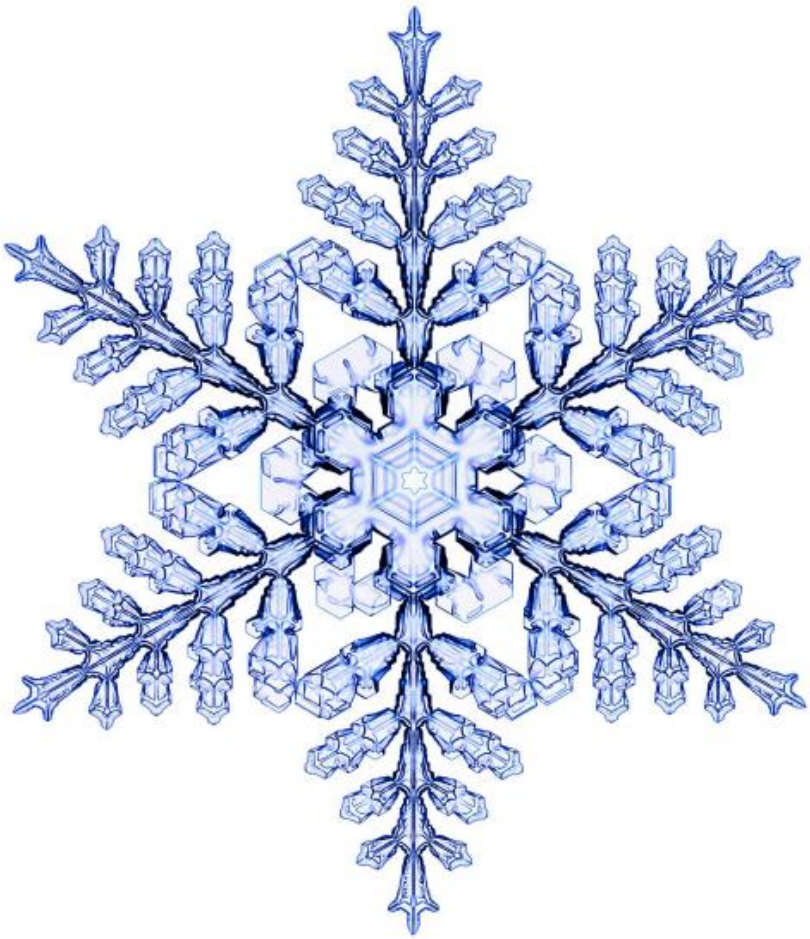
Combinatorial Decision Making and Optimization (A. Y. 2019-2020)

Project Work

Modelling and Solving the Present Wrapping Problem



Report by
Johnny Agosto, Chiara Malizia

Professor
Zeynep Kiziltan

* Johannes Kepler, On the Six-Cornered Snowflake, 1611: six-fold rotation symmetry of snowflakes.

Agenda

Introduction

The purpose of the following report is to describe how we solved the Present Wrapping Problem using the MiniZinc CP solver [1]. Given a wrapping paper roll of a certain dimension and a list of presents, the aim of the Present Wrapping Problem (PWP) is to decide how to cut off pieces of paper so that all the presents can be wrapped.

Specifications

We were asked to produce both trivial and optimised models, and to find solutions for 33 different instances with increasing complexity. Some initial suggestions were given to help us reason and gradually create a model able to face all the problem challenges. In total, we developed five models, named Base Model, Symmetry Model, Rotation Model, Multiple Instances Model and Final Model. We also realized a CLI python program *CP.py* to have a user-friendly interface that allowed us to choose the instance(s) to solve, which model to consider, where to save the outputs and other optional parameters, such as the timeout to use.

Input Data

An input instance of PWP is a text file consisting of lines of integer values:

- The first line gives the width and the height of the paper roll.
- The second line gives the number of necessary pieces of paper to cut off.
- Finally, it follows as many lines as the number of necessary pieces of paper to cut off, each with the horizontal and vertical dimensions of the *i*-th piece of paper.

However, the input instances needed to be first formatted to get fed into our models, and then saved onto .dzn files, which are easier to use with Minizinc. We performed these operations in python by means of the *convert_txt2dzn.py* file. The image below shows an example of the translation from a given .txt file to the corresponding .dzn file.

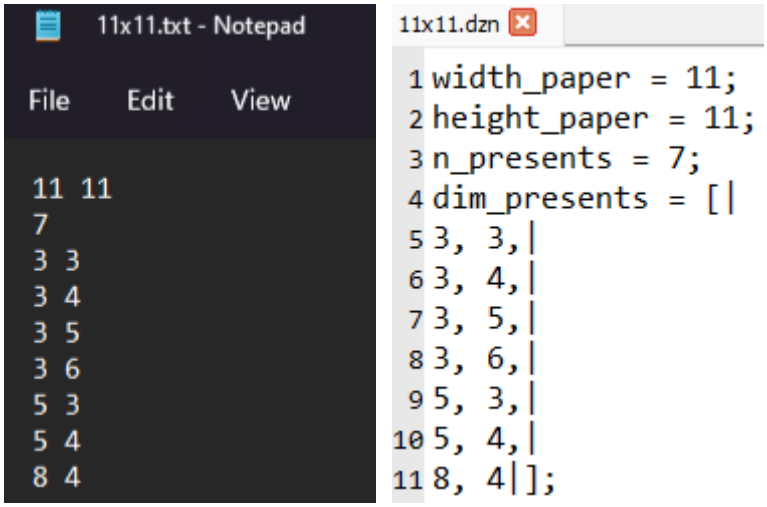


Figure 1: Example of conversion from .txt file (left) to .dzn file (right).

Output Data

The specifications require that each instance output should indicate the position of each *i*-th present by the coordinates of its left bottom corner, by adding them next to the numbers describing its horizontal and vertical dimensions in the instance file. Moreover, the output file corresponding to an instance *w* × *l.txt* is named *w* × *l–out.txt*. The figure below illustrates for example the instance *w* × *l.txt* and the output file *w* × *l–out.txt*.

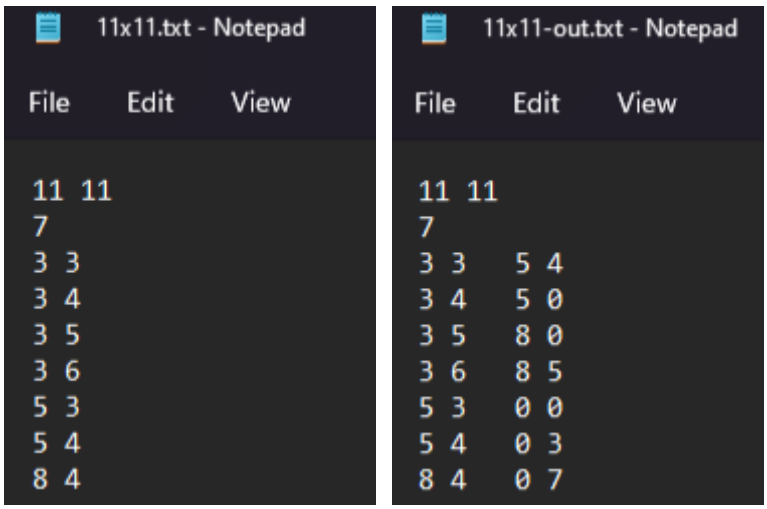


Figure 2: Example of input instance and its corresponding output solution.

In addition, when asked to consider the possibility to rotate a piece, each row describing a present in the output file ends with “True” or “False” to say that the present is rotated or not respectively.

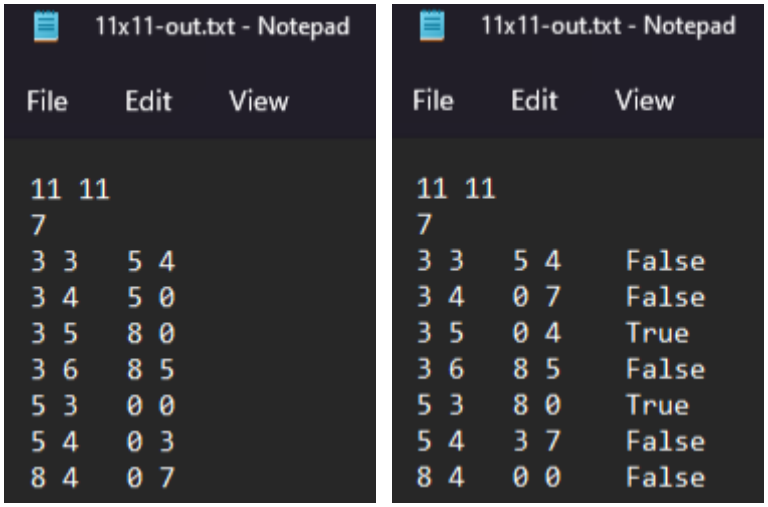


Figure 3: Example of input instance ad its corresponding output solution when pieces rotation is allowed.

In addition to the *-out.txt* files, we decided to save:

- The plot of the solution of each instance, as .jpg images, for visualization purpose.
- The statistics of each instance obtained by the solving process in order to better understand the behaviour of the model. The statistics are all saved into a csv file: each row is related to an instance. We created a csv file per model.

1. Base Model

The first model we created is the Base Model, which is the baseline model providing the starting point of the project. We employed it to define parameters and variables, to analyse the essential constraints for the model functioning and to investigate different search strategies to find the problem solution.

1.1 Parameters and Variables

In this section we describe the parameters and the variables used to model the problem.

1. Parameters

- An integer value *width_paper* representing the paper roll width.
- An integer value *height_paper* representing the paper roll height.
- An integer value *n_presents* representing the number of presents to place in the paper roll.
- A set *PRESENTS* of integer values defining the n presents to place as integers from 1 to *n_presents*.
- An integer value *x_axis* representing the x axis of the paper roll.
- An integer value *y_axis* representing the x axis of the paper roll.
- A set of integer values *AXES* representing the possible axes on which the problem is defined.
- A matrix *dim_presents* of integer values representing the dimensions of piece of paper needed to wrap each present. *dim_presents* has as many rows as *PRESENTS* and as many columns as *AXES.*
- A set *X_COORDS* of integer values defining all the possible values that x coordinates can assume.
- A set *Y_COORDS* of integer values defining all the possible values that y coordinates can assume.

2. Variables

Our goal is to find the bottom left coordinates of each present. We modeled the problem with a 2-dimensional matrix of variables, ***coord_presents***, having as many rows as *PRESENTS* and as many columns as *AXES*. Hence each row *i* of the ***coord_presents*** matrix will contain the *(x,y)* bottom left coordinate of the *i*-th present. The variables can assume a value belonging to the domain $0\,..\,max([width\_paper, height\_paper]) - 1$.

```
array[PRESENTS, AXES] of var 0..max([width_paper, height_paper])-1: coord_presents;
```

1.2 Main Problem Constraints

1. Domain Constraint

The domain of the decision variable has been modified in a way such that the bottom left coordinates summed with its corresponding dimension are lesser than or equal to the paper roll dimensions.
By isolating the decision variable, the resultant constraint is:

```
constraint forall(i in PRESENTS)(
        (coord_presents[i, x_axis] <= width_paper - dim_presents[i, x_axis]) /\
        (coord_presents[i, y_axis] <= height_paper - dim_presents[i, y_axis]));
```

Constraint 1.1

2. Non-Overlapping Constraint

We constrained that two presents do not overlap.
Two presents do not overlap if one of them has zero size or if there exists at least one dimension where their projections do not overlap [2].

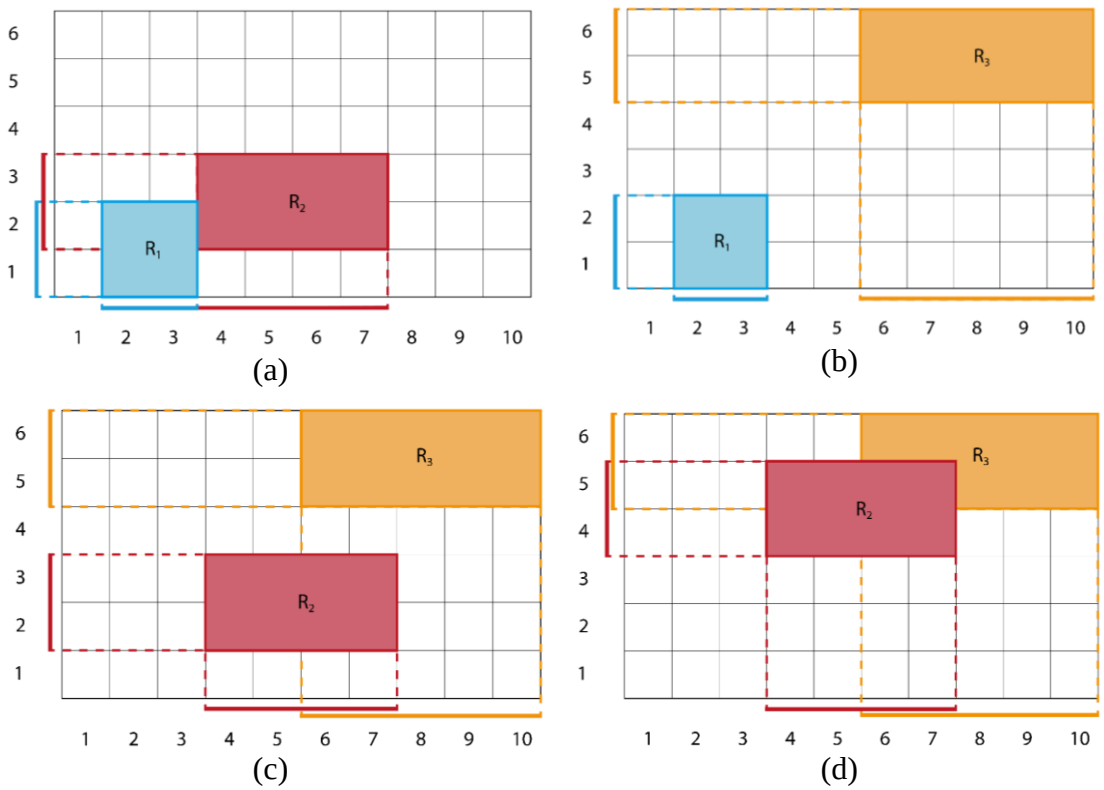


Figure 4: Illustration of why pairs of rectangles overlap or do not.

(a) R1 and R2 do not overlap since their projections onto x-axis do not intersect. (b) R1 and R2 do not overlap since their projections onto both dimensions do not intersect. (c) R1 and R2 do not overlap since their projections onto y-axis do not intersect. (d) R1 and R2 overlap since their projections onto both dimensions intersect.

We implemented the non-overlapping constraint by using the global constraint *diffn* provided by MiniZinc.

```
constraint diffn(coord_presents[..,x_axis],
                 coord_presents[..,y_axis],
                 dim_presents[.., x_axis],
                 dim_presents[.., y_axis]);
```

Constraint 1.2

1.3 Implied Constraints

In any solution, if we draw a vertical line and sum the vertical sides of the traversed pieces, the sum can be at most equal to the paper height. A similar property holds if we draw a horizontal line.
To design the implied constrained we reasoned as follows:
for each value that the x (*resp.* y) coordinate can assume, we check if the sum of the heights (*resp.* widths) of the presents, whose projection onto x-axis (*resp.* y-axis) encounters that value, is at most the paper height (*resp.* width).

```
constraint forall (x in X_COORDS) (
                sum(i in PRESENTS)
                    (if x >= coord_presents[i, x_axis] /\
                     x < coord_presents[i, x_axis] + dim_presents[i, x_axis]
                    then dim_presents[i, y_axis] else 0 endif) <= height_paper)
                /\
               forall (y in Y_COORDS) (
                 sum(i in PRESENTS)
                    (if y >= coord_presents[i, y_axis] /\
                    y < coord_presents[i, y_axis] + dim_presents[i, y_axis]
                    then dim_presents[i, x_axis] else 0 endif) <= width_paper);
```

Constraint 1.3

However, we reformulated the constraint from a time-scheduling point of view by exploiting the cumulative global constraint provided by MiniZinc. It considers a set T of tasks described by start time *s*, duration *d* and resource requirements *r*, and ensures that they don't exceed a global resource bound *b* at any one time.
In our case, the cumulative constraint enforces that at each point in time (i.e., each value of *X_COORD*), the cumulated height of the set of tasks (i.e., *PRESENTS*) that overlap that point, does not exceed a given limit (i.e., *height_paper*) [3].
A task *t* overlaps a point *i* if and only if:

1. its origin (i.e., the x bottom left coordinate) is less than or equal to *i*, and
2. its end (i.e., the sum of the x bottom left coordinate and *width_present*) is strictly greater than *i*.

```
constraint cumulative(coord_presents[..,x_axis],
                      dim_presents[..,x_axis],
                      dim_presents[..,y_axis],
                      height_paper);
```

Constraint 1.4.1

The same constraint is applied to the other axis.

```
constraint cumulative(coord_presents[..,y_axis],
                      dim_presents[..,y_axis],
                      dim_presents[..,x_axis],
                      width_paper);
```

Constraint 1.4.1

1.4 Search and Results

Since the model is meant to be independent of solving technique in MiniZinc, the search is encoded in annotations that allow to specify how to search in order to find a solution to the problem. MiniZinc provides both variable selection and value choice annotations.
For variable selection strategy we tested:

- *input_order*: choose in order from the array of variables.
- *first_fail*: choose the variable with the smallest 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.

For value selection strategy we tried:

- *indomain_min*: assign to the variable its smallest domain value.
- *indomain_random*: assign to the variable a random value from its domain.

Moreover, in the case of *dom_w_deg*, we added restart annotations, which control how frequently a restart occurs. We investigated:

- *restart_constant(100)* : restart after 100 nodes.
- *restart_linear(100):* first restart after 100 nodes. The second restart gets twice as many nodes, the third gets three times, etc.
- *restart_luby(100)* : the *k*-th restart gets 100*L[*k*] where L[*k*] is the *k*-th number in the Luby sequence. The Luby sequence looks like: 1 1 2 1 1 2 4 1 1 2 1 1 2 4 8 …, that is it repeats two copies of the sequence ending in *2^i* before adding the number *2^{i+1}*.
- *restart_luby(1000)* : same as the previous one but the *k*-th restart gets 1000*L[*k*].

All the experiments have been done by imposing a timeout of 5 minutes, using as CPU an AMD Ryzen 7 3700X with 16 threads. In the end we tried the following combinations:

- First_fail_indomain_min (ff_im)
- Input_order/indomain_min (io_im)
- Input_order/indomain_random (io_ir)
- Dom_w_deg/indomain_min with restart luby 1000 (dwd_im)
- Dom_w_deg/indomain_min with restart luby 100 (dwd_im100)
- Dom_w_deg/indomain_min with restart costant 100 (dwd_imcost)
- Dom_w_deg/indomain_min with restart linear 100 (dwd_imlin)
- Dom_w_deg/indomain_random with restart luby 1000 (dwd_ir)
- Dom_w_deg/indomain_random with restart luby 100 (dwd_ir100)
- Dom_w_deg/indomain_random with restart costant 100 (dwd_ircost)
- Dom_w_deg/indomain_random with restart linear 100 (dwd_irlin)

To find the best search, we used as metrics the number of solutions found per search strategy [4]. The figure below shows the comparison among the different trials.

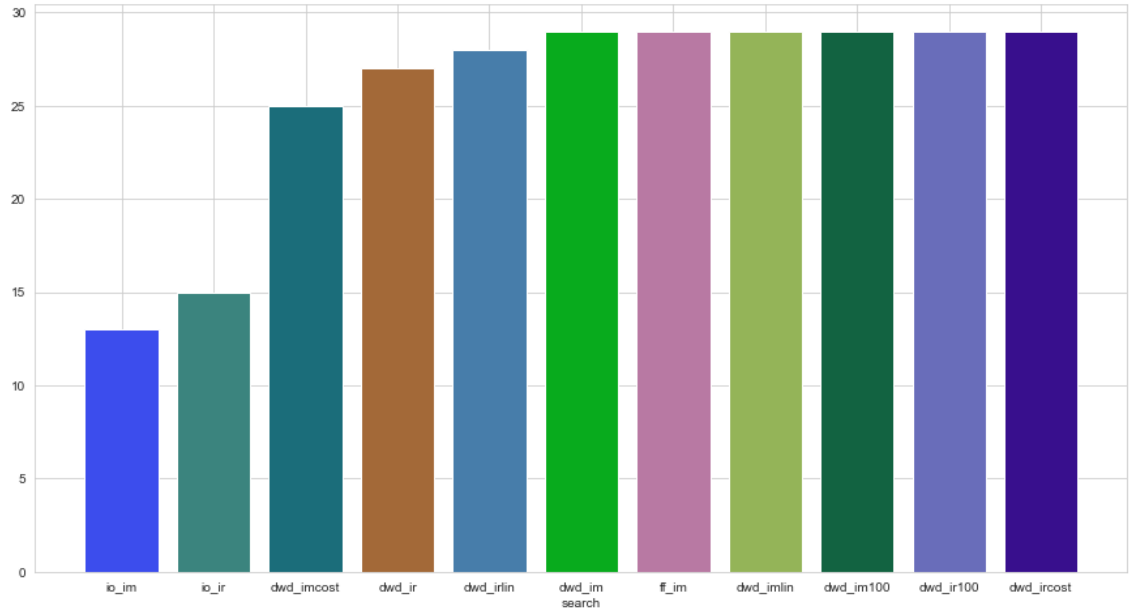


Figure 5: Number of solutions found per search strategy.

In case of equal podium, we decided to further investigate in order to find the best search. In such a case, the comparison is done using as metrics the average solving time and the average number of failures among all the instances for which we find a solution. Thus, we created two different ranks, one for each metric, and each search strategy receives a score depending on the chart position: given n search strategies, we assigned a value equal to *n* to the top one, *n-1* to the second ranked strategy etc. It follows that each search strategy had two scores, and the final score was obtained by averaging them. The final score was then normalized and expressed as a percentage. All these operations are in the *analysis.ipynb* file.

To exemplify, a score equals to 100 means that the considered search strategy had the best average solving time and the best average failures.

Search	**Score**
ff_im	100.00
dwd_im100	83.33
dwd_imlin	66.66

Table 1: Score of the top three search strategies.

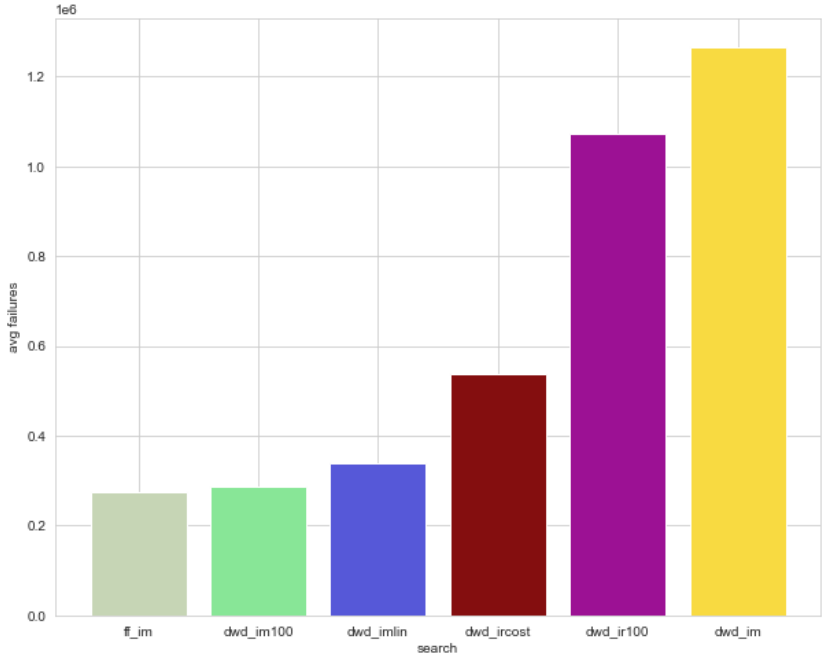


Figure 6: Average number of failures per search strategy.

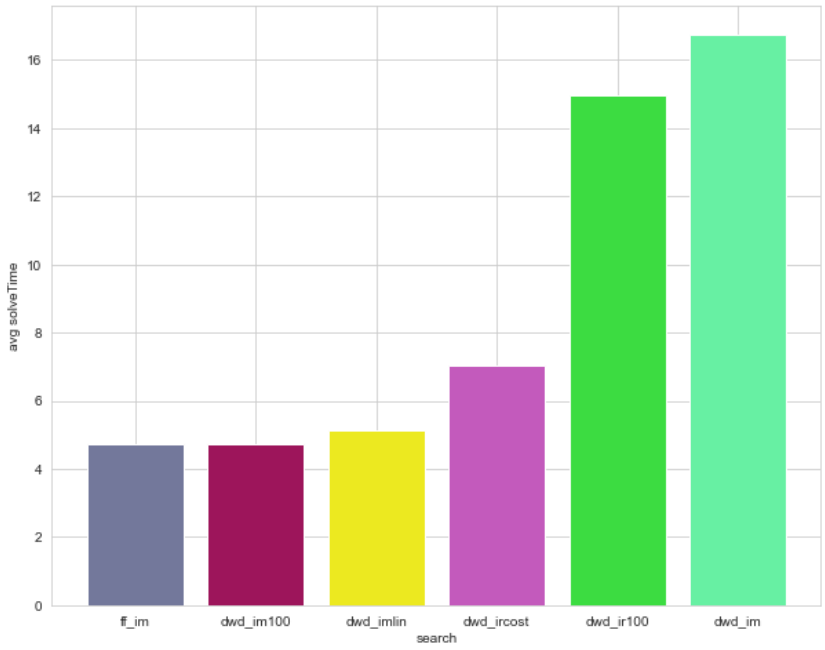


Figure 7: Average solving time per search strategy.

According to our metrics, figure 6 and figure 7 proves that first_fail/indomain_min and dom_w_deg/indomain_min with restart_luby(100) are those having the lowest average solving time and the average number of failures.

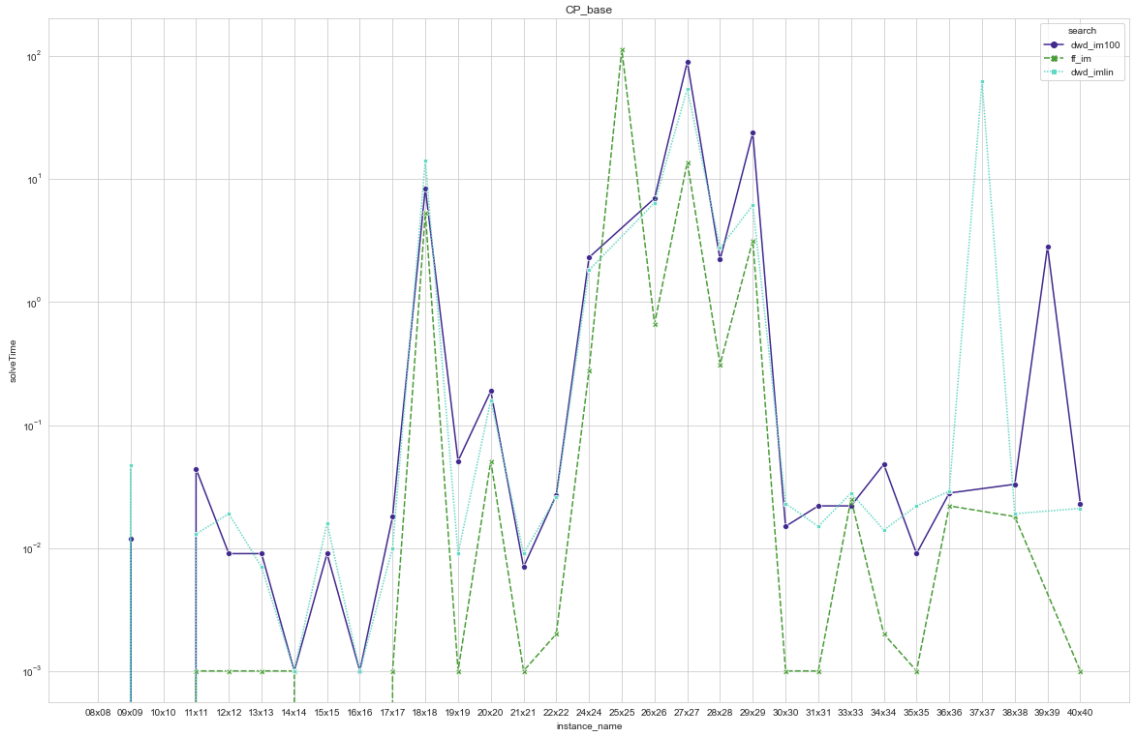


Figure 8: Solving time per instance.

Figure 9: Failures per instance.

2. Symmetry Model

The Symmetry Model is basically an enhanced Base Model which benefits from the symmetry breaking technique.

The main point of this approach is to exploit symmetry to reduce the amount of search needed to solve the problem. A common way to proceed is to add a **symmetry breaking constraint**, whose goal is never to explore two search states which are symmetric to each other, since we know the result in both cases must be the same. However, it is worth pointing out that the existence of at least one solution will be preserved.

Given a solution, we wanted to remove its vertical flip, its horizontal flip, and a combination of both (namely its 180 degrees rotation), as shown in the figure below.

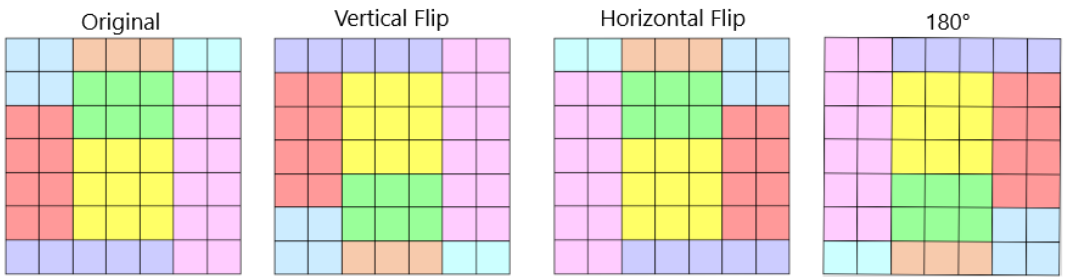


Figure 10: Symmetric variants of a solution (labeled “Original”).

A fast and easy way to remove symmetry is to divide the paper roll in four quadrants and enforce a present to be placed in one of them. We decided to constrain the presents having the largest area: we considered the first element of the array *order_indexes*, a new array of integer representing the index of presents sorted by area in decreasing order.

```
array[PRESENTS] of int: order_indexes = reverse(sort_by( PRESENTS,
                                [dim_presents[i,x_axis] * dim_presents[i,y_axis] | i in PRESENTS]));
```

We forced then the considered present to stay in the bottom left quadrant using the following symmetry breaking constraint:

```
constraint symmetry_breaking_constraint (
        coord_presents[order_indexes[1], x_axis] < (width_paper) / 2) /\
        (coord_presents[order_indexes[1], y_axis] < (height_paper) / 2);
```

Constraint 2.1

We also decided to impose an order between pairs of presents: considering two subsequent presents from biggest to smallest, we checked if the two had the same bottom left value along the y_axis. In such a case, we constrained the value of the bottom left coordinate along the x_axis of the first present (namely, the bigger one) to be smaller than the value of the bottom left coordinate along the same axis of the second present.

```
constraint forall (i in PRESENTS) (if (i!=n_presents /\
        coord_presents[order_indexes[i],y_axis]=coord_presents[order_indexes[i+1],y_axis])
        then (coord_presents[order_indexes[i],x_axis] + dim_presents[order_indexes[i],x_axis]
        <= coord_presents[order_indexes[i+1],x_axis])
        endif);
```

Constraint 2.2

2.1 Search and Results

Heretofore we preliminary sorted the presents according to a prefixed policy (namely, decreasing area), and placed the biggest present in the bottom left quadrant. Moreover, whenever two presents have the same bottom left coordinate value along the y_axis, we imposed the bigger one to have a smaller bottom left coordinate value along the x_axis.

Guided by common sense, we decided that first_fail and dom_w_deg are those that suit best: both basically select variable which has the lowest number of available values, that is the variable encoding the biggest present. Once chosen the variable to consider, the smallest available value in the current domain should be selected. In an indomain_min fashion, we put the bottom left coordinates of each present in the first available lower left corner of the paper roll.

Henceforth, this and future models will be tested by using a timeout of 5 minutes and the following search heuristics:

- first fail/indomain_min (ff_im)
- dom_w_deg/indomain_min with luby_restart(100) (dwd_im100)
- dom_w_deg/indomain_min with luby_restart(1000) (dwd_im1000)

Using the same approach done in the Base Model, we choose the best search starting by considering first the number of solved instances, and then in case of equality, the score obtained from the average number of failures and the average solving time.

Search	**Score**
dwd_im100	100.00
dwd_im	50.00

Table 2: Score of the best search strategies.

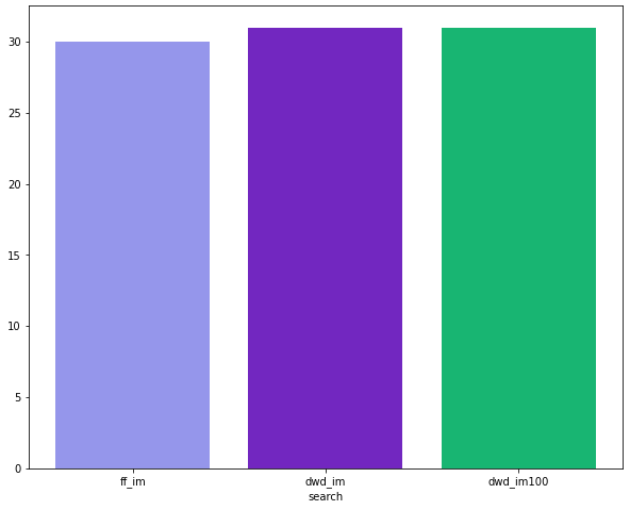


Figure 11: Number of solutions found per search strategy.

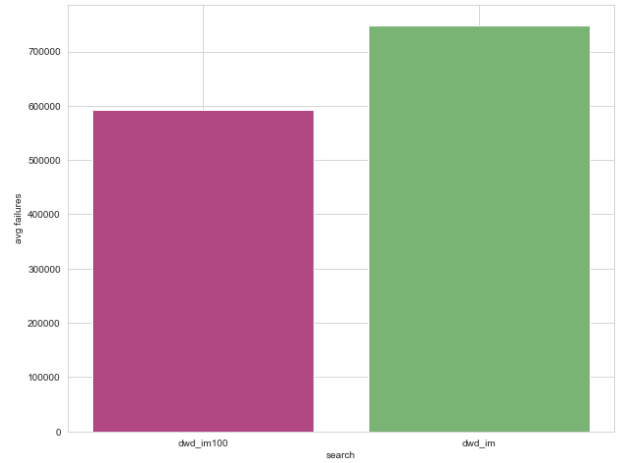


Figure 12: Average number of failures per search strategy.

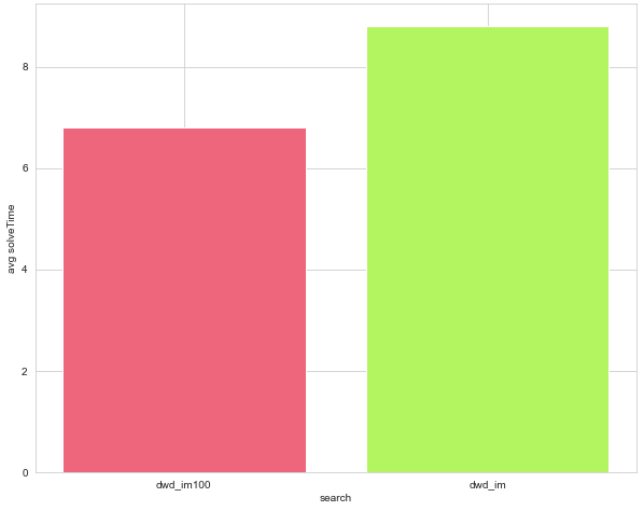


Figure 13: Average solving time per search strategy.

Figure 14: Solving time per instance.

Figure 15: Failures per instance.

3. Rotation Model

The problem requirements do not allow the rotation of the presents. This means that a *n*x*m* present cannot be positioned as an *m*x*n* present in the paper roll. Now we want this restriction to relax and modify the model accordingly. We introduce hence the **Rotation Model**, that is a new version of the Symmetry Model dealing with pieces rotation.

Generally, we can rotate an object by 90, 180, and 270 degrees. However, given that our objects have rectangular shapes, the 90 and 270 degrees rotations give the same results. The same holds for the original orientation and its 180 degrees rotation. Thus, we considered only two possible rotations.

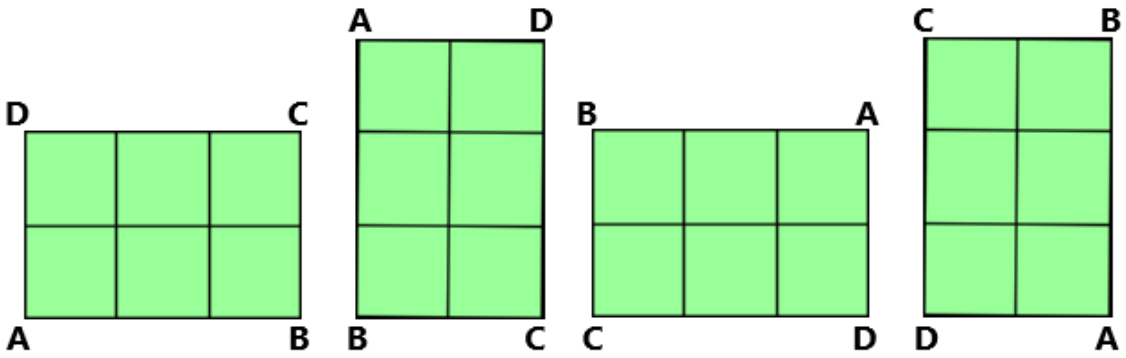


Figure 16: Possible rotation of a present.

To deal with it, we assigned an additional array ***rot*** of Boolean variables. Each variable could be *True* if the piece will be rotated (i.e., 90 degrees rotation) in our solution, or *False* if it will be not (i.e., original orientation).

```
array[PRESENTS] of var bool: rot;
```

When the present is rotated its width and height are swapped. Therefore, we need to modify the constraints considering whether the piece is rotated.

We introduced an auxiliary function ***axis_rot*** that given the *i*-th presents and its axis as input it returns the same axis if it is not rotated or the other axis if it is rotated.

```
function var int: axis_rot(int: i, int:axis) =
                  if axis = x_axis then
                  rot[i]+1
                  else not(rot[i])+1 endif;
```

The constraints 1.1, 1.2, 1.4.1, 1.4.2 and 2.2 become:

```
constraint forall(i in PRESENTS)(
    (coord_presents[i, x_axis] <= width_paper - dim_presents[i, axis_rot(i,x_axis)]) /\
    (coord_presents[i, y_axis] <= height_paper - dim_presents[i, axis_rot(i,y_axis)]));
```

Constraint 3.1: Domain constraint.

```
constraint diffn(coord_presents[..,x_axis],
                 coord_presents[..,y_axis],
                 [dim_presents[i, axis_rot(i,x_axis)] | i in PRESENTS],
                 [dim_presents[i, axis_rot(i,y_axis)] | i in PRESENTS]);
```

Constraint 3.2: Non-overlapping constraint.

```
constraint cumulative(coord_presents[..,x_axis],
                      [dim_presents[i, axis_rot(i,x_axis)] | i in PRESENTS],
                      [dim_presents[i, axis_rot(i,y_axis)] | i in PRESENTS],
                      height_paper);
```

Constraint 3.3.1: Implied constraints.

```
constraint cumulative(coord_presents[..,y_axis],
                      [dim_presents[i, axis_rot(i,y_axis)] | i in PRESENTS],
                      [dim_presents[i, axis_rot(i,x_axis)] | i in PRESENTS],
                      width_paper);
```

Constraint 3.3.2: Implied constraints.

```
constraint forall(i in PRESENTS)(
                      if (i!=n_presents /\
                      coord_presents[order_indexes[i],y_axis]=
                      coord_presents[order_indexes[i+1],y_axis])
                      then (coord_presents[order_indexes[i],x_axis] +
                      dim_presents[order_indexes[i],
                      axis_rot(order_indexes[i],x_axis)] <=
                      coord_presents[order_indexes[i+1],x_axis])
                      endif);
```

Constraint 3.4: Ordering constraint between pairs of presents.

3.1 Rotation for squared presents

Among the presents to place in the paper roll there could be squared pieces. In such a case their dimensions remain the same whatever rotation we apply. Therefore, we added a constraint to avoid rotation for squared pieces and reduce the search space.

```
constraint forall(i in PRESENTS where dim_presents[i, x_axis] == dim_presents[i, y_axis])
    (rot[i] = false);
```

Constraint 3.5

3.2 Symmetry in the Rotation Model

It is worth to point out that the Rotation Model excludes the symmetrical solutions, as it arises from the Symmetry Model that already took them into account. However, now a solution can have seven symmetric variants: rotated by 90, 180 and 270 degrees, and flipped vertically, as shown in the figure below.

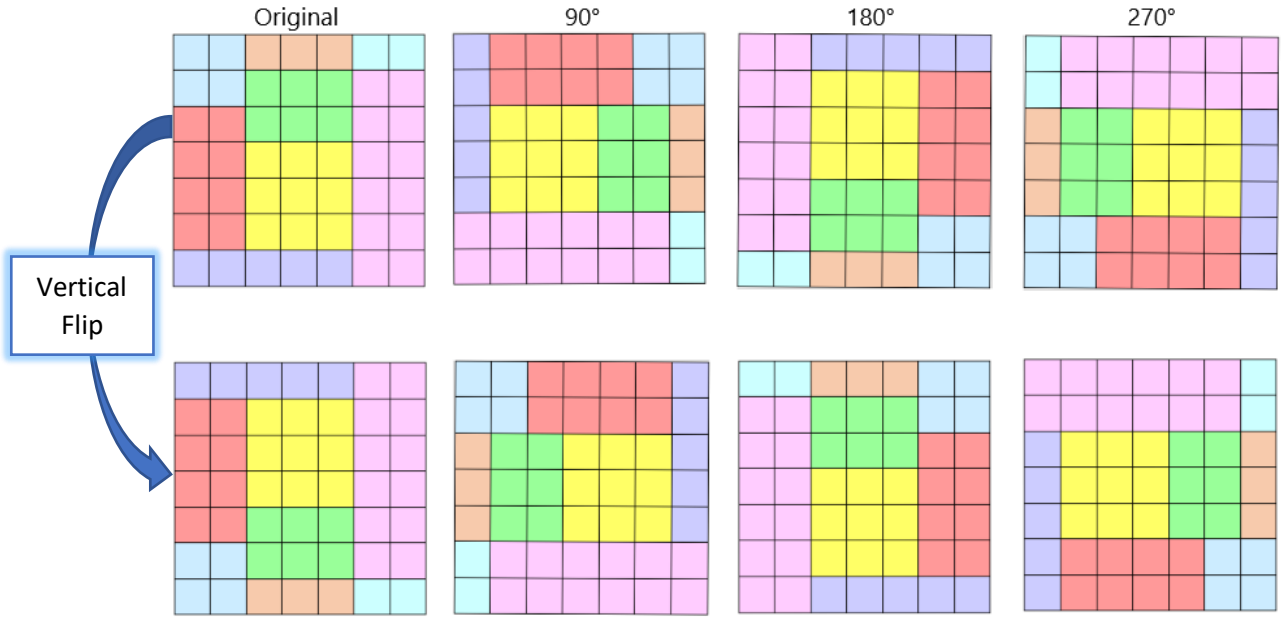


Figure 17: Symmetric variants of a solution (labeled “Original”).

3.3 Search and Results

When pieces rotation is allowed, ff_im wins the first place in the ranking of the tested search strategies. Indeed, according to the criteria defined in section 1.4, ff_im reaches the highest number of solutions found, as illustrated in the figure 18.

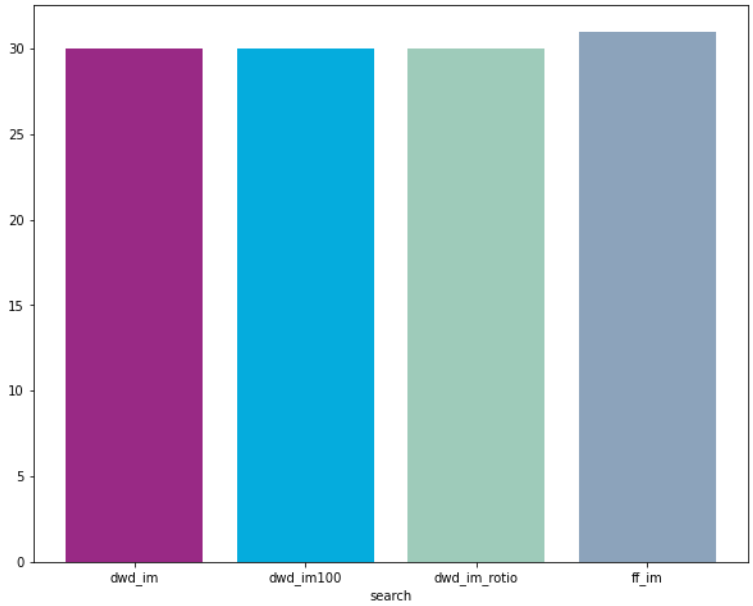


Figure 18: Number of solutions found per search strategy.

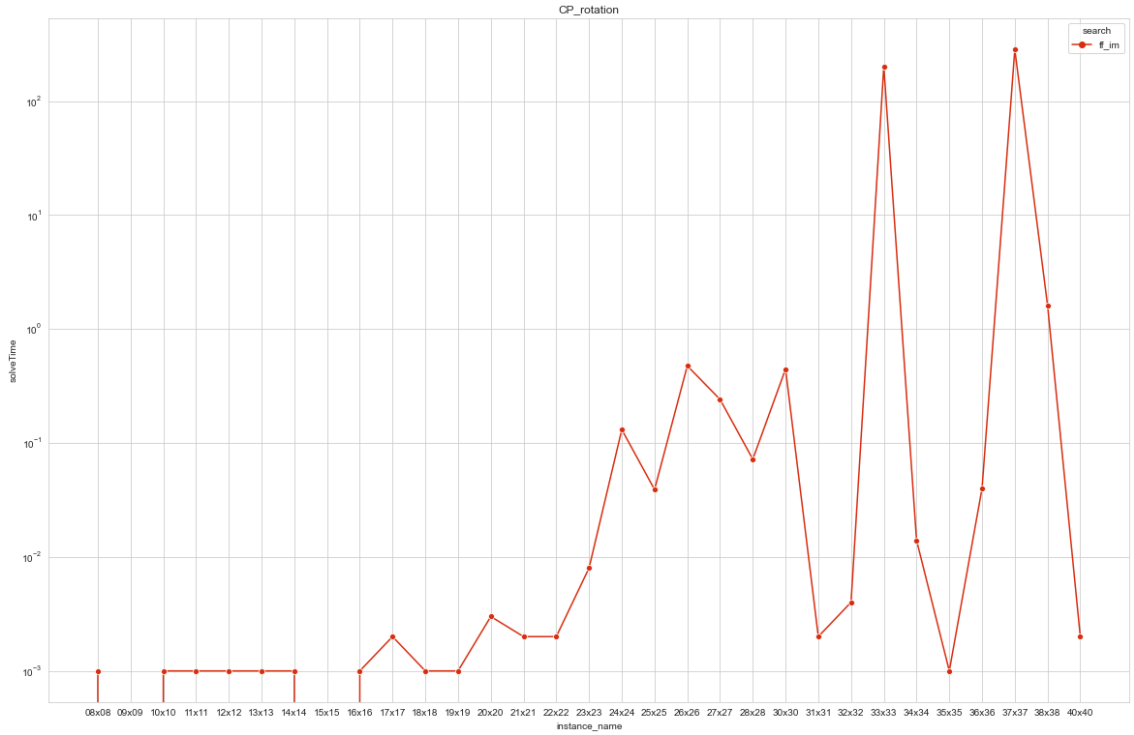


Figure 19: Number of solutions found per search strategy.

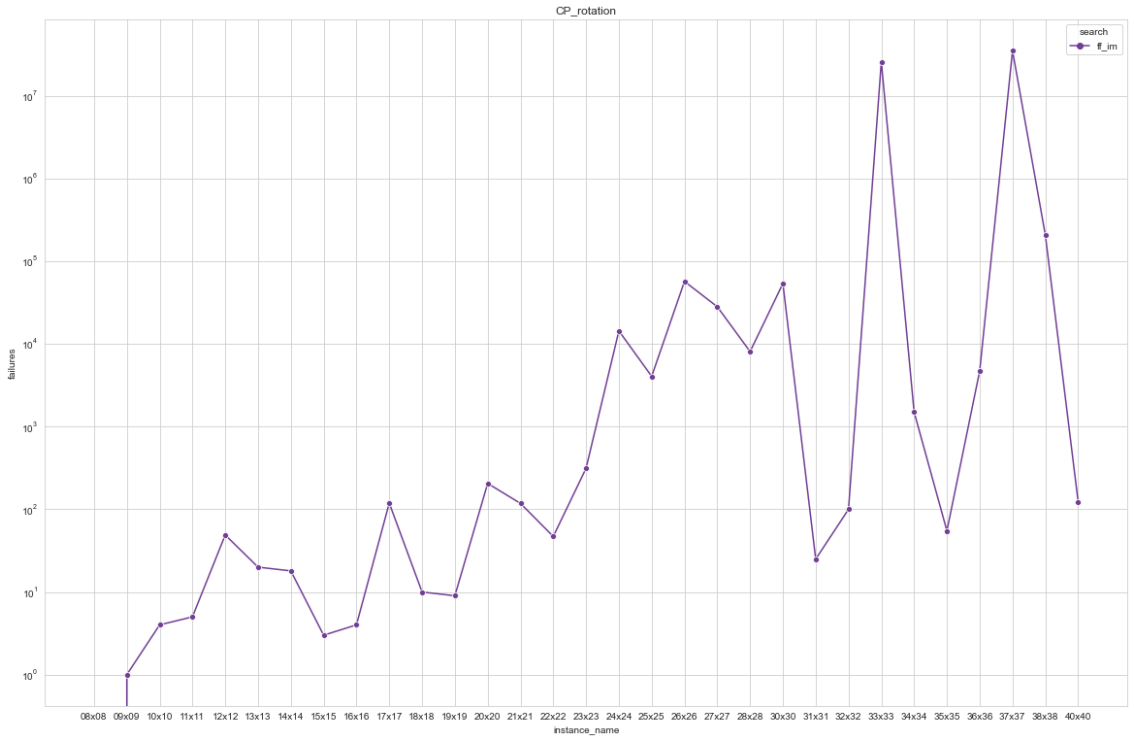


Figure 20: Failures per instance.

4. Multiple Instances Model

The provided problem instances do not have pieces of identical dimension, but in a more general case there could be several presents having same width and height. In such a case we should handle them by improving our model. Given multiple presents of the same dimensions, the solutions where their positions are swapped will be equivalent: since they have the same dimensions, the solutions will look the same.

The **Multiple Instance Model** is a variant of the Symmetry Model enlarged with the addition of a lexicographic order constraint. The goal of the lexicographic ordering constraint is to impose an order at the bottom left coordinates of the presents with same dimensions in order to reduce the number of solutions. In the following constraint we exploited the *lex_less* global constraint provided by Minizinc.

```
constraint forall(i,j in PRESENTS where j>i) (
        if (dim_presents[order_indexes[i], x_axis] == dim_presents[order_indexes[j], x_axis]
        /\
        dim_presents[order_indexes[i], y_axis] == dim_presents[order_indexes[j], y_axis])
        then lex_less(coord_presents[order_indexes[i],..],coord_presents[order_indexes[j],..])
        endif);
```

Constraint 4.1

4.1 Search and Results

Following the score criteria defined in the section 1.4, we found out the results shown in the table below.

Search	Score
dwd_im	100.00
dwd_im100	50.00

Table 3: Score of the best search strategies.

In particular, we obtained that dwd_im and dwd_im100 have the same number of solutions found per search strategy, but dwd_im has both lower average solving time per search strategy and lower average number of failures per search strategy than dw_im100.

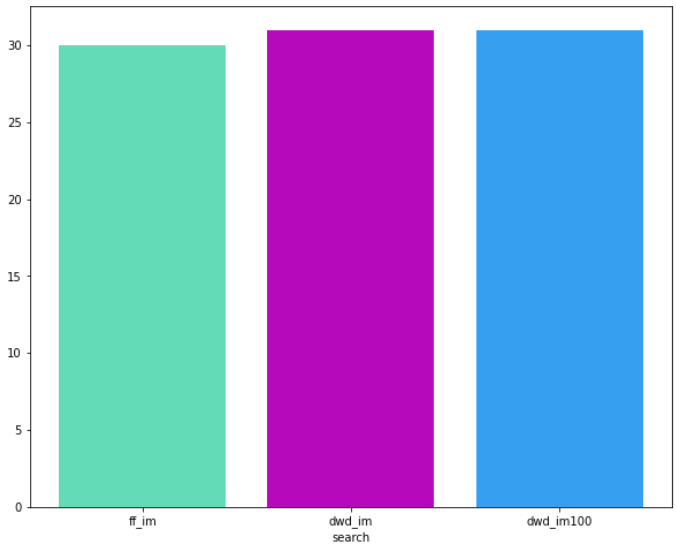


Figure 21: Number of solutions found per search strategy.

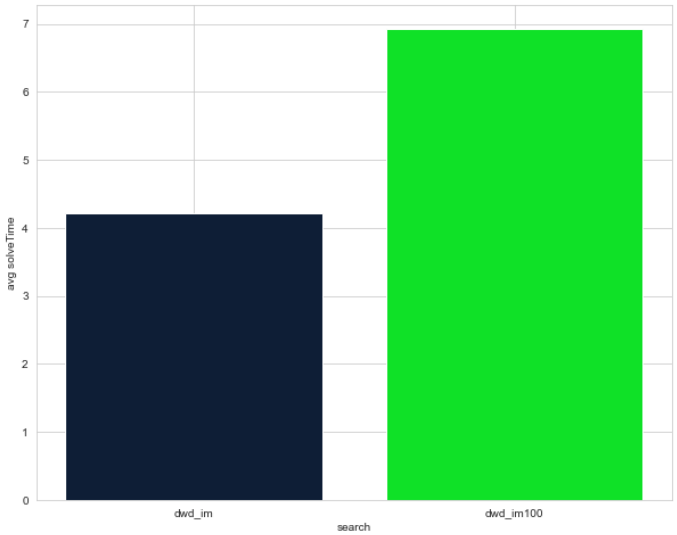


Figure 22: Average solving time per search strategy.

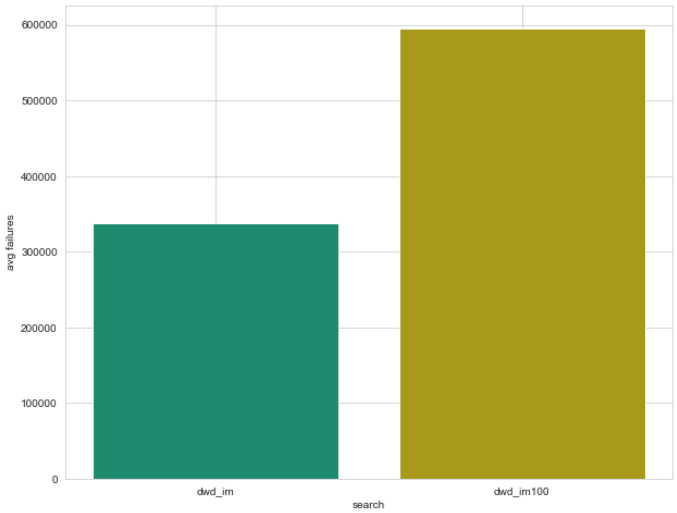


Figure 23: Average number of failures per search strategy.

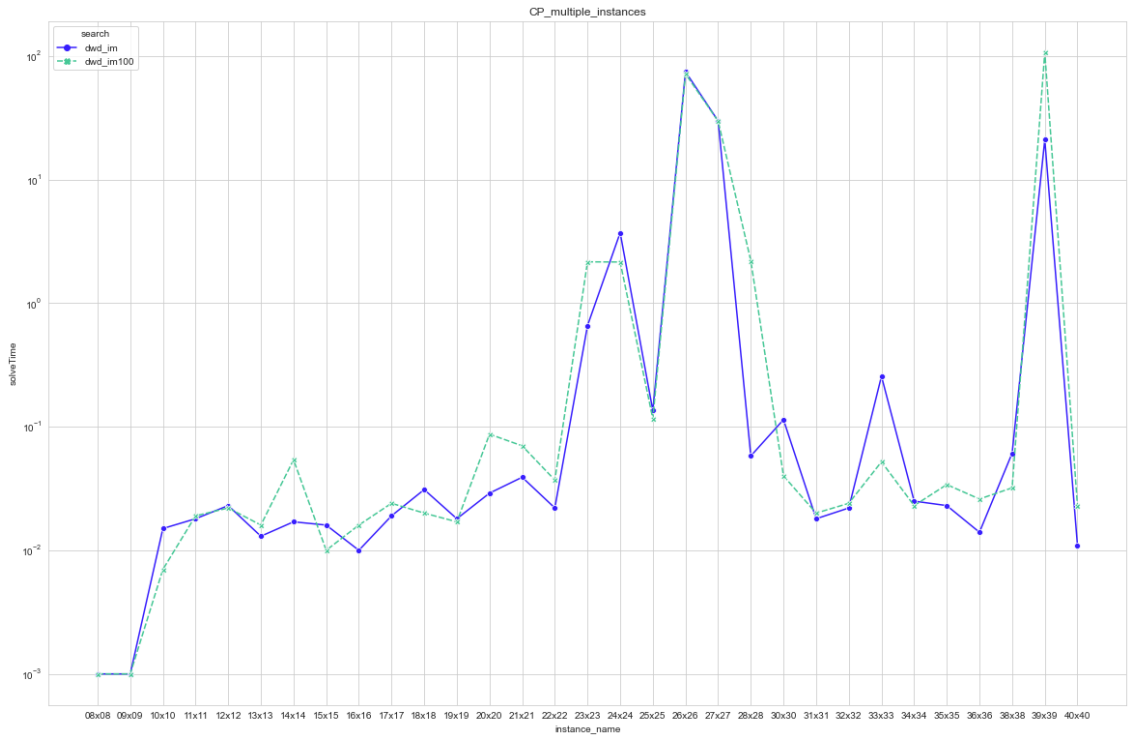


Figure 24: Number of solutions found per search strategy.

Figure 25: Failures per instance.

5. *Final Model*

The **Final Model** has the ability to handle both pieces rotations and presents with the same dimensions, and tackle symmetric solutions too.

The Final Model benefits from the constrained used so far in the Rotation and Multiple Instances Models but the constraint 4.1 should now take into account that pieces can result having the same dimension because of a possible rotation. Therefore, we exploited the helper function *axis_rot* and the constraint 4.1 becomes as follows:

```
constraint forall(i,j in PRESENTS where j>i) (
    if (dim_presents[order_indexes[i], axis_rot(i,x_axis)] ==
       dim_presents[order_indexes[j],axis_rot(j,x_axis)] /\
       dim_presents[order_indexes[i], axis_rot(i,y_axis)] ==
       dim_presents[order_indexes[j], axis_rot(i,y_axis)])
    then lex_less(coord_presents[order_indexes[i],..],coord_presents[order_indexes[j],..])
    endif);
```

Constraint 5.1

To summarize, here is the list of constraints involved in the Final Model:

- 3.1
- 3.2
- 3.3.1 and 3.3.2
- 3.4
- 3.5
- 5.1

5.1 Search and Results

In the Final Model we found out that dwd_im_100 obtains the highest score: it has lowest average solve time and lowest number of failures.

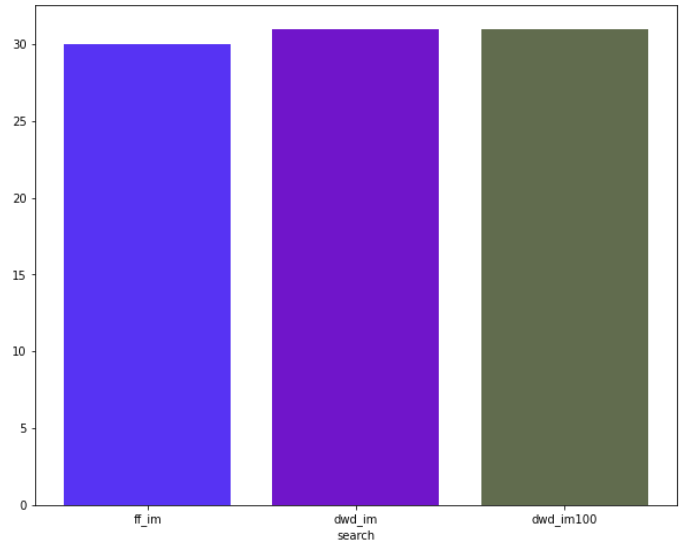


Figure 26: Number of solutions found per search strategy.

Search	Score
dwd_im100	100.00
dwd_im	50.00

Table 4: Score of the best search strategies.

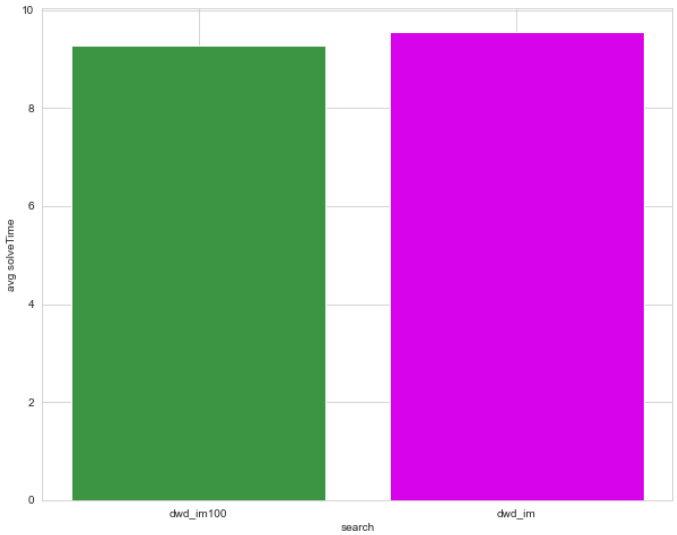


Figure 27: Average solving time per search strategy.

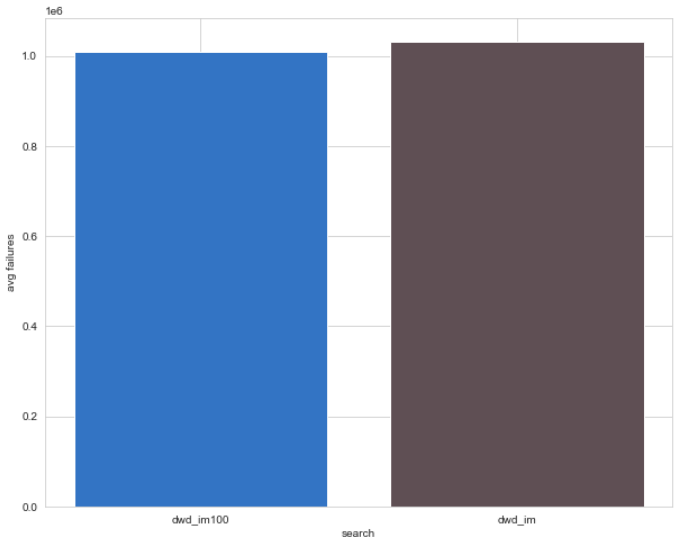


Figure 28: Average number of failures per search strategy.

Figure 29: Number of solutions found per search strategy.

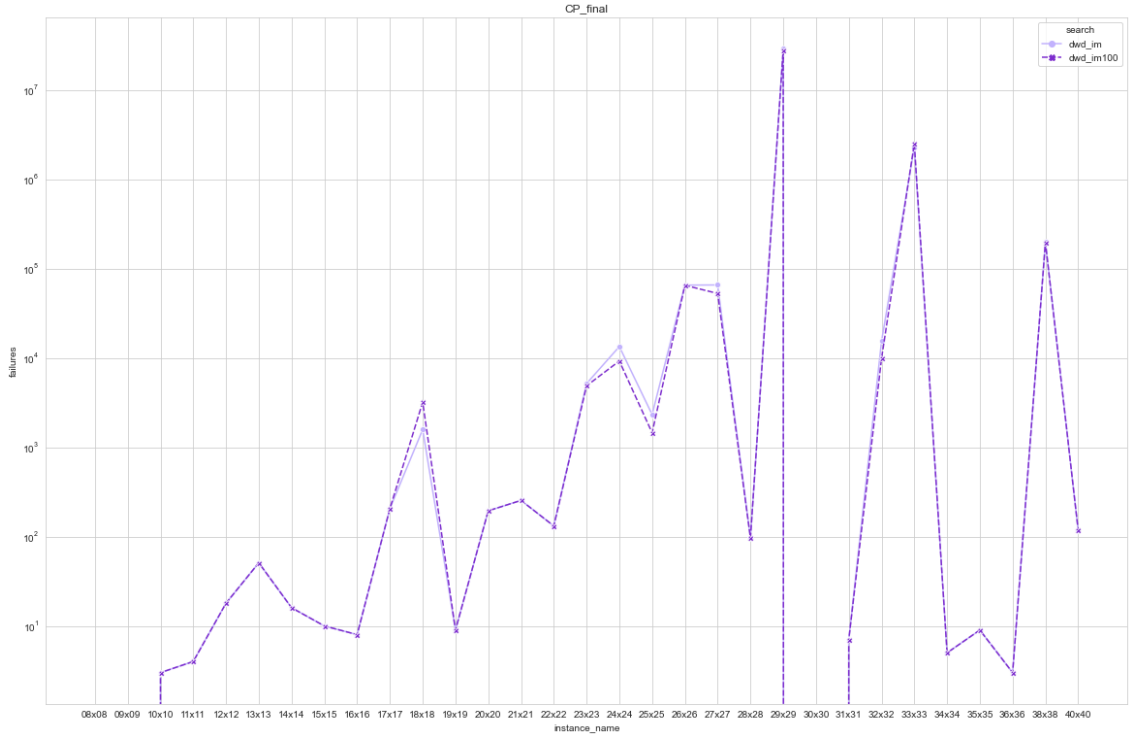


Figure 30: Failures per instance.

6. Conclusions and Remarks

In the end, this report described the way we solved the Present Wrapping Problem using the MiniZinc CP solver. We proposed different models facing varied subproblems, such as the possibility to rotate the pieces or to have pieces with the same dimensions. Coming up with all these models required us a lot of efforts as we did several experimentations. Some of them were happily successful, such as using some optimization constraints, like the symmetry breaking and the ordering ones. We also tested different search strategies and tried to define a criterion to choose the one that best suited the problem. It is worth pointing out that while in the case of the Base Model we tried a lot of combinations of variable selection strategies and value selection strategies, in all the other models we focused only on three of them: ff_im, dwd_im and dwd_im100. We found out that dwd_im100 obtained the highest score in the Symmetry and Final Models, while ff_im succeeded in the Base and Rotation Models and dwd_im in the Multiple Instances Model.

References

[1] https://www.minizinc.org/doc-2.5.5/en/index.html

[2] https://sofdem.github.io/gccat/gccat/Cdiffn.html

[3] https://sofdem.github.io/gccat/gccat/Ccumulative.html

[4] https://www.coursera.org/lecture/solving-algorithms-discrete-optimization/3-2-2-restart-and-advanced-search-LKXoR