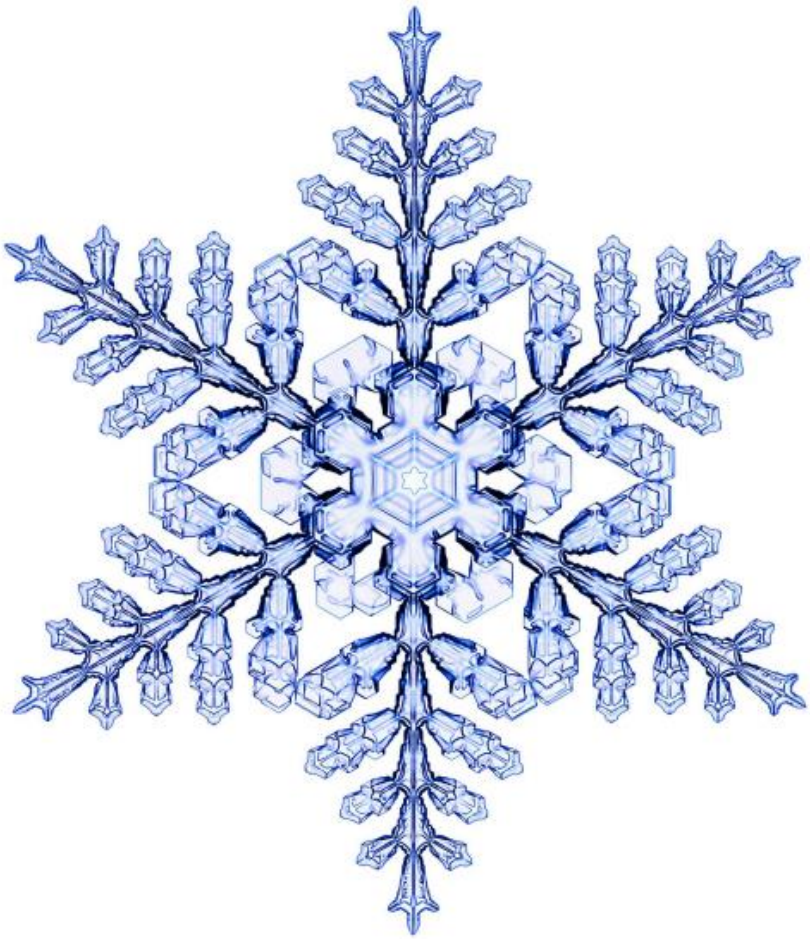
Combinatorial Decision Making and Optimization (A. Y. 2019-2020)

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.

1. Agenda

Introduction

The purpose of the following report is to describe how we solved the Present Wrapping Problem using the Z3Py, the API for python to use Z3 [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 *SMT.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.

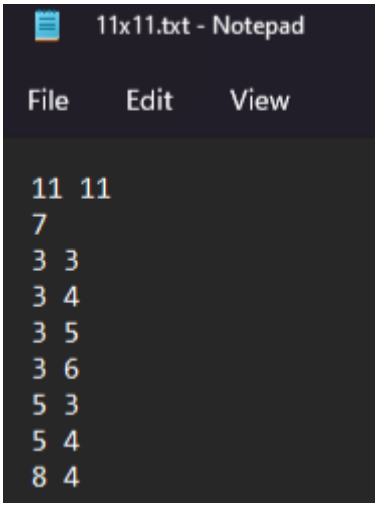


Figure 1: Example of .txt file.

The figure above shows an example of input files, which we directly used as input data for our CLI python program.

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*.

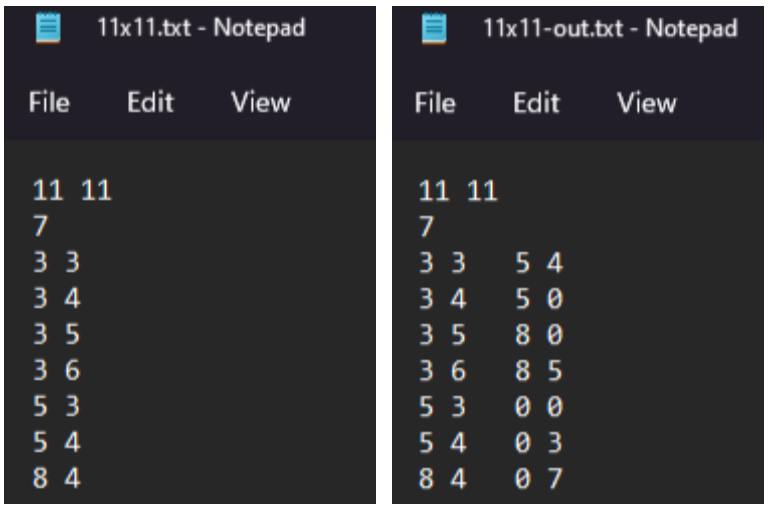


Figure 2: Example of input instance and its corresponding output solution.

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.

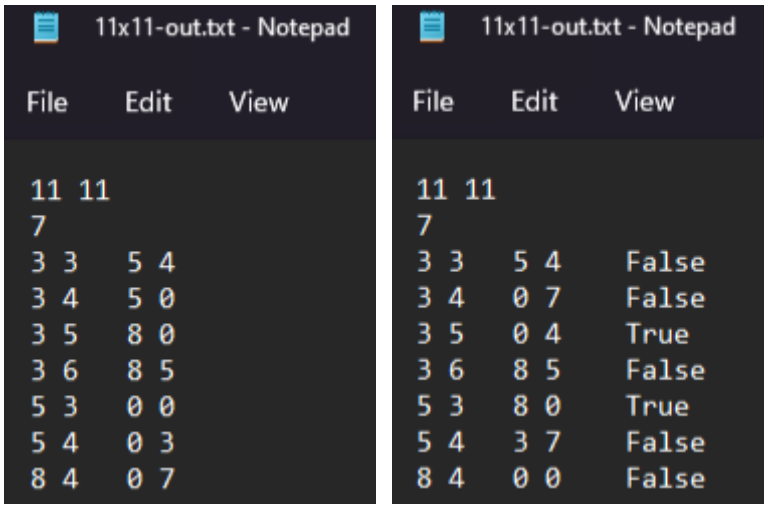


Figure 3: Example of input instance ad its corresponding output solution when pieces rotation is allowed.

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 time needed to solve an instance by the solving process in order to better understand the behaviour of the model. They 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 and to analyse the essential constraints for the model functioning.

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 0 to *n_presents -1*.
- 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 2-D 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 has two columns (namely, the number of possible 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 two columns related to the x_axis and y_axis. Hence each row *i* of the ***coord_presents*** matrix will contain the *(x,y)* bottom left coordinate of the *i*-th present.

```
coord_presents = [[Int("x_axis_%s" % i), Int("y_axis_%s" % i)] for i in 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. In addition, the bottom left coordinates should be strictly positive, as the presents should fall inside the paper roll dimensions.

By isolating the decision variable, the resultant constraint is:

```
domain_constraint = [
    And(And(coord_presents[i][x_axis] >= X_COORDS[0],
            coord_presents[i][x_axis] <= width_paper - dim_presents[i][x_axis]),
        And(coord_presents[i][y_axis] >= Y_COORDS[0],
            coord_presents[i][y_axis] <= height_paper - dim_presents[i][y_axis]))
        for i in PRESENTS]
```

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].

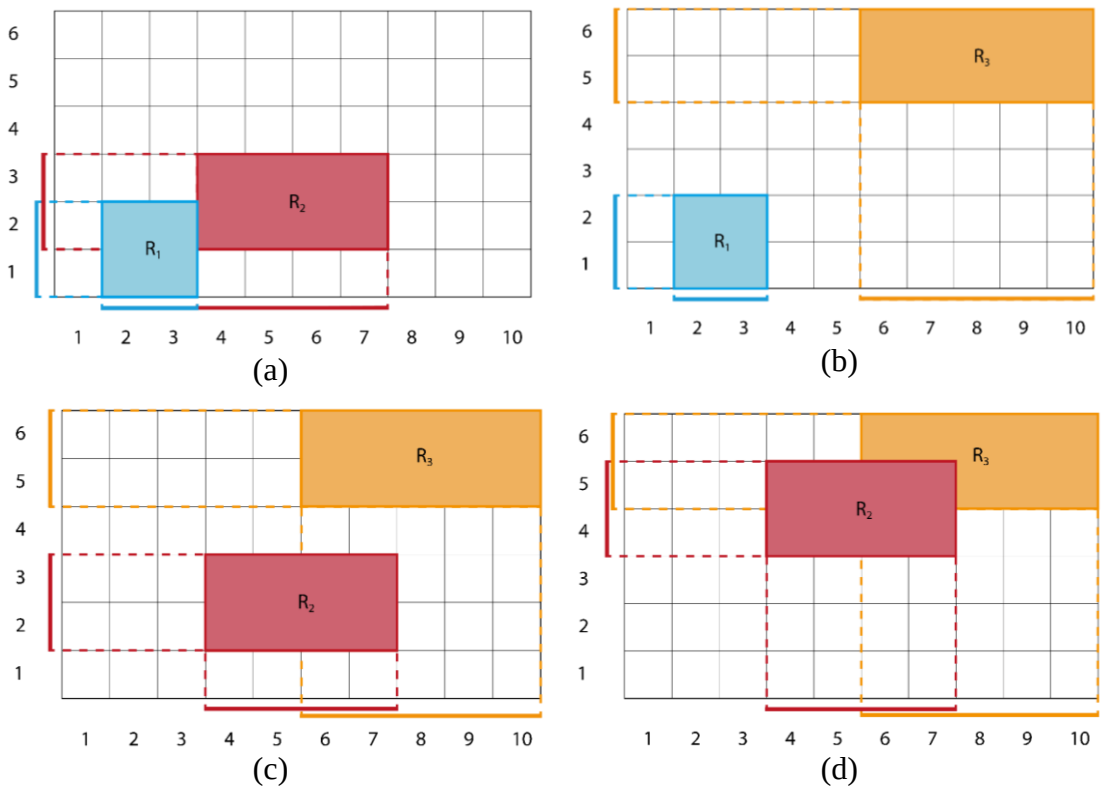


Figure 4: Illustration of why pairs of rectangles overlap or do not.

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

Inspiring from [3] and [4], we implemented the non-overlapping constraint as follows:

```
no_overlap_constraint =
    [Or(Or(coord_presents[i][x_axis] + dim_presents[i][x_axis] <= coord_presents[j][x_axis],
           coord_presents[i][x_axis] >= coord_presents[j][x_axis] + dim_presents[j][x_axis]),
        Or(coord_presents[i][y_axis] + dim_presents[i][y_axis] <= coord_presents[j][y_axis],
           coord_presents[i][y_axis] >= coord_presents[j][y_axis] + dim_presents[j][y_axis]))
     for i in PRESENTS for j in PRESENTS if i < j]
```

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).
The constraint applied to the x_axis is the following:

```
implied_constraint_x = [Sum(
        [If(And(y >= coord_presents[i][y_axis],
        y < coord_presents[i][y_axis] +dim_presents[i][y_axis]),
        dim_presents[i][x_axis], 0) for i in PRESENTS]) = = width_paper
        for y in Y_COORDS]
```

Constraint 1.3.1

The same constraint is applied to the other axis.

```
implied_constraint_y = [Sum(
        [If(And(x >= coord_presents[i][x_axis],
        x < coord_presents[i][x_axis] + dim_presents[i][x_axis]),
        dim_presents[i][y_axis], 0) for i in PRESENTS]) = = height_paper
       for x in X_COORDS]
```

Constraint 1.3.2

1.4 Results

The table below shows the results obtained with the Base Model, using the default resolution strategy for the Z3 Solver and a timeout of 5 minutes. The solve time shown in the table below is expressed as seconds.

instance_name	solveTime
8x8	0.010
9x9	0.011
10x10	0.013
11x11	0.016
12x12	0.022
13x13	0.040
14x14	0.044
15x15	0.064
16x16	0.060
17x17	0.116
18x18	0.195
19x19	0.271
20x20	0.176
21x21	0.249
22x22	0.202
23x23	0.832
24x24	0.373
25x25	0.251
26x26	3.151
27x27	0.998
28x28	3.732
29x29	2.768
30x30	1.141
31x31	1.476
32x32	52.265
33x33	18.403
34x34	3.700
35x35	3.092
36x36	4.632
37x37	62.103
38x38	1.318
39x39	86.798
40x40	2.354

Table 1: Solve time 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 should be preserved.

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

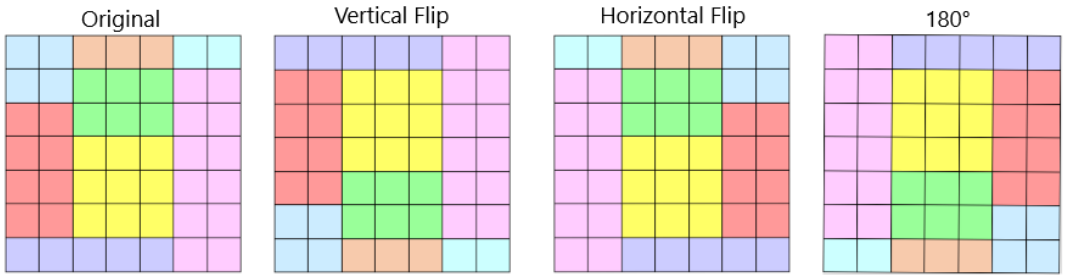


Figure 5: Symmetric variants of a solution (labeled “Original”).

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.

```
order_indexes = sorted(PRESENTS,
                       key= lambda x: dim_presents[x][x_axis] * dim_presents[x][y_axis],
                       reverse= True)
```

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

```
symmetry_breaking_constraint = [
        And((coord_presents[order_indexes[0]][x_axis] < (width_paper) / 2),
            (coord_presents[order_indexes[0]][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.

```
same_y_constraint =
  [coord_presents[order_indexes[i]][x_axis] < coord_presents[order_indexes[i+1]][x_axis]
  for i in range(n_presents-1)
      if coord_presents[order_indexes[i]][y_axis] == coord_presents[order_indexes[i+1]][y_axis]]
```

Constraint 2.2

2.1 Results

As in the section 1.2, here we report the results obtained in the Symmetry Model, imposing a timeout of 5 minutes. The solve time shown in the table below is expressed as seconds.

instance_name	solveTime
8x8	0.009
9x9	0.01
10x10	0.015
11x11	0.017
12x12	0.02
13x13	0.035
14x14	0.031
15x15	0.029
16x16	0.046
17x17	0.072
18x18	0.216
19x19	0.128
20x20	0.177
21x21	0.225
22x22	0.338
23x23	0.953
24x24	0.643
25x25	0.814
26x26	1.894
27x27	1.465
28x28	4.181
29x29	2.278
30x30	1.997
31x31	0.804
32x32	30.604
33x33	15.319
34x34	1.889
35x35	1.941
36x36	3.012
37x37	72.098
38x38	1.458
39x39	57.758
40x40	2.179

Table 2: Solve time 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 each 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.

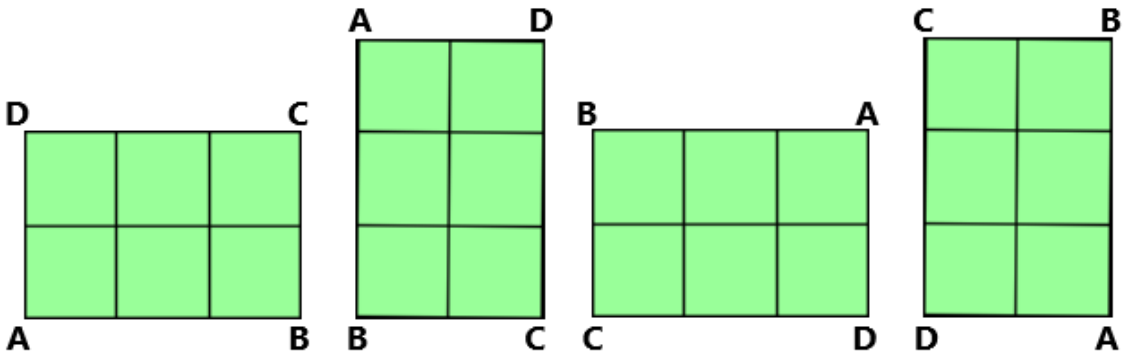


Figure 6: Possible rotation of a present.

To deal with it, we assign 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).

```
rot = [Bool("rot_%s" % i) for i in PRESENTS]
```

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 ***dim_rot*** that given the *i*-th presents and its axis as input it returns the original present dimension if it is not rotated or its swapped dimension if it is rotated.

```
def dim_rot(i, axis):
    if axis == x_axis:
        return If(rot[i],dim_presents[i][y_axis],dim_presents[i][x_axis])
    else:
        return If(rot[i],dim_presents[i][x_axis],dim_presents[i][y_axis])
```

The constraints 1.1, 1.2, 1.4.1 and 1.4.2 become:

```
domain_constraint = [
    And(And(coord_presents[i][x_axis] >= X_COORDS[0],
        coord_presents[i][x_axis] <= width_paper - dim_rot(i,x_axis),
    And(coord_presents[i][y_axis] >= Y_COORDS[0],
        coord_presents[i][y_axis] <= dim_rot(i,y_axis))
      for i in PRESENTS]
```

Constraint 3.1: Domain constraint.

```
no_overlap_constraint =
    [Or(Or(coord_presents[i][x_axis] + dim_rot(i,x_axis)<= coord_presents[j][x_axis],
            coord_presents[i][x_axis] >= coord_presents[j][x_axis] + dim_rot(j,y_axis)),
        Or(coord_presents[i][y_axis] + dim_rot(i,y_axis)<= coord_presents[j][y_axis],
            coord_presents[i][y_axis] >= coord_presents[j][y_axis] + dim_rot(j,y_axis)))
         for i in PRESENTS for j in PRESENTS if i < j]
```

Constraint 3.2: Non-overlapping constraint.

```
implied_constraint_x = [Sum(
        [If(And(y >= coord_presents[i][y_axis],
                y < coord_presents[i][y_axis] + dim_rot(i,y_axis)),
            dim_rot(i,x_axis ),0) for i in PRESENTS]) = = width_paper
         for y in Y_COORDS]
```

Constraint 3.3.1: Implied constraints.

```
implied_constraint_y = [Sum(
        [If(And(x >= coord_presents[i][x_axis],
                x < coord_presents[i][x_axis] + dim_rot(i,x_axis)),
            dim_rot(i,y_axis ),0) for i in PRESENTS]) = = height_paper
         for x in X_COORDS]
```

Constraint 3.3.2: Implied constraints.

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.

```
square_constraint = [Not(rot[i])
                     for i in PRESENTS
                         if dim_presents[i][x_axis] == dim_presents[i][y_axis]]
```

Constraint 3.4

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.

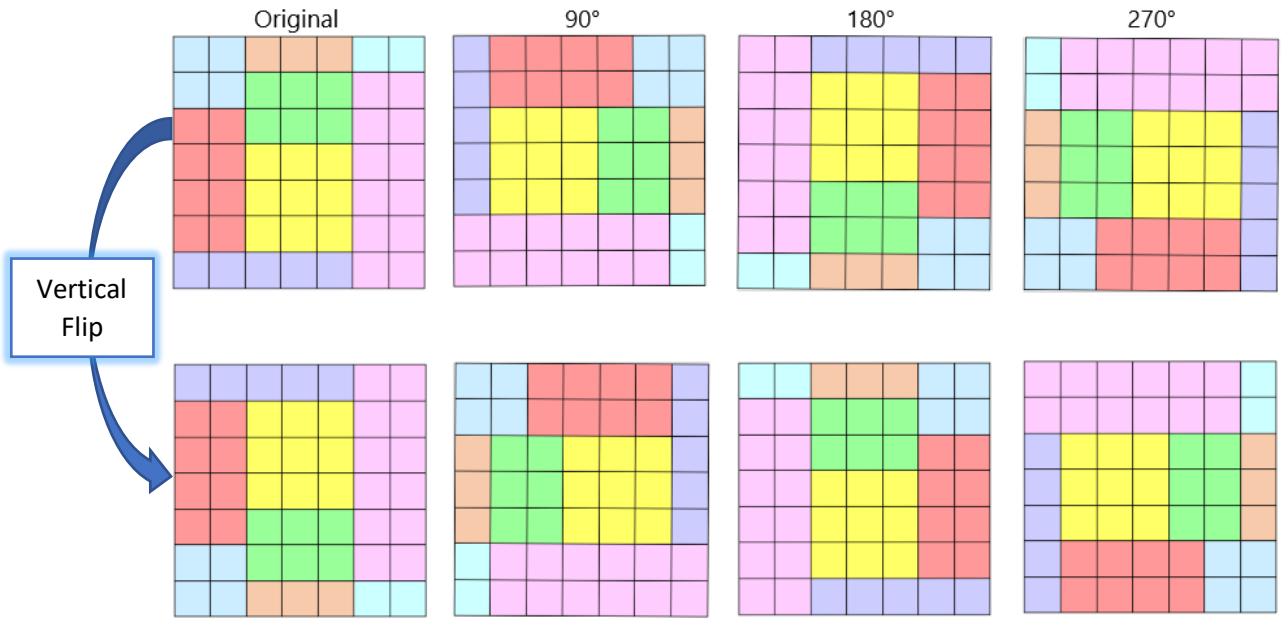


Figure 7: Symmetric variants of a solution (labeled “Original”)

3.3 Results

instance_name	solveTime
8x8	0.013
9x9	0.015
10x10	0.019
11x11	0.042
12x12	0.11
13x13	0.081
14x14	0.105
15x15	0.127
16x16	0.086
17x17	0.726
18x18	3.557
19x19	1.309
20x20	0.892
21x21	2.412
22x22	1.655
23x23	46.553
24x24	10.358
25x25	52.822
26x26	164.459
27x27	194.628
28x28	192.432
29x29	242.19
30x30	86.978
31x31	17.399
32x32	-
33x33	62.149
34x34	87.708
35x35	63.932
36x36	114.345
37x37	-
38x38	6.885
39x39	-
40x40	16.599

Table 3: Solve time per instance.

”–“ means that no solution has been found within the time limit of 5 minutes.

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.

```
multiple_instances_constraint =
        [lex_less(coord_presents[order_indexes[i]],coord_presents[order_indexes[j]])
            for i in PRESENTS for j in PRESENTS if j > i
            if dim_presents[order_indexes[i]][x_axis] ==dim_presents[order_indexes[j]][x_axis]
               and
           dim_presents[order_indexes[i]][y_axis] == dim_presents[order_indexes[j]][y_axis]]
```

Constraint 4.1

The lex_less function used in the constraint 4.1 is a custom function, defined as follows [5]:

```
def lex_less(a, b):
    if not a:
        return True
    if not b:
        return False
    return Or(a[0] < b[0], And(a[0] == b[0], lex_less(a[1:], b[1:])))
```

4.1 Results

instance_name	solveTime
8x8	0.008
9x9	0.01
10x10	0.012
11x11	0.017
12x12	0.022
13x13	0.034
14x14	0.032
15x15	0.03
16x16	0.046
17x17	0.072
18x18	0.233
19x19	0.13
20x20	0.183
21x21	0.228
22x22	0.363
23x23	0.986
24x24	0.705
25x25	0.797
26x26	2.056
27x27	1.56
28x28	4.442
29x29	2.386
30x30	2.02
31x31	0.839
32x32	30.856
33x33	15.522
34x34	2.076
35x35	2.221
36x36	3.139
37x37	71.445
38x38	1.45
39x39	59.27
40x40	2.286

Table 4: Solve time 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 exploit the helper function *dim_rot* and the constraint 4.1 becomes as follows:

```
multiple_instances_constraint =
        [lex_less(coord_presents[order_indexes[i]],coord_presents[order_indexes[j]])
                for i in PRESENTS for j in PRESENTS if j > i
                if dim_rot(i,x_axis) == dim_rot(j,x_axis) and
                dim_rot(i,y_axis) == dim_rot(j,y_axis)]
```

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
- 2.2
- 3.4
- 5.1

5.1 Results

instance_name	solveTime
8x8	0.011
9x9	0.019
10x10	0.022
11x11	0.028
12x12	0.052
13x13	0.095
14x14	0.12
15x15	0.085
16x16	0.209
17x17	0.28
18x18	1.275
19x19	2.524
20x20	1.537
21x21	0.499
22x22	4.515
23x23	14.419
24x24	6.255
25x25	44.832
26x26	237.395
27x27	241.603
28x28	-
29x29	-
30x30	48.682
31x31	0.877
32x32	-
33x33	39.139
34x34	152.758
35x35	93.554
36x36	93.167
37x37	-
38x38	38.976
39x39	-
40x40	13.377

Table 5: Solve time per instance.

”–“ means that no solution has been found within the time limit of 5 minutes.

6. Conclusions and Remarks

In the end, this report described the way we solved the Present Wrapping Problem using the Z3 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.

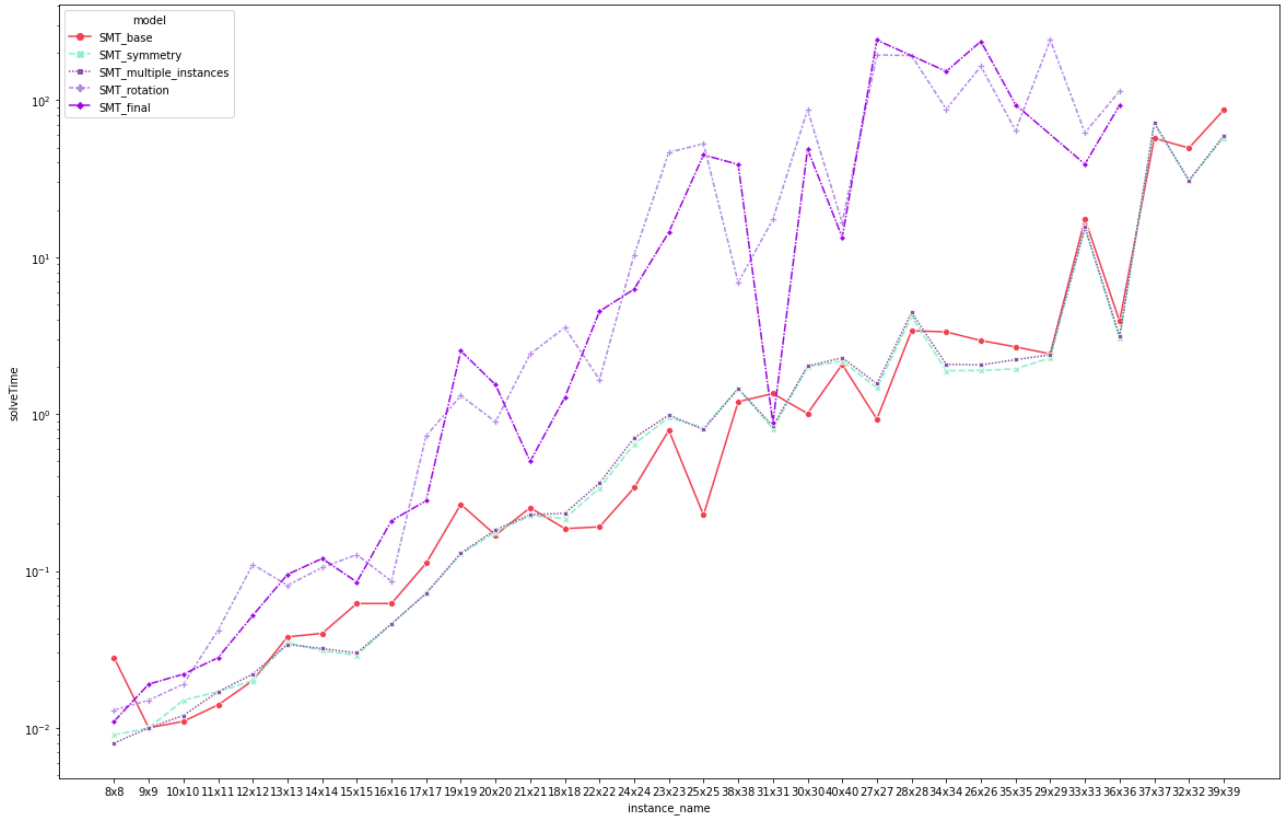


Figure 8: Solve time per instance.

We found out that the Base, Symmetric and Multiple Instances Models have about the same value of solving time per instance. On the other hand, in general the solving time increases a lot when considering the Rotation and Final Models, as shown in the figure above.

7. References

[1] https://github.com/Z3Prover/z3

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

[3] https://stackoverflow.com/questions/40795709/checking-whether-two-rectangles-overlap-in-python-using-two-bottom-left-corners

[4] https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169

[5] https://stackoverflow.com/questions/68557254/z3py-symmetry-breaking-constraint-by-lexicographic-order