Present Wrapping Problem: SAT/SMT

Federico Battistella, Alessandro Pavesi

April 10, 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 approach the problem with the use of Z3Py, the API for Python to use Z3, a high-performance theorem prover developed at Microsoft Research.

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 it 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 with the chosen Program and run it:

```
python3 SMT/PWZ3.py
```

1.2 Initial setting

The project comes with some initial suggestions on how to 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), we use directly this file as input data for the python program. Below it's shown an example.

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

Figure 1: .txt 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 2: Output file

2 Solution of the problem

In this section we describe the solution adopted, focusing on each part of the Z3 solution. Z3 is a solver for logic formulas so every constraint needs to be designed to become a logical formula.

2.1 Input data

We get in input few data: the container dimensions, the number of pieces of papers and the dimension of each of them. We use a function that read the txt file and return directly the initial variables.

```
def read_txt(path):
    file = open(path,"r").readlines()
    w , h = tuple(map(int, file[0].rstrip("\n").split(" ")))
    n_papers = int(file[1].rstrip("\n"))
    papers = []
    for i in range(2, n_papers + 2):
        papers.append(list(map(int, file[i].rstrip("\n").split(" "))))
    return w, h, n_papers, papers
```

Figure 3: Result variable

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.

```
coords = [[Int("c_{}_{}".format(i, j)) for j in range(2)]
          for i in range(n_papers)]
```

Figure 4: Result variable

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 input data provided.

2.3.1 Cumulative

We have decided to use the *cumulative* constraint as the problem can be view as a task scheduling problem. In the case of Z3 the API isn't provided so we choose to implement the *cumulative* constraint based on the code we used on the MiniZinc program. Below the code:

```
def cumulative(solver, h, w, n_papers, coords, rotations, papers):
    for coord_y in range(h):
        listsum = []
        for i in range(n_papers):
            listsum.append(If(
                And(
                    coord_y >= coords[i][1],
                    coord_y < coords[i][1] + getdimension(i, 1, rotations, papers)
                ),
                getdimension(i, 0, rotations, papers),
                0
            ))
        solver.add(Sum(listsum) == w)

    for coord_x in range(w):
        listsum = []
        for i in range(n_papers):
            listsum.append(If(
                And(
                    coord_x >= coords[i][0],
                    coord_x < coords[i][0] + getdimension(i, 0, rotations, papers)
                ),
                getdimension(i, 1, rotations, papers),
                0
            ))
        solver.add(Sum(listsum) == h)
```

Figure 5: Our implementation of *cumulative* constraint

2.3.2 Stay In Limit

The second constraint that we apply is to limit the coordinates to stay in the container dimension, taking into account the dimension of each piece of paper. Moreover, we need to set the minimum of the coordinates to 0, while the maximum of the domain will be the dimension of the container minus the dimension of the piece of paper corresponding to that coordinates.

```
# Stay in limits constraints W and H
def stay_in_limits(solver, coords, n_papers, w, h, rotations, papers):
    for i in range(n_papers):
        solver.add(
            And(
                And(
                    (coords[i][0]+getdimension(i, 0, rotations, papers))<=w,
                    (coords[i][0]>=0)
                ),
                And(
                    (coords[i][1]+getdimension(i, 1, rotations, papers))<=h,
                    (coords[i][1]>=0)
                )
            )
            )
```

Figure 6: Our implementation of the *stay in limit* constraint

2.3.3 No Overlapping

The problem needs that each piece does not overlap with the others, we implement this restriction using a function called *no_overlapping* that constrain the sum between each coordinate and the corresponding dimension of each piece to be smaller than the coordinates of the other pieces.

```
# no overlapping bewtween different pieces
def no_overlapping(solver, coords, n_papers, rotations, papers):
    for i in range(n_papers):
        for j in range(i+1, n_papers):
            solver.add(
                        Or(
                            Or(
                                coords[i][0]+getdimension(i, 0, rotations, papers)<=coords[j][0],
                                coords[i][1]+getdimension(i, 1, rotations, papers)<=coords[j][1]
                                ),
                            Or(
                                coords[j][0]+getdimension(j, 0, rotations, papers)<=coords[i][0],
                                coords[j][1]+getdimension(j, 1, rotations, papers)<=coords[i][1]
                                )
                        )
                       )
```

Figure 7: Our implementation of the *no overlapping* constraint

2.3.4 Lex Lesseq

This constraint is used mainly to improve the performances, forcing to have the smaller pieces at the initial coordinates. To create this constraint we took the idea from MiniZinc, in that program we used to constrain the largest piece in the bottom left coordinate of the container, here we shot the idea and constrain the smallest one. The code is below:

```
def lex_lesseq(solver, list1, list2):
    temp = list1[0] <= list2[0]
    for i in range(1, len(list1)):
        temp = And(temp, append_or(i, list1, list2))

    solver.add(temp)
```

Figure 8: Our implementation of the *lex lesseq* constraint

This constraint is used also to remove some solution that are symmetric w.r.t. the axis. In this case the differences are the lists with which we call the function. The first line tries to remove the symmetry w.r.t. the main diagonal, instead the other two removes the symmetry w.r.t. the x and y axis.

```
lex_lesseq(solver,
           [coords[i][0]*100+coords[i][1] for i in range(n_papers)],
           [coords[i][1]*100+coords[i][0] for i in range(n_papers)])
```

Figure 9: Remove *main diagonal* symmetry

```
lex_lesseq(solver,
           [coords[i][0] for i in range(n_papers)],
           [w - getdimension(p, 0, rotations, papers) - coords[p][0] for p in range(n_papers)])
```

Figure 10: Remove x *axis* symmetry

```
lex_lesseq(solver,
           [coords[i][1] for i in range(n_papers)],
           [h - getdimension(p, 1, rotations, papers) - coords[p][1] for p in range(n_papers)])
```

Figure 11: Remove y *axis* symmetry

3 Solution of the problem with pieces rotation

We showed a solution to the traditional Present Wrapping problem using Z3 and in particular the Z3Py module. Now we have to face a 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. *No Overlapping* : an array containing the highest admissible value for x and y coordinates of paper pieces
4. *Lex Lesseq* : the index of the smallest piece of paper used to apply some constraints

3.1 All Different

We implemented a constraint that checks if two values are different, we used as a concept the all_different from MiniZinc. We use an equation

$$x * 100 + y$$

that correlate the x and y without change the information they bring. We use this equation given that the dimension of paper doesn't exceed the hundreds. This constraint is useful to improve the performance in the specific rotation-enabled problem.

```
# coordinates of each paper cut all differents
def alldifferent(solver, coords, n_papers):
    for i in range(n_papers):
        for j in range(i+1, n_papers):
            solver.add(
                Distinct(
                    coords[i][0]*100+coords[i][1],
                    coords[j][0]*100+coords[j][1]
                )
            )
```

Figure 12: Our implementation of the *all different* constraint

3.2 Rotation Variable

To manage the rotation for each present we introduced an array of the same length as the number of paper pieces, each element of the array is a boolean that indicates true if the corresponding index piece is rotated or not.

```
rotations = [Bool("r_{}".format(i)) for i in range(n_papers)]
```

Figure 13: Rotations variable

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

```
def getdimension(i, axis, rotations, papers):
    if axis == 0:
        return If(rotations[i], papers[i][1], papers[i][0])
    else:
        return If(rotations[i], papers[i][0], papers[i][1])
```

Figure 14: Help function the get the right dimension on each of the axis of a paper piece

3.3 Rotation on Squared Presents

We added 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.

```
# remove rotation from squared papers
def no_rotation_on_squared(solver, n_papers, rotations, papers):
    for i in range(n_papers):
        if getdimension(i, 0, rotations, papers)==getdimension(i, 1, rotations, papers):
            solver.add(Not(rotations[i]))
```

Figure 15: Remove rotation from squared pieces constraint

3.4 At Least One Rotation

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.

```
def at_least_one_rotation(solver, n_papers, rotations, papers):
    solver.add(Or([rotations[i] for i in range(n_papers)]))
```

Figure 16: Constraint the number of rotated pieces are greater or equal than one

3.5 No Rotation

Given that we develop a single program that can run the solution search with or without the rotation, we need to create a constraint that, in case the user chooses to do not to use the rotation, sets the corresponding Boolean to False.

```
def no_rotation(solver, n_papers, rotations, papers):
    for i in range(n_papers):
        solver.add(Not(rotations[i]))
```

Figure 17: Our implementation of the *No Rotation* constraint

4 Algorithm

The Z3Py APIs allow us to instantiate a Solver and add each constraint to it very easily. We built the constraint using the functions described above, each constraint adds some logical formulas, at the end all of these are combined to provide the solver the final formula to solve. We implemented both the rotation enabled and not enables solutions in the same file, providing the user the ability to choose whether the rotation should be enabled or not. The rotation enabled version of the problem uses the same constraints coded for the classical version, then it adds to them some rotation-specific functions.

```
solver = Solver()

# adding the constraints
cumulative(solver, h, w, n_papers, coords, rotations, papers)
stay_in_limits(solver, coords, n_papers, w, h, rotations, papers)
no_overlapping(solver, coords, n_papers, rotations, papers)
lex_lesseq(solver,
            [coords[i][0]*100+coords[i][1] for i in range(n_papers)],
            [coords[i][1]*100+coords[i][0] for i in range(n_papers)])


lex_lesseq(solver,
            [coords[i][0] for i in range(n_papers)],
            [w - getdimension(p, 0, rotations, papers) - coords[p][0] for p in range(n_papers)])


lex_lesseq(solver,
            [coords[i][1] for i in range(n_papers)],
            [h - getdimension(p, 1, rotations, papers) - coords[p][1] for p in range(n_papers)])


# check if Rotation is selected
if user_rotation:
    no_rotation_on_squared(solver, n_papers, rotations, papers)
    at_least_one_rotation(solver, n_papers, rotations, papers)
    alldifferent(solver, coords, n_papers)
else:
    no_rotation(solver, n_papers, rotations, papers)
```

Figure 18: The main program

5 Results

5.1 Solution without rotation

We managed to obtain a result for each of the input files provided, without considering the rotation version of the problem. 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 using constraints like the *lex_lesseq* that have various task to limit the search space. The most difficult solutions to find were from the input files: 32x32, 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:

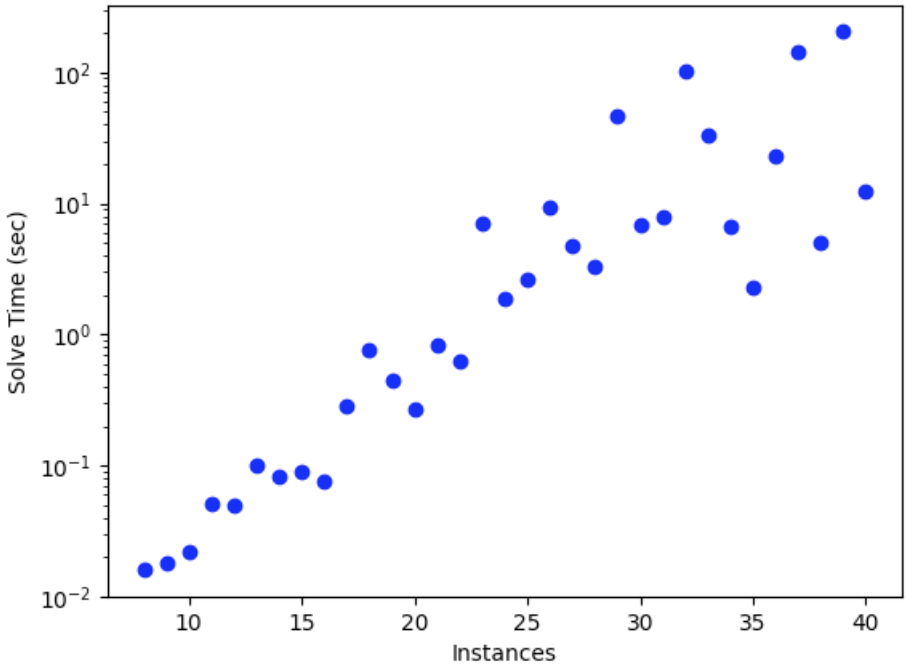

Figure 19: Solve times

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.

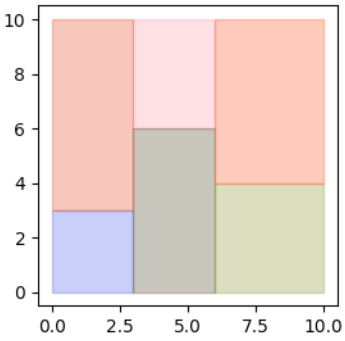

Figure 20: 10x10 instance visual solution

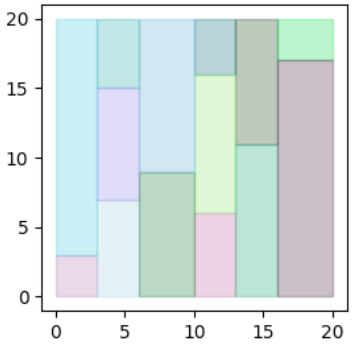

Figure 21: 20x20 instance visual solution

5.2 Performance

These graphs show how the complexity in terms of restart and propagation increases along with the instance dimension. We know that a large number of propagation can be optimal from the point of view of how the constraints work together but on the other side a large number of propagations is correlated to an increase of the solve time.

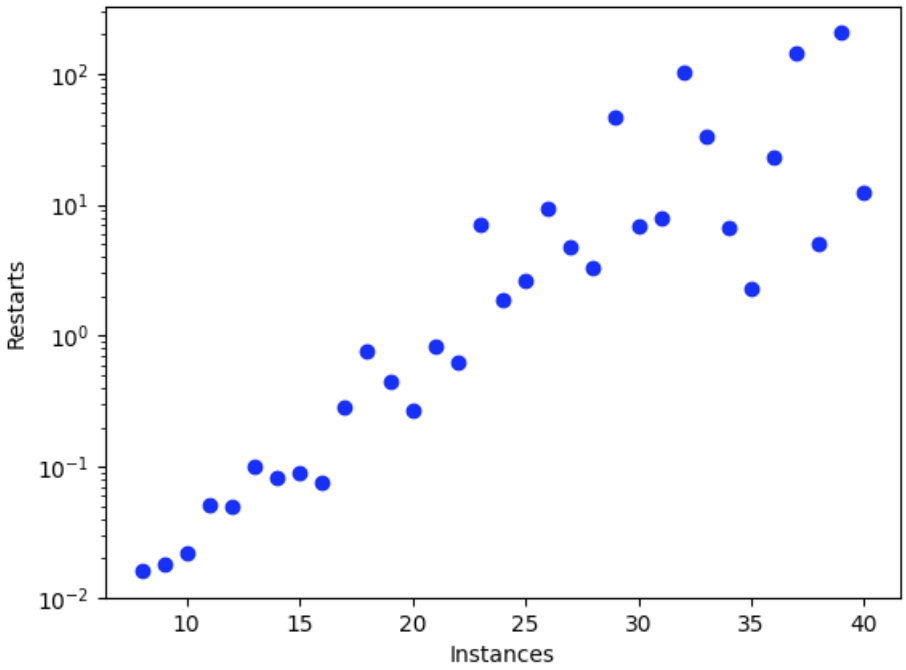

Figure 22: Restarts

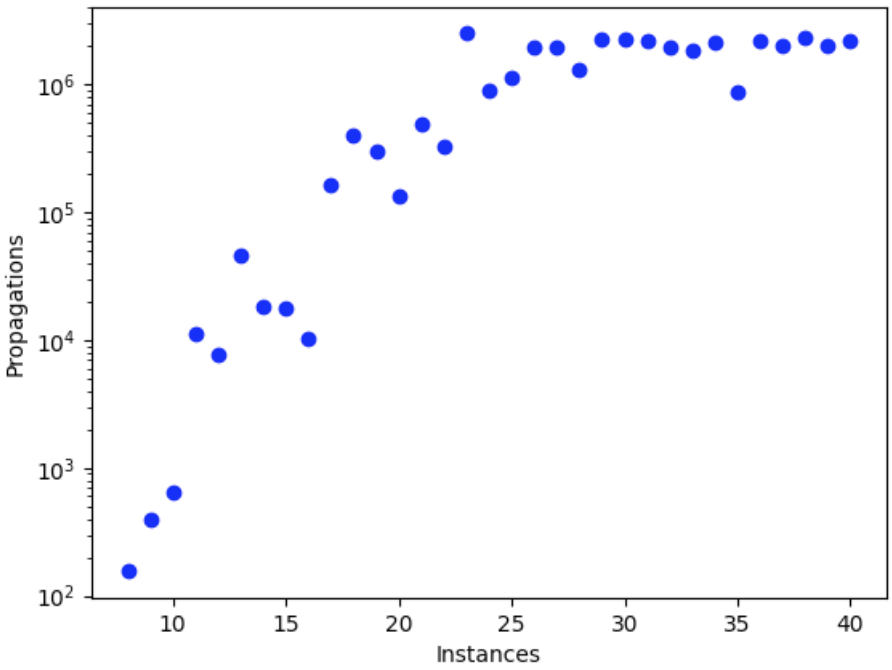

Figure 23: Propagations

5.3 Solution with rotation

Considering the rotation enabled for each of the pieces, the time to find a solution strongly increases, due to the larger dimension of the search space, which contains also the possible combinations of rotations for the entire set of pieces. We have an increase in solve time of about 2 order of magnitude w.r.t. the solution without the rotation.

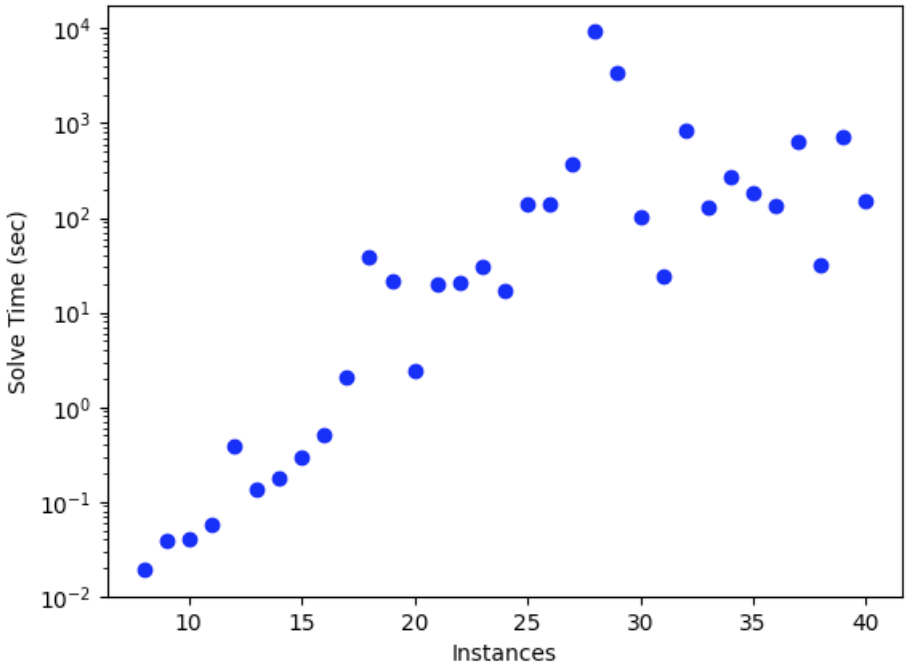

Figure 24: Solve times

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.

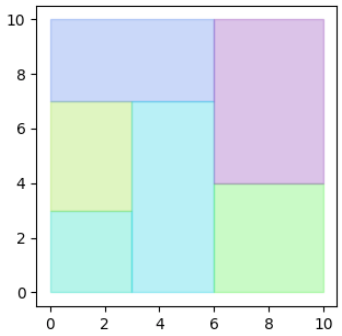

Figure 25: 10x10 instance with rotation

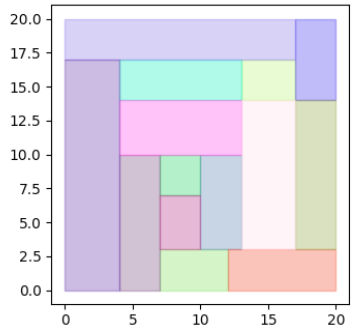

Figure 26: 20x20 instance with rotation

5.4 Performance

In case of the use of rotation, the number of restarts and propagation have an increase of about 2 order of magnitude, comparable to the increase in solve time.

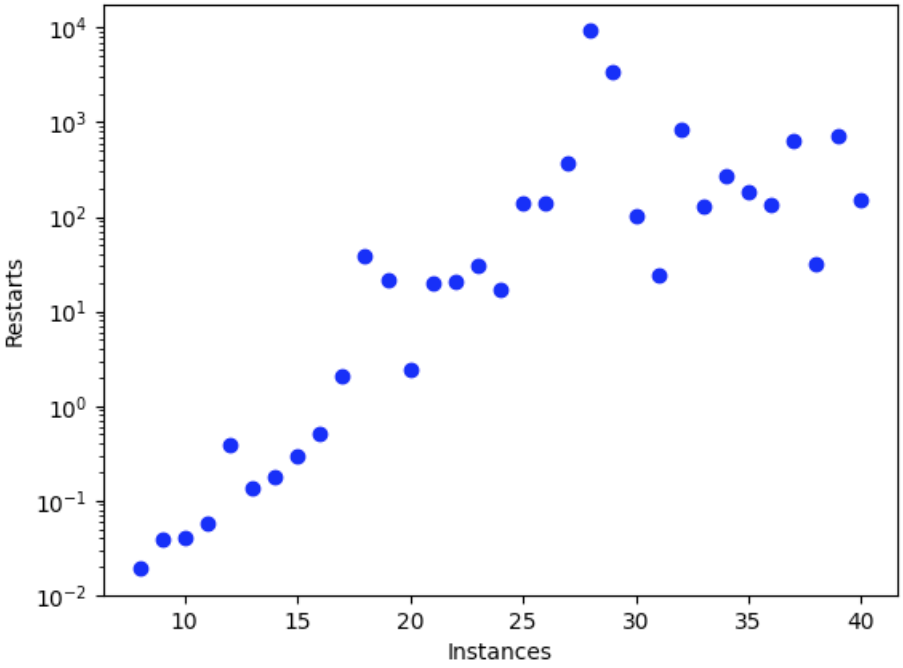

Figure 27: Restarts

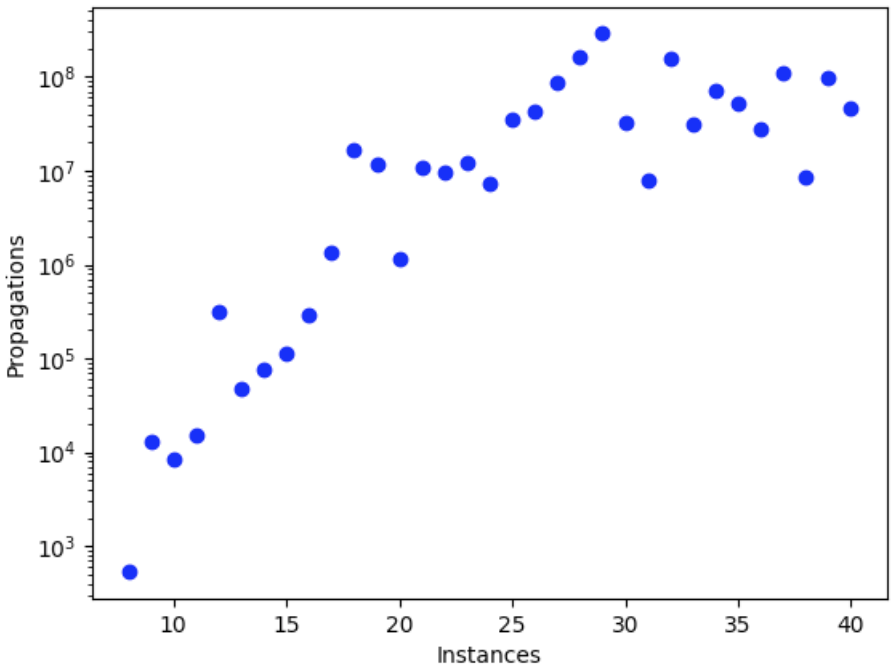

Figure 28: Propagation

6 Conclusions

In this report, we describe our way to solve the Present Wrapping Problem proposed. We used Z3 to code a logical solution that needs to build a SAT formula. 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 an acceptable time span.