*University of Bologna*
*Academic year: 2021/22*
*Department: Computer Science and Engineering*
*Master in: Artificial Intelligence*
*Course: Combinatorial Decision Making and Optimization*

# Project work
# VLSI Design

*Student: Dinno Koluh*
*e-mail: dinno.koluh@studio.unibo.it*
*ID number: 0001034376*

*August 2022*
*Bologna*

# 1 Introduction

In this paper we are going to explain how the problem of stacking rectangles in a container was tackled, the encountered obstacles, the experimental results and ideas used to solve the problem. The problem will be done in three ways, namely:

- Constraint Programming (CP)
- Satisfiability Modulo Theory (SMT)
- Mixed-Integer Linear Programming (MIP)

During the description of the model of the problem, depending on the approach we are explaining, the name of the respective approach will be outlined with its acronym (CP, SMT or MIP).

# 2 Defining the Variables

**CP - SMT - MIP**

Let us firstly introduce some notation for the problem parameters and variables. The width of the container is $w$, the height of the container is $l$ which is the variable we are trying to minimize.

The dimensions of the $i$-th rectangle are given as an ordered pair $(x_i, y_i)$ and for later convenience we are going to introduce the vectors $D_x$ and $D_y$ which hold the values of the dimensions of all rectangles respectively and to obtain the $i$-th $x$ dimension we can write $D_x(i)$.

The problem decision variables are the coordinates of the left-bottom corner of a rectangle and they are given as an ordered pair $(\hat{x}_i, \hat{y}_i)$ and for later convenience we are going to introduce the vectors $X$ and $Y$ which hold the values of the coordinates of all rectangles respectively and to obtain the $i$-th $\hat{x}$ coordinate we can write $X(i)$.

**MIP**

To encode a disjunction of literals (which we are going to encounter later) as a MIP set of inequalities we are going to need the binary variable $\delta$ as we are going to use the concept of the Big-M constraints. When we introduce these constraints then we will also explain $\delta$ in more detail.

# 3 Lower, Upper and Container Bounds

## 3.1 Container Constraints

**CP - SMT - MIP**

The reference point for the container where the rectangles are stacked will be the point $(0, 0)$. This implies that the lower bounds for the coordinates are given as:

$$\forall i \in [1, n] \quad X(i) \geq 0, \tag{3.1}$$

$$\forall i \in [1, n] \quad Y(i) \geq 0. \tag{3.2}$$

The $y$ coordinate of the upper left corner of a rectangle $i$ is given as $Y(i)+D_y(i)$ and the $x$ coordinate of the bottom-right corner of rectangle $i$ is given as $X(i)+D_x(i)$. As the container width is constrained by $w$ then the upper bounds for the coordinates are given as:

$$\forall i \in [1, n] \quad X(i) + D_x(i) \leq w, \tag{3.3}$$

$$\forall i \in [1, n] \quad Y(i) + D_y(i) \leq l. \tag{3.4}$$

As mentioned, $l$ is the variable we are trying to minimize so let us now define the bounds for the height $l$ of the container.

## 3.2 The Value of H and h

Let us call the upper bound for the height of the container as $H$. We will present different ways of computing the value of the upper bound that build on top of each other where the last way is some combination of all previously mentioned:

1. The most trivial way of computing the upper bound is to just stack all the rectangles on top of each other which gives:
$$H = \sum_i D_y(i)$$
This is also the most inefficient way of computing it.

2. The next way would be to firstly sort the dimensions of the rectangles in decreasing order by height. Then we put the first (the tallest as well) rectangle at position $(0, 0)$ and then stack others to the right of it (so the coordinates of the next rectangle will be $(D_x(1), 0)$) until we reach the $i$-th rectangle for which $X(i) + D_x(i) > w$. If that happens we stack it at position $(0, D_y(1))$ which is on top of the first rectangle. We do this until we run out of rectangles. The end value of $H$ will be just the sum of $D_y(i)$ of rectangles for which $D_x(i) = 0$:
$$H = \sum_{i \text{ for } D_x(i)=0} D_y(i).$$
This reduces substantially $H$ from the previous method but we can do better.

3. We again start by sorting and putting the tallest rectangle first at the position $(0, 0)$. Then we put the second highest rectangle next to the previous one at position $(D_x(1), 0)$. So, we have some empty space on top of the second rectangle and below the height of the first one. Now we search through the remaining rectangles and check if there is a rectangle which satisfies the condition $(D_y(i) \leq D_y(1) - D_y(2)) \wedge (D_x(i) \leq D_x(2))$ and stack it on top of the second rectangle, and we do this until we reach the height of the first one. And we move on like this for all rectangles until non are left. An example of such stacking is shown below:

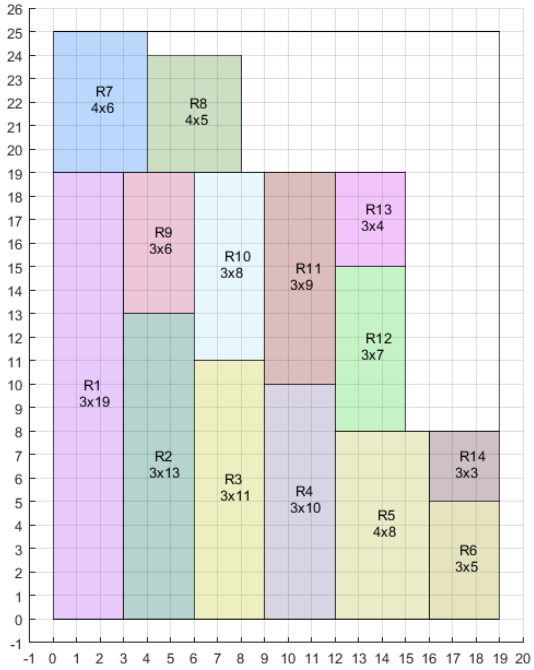



Figure 1: Stacking rectangles to get $H$ (`ins-12`)

This method is not perfect but it gives us a decently tight upper bound for $l$ for which we know that there is a solution and we are going to use this $H$ as the upper bound for the height of the container.

Let us call the lower bound for the height of the container $h$ and let us call the total area of all the rectangles $A = \sum_i D_x(i)D_y(i)$. Because the width of the container is fixed then the lowest possible height of the container if all the rectangles stack "nicely" must be:

$$h = \frac{A}{w}$$

So, in the end the constraint for the height $l$ of the container can be put like:

$$h \leq l \tag{3.5}$$

$$H \geq l \tag{3.6}$$

# 4 Constraints on Rectangle Non-overlapping

We introduced the constraints for the container bounds and the height of the container and they will be the same for all three approaches. In this section, however, we are introducing the next set of constraint between respective rectangles, namely the conditions that there are no overlaps between rectangles. Let us firstly define the main constraint for the problem, and then we are going to refine it for the purpose of the respective approach.

## 4.1 Main Constraint

The main constraints are the ones of non-overlapping rectangles. There are four arrangements for which any two rectangles $i$ and $j$ do not overlap:

1. Rectangle $i$ is under the rectangle $j$
2. Rectangle $j$ is under the rectangle $i$
3. Rectangle $i$ is to the left of rectangle $j$
4. Rectangle $j$ is to the left of rectangle $i$

At least one of these conditions must be satisfied so that two rectangles do not overlap (e.g. condition one is satisfied, then condition two cannot be satisfied, and if the rectangles have the same $x$ coordinate then conditions three and four are not satisfied).

And at most two of these conditions can be satisfied (e.g. we fix rectangle $i$ in place and rectangle $j$ can be above or under it and left or right to it).

We must ensure that $i \neq j$ for obvious reasons. A major symmetry breaking is to look at non-overlapping only for $i < j$. The reason is if, for example, rectangle $i$ is under rectangle $j$ then $j$ cannot be under rectangle $i$ and this holds for all of them. So instead of looking at all $i$ and $j$ (except for $i = j$) for are only required to look at all $i$ and $j$ s.t. $i < j$. This can written in mathematical terms as:

$$\begin{aligned} \forall i, j \text{ s.t. } i < j : \quad & (Y(i) + D_y(i) \leq Y(j)) \\ & \vee (Y(j) + D_y(j) \leq Y(i)) \\ & \vee (X(i) + D_x(i) \leq X(j)) \\ & \vee (X(j) + D_x(j) \leq X(i)) \end{aligned} \tag{4.1}$$

With the previous three constraints and the the one above the problem is fully defined and constrained so let us see now how are they implemented for different approaches.

## 4.2 Additional Constraints for Different Approaches

**CP**

We firstly have the constraints 3.1, 3.2, 3.3, 3.4, 3.5 and 3.6 for the container bounds and the bounds for the coordinates of the rectangles.

It would be enough to write the non-overlapping constraints as they are given in equation 4.1 but in the MiniZinc documentation there is a prewritten non-overlapping rectangle global constraint `diffn` [1] which is implemented on the basis of 4.1. This global constraint takes parameters in the form:

$$\texttt{diffn}(X, Y, D_x, D_y) \tag{4.2}$$

so we are going to replace constraint 4.1 with this single global constraint and the problem if fully modeled for CP.

**SMT**

As for the previous approach, also here the constraints 3.1, 3.2, 3.3, 3.4, 3.5 and 3.6 hold and are implemented. For the SMT approach the main non-overlapping constraint 4.1 will be implemented and extended with these additional constraints:

- For only one pair of rectangles $i$ and $j$ we can restrict their positional relation (shown in the figure 2 [2]. For example, we can restrict that $i$ has to be under or left to $j$. If we sort $D_x$ and $D_y$ based on the heights of $D_y$ then we can restrict that the two tallest rectangles are fixed with respect to each other. Written mathematically:

$$\texttt{if } (i = 1 \wedge j = 2): \quad (Y(i) + D_y(i) \leq Y(j)) \vee (X(i) + D_x(i) \leq X(j)) \tag{4.3}$$

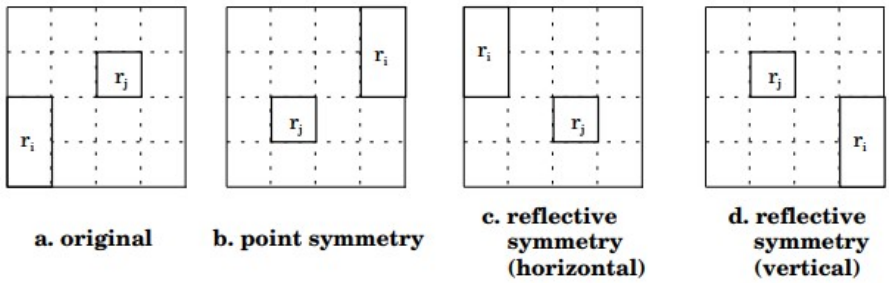



Figure 2: Positional relation between rectangles

- The next additional constraint is for rectangles which are tall or wide. Namely, if for any two rectangles $i$ and $j$, $D_y(i) + D_y(j) > l$ or $D_x(i) + D_x(j) > w$ then we do not need to look at

their respective overlapping but just define the proposition as false as they are out of bounds of the container. The condition can be written mathematically as:

$$\begin{aligned}&\texttt{if } (D_y(i) + D_y(j) > H)\\&\quad \texttt{false}\\&\texttt{else}\\&\quad (Y(i) + D_y(i) \leq Y(j)) \vee (Y(j) + D_y(j) \leq Y(i))\end{aligned} \tag{4.4}$$

$$\begin{aligned}&\texttt{if } (D_x(i) + D_x(j) > w)\\&\quad \texttt{false}\\&\texttt{else}\\&\quad (X(i) + D_x(i) \leq X(j)) \vee (X(j) + D_x(j) \leq X(i))\end{aligned} \tag{4.5}$$

To sum up, the non-overlapping constraints for the SMT approach is the combination of 4.1, 4.3, 4.4 and 4.5

**MIP**

As mentioned, when defining the binary variable $\delta$ the MIP approach will require some additional work to encode the non-overlapping constraint as a set of inequalities. Firstly, as the constraints for container bounds and the bounds of the coordinates of rectangles (3.1, 3.2, 3.3, 3.4, 3.5 and 3.6) are in the form of inequalities they are a part of the MIP model.

We will need to encode the disjunction of literals of the non-overlapping constraints from 4.1 as a set of inequalities and in order to do that we are going to use the Big-M constraints [3] with the help of the binary variable $\delta$. The dimensions of $\delta$ are $n \times n \times 4$. Let us introduce the constraint and then explain it in detail:

$$\begin{aligned}\forall i, j \text{ s.t. } i < j: \quad X(i) + D_x(i) &\leq X(j) + M_1\delta_{i,j,1}\\ X(j) + D_x(j) &\leq X(i) + M_2\delta_{i,j,2}\\ Y(i) + D_y(i) &\leq Y(j) + M_3\delta_{i,j,3}\\ Y(j) + D_y(j) &\leq Y(i) + M_4\delta_{i,j,4}\\ \sum_k \delta_{i,j,k} &\leq 3\end{aligned}$$

We have 4 binary variables for each ordered pair $(i, j)$ s.t. $i < j$. They are multiplied with constants $M_k$ and with the constraint $\sum_k \delta_{i,j,k} \leq 3$ (there must be a $k$ for which $\delta_{i,j,k} = 0$ for each pair of $i$ and $j$) we are making sure that at least one of the four constraints is active in its original form from 4.1.

If $\delta_{i,j,k} = 1$ then we must find an appropriate $M_k$ such that the constraint is kept active. For $x$ coordinate constraints a sufficiently large constant is the width of the container $w$ (as $w > D_x(i)\ \forall i$), so $M_1 = w$ and $M_2 = w$. The same logic holds for the $y$ coordinate, only the height of the container $l$ will be the constant (as $l > D_y(i)\ \forall i$), so $M_3 = l$ and $M_4 = l$.

After initializing the $M_k$ constants we have the final form of the constraints as:

$$\begin{aligned} \forall i,j \text{ s.t. } i<j: \quad X(i)+D_x(i) &\le X(j)+w\delta_{i,j,1} \\ X(j)+D_x(j) &\le X(i)+w\delta_{i,j,2} \\ Y(i)+D_y(i) &\le Y(j)+l\delta_{i,j,3} \\ Y(j)+D_y(j) &\le Y(i)+l\delta_{i,j,4} \\ \sum_k \delta_{i,j,k} &\le 3 \end{aligned} \tag{4.6}$$

And with this non-overlapping constraint and the constraint for the bounds of the container and the coordinates of the rectangles the MIP model is fully defined.

# 5 Rotations

**CP - SMT**

Let us introduce a new decision variable which is a boolean vector $R$ where the $i$-th element of this vector stores a boolean value where a logical true means that the $i$-th rectangle is rotated and a logical false means it is not. A way to introduce it to the basic model is such that wherever we have a constraint which depends on vectors $D_x$ and $D_y$ which hold the dimensions of the rectangles we need to branch it out to switch the dimensions. A general rule with a constraint dependent on $D_x(i)$ and $D_y(i)$ would be:

$$\begin{aligned} &\texttt{if } R(i) \\ &\quad \text{constraint}(D_y(i), D_x(i)) \\ &\texttt{else} \\ &\quad \text{constraint}(D_x(i), D_y(i)) \end{aligned} \tag{5.1}$$

The other way would to use a disjunction of conjuctions like this:

$$\begin{aligned} &\texttt{Or}(\texttt{And}(R(i), \text{constraint}(D_y(i), D_x(i)), \\ &\quad \texttt{And}(\texttt{Not}(R(i)), \text{constraint}(D_x(i), D_y(i))))) \end{aligned} \tag{5.2}$$

In the end the values of $R$ will tell us if the rectangle is rotated or not. Then in the output it is enough to just rotate the $i$-th dimension depending on $R(i)$ and to write it to the output file.

To sum up, to enable the rotations of rectangles it is enough to introduce the decision variable $R$ and to modify the already mentioned model constraints as presented above.

**CP - SMT - MIP**

As already mentioned, for MIP we cannot have disjunctions but a set of inequalities so the previous two ways of dealing with rotations will not work. But there is an idea which would work for all approaches and it is quite easy to implement.

The idea is that the value of $R(i)$ can be used in a similar manner like the values of the $\delta$ binary variable. If, for example, we want that $D_x(i)$ is not rotated then we can multiply it with $(1 - R(i))$ (as the value of $R(i) = 0$) and in the other case with $R(i)$ (as the value $R(i) = 1$). Mathematically we can introduce a transformation of non-rotational constraints as:

$$\begin{aligned} &\text{constraint}(D_x(i), D_y(i)) \rightarrow \\ &\text{constraint}(D_x(i)(1 - R(i)) + D_y(i)R(i), D_y(i)(1 - R(i)) + D_x(i)R(i)). \end{aligned} \tag{5.3}$$

We can check that the transformation boils down to the non-rotational constraints if we input $R(i)$ being 0 $\forall i$ and to the rotational constraints if we put $R(i)$ being 1 $\forall i$.

This is the model (5.3) that we are going to use for SMT. For CP we are going to use this model as well after experimentally testing that it produces better results than the models 5.1 and 5.2. For SMT we are going to use model 5.2 as it gave the best experimental results.

# 6 The Objective

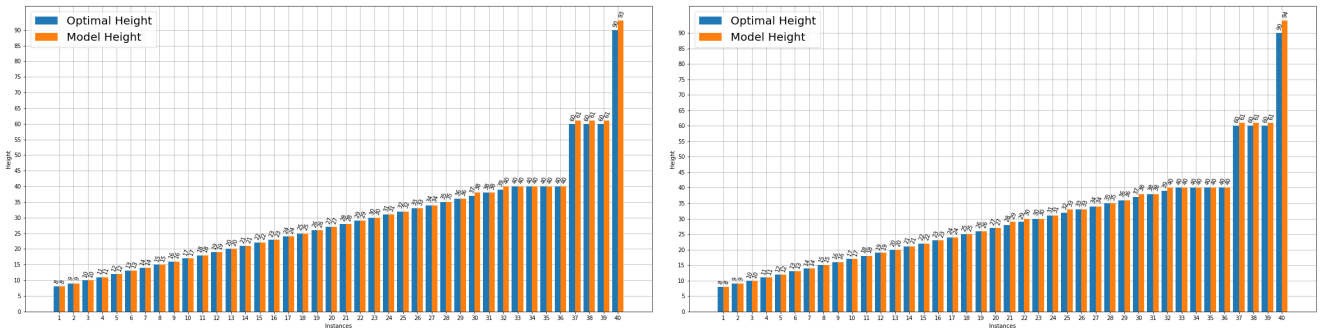
The objective to be minimized is the container height $l$ bounded by equations 3.5 and 3.6. For all approaches the objective to be minimized is just $l$. A remark to make is that after experimentally testing the SMT approach, it was concluded that instead of directly trying to minimize the objective, it is better to just try to find a solution of the problem with the constraints as they are. Then if there exists a solution we can lower the upper bound $H$ of $l$ and restart the solver. This method uses also the notion of restarting. And we are going to do this until we reach the condition that $H = h$ which ensures that we reached the minimum. In the next section we are going to present in more detail the used solvers and techniques for speeding up the process.

# 7 Results and Discussion

As far as the results are concerned a time limit of 300 [s] is put on the solver execution. To analyze the results we are going to look at the number of solved instances, the solution found and the time it took to find it. Also for each approach the solver used and the solver-related techniques for speeding up the process will be discussed. We will look at the base model and the one with rotations for each approach.

## CP

Two different solvers were used to solve the CP approach, *gecode* and *chuffed* inside the MiniZinc environment for Python. Chuffed has proven to be much faster than gecode so it has been chosen as the solver for CP. We also tried to include different solver restarting and for some instances it was a crucial step as the default solver could not solve them. Concretely, the instances $22, 34, 36$ were solved using linear restarting while the instances 25 and 27 were solved using Luby restarting as they could not be solved by the default options. For the rotational model restarting was even more crucial as after the 11-th instance it was almost impossible to solve any instance by the default settings.
For the base model, out of the 40 instances the used model was able to find 34 optimal solutions while 6 were suboptimal. For the model with rotations, out of the 40 instances the used model was able to find 31 optimal solutions while 9 were suboptimal. In figures 3a and 3b we can see the model vs. optimal heights barplot.

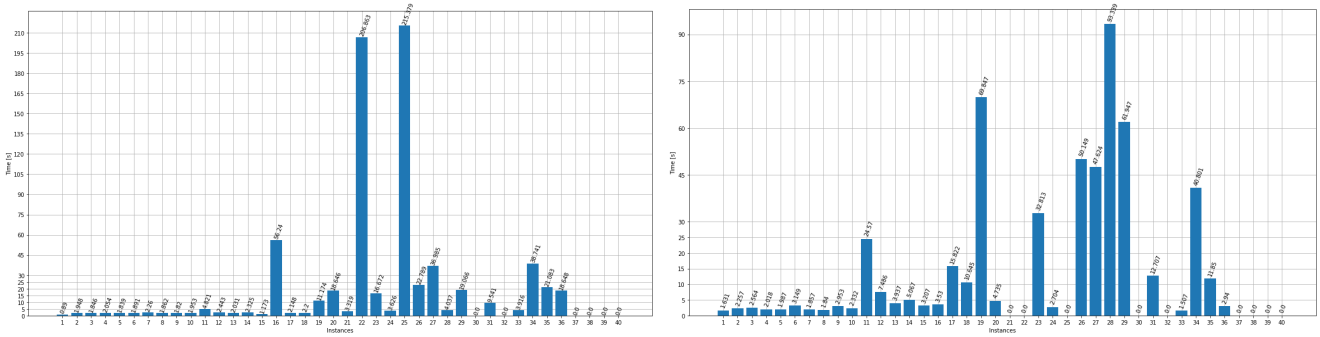



(a) Base model.

(b) Rotational model.

Figure 3: Barplot of model vs. optimal heights for CP.

In figures 4a and 4b we can see the time needed to solve each instance for the base and rotational model. The suboptimal solutions were terminated after a timeout (indicated as lasting 0 [s]).

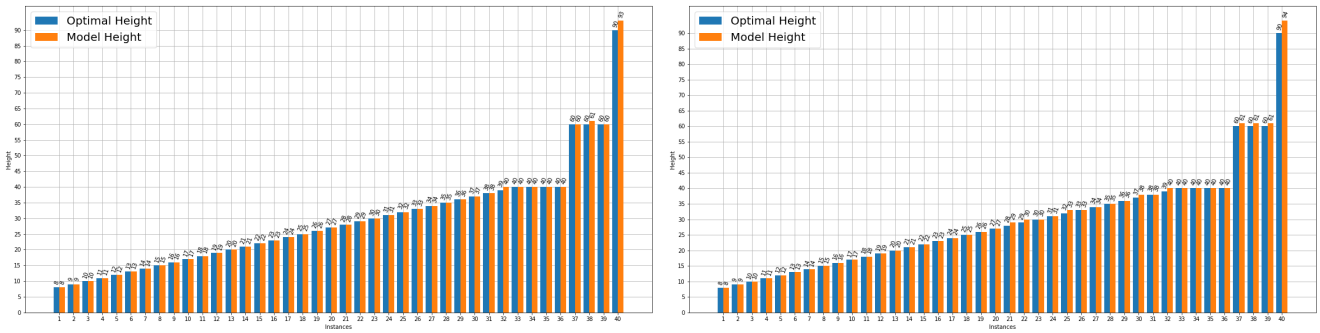


(a) Base model. (b) Rotational model.

Figure 4: Barplot of time needed to solve instances for CP.

## SMT

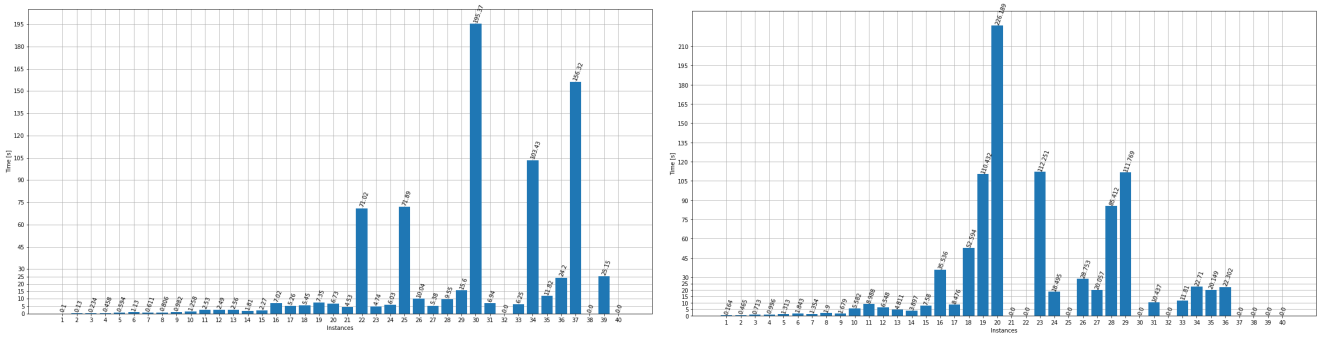
The solver used to tackle this approach was the default solver from z3. As already mentioned, after experimentally testing it, it was concluded that running the solver inside a loop, and decreasing $H$ after it finds a solution was faster than to directly minimize it.
For the base model, out of the 40 instances the used model was able to find 37 optimal solutions while 3 were suboptimal. For the model with rotations, out of the 40 instances the used model was able to find 31 optimal solutions while 9 were suboptimal. In figures 5a and 5b we can see the model vs. optimal heights barplot.




(a) Base model. (b) Rotational model.

Figure 5: Barplot of model vs. optimal heights for SMT.

In figures 6a and 6b we can see the time needed to solve each instance for the base and rotational model. The suboptimal solutions were terminated after a timeout (indicated as lasting 0 [s]).



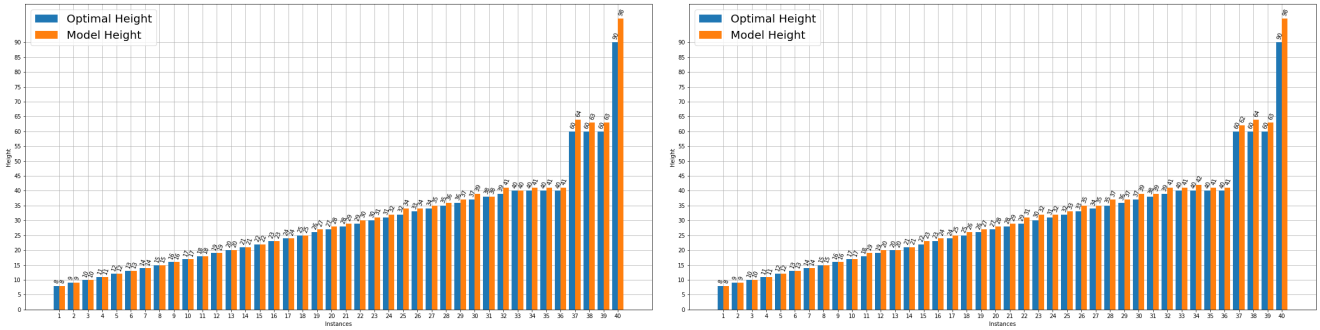
(a) Base model. (b) Rotational model.

Figure 6: Barplot of time needed to solve instances for SMT.

## MIP

Three different solvers were used to tackle the MIP approach, namely the *cplex* and *coin-bc* solver inside the MiniZinc for Python environment, and the *gurobi* solver inside the gurobipy Python environment. Of these three solvers gurobi was the fastest one but cplex was used in the end due to technical problems with the gurobi solver license (unfortunately I had no access to the University WiFi to activate the license).
For the base model, out of the 40 instances the used model was able to find 20 optimal solutions while 20 were suboptimal. For the model with rotations, out of the 40 instances the used model was able to find 12 optimal solutions while 28 were suboptimal. In figures 7a and 7b we can see the model vs. optimal heights barplot.

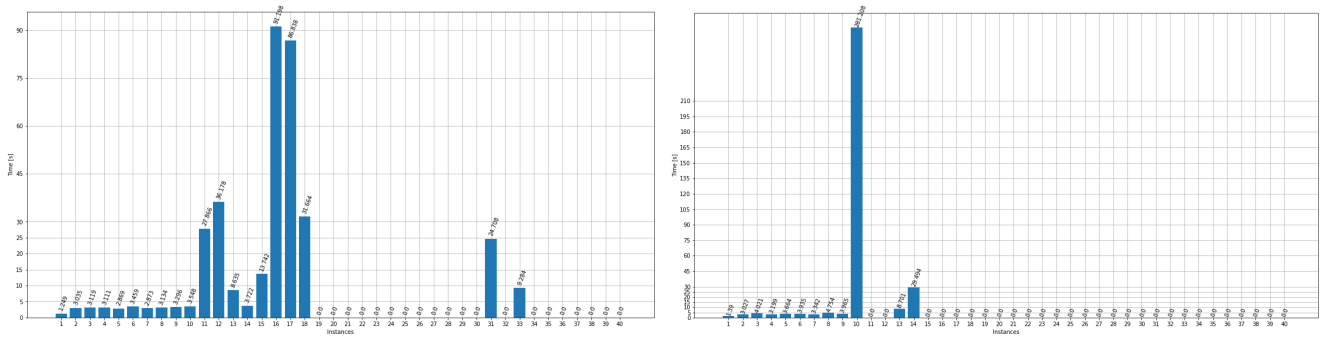
The low number of solved instances is again due to the usage of the cplex solver, whereas using gurobi more instances can be solved (there is a python script written for gurobi so you can test it for yourself).




(a) Base model. (b) Rotational model.

Figure 7: Barplot of model vs. optimal heights for MIP.

In figures 8a and 8b we can see the time needed to solve each instance for the base and rotational model. The suboptimal solutions were terminated after a timeout (indicated as lasting 0 [s]).



(a) Base model.

(b) Rotational model.

Figure 8: Barplot of time needed to solve instances for MIP.

# 8 Conclusion and Further Ideas

Analyzing the experimental results we can conclude that the model gave satisfactory results even though there are most certainly other constraints which could be added to further improve the results. But still, even though with the presented model an optimal solution for some instances was not found, a rather good suboptimal solution is also acceptable in real life problems and here we have quite good ones. As far as further ideas, there are two ideas that we will mention.

The first idea would be data preprocessing in the sense as to decrease the number of rectangles. This could be done by "gluing" rectangles of same height or same width into bigger rectangles where the other dimension is conserved.

The other idea would be of stacking smaller parts of the container. For instance, we could insert the tallest rectangle at the origin. Then the empty space to the right of that rectangle would have dimensions $(w - D_x(1), D_y(1))$. Then filling up this new, smaller, container could be a subtask which would be much easier than filling up the whole container. After we find the appropriate fitting, we can repeat the process for a new container on top of the already inserted rectangles. There are instances for which this might not work, but it is a good idea to build upon and combining it with the first one.

And of course as already mentioned, using gurobi for the MIP part would certainly improve the results.

# 9 Literature

[1] MiniZinc Packing constraints.
URL: https://www.minizinc.org/doc-2.6.2/en/lib-globals-packing.html

[2] Soh Takehide, Inoue Katsumi, Tamura Naoyuki, Banbara Mutsunori and Nabeshima Hidetomo. (2010). *"A SAT-based Method for Solving the Two-dimensional Strip Packing Problem"*. Fundam. Inform.. 102. 467-487. 10.3233/FI-2010-314.
URL: https://www.researchgate.net/publication/220445013_A_SAT-based_Method_for_Solving_the_Two-dimensional_Strip_Packing_Problem

[3] Blog on No-overlap constraints.
URL: http://yetanothermathprogrammingconsultant.blogspot.com/2017/07/rectangles-no-overlap-constraints.html