Present Wrapping Problem: CP

Federico Battistella, Alessandro Pavesi

September 6, 2021

Contents

1 Introduction

The Present Wrapping Problem (PWP) is the problem where given a wrapping paper roll of a certain dimension and a list of presents, an agent needs to decide how to cut pieces of paper to wrap each present. The problem can be represented as a grid of the same dimension of the paper roll and each present as a rectangle or square of dimension equal to each present, we call this *instance*. The problem is represented by how we arrange the rectangle/square on the grid. A natural improvement for this problem is to optimize the cut of the paper roll to minimize the lost paper, given that the presents are not a perfect fit for them. We can ignore this latter case as, in our case, the presents fit perfectly on the paper roll.
At first, it is required to use the presents as they are, without the possibility to rotate the pieces. Later, in the last part of the project, we implemented a program that also uses rotation to find the solution. We approached the problem with the use of MiniZinc, a free, open-source constraint modeling language that provides methods to model constraint satisfaction and optimization problems in a high-level, solver-independent way.

1.1 How to use

We have created a user-friendly interface to be used by the Terminal/Command-Prompt. Once the user is in the main folder it can use *python* to launch the program.

```
python3 PresentWrapping.py
```

Once launched, the program will ask what kind of technique you want to use:

```
Welcome to the present wrapping problem! Make your choice:
1)Minizinc
2)Z3
```

Picked one of the choices the program will ask what instance do you want to use.

```
['08x08.txt', '09x09.txt', '10x10.txt', ...]

Choose an instance: [without the extension]
```

Lastly, one more question is asked about the use of rotation:

```
Do you want to use rotation: [Y/N]
```

In the end, the program will plot the solution if find it as well as some useful statistics. The use of this interface as a main is optional, it's possible to launch the file associated to the chosen Program and run it:

```
python3 CP/PWMinizinc.py
```

1.2 Initial setting

The project comes with some initial suggestions on how the program the work. The following points are the basic suggestion that we used to start:

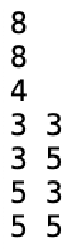
1. Start with the variables and the main problem constraints.
2. In any solution, if we draw a vertical line and sum the vertical sides of the traversed pieces, the sum can be at most l. A similar property holds if we draw a horizontal line. Use these implied constraints in both your CP model and SAT/SMT encoding.
3. Use global constraints to impose the main problem constraints and the implied constraints in your CP model.
4. Investigate the best way to search for solutions in CP.

5. the rotation is allowed: which of the CP model and the SAT/SMT encoding is easier to modify to take this into account? How would you modify that model/encoding?
6. there can be multiple pieces of the same dimension: how would you improve the CP model and the SAT/SMT encoding?

The first 4 points are the best way to start to write the code and see the result with some of the basic data that we have. Going forward the data that we need to use increases in difficulty and size so a better model is needed. Moreover, we are asked to implement the other two final points, that bring the complexity of the final problem to an upper level.

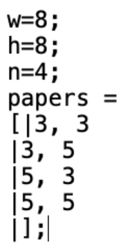
1.2.1 Input File

Input files are provided as a text file (.txt) and need to be modified in structure and extension to create the correct input file for the MiniZinc program, the extension of the file is .dzn. We decided arbitrarily the internal structure of the input files. There are 32 data files, each of them contains a different problem with increasing complexity, the first input is shown, as well as the formatted data used as input for the program.



```
8
8
4
3 3
3 5
5 3
5 5
```

Figure 1: .txt file



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

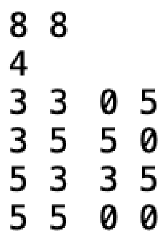
Figure 2: .dzn file

Each input file represents a dimension of the paper that needs to be cut. The structure of the file has a meaning:

- the first line is the width of the paper (**w**)
- the second line is the height of the paper (**h**)
- the third line is the number of pieces to be cut (**n**)
- lastly, there are **n** lines, each represents the width and height of one paper to wrap (**papers**)

1.2.2 Output File

The output file have a specific structure to follow, it needs to have the same format as the basic input file, but for each paper it is necessary to write, next to the dimensions, the coordinates of the bottom left corner point. An example is provided:



```
8 8
4
3 3  0 5
3 5  5 0
5 3  3 5
5 5  0 0
```

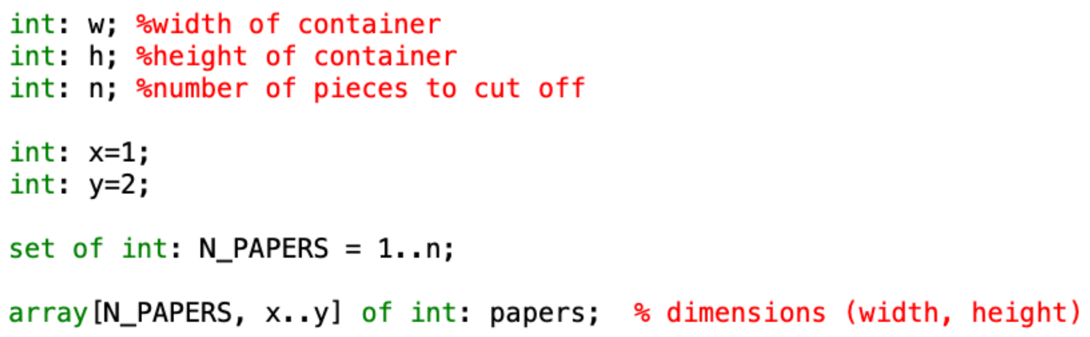
Figure 3: Output file

2 Solution of the problem

In this section we describe the solution adopted, focusing on each part of the MiniZinc solution:

2.1 Input data

We get in input few data: the container dimensions, the number of pieces of papers and the dimensions of each of them. We indicate with 1 the x axis of the coordinates, and with 2 the y axis.

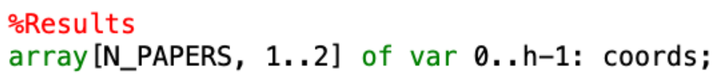


```
int: w; %width of container
int: h; %height of container
int: n; %number of pieces to cut off

int: x=1;
int: y=2;

set of int: N_PAPERS = 1..n;

array[N_PAPERS, x..y] of int: papers;  % dimensions (width, height)
```

Figure 4: Input variables

2.2 Variables

The variables we are interested in for our solution are the coordinates of the bottom left corner of each piece of paper, which belong to the domain [0, container width] for the x coordinates and [0, container height] for the y coordinates. We encoded the coordinates variables in a matrix called *coords*.



```
%Results
array[N_PAPERS, 1..2] of var 0..h-1: coords;
```

Figure 5: Result variable

Then we added some variables to help the solver and to get a more readable code:

1. *ordered_index_array* : a sorted index array from biggest to smallest piece of paper
2. *coords_domain_lower* : an array containing the lowest admissible value for x and y coordinates of paper pieces
3. *coords_domain_upper* : an array containing the highest admissible value for x and y coordinates of paper pieces
4. *index_of_largest_paper* : the index of the largest piece of paper used to apply some constraints
5. *X_COORDS : the set of all* the possible values that x coordinates can assume
6. *Y_COORDS : the set of all* the possible values that y coordinates can assume

```
array[N_PAPERS] of int: ordered_index_array = sort_by(N_PAPERS, [papers[i,x] + papers[i,y] | i in N_PAPERS]);

array[1..2] of int: coords_domain_lower = [0, 0];
array[1..2] of int: coords_domain_upper = [w, h];

int: index_of_largest_paper = ordered_index_array[n];
set of int: X_COORDS = coords_domain_lower[x]..coords_domain_upper[x]-1;
set of int: Y_COORDS = coords_domain_lower[y]..coords_domain_upper[y]-1;
```

Figure 6: Variables

2.3 Constraints

In the following sections, we describe the constraints used to reach the basic model that includes the first 4 points of the aforementioned to-do list. The use of these constraints brings us to solve the problem for every data input that is provided.

2.3.1 Cumulative

We have decided to use the *cumulative* constraint as the problem can be view as a task scheduling problem. Initially, we chose to use the cumulative function built-in in MiniZinc but, after many trials, we were forced to replace the function with an implied version of them. To build this version we imposed that the sum of each dimension for each piece, given the coordinates, is equal to the container dimension. This procedure needs to be done for both the height and the width as shown below.

$$\forall x\_coord \in X\_COORDS$$

$$\sum \forall i \in N\_PAPERS \begin{cases} papers(i,y) & x\_coord \geq coords(i,x) \wedge x\_coord < coords(i,x) + papers(i,x) \\ 0 & \text{otherwise} \end{cases} = w$$

$$\wedge$$

$$\forall y\_coord \in Y\_COORDS$$

$$\sum \forall i \in N\_PAPERS \begin{cases} papers(i,y) & y\_coord \geq coords(i,y) \wedge y\_coord < coords(i,y) + papers(i,y) \\ 0 & \text{otherwise} \end{cases} = h$$

The implementation in Minizinc is the following:

```
constraint forall (x_coord in X_COORDS) (
                sum(i in N_PAPERS)
                (if x_coord >= coords[i, x] /\ x_coord < coords[i, x] + papers[i, x]
                then papers[i, y]
                else 0
                endif) = h)
           /\
           forall (y_coord in Y_COORDS) (
                sum(i in N_PAPERS)
                (if y_coord >= coords[i, y] /\ y_coord < coords[i, y] + papers[i, y]
                then papers[i, x]
                else 0
                endif) = w);
```

Figure 7: Our implementation of *cumulative*

2.3.2 Stay In Limit

The second constraint that we apply is necessary to limit the coordinates of each piece to stay in the container dimensions, taking also into account the dimension of each piece.

$$\begin{aligned} &\forall i \in N\_PAPERS \\ &coords(i,x) \leq (w - papers(i,x)) \\ \wedge\ &coords(i,y) \leq (w - papers(i,y)) \end{aligned}$$

The implementation in Minizinc is the following:

```
constraint forall(i in N_PAPERS)
              (coords[i,x] <= (w - papers[i,x]));
constraint forall(i in N_PAPERS)
              (coords[i,y] <= (h - papers[i,y]));
```

Figure 8: Our implementation of the stay in limit constraint

2.3.3 Diffn

The problem needs that each piece does not overlap every other, to write this restriction we used the *diffn_k* constraint present inside MiniZinc api.

$$\begin{aligned} &\forall i,j \in N\_PAPERS,\ i<j, \\ &coords(i,x) + papers(i,x) \leq coords(j,x) \\ \wedge\ &coords(i,y) + papers(i,y) \leq coords(j,y) \\ \wedge\ &coords(j,x) + papers(j,x) \leq coords(i,x) \\ \wedge\ &coords(j,y) + papers(j,y) \leq coords(j,y) \end{aligned}$$

The implementation in Minizinc is the following:

```
constraint diffn_k(coords, papers);
```

Figure 9: Diffn_k

2.3.4 Lex Greatereq

This constraint is used mainly to improve the performances, forcing to have the biggest piece at the left bottom corner of the container. This constraint is based on the lex_greatereq constraint that requires that the first array is lexicographically greater than the second array.

$$\begin{aligned} &\forall i,j \in N\_PAPERS, i<j, \\ &coords(i,x) \leq coords(index\_of\_largest\_paper, x) \\ \wedge\ &coords(i,y) \leq coords(index\_of\_largest\_paper, y) \end{aligned}$$

The implementation in Minizinc is the following:

```
constraint forall(i in N_PAPERS) (
            lex_greatereq(coords[i, x..y], coords[index_of_largest_paper, x..y]));
```

Figure 10: Lex Greatereq

3 Solution of the problem with pieces rotation

We showed a solution to the traditional Present Wrapping problem using Cp in MiniZinc. Now we have to face the rotation enabled version of the above problem, which is harder, because the search space gets much larger than before and we have to consider all the combinations of the pieces in which they can be rotated by 90 degrees or not. We decided to adopt the same constraints used for the traditional problem, specifically:

1. *Cumulative*
2. *Stay in limits* : an array containing the lowest admissible value for x and y coordinates of paper pieces
3. *Diffn* : an array containing the highest admissible value for x and y coordinates of paper pieces
4. *Lex Greatereq* : the index of the largest piece of paper used to apply some constraints

We added only a constraint to set the rotation of all squared pieces to false, because they remain identical whatever rotation you apply, we can this way reduce the search space.

```
constraint forall(i in N_PAPERS where get_dimension(i,x) == get_dimension(i,y)) (rotations[i]=0);
```

Figure 11: Remove rotation from squared pieces constraint

In addition to the constraints above we introduced an array of the same length as the number of paper pieces to represent their rotations, each element of the array is a boolean that indicates true if the corresponding index piece is rotated or not.

```
% Define rotation property for the pieces of paper
array[N_PAPERS] of var bool: rotations;
```

Figure 12: Rotation array

Furthermore, we added a function to get the right dimension on the x or y axis of each piece considering their rotation:

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

Figure 13: Help function the get the right dimension on each of the axis of a paper piece

In the end, we decided to impose the rotation of at least one of the paper pieces, as we were not interested in the solution where none of them is rotated.

4 Search algorithm

In MiniZinc the method for searching the solution(s) can be modified to optimize the time spent. The default Search Algorithm implemented in MiniZinc is a depth-first traversal of the search tree. The use of this method involves that there is no optimization, for example before the solver check if a constraint is valid it needs that all the variables of that constraint are instantiated. This behaviour is not optimal in our case because of the complex and large solution space. To overcome the limitation of the default search strategy MiniZinc expose some search annotations. We used the *int_search* annotation that is specific for searching over integer values. Using an annotation, we can also specify how the variables are chosen, the *heuristic*, the options are *first-fail*, *dom_w_deg* and *input-order*. We chose the *dom_w_deg* order that is the best one based on solve time and number of propagation (More evaluation in Section 4.1). This technique chooses the next variable based on a middle ground between the variable with a smaller domain (dom) and the variable that is most constrained (deg). Furthermore, we adopted the function *restart_luby* that restart the search after a defined number of nodes of the search tree was visited; the number of nodes, using this function, is variable and is based on the Luby sequence. The number 2000 as paramter fot the restart brought us the best results. The dom_w_deg strategy works very well with the restart function because:

”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.”

Moreover, we use the *indomain_min* as the way to constrain a variable. Also, this choice was done based on execution time.

```
ann: search_ann = int_search([coords[i, x] | i in ordered_index_array], dom_w_deg, indomain_min, complete);
ann: restart_annotation = restart_luby(5000);
solve :: search_ann :: restart_annotation satisfy;
```

Figure 14: Search annotation

4.1 Annotations Comparison

Here we compare different annotation over some attributes of the search. The first thing to choose in the annotation is the heuristic, so holding the other parameters we analyse them. During the experiments we have seen that the harder instances could not be solved in a reasonable amount of time. We decided to focus on a set of instances that are solvable from, at least, the majority of the heuristics, and ended up with five instances: 15x15, 18x18, 24x24, 27x27, 40x40. Below the result divided by instances.

Heuristic	Instances	Solve Time	Propagation	Restart	Failures
dom_w_deg	15x15	0.003	8911	2	3
first_fail	15x15	0.007	6383	2	6
input_order	15x15	0.006	9142	2	13
smallest	15x15	0.007	8522	2	14
dom_w_deg	18x18	0.018	53489	2	38
first_fail	18x18	0.028	54817	2	34
input_order	18x18	0.643	2071578	2	3532
smallest	18x18	0.012	38169	2	27
dom_w_deg	24x24	0.114	379319	2	374
first_fail	24x24	0.065	243310	2	195
input_order	24x24	94.791	363601891	32	374051
smallest	24x24	0.06	62087	2	59
dom_w_deg	27x27	16.103	64120871	8	47582
first_fail	27x27	6.396	38630973	4	17736
input_order	27x27	—	—	—	—
smallest	27x27	1.382	3097294	2	4711
dom_w_deg	40x40	0.02	45585	2	4
first_fail	40x40	0.021	47487	4	8
input_order	40x40	—	—	—	—
smallest	40x40	0.432	527556	2	693

Table 1: Heuristic comparison (indomain_min, restart_luby(5000))

Given that the best heuristic is *dom_w_deg* the next parameter that we need to optimize is the one that defines how the values of the domain are chosen. This parameter is fundamental to improve the performance, as it's possible to see in the table below *indomain_min* and *indomain_split* are similar w.r.t. the parameter but the latter doesn't solve the 27x27 instance.That's why we choose the *indomain_min.*

Restart Strategy	Instances	Solve Time	Propagation	Restart	Failures
indomain_min	15x15	0.003	8911	2	493
indomain_split	15x15	0.003	8658	2	3
indomain_median	15x15	0.071	199523	2	493
indomain_random	15x15	0.101	210406	2	493
indomain_min	18x18	0.015	53488	2	38
indomain_split	18x18	0.026	51378	2	38
indomain_median	18x18	0.06	91639	2	146
indomain_random	18x18	75.732	224467987	32	320374
indomain_min	24x24	0.108	379319	2	374
indomain_split	24x24	0.108	232971	2	223
indomain_median	24x24	77.816	161608517	29	256208
indomain_random	24x24	11.058	33465199	8	41590
indomain_min	27x27	20.306	64120869	8	47582
indomain_split	27x27	—	—	—	—
indomain_median	27x27	—	—	—	—
indomain_random	27x27	—	—	—	—
indomain_min	40x40	0.022	45585	2	4
indomain_split	40x40	0.04	48731	2	6
indomain_median	40x40	0.263	190499	2	147
indomain_random	40x40	—	—	—	—

Table 2: Domain comparison (dom_w_deg, restart_luby(5000)

Then the last parameter to optimize is the number of resources used before the program restarts. This is a crucial parameter because if it's set to a low number maybe the solution isn't reached due to too limited resources, instead, a high number isn't useful because it can be larger than the search graph. It's visible the trend that with an easier instance the differences between the trials are minimal or nil. The most important trial with this parameter is with the instance 27x27 where the difference in solving time is visible.

Restart Strategy	Instances	Solve Time	Propagation	Restart	Failures
restart_luby(500)	15x15	0.002	8911	2	3
restart_luby(1000)	15x15	0.037	8911	2	3
restart_luby(2000)	15x15	0.003	8911	2	3
restart_luby(5000)	15x15	30.005	8911	2	3
restart_constant(5000)	15x15	0.003	8911	2	3
restart_geometric(10, 500)	15x15	0.003	8911	2	3
restart_linear(5000)	15x15	0.003	8911	2	3
restart_luby(500)	18x18	0.017	53489	2	38
restart_luby(1000)	18x18	0.017	53489	2	38
restart_luby(2000)	18x18	0.016	53489	2	38
restart_luby(5000)	18x18	0.016	53488	2	38
restart_constant(5000)	18x18	0.016	53489	2	38
restart_geometric(10, 500)	18x18	0.015	53489	2	38
restart_linear(5000)	18x18	0.017	53489	2	38
restart_luby(500)	24x24	0.123	379319	2	374
restart_luby(1000)	24x24	0.115	379319	2	374
restart_luby(2000)	24x24	0.111	379319	2	374
restart_luby(5000)	24x24	0.121	379319	2	374
restart_constant(5000)	24x24	0.114	379319	2	374
restart_geometric(10, 500)	24x24	0.0112	379319	2	374
restart_linear(5000)	24x24	0.11	379319	2	374
restart_luby(500)	27x27	28.91	122754312	64	80066
restart_luby(1000)	27x27	39.267	193636898	48	112133
restart_luby(2000)	27x27	104.241	553841966	62	287952
restart_luby(5000)	27x27	16.539	64120870	8	47582
restart_constant(5000)	27x27	10.643	34445946	8	30098
restart_geometric(10, 500)	27x27	30.34	107935513	4	86277
restart_linear(5000)	27x27	50.868	208035969	8	144173
restart_luby(500)	40x40	0.024	45585	2	4
restart_luby(1000)	40x40	0.02	45585	2	4
restart_luby(2000)	40x40	0.021	45585	2	4
restart_luby(5000)	40x40	0.02	45585	2	4
restart_constant(5000)	40x40	0.02	45585	2	4
restart_geometric(10, 500)	40x40	0.02	45585	2	4
restart_linear(5000)	27x27	0.019	45585	2	4

Table 3: Restarts comparison (dom_w_deg, indomain_min)

5 Results

5.1 Solution without rotation

We managed to obtain a result for each of the input files provided, without considering the rotation. As the dimension of the container and the number of pieces increases the problem becomes harder and harder and the search space inevitably expands, we successfully limited it by adding some constraints like the *lex_gretereq* that force the biggest piece to be in coordinate (0,0). The most difficult solutions to find were from the input files: 29x29, 37x37, 39x39; although they required a much larger execution time compared to the other input files, after few minutes of execution we finally got a solution. In the following picture we show the execution time necessary to find a solution for each of the instances of the problem:

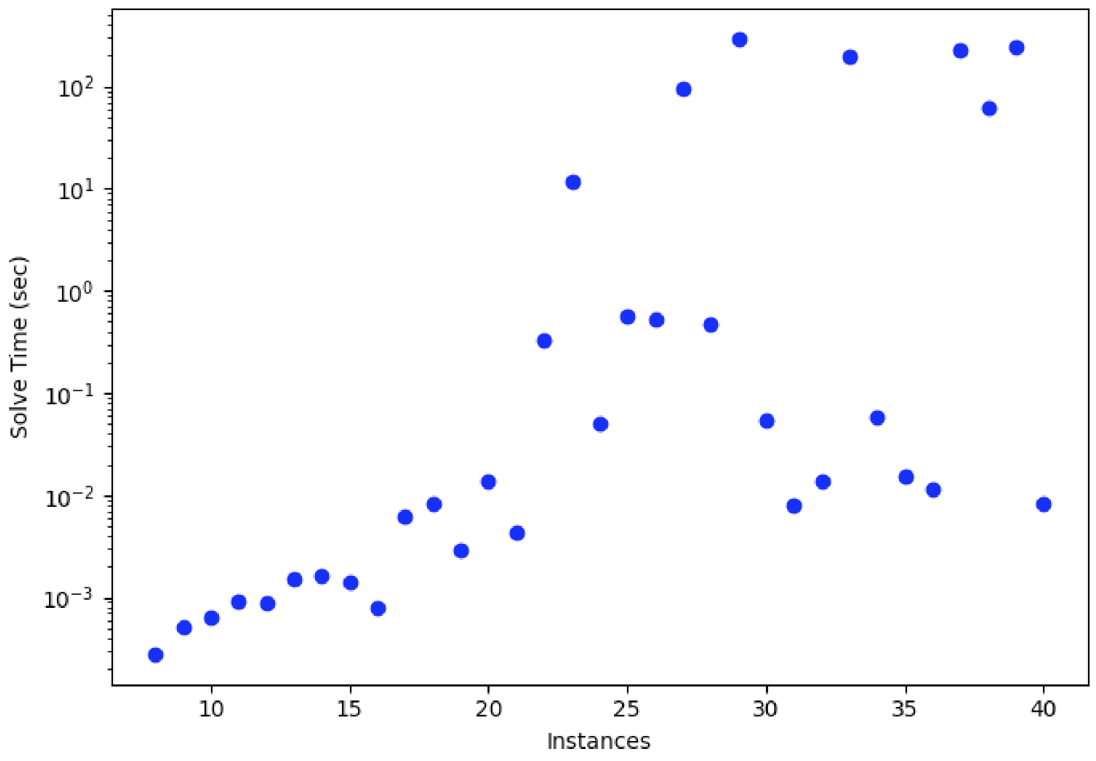


Figure 15: Solve times

We also implemented a python script to print a visual solution for each instance of the problem, some examples can be found in the following images.

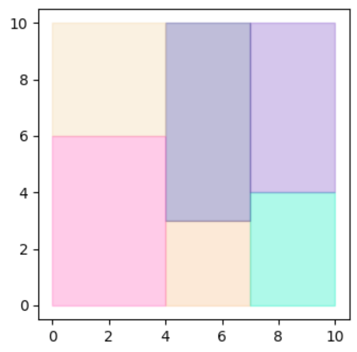


Figure 16: 10x10 instance visual solution

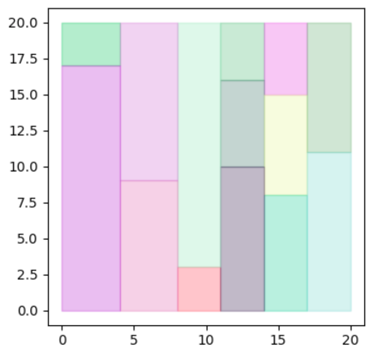


Figure 17: 20x20 instance visual solution

5.2 Solution with rotation

Considering the rotation the time to find a solution increases by far. With the rotation enabled we have a reduction on the number in solved instances and an increase in solve times.

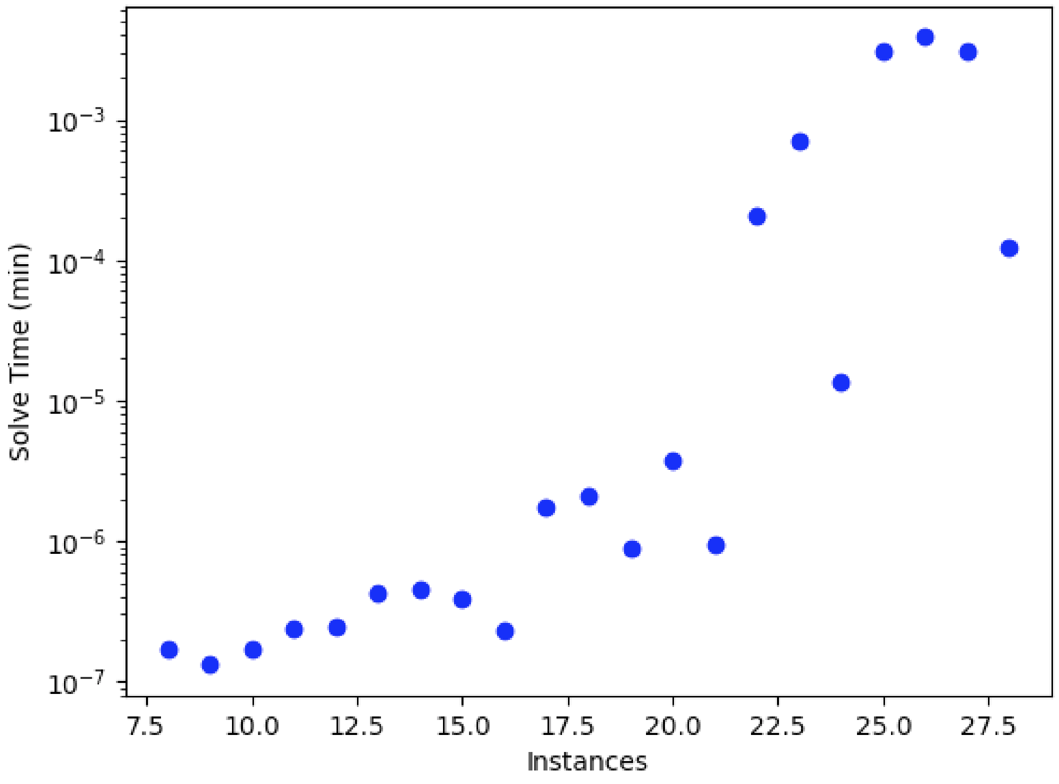


Figure 18: Solve times

We choose to impose that at least one piece is rotated, it can be viewed that in the 20x20 example below every piece is rotated but the one on the bottom-right corner is not.

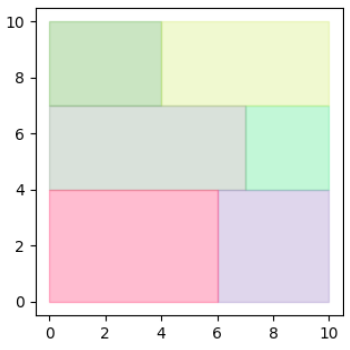


Figure 19: 10x10 instance with rotation

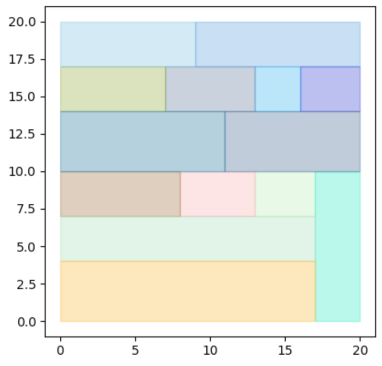


Figure 20: 20x20 instance with rotation

6 Conclusions

In this report, we describe our solution for the Present Wrapping Problem proposed. We use MiniZinc to program a constraint problem. The basic model can easily solve all instances in a fair amount of time. When rotation is added to the problem our model can solve most of the instances within the limit of time. The building of the constraint is the most important part of this project, we carefully craft each constraint to succeed in every instance we got. This means that we have a model which is working good in the general case but not in the optimal way w.r.t. each instance. We discovered that some instances could have a smaller solving time with constraints that differ from those described here. In the end we decided to adopt a general approach capable of solving every instance of the problem, we avoided the idea of having specific constraints for each of the proposed instances of the problem.