Combinatorial Decision Making
and
Optimization

Project Report:

VLSI Design

Gianluca Di Tuccio - gianluca.dituccio@studio.unibo.it,
Lorenzo Orsini - lorenzo.orsini4@studio.unibo.it

Index

Abstract

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. A typical example is the smartphone. The modern trend of shrinking transistor sizes, allowing engineers to fit more and more transistors into the same area of silicon, has pushed the integration of more and more functions of cellphone circuitry into a single silicon die (i.e. plate). This enabled the modern cellphone to mature into a powerful tool that shrank from the size of a large brick-sized unit to a device small enough to comfortably carry in a pocket or purse, with a video camera, touchscreen, and other advanced features.

In this project, we developed four different approaches for the VLSI problem: in particular, we adopted Constraint Programing (CP), Propositional Satisfiability (SAT), Satisfiability Modulo Theories (SMT) and Mixed Integer Linear Programming (MIP).

All the codes and notebooks can be available also on https://github.com/DitucSpa/VLSI.

1. Introduction

For this project, we designed the VLSI of the circuits defining their electrical device: given a fixed-width plate and a list of rectangular circuits, we have to place them on the plate so that the length of the final device is minimized (improving its portability). Here we have two variants of the problem. In the first, each circuit must be placed in a fixed orientation with respect to the others. This means that, an $n \times m$ circuit cannot be positioned as an $m \times n$ circuit in the silicon plate. In the second case, the rotation is allowed, which means that an $n \times m$ circuit can be positioned either as it is or as $m \times n$. In the end, for both problems we have 40 instances to design.
All models were run on an Anaconda Jupyter Notebook using an Apple M1 CPU with 16GB of RAM.

1.1 Variables

For each instance i, we defined the following variables for all the four approaches:

- w^i, the integer variable that represents the width of the plate;
- n^i, the integer variable that represents the number of rectangles to place;
- $x\_components^i$, which is an integer array of size n^i where each value $x\_components^i_j$ is the width of rectangle j;
- $y\_components^i$, which is an integer array of size n^i where each value $y\_components^i_j$ is the width of the rectangle j;
- $x\_positions^i$, which is an integer array of size n^i where each value $x\_components^i_j$ is the bottom-left x coordinate of the rectangle j. The domain of each value is the range $[0, w^i - \min(x\_components^i)]$;
- $y\_positions^i$, which is an integer array of size n^i where each value $y\_components^i_j$ is the bottom-left y coordinate of the rectangle j. The domain of each value is the range $[0, h^i - \min(y\_components^i)]$.

All these variables are shown in *Figure 1*.

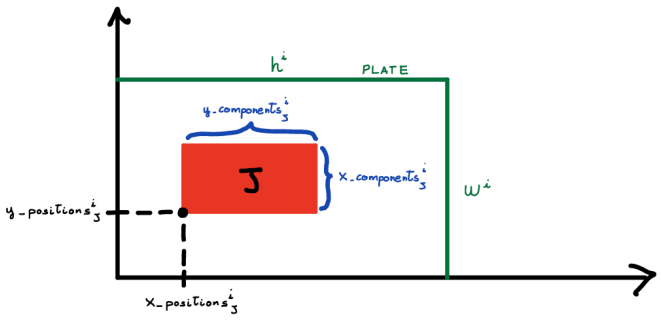

Figure 1: variables for a rectangle j of an instance i.

Regarding the upper bounds for both $x\_positions^i$ and $y\_positions^i$, we set them as the difference between the width/height of the plate and the minimum width/height of the components, to enhance the performance of the model. Finally, it is noteworthy to mention the variable rot^i_j, which is a Boolean arrays of dimension n^i, where each element rot^i_j is *True* or *False* if the rectangle j has been rotated or not.

The objective function to minimize is the *height* of the plate of each instance, which is the variable h^i. The upper bound is the sum of the heights of all the rectangles to place, while the lower bound is the ration between the sum of all the area of the rectangles to place and the width of the plate, as in *[1]*. We can formalize it as follows:
$h^i \in [lb^i, ub^i]$ where

$$lb^i = \frac{\sum_{j=1}^{n^i} (x\_components^i[j] \cdot y\_components^i[j])}{w^i}$$

$$ub^i = \sum_{j=1}^{n^i} (y\_components^i[j])$$

From now on all the formulae and variables are presented for a generic istance i, and so we will not use the index i to lighten the notation.
For each instance, the output of a model is a text file containing the width and the height of the plate and the two bottom left coordinates for each rectangle.

1.2 Constraints

Here we explain the main constraints implemented for the models. In particular, we used:

- the *position constraint*, which ensures that the length of each rectangle in both dimensions (width and height) does not exceed the length of the plate, as illustrated in the following formulae:

$$x\_positions^i[j] + x\_components^i[j] \leq w^i \quad \text{with } j \in [1,n^i]$$
$$y\_positions^i[j] + y\_components^i[j] \leq h^i \quad \text{with } j \in [1,n^i]$$

- the *no overlappping constraint*, which ensures that the placed rectangles must not overlap in both dimensions, as shown in the following formulas:

$$\begin{aligned} &x\_positions^i[j] + x\_components^i[j] \leq x\_positions^i[k] \quad \vee \\ &x\_positions^i[k] + x\_components^i[k] \leq x\_positions^i[j] \quad \vee \\ &y\_positions^i[j] + y\_components^i[j] \leq y\_positions^i[k] \quad \vee \\ &y\_positions^i[k] + y\_components^i[k] \leq y\_positions^i[j] \end{aligned}$$
$$\text{with } j \in [1,n^i], \quad k \in [1,n^i] \quad \text{and } j \neq k$$

- the *simmetry breaking constraint*, which allows to break some simmetries and so improving the performance of the model. For each approach, we considered different simmetry breaking constraints, which will be explained in the dedicated sections. The only simmetry breaking constraints that we adopted in all the models were the *highest rectangle constraint* and the *square constraint*. The *highest rectangle constraint* requires the rectangle with the highest height to always be placed at the origin. We observed that this constraint improves the model's performance by breaking the symmetries of this rectangle and eliminating solutions with suboptimal height *(Figure 2)*. This constraint also applies to the rotated case, where rotation is not allowed for the highest rectangle. The *square constraint* prevents rotation for rectangles where width equals height, meaning when the rectangle is actually a square.

- the *opposit component constraint*, which sets rot_j to *False* for the rectangles with a component higher than the opposite plate limit (i.e. if the rectangle has the x component higher than the height of the plate, the rotation is set to *False*).

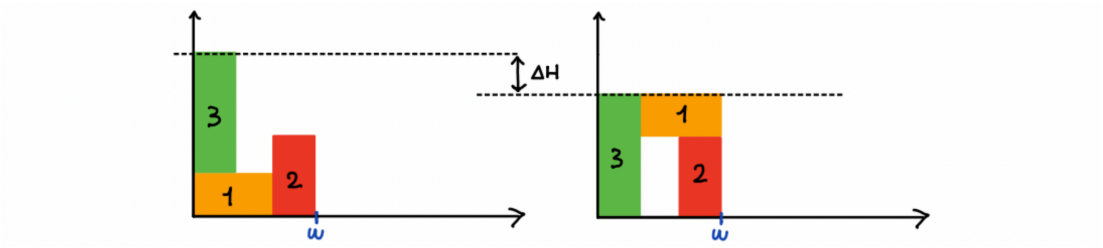

Figure 2: Solution with and without the highest rectangle placed in the origin. As we can see, its position can lead to a not-optimal solution.

2. CP

In this chapter we illustrate our CP approach to solve both variants of the VLSI problem by using two solvers, namely Chuffed and Geocode. Both solutions were implemented with Minizinc.

2.1 Decision Variables and Encodings

All the variables utilized in the CP approach are previously explained in *Section 1.1*, retaining the same meaning and domains.

2.2 Objective Function

The objective function is to minimize the height h of the plate, whose bounds were previously discussed in *Section 1.1*.

2.3 Constraints

For the CP models we adopted all the constraints explained in *Section 1.2*. In paricular, we implemented the *position constraint* for both dimensions with the *forall constraint:*

forall (j in 1..n) (y_components[j] + y_positions[j] <= h)

forall (j in 1..n) (x_components[j] + x_positions[j] <= w)

Then, we also applied the *no overlapping constraint* by using the *diffn constraint [2]*:

diffn (x_positions, y_positions, x_components, y_components)

2.3.1 Implied Constraints

In addition to the previous constrains, we also osberve that the *cumulative constraint [2]* improves the performance of the model. To implement this condition, we noticed that the VSLI design problem can be approached as a resource allocation problem: each rectangle has been considered as a task whose duration is $x\_components$, its amount of resources is $y\_components$, and the capacity is h. The same consideration can be applied for the other dimension in the opposite way, obtaining:

cumulative (*x_positions, x_components, y_components, h)*

cumulative (*y_positions, y_components, x_components, w)*

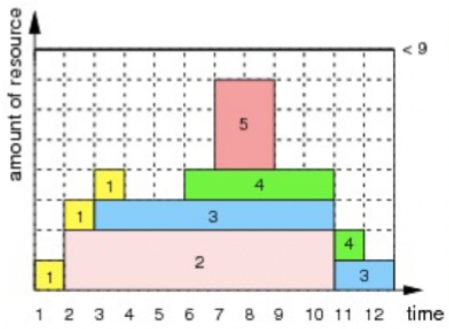

Figure 3: the resource allocation for a capacity ≤ 9 (horizontal line) seen as h in our case, *x_components* as duration and *y_components* as amount of resources (figure taken from).

2.3.2 Simmetry Breaking Constraints

This problem features two types of symmetries, which result in equivalent configurations of the same plate. In particular, one symmetry can be observed along the x-axis, where the position of two rectangles can be swapped horizontally; similarly, the same type of symmetry is present along the y-axis, where the swap can be made vertically. To break both simmetries, we imposed an order between the rectangles with the *lexicographic constraint [3]*:

lex_lesseq([x_positions[j] | j in 1..n], [rightest_coord_x - x_positions[j] - x_components[j] | j in 1..n])
lex_lesseq([y_positions[j] | j in 1..n], [hightest_coord_y - y_positions[j] - y_components[j] | j in 1..n])

where the first one breaks the simmetries along the x and the second along the y-axis. For each constraint the *righest_coord_x* and the *righest_coord_y* are defined as

rightest_coord_x = max([x_positions[j] + x_components[j] | j in RECTANGLES])
rightest_coord_y = max([y_positions[j] + y_components[j] | j in RECTANGLES])

Finally, we adotped also the *highest rectangle constraint* (*Section 1.2*), which enhances the performance of the model.

2.4 Rotation

For the rotation variant of the problem we introduced some new variables:

- $x\_components\_rot_j$, which is an array that contains the width of each rectangle *j*;
- $y\_components\_rot_j$, which is an array that contains the height of each rectangle *j*;

Here, we implement the rotation by using the variable rot_j *(Section 1.1)* with the following condition:

forall (j in RECTANGLES) (if rotations[j] then y_components_rot[j] == x_components[j] ∧
x_components_rot[j] == y_components[j] else x_components_rot[j] == x_components[j] ∧
y_components_rot[j] == y_components[j] endif);

which allows to swap the components of each rectangle if rotation is applied.

Regarding the constraints, we employed the previous constraints such as non-overlapping, position, opposite component (*Section 1.2*) and cumulative, which here is imposed only for the y-component. We noticed that including the cumulative constraint also for the x-coordinate worsens the performance of the model.

Regarding the symmetries, we utilized the same symmetry-breaking constraints from the case without rotation, and also added the *square constraint (Section 1.2)*:

forall (j in RECTANGLES) (if x_components[j] == y_components[j] then rotations[j]==false else true endif)

2.5 Validation

We implemented the CP model with two solvers, *Gecode* and *Chuffed* with the *free search* strategy and a constant *restart* condition, after seeing that the input order strategy and the no restart condition led to worse performances. The objective values without rotation are shown in *Table 1,* while *Table 2* contains the results with rotation:

Instance	h_min	Gecode w/out SB	Gecode + SB	Chuffed w/out SB	Chuffed + SB	Instance	h_min	Gecode w/out SB	Gecode + SB	Chuffed w/out SB	Chuffed + SB
1	8	**8**	**8**	**8**	**8**	**21**	28	N/A	N/A	**28**	**28**
2	9	**9**	**9**	**9**	**9**	**22**	29	N/A	N/A	**29**	**29**
3	10	**10**	**10**	**10**	**10**	**23**	30	N/A	N/A	**30**	**30**
4	11	**11**	**11**	**11**	**11**	**24**	31	N/A	N/A	**31**	**31**
5	12	**12**	**12**	**12**	**12**	**25**	32	N/A	N/A	**32**	**32**
6	13	**13**	**13**	**13**	**13**	**26**	33	N/A	N/A	**33**	**33**
7	14	**14**	**14**	**14**	**14**	**27**	34	N/A	N/A	**34**	**34**
8	15	**15**	**15**	**15**	**15**	**28**	35	N/A	N/A	**35**	**35**
9	16	**16**	**16**	**16**	**16**	**29**	36	N/A	N/A	**36**	**36**
10	17	**17**	**17**	**17**	**17**	**30**	37	N/A	N/A	**37**	**37**
11	18	N/A	N/A	**18**	**18**	**31**	38	N/A	N/A	**38**	**38**
12	19	**19**	**19**	**19**	**19**	**32**	39	N/A	N/A	**39**	**39**
13	20	**20**	**20**	**20**	**20**	**33**	40	N/A	N/A	**40**	**40**
14	21	**21**	**21**	**21**	**21**	**34**	40	N/A	N/A	**40**	**40**
15	22	N/A	**22**	**22**	**22**	**35**	40	N/A	N/A	**40**	**40**
16	23	N/A	N/A	**23**	**23**	**36**	40	N/A	N/A	**40**	**40**
17	24	N/A	N/A	**24**	**24**	**37**	60	N/A	N/A	**60**	**60**
18	25	N/A	N/A	**25**	**25**	**38**	60	N/A	N/A	**60**	**60**
19	26	N/A	N/A	**26**	**26**	**39**	60	N/A	N/A	**60**	**60**
20	27	N/A	N/A	**27**	**27**	**40**	90	N/A	N/A	91	91

*Table 1 - **No Rotation**:* the results of each instance for *Geocode* and *Chuffed*, comparing the solutions with and without simmetry breaking constraints.

Instance	h_min	Gecode w/out SB	Gecode + SB	Chuffed w/out SB	Chuffed + SB	Instance	h_min	Gecode w/out SB	Gecode + SB	Chuffed w/out SB	Chuffed + SB
1	8	**8**	**8**	**8**	**8**	**21**	28	N/A	N/A	**28**	**28**
2	9	**9**	**9**	**9**	**9**	**22**	29	N/A	N/A	**29**	**29**
3	10	**10**	**10**	**10**	**10**	**23**	30	N/A	N/A	**30**	**30**
4	11	**11**	**11**	**11**	**11**	**24**	31	N/A	N/A	**31**	**31**
5	12	**12**	**12**	**12**	**12**	**25**	32	N/A	N/A	33	**32**
6	13	**13**	**13**	**13**	**13**	**26**	33	N/A	N/A	**33**	**33**
7	14	**14**	**14**	**14**	**14**	**27**	34	N/A	N/A	**34**	**34**
8	15	**15**	**15**	**15**	**15**	**28**	35	N/A	N/A	**35**	**35**
9	16	**16**	**16**	**16**	**16**	**29**	36	N/A	N/A	**36**	**36**
10	17	**17**	**17**	**17**	**17**	**30**	37	N/A	N/A	38	**37**
11	18	N/A	N/A	**18**	**18**	**31**	38	N/A	N/A	**38**	**38**
12	19	**19**	**19**	**19**	**19**	**32**	39	N/A	N/A	**39**	40
13	20	**20**	**20**	**20**	**20**	**33**	40	N/A	N/A	**40**	**40**
14	21	**21**	**21**	**21**	**21**	**34**	40	N/A	N/A	**40**	**40**
15	22	**22**	**22**	**22**	**22**	**35**	40	N/A	N/A	**40**	**40**
16	23	N/A	N/A	**23**	**23**	**36**	40	N/A	N/A	**40**	**40**
17	24	N/A	N/A	**24**	**24**	**37**	60	N/A	N/A	**60**	61
18	25	N/A	N/A	**25**	**25**	**38**	60	N/A	N/A	**60**	61
19	26	N/A	N/A	**26**	**26**	**39**	60	N/A	N/A	**60**	**60**
20	27	N/A	N/A	**27**	**27**	**40**	90	N/A	N/A	91	91

*Table 2 - **Rotation**:* the results of each instance for Geocode and Chuffed, comparing the solutions with and without simmetry breaking constraints.

The results indicate that Chuffed surpasses Geocode in both problem variants. Geocode was unable to solve more than half of the instances within the time limit, while Chuffed consistently found a feasible solution within the time limit and, except for a few instances, the solution was optimal. The symmetry breaking constraints did not have an impact on the solution found in either variant of the

problem. The solution was always optimal, with a minimal difference in a few instances. Given Chuffed's superior performance compared to Geocode, the following discussion will exclusively focus on Chuffed.

The following figure (*Figure 4*) shows the time required by Chuffed to solve each instance for both problem variants:

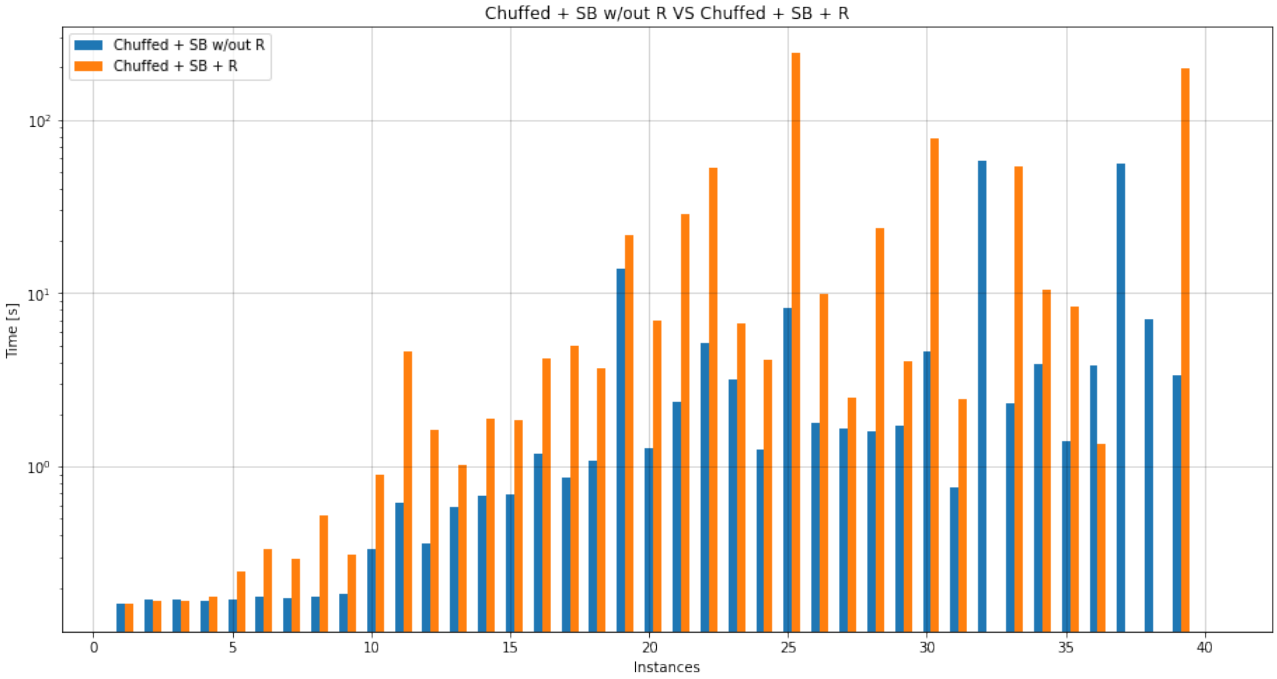

Figure 4: compared times between the two variants of the problem. The results are shown only for the best solver, Chuffed.

As expected, the results demonstrate that instances with rotation take longer to solve compared to the others, with a significant difference starting from instance 5. Finally, *Table 3* illustrates some statistics of the model:

N/A is seen as 300s.
Non Optimal solution has required 300s.

	CH w/out SB w/out R	CH + SB w/out R	CH w/out SB + R	CH + SB + R
Total Time [s]	630.27	491.659	2001.613	1984.004
Max [s]	300.0	300.0	300.0	300.0
Min [s]	0.159	0.161	0.163	0.162
Mean [s]	15.757	12.291	50.04	49.6
Std [s]	50.753	48.287	94.943	97.914
Instances Solved with H optimal	39/40	39/40	37/40	36/40

Table 3: some statistics on the Chuffed model (with and without simmetry breaking plus with and without rotation).

Table 3 reveals that the model performs better globally for the non-rotated variant, where the symmetry breaking constraints reduce the total time but do not increase the number of instances solved with the optimal h, which remains at 39. In the rotated variant, the model achieved an optimal solution for instances 25 and 30 only when using *symmetry breaking constraints*, and for instances 32, 37, and 38 without. Furthermore, the constraints did not impact the model's overall running time.

3. SAT

In this chapter we illustrate our SAT approach for the VSLI design problem by using Z3. Initially, we started developing a 3D SAT solver, as suggested in the tutorials *[4]*. However this approach led to poor performances of the model, with only 12 instances solved. So, we implemented a 2D SAT solver, as proposed in *[5]*.

3.1 Decision Variables and Encodings

In our first approach we used the 3D SAT solver proposed in *[4],* which introduces a new variable $plate^i$ for each instance i, defined as a 3D matrix of Bools $w^i \times h^i_{min} \times n^i$. Here, w^i is defined as width of the plate of the instance i, h^i_{min} is the lower bound of h^i discussed in *Section 1.1* and n^i as number of rectangles.

Since the performance of the previous approach led to only 12 instances solved within the time limit, we relied on the order encoding *[5]*, whose main feature is that a CSP comparison $x \leq e$ is encoded into a Boolean variable, which enables a compact encoding for our VLSI design problem. More specifically, the following new variables were introducted, while n^i, w^i and h^i_{min} are the same variables discussed in *Section 1.1*.

- $px^i_{j,e}$, a 2D Boolean array of dimension $n^i \times w^i$ where e refers to the x coordinate of the plate for the rectangle j. In particular, $px^i_{j,e}$ is *True* when $x_j \leq e$, so when the rectangle is placed at less than or equal to the x coordinate $e \in [0, w^i - x\_components^i_j]$;
- $py^i_{j,f}$, a 2D Boolean array of dimension $n^i \times h^i_{min}$ where f refers to the y coordinate of the plate for the rectangle j. In particular, $py^i_{j,f}$ is *True* when $y_j \leq f$, so when the rectangle is placed at less than or equal to the y coordinate $f \in [0, h^i_{min} - y\_components^i_j]$;
- $lr^i_{j,k}$, a Boolean variable that is *True* if the rectangle j is placed at the left of the rectangle k;
- $ud^i_{j,k}$, a Boolean variable that is *True* if the rectangle j is placed at the at the downward of the rectangle k.

3.2 Objective function

The objective function is to minimize the height h of the plate, whose bounds were previously discussed in *Section 1.1*. More specifically, our model starts by checking if the value of h^i_{min} satisfies the constraints within the time limit and so returning sat. However, if the instance with h^i_{min} isn't the solution the model checks the next feasible value for h^i, considering as time left the difference between the timeout time (5 minutes) and the time occured to check the previous value and to create the constraints.

3.3 Constraints

Regarding the 3D encoding, we adopted the no-overlapping and position constraints previously explained in *Section 1.1*. However, we noticed significantly inferior performance compared to the 2D encoded SAT solver. Specifically, the time required for constructing the constraints exceeded the allotted time limit without even verifying the model. As a result, we resolved to concentrate our efforts on the 2D encoded SAT, incorporating the simmetries discussed in *Section 1.2* and including the capability for rotation.

For the 2D SAT encoding we have the 2-literal axiom clauses due to order encoding for each rectangle *j*: (for semplicity, we don't report the *i* instance)

$$\neg px_{j,e} \vee px_{j,e+1}$$

$$\neg py_{j,f} \vee py_{j,f+1}$$

$$\text{with } 0 \leq e < w - x\_components_j \text{ and } 0 \leq f < h_{min} - y\_components_j$$

Then we implemented the no-overlapping constraint (previuosly discussed in *Section 1.2*) with the following clauses:

$$lr_{j,k} \vee lr_{k,j} \vee ud_{j,k} \vee ud_{k,j}$$

$$\neg lr_{j,k} \vee px_{j,e} \vee \neg px_{k,e+x\_components_j}$$

$$\neg lr_{k,j} \vee px_{k,e} \vee \neg px_{j,e+x\_components_k}$$

$$\neg ud_{j,k} \vee py_{j,f} \vee \neg py_{k,f+y\_components_j}$$

$$\neg ud_{k,j} \vee py_{k,f} \vee \neg py_{j,f+y\_components_k}$$

$$\neg lr_{j,k} \vee \neg px_{k,x\_components_j-1}$$

$$\neg lr_{j,k} \vee px_{j,\ w-x\_components_j-1}$$

$$\neg ud_{j,k} \vee \neg py_{k,y\_components_j-1}$$

$$\neg ud_{j,k} \vee py_{j,\ h_{min}-y\_components_j-1}$$

$$\text{with } r_j, r_k \text{ two rectangles such that } i < j$$

Finally, we ensure that all the rectangles are placed with the following constraint:

$$px_{j,e} \text{ with } w - x\_components_j \leq e < w$$

$$py_{j,f} \text{ with } h_{min} - y\_components_j \leq f < w$$

3.3.1 Implied Constraints

In addition we also added the *Large Rectangle constraint,* where for each pair of placed rectangles their width and height shouldn't exceed the width and the height of the plate: This constraint is implemented by adding the clause:

$$\neg lr_{j,k} \wedge \neg lr_{k,j}.$$

3.3.2 Simmetry Breaking Constraints

Regarding the simmetry breaking constraints, we implemented the *highest rectangle constraint (Section 1.2)*, by setting $px_{1,0} = True$ and $py_{1,0} = True$, remembering that the first rectangle is the one with the highest y component. Besides, we excluded the flip between two rectangles with the same dimensions (i.e. two rectangles r_j, r_k such that if $(x\_components_j, y\_components_j) = (x\_components_k, y\_components_k)$ then we add $\neg lr_{j,k}$ and $lr_{j,k} \vee \neg ud_{k,j}$.

3.4 Rotation

For the rotation variant we used the rot_j variable (*Section 1.2*). The constraints implemented for this variant are the same of the problem without rotation and the rot variable ensures that if the rotation is applied, the previous constraints are executed by swapping the conditions on the x and y components of the rectangles, according to the following formulae:

- *order constraints*:

$$\left[\left(\bigwedge_{e=0}^{w-x\_components_j-1} \neg px_{j,e} \vee px_{j,\,e+1}\right) \wedge \neg rot_j\right] \vee \left[\left(\bigwedge_{e=0}^{w-y\_components_j-1} \neg px_{j,e} \vee px_{j,\,e+1}\right) \wedge rot_j\right]$$

$$\left[\left(\bigwedge_{f=0}^{h_{min}-y\_components_j-1} \neg py_{j,f} \vee py_{j,\,f+1}\right) \wedge \neg rot_j\right] \vee \left[\left(\bigwedge_{f=0}^{h_{min}-x\_components_j-1} \neg py_{j,f} \vee py_{j,\,f+1}\right) \wedge rot_j\right]$$

- *placement constraints*:

$$\left[\left(\bigwedge_{e=0}^{w-x\_components_j-1} px_{j,e}\right) \wedge \neg rot_j\right] \vee \left[\left(\bigwedge_{e=0}^{w-y\_components_j-1} px_{j,e}\right) \wedge rot_j\right]$$

$$\left[\left(\bigwedge_{f=0}^{h_{min}-y\_components_j-1} py_{j,f}\right) \wedge \neg rot_j\right] \vee \left[\left(\bigwedge_{f=0}^{h_{min}-x\_components_j-1} py_{j,f}\right) \wedge rot_j\right]$$

- *non overlap constraints*:

$$\left[\left(\bigwedge_{e=0}^{w-x\_components_j-1} \neg lr_{j,k} \vee px_{j,e} \vee \neg px_{k,e+x\_components_j}\right) \wedge \neg rot_j\right] \vee \left[\left(\bigwedge_{e=0}^{w-y\_components_j-1} \neg lr_{j,k} \vee px_{j,e} \vee \neg px_{k,e+y\_components_j}\right) \wedge rot_j\right]$$

$$\left[\left(\bigwedge_{e=0}^{w-x\_components_k-1} \neg lr_{k,j} \vee px_{k,e} \vee \neg px_{j,e+x\_components_k}\right) \wedge \neg rot_j\right] \vee \left[\left(\bigwedge_{e=0}^{w-y\_components_k-1} \neg lr_{k,j} \vee px_{k,e} \vee \neg px_{j,e+y\_components_k}\right) \wedge rot_j\right]$$

$$\left[\left(\bigwedge_{f=0}^{h_{min}-y\_components_j-1} \neg ud_{j,k} \vee py_{j,f} \vee \neg py_{k,f+y\_components_j}\right) \wedge \neg rot_j\right] \vee \left[\left(\bigwedge_{f=0}^{h_{min}-x\_components_j-1} \neg ud_{j,k} \vee py_{j,f} \vee \neg py_{k,f+x\_components_j}\right) \wedge rot_j\right]$$

$$\left[\left(\bigwedge_{f=0}^{h_{min}-y\_components_k-1} \neg ud_{k,j} \vee py_{k,f} \vee \neg py_{j,f+y\_components_k}\right) \wedge \neg rot_j\right] \vee \left[\left(\bigwedge_{f=0}^{h_{min}-x\_components_k-1} \neg ud_{k,j} \vee py_{k,f} \vee \neg py_{j,f+x\_components_k}\right) \wedge rot_j\right]$$

$$\left[\left(\neg lr_{j,k} \vee \neg px_{k,x\_components_j-1}\right) \wedge \neg rot_j\right] \vee \left[\left(\neg lr_{j,k} \vee \neg px_{k,y\_components_j-1}\right) \wedge rot_j\right]$$

$$\left[\left(\neg ud_{j,k} \vee py_{j,y\_components_j-1}\right) \wedge \neg rot_j\right] \vee \left[\left(\neg ud_{j,k} \vee py_{j,\,w-x\_components_j-1}\right) \wedge rot_j\right]$$

Besides, the following clause of the no overlapping constraint is not tied to variable rot_j, despite it being active for the rotation: $lr_{j,k} \vee lr_{k,j} \vee ud_{j,k} \vee ud_{k,j}$.
Finally, we also added the *highest rectangle constraint* and the *square constraint* (*Section 1.2*).

3.5 Validation

We implemented the SAT model with the Z3 solver. The results of each instance for both variants of the problem are shown in *Table 4:*

Instance	h_min	SAT w/out SB w/out R	SAT + SB w/out R	SAT w/out SB + R	SAT + SB + R	Instance	h_min	SAT w/out SB w/out R	SAT + SB w/out R	SAT w/out SB + R	SAT + SB + R
1	8	8	8	8	8	21	28	28	28	28	28
2	9	9	9	9	9	22	29	29	29	N/A	29
3	10	10	10	10	10	23	30	30	30	30	30
4	11	11	11	11	11	24	31	31	31	31	31
5	12	12	12	12	12	25	32	32	32	N/A	N/A
6	13	13	13	13	13	26	33	33	33	33	33
7	14	14	14	14	14	27	34	34	34	34	34
8	15	15	15	15	15	28	35	35	35	35	35
9	16	16	16	16	16	29	36	36	36	36	36
10	17	17	17	17	17	30	37	37	37	N/A	37
11	18	18	18	18	18	31	38	38	38	38	38
12	19	19	19	19	19	32	39	N/A	39	N/A	N/A
13	20	20	20	20	20	33	40	40	40	40	40
14	21	21	21	21	21	34	40	40	40	40	40
15	22	22	22	22	22	35	40	40	40	40	40
16	23	23	23	23	23	36	40	40	40	40	40
17	24	24	24	24	24	37	60	60	60	N/A	N/A
18	25	25	25	25	25	38	60	N/A	N/A	60	60
19	26	26	26	26	26	39	60	60	60	N/A	60
20	27	27	27	27	27	40	90	N/A	N/A	N/A	N/A

Table 4: the results of each instance for SAT with and without simmetry breaking and with and without rotation.

Table 4 demonstrates that for both problem variants, the model consistently finds the optimal solution as long as it does not exceed the time limit. In the non-rotated variant, instances 38 and 40 cannot be solved regardless of the symmetry breaking constraints, but the constraints do affect instance 32. In the same way, for the rotated variant instances 25, 32, 37 and 40 cannot be solved with or without simmetry breaking constraint; however, adding these constraints helps the model to find the optimal solution for instances 22, 30 and 39. *Table 5* reports some statistics of our SAT model:

	SAT w/out SB w/out R	SAT + SB w/out R	SAT w/out SB + R	SAT + SB + R
Total Time [s]	1134.767	993.825	2632.973	2433.407
Max [s]	300.0	300.0	300.0	300.0
Min [s]	0.044	0.043	0.123	0.134
Mean [s]	28.369	24.846	65.824	60.835
Std [s]	78.957	71.622	110.514	101.056
Instances Solved with H optimal	37/40	38/40	33/40	36/40

Table 5: some statistics on SAT (with and without simmetry breaking plus with and without rotation).

Table 5 shows that our SAT model solved 38 instances without rotation and 36 instances with rotation, both with symmetry breaking constraints.The impact of the symmetry breaking constraint was more evident in the rotated case, resulting in a reduction of 3 solved instances, rather than just 1. Regarding the total time, the rotated variant has a higher value, as expected, while both variants benefit from the addition of symmetry breaking constraints. *Figure 5* shows the time required by the model to solve each instance for both variants.

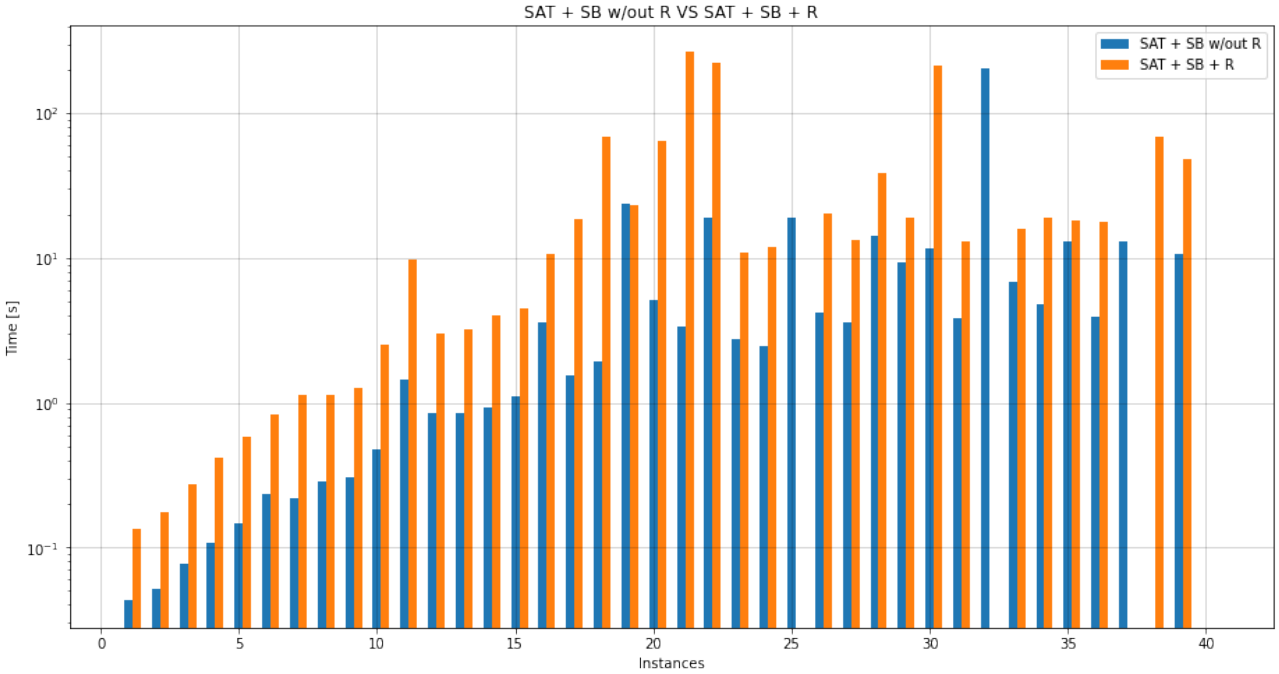

Figure 5: compared times between the two variants of the problem. The results are shown only for the variant with the simmetry breaking constraints.

4. SMT

In this chapter we illustrate our SMT approach for the VSLI design problem by using Z3 with Python. We adopted a simple encoding that exploits the variables defined in Section 1.1, while here we also implemented the cumulative constraint, which is shown to be oosts the performance of our model.

4.1 Decision Variables and Encodings

For our SMT model we adopted a simple encoding that uses the same variables discussed in *Section 1.1* plus the two Boolean variables previously defined for the SAT model (*Section 3.1*), namely $lr_{j,k}$ and $ud_{j,k}$.

4.2 Objective function

The objective function is to minimize the height h of the plate, whose bounds were previously discussed in *Section 1.1*.

4.3 Constraints

For our SMT approach we adopted all the constraints defined in *Section 1.2*. The no overlapping constraint is here implemented by using the variables $lr_{j,k}$ and $ud_{j,k}$:

$$\begin{gathered} lr_{j,k} == x\_positions_j + x\_component_j \leq x\_positions_k \\ lr_{k,j} == x\_positions_k + x\_component_k \leq x\_positions_j \\ ud_{j,k} == y\_positions_j + y\_component_j \leq y\_positions_k \\ ud_{k,j} == y\_positions_k + y\_component_k \leq y\_positions_j \\ \text{with } j \text{ and } k \text{ two any rectangles such that } j < k \end{gathered}$$

then, as we have done for the SAT model, we impose that at least one of the previous constraint should be active, as follow:

$$lr_{j,k} \vee lr_{k,j} \vee ud_{j,k} \vee ud_{k,j}$$

In addition, the position constraint is implemented as defined in *Section 1.2*:

$$\begin{aligned} x\_positions[j] + x\_components[j] \leq w \quad \text{with } j \in [1,n] \\ y\_positions[j] + y\_components[j] \leq h \quad \text{with } j \in [1,n] \end{aligned}$$

4.3.1 Implied Constraints

We have also adopted some other constraits to enhance the performacne of our model. In particular, we added the large rectangle constraint defined previously for the SAT model (Section 3.3.1), here implemented as:

$$\begin{gathered} \textit{if } (x\_components_j + x\_components_k > w) \textit{ then } lr_{j,k} == \textit{False} \wedge lr_{k,j} == \textit{False} \\ \textit{if } (y\_components_j + y\_components_k > h) \textit{ then } ud_{j,k} == \textit{False} \wedge ud_{k,j} == \textit{False} \\ \textit{with j and k two any rectangles such that } j < k \end{gathered}$$

To ensure that the x and y coordinates of the placed rectangles should be equal or greater than 0 we impose that:

$$x\_positions_j \geq 0 \text{ and } y\_positions_j \geq 0 \quad \text{with } j \in [1,n]$$

Finally, we osberved that with the previous constraintsthe model was able to solve only 25 instances without rotation: for this reason we used the cumulative constraint (already adopted in *Section 2.3.1*), which here we applied only on the horizontal axis:

$$\bigwedge_{q=0}^{w}\left(\sum_{j=1}^{n} if\left((x\_positions_j \le q) \wedge (x\_positions_j + x\_components_j > q),\ x\_components_j,\ 0\right) \le h\right)$$

$$\text{with } j \in [1,n]$$

4.3.2 Simmetry Breaking Constraints

Regarding the simmetry breaking constraints, here we used the *highest rectangle constraint* and the *square constraint* (both defined in *Section 1.2*). The latter constraint is here implemented with a simple *if condition*:

$$If\ (x\_components_j == x\_components_k) \wedge (y\_components_j == y\_components_k)$$
$$then\ \neg lr_{j,k} \wedge (lr_{j,k} \vee ud_{j,i})$$
$$with\ j\ and\ k\ two\ any\ rectangles\ such\ that\ j < k$$

4.4 Rotation

To implement the rotation variant we used the variable rot_j defined in *Section 1.2*. With this variable, the new component of ech rectangle *j* are defined as follow:

$$if\ \ rot_j\ \ then\ \ x\_components\_rot_j == y\_components_j\ \ else\ x\_components\_rot_j == x\_components_j$$
$$if\ \ rot_j\ \ then\ \ y\_components\_rot_j == x\_components_j\ \ else\ y\_components\_rot_j == y\_components_j$$

These variables are now used in the no overlapping and position constraints, where $x\_components_j$ and $y\_components_j$ are now substituted by $x\_components\_rot_j$ $y\_components\_rot_j$.

We also added the following constraints to avoid rotation if the component of the rectangle exceeds the dimensions of the opposite plate limits:

$$if\ x\_components_j > h\ \ then\ \ rot_j == False$$
$$if\ y\_components_j > w\ \ then\ \ rot_j == False$$
$$with\ \ j \in [1,n]$$

Finally, we used the square constraint and the highest rectangle constraints for this rotation variant, as discussed in *Section 1.2*.

4.5 Validation

We implemented our SMT model with the Z3 solver on Python. The results for each instance are shown in Table:

Instance	h_min	SMT w/out SB w/out R	SMT + SB w/out R	SMT w/out SB + R	SMT + SB + R	Instance	h_min	SMT w/out SB w/out R	SMT + SB w/out R	SMT w/out SB + R	SMT + SB + R
1	8	8	8	8	8	21	28	28	28	N/A	28
2	9	9	9	9	9	22	29	29	29	N/A	29
3	10	10	10	10	10	23	30	30	30	30	30
4	11	11	11	11	11	24	31	31	31	31	31
5	12	12	12	12	12	25	32	32	32	N/A	N/A
6	13	13	13	13	13	26	33	33	33	33	33
7	14	14	14	14	14	27	34	34	34	N/A	34
8	15	15	15	15	15	28	35	35	35	N/A	35
9	16	16	16	16	16	29	36	36	36	N/A	36
10	17	17	17	17	17	30	37	N/A	N/A	N/A	N/A
11	18	18	18	18	18	31	38	38	38	38	38
12	19	19	19	19	19	32	39	N/A	N/A	N/A	N/A
13	20	20	20	20	20	33	40	40	40	40	40
14	21	21	21	21	21	34	40	40	40	40	N/A
15	22	22	22	22	22	35	40	40	40	N/A	N/A
16	23	23	23	N/A	23	36	40	40	40	40	N/A
17	24	24	24	24	24	37	60	60	60	N/A	N/A
18	25	25	25	N/A	25	38	60	N/A	N/A	N/A	N/A
19	26	26	26	N/A	26	39	60	N/A	N/A	N/A	N/A
20	27	27	27	N/A	N/A	40	90	N/A	N/A	N/A	N/A

Table 6: the results of each instance for SMT with and without simmetry breaking and with and without rotation.

As seen in Table 6, the model consistently finds the optimal solution as long as it stays within the time limit. For the non-rotated variant, it is observed that the model found the optimal solution without the symmetry breaking constraints and that the constraints did not improve the time-exceeded instances. For the rotated variant, the impact of symmetry breaking constraints is more apparent, as 8 instances go from not being solved within the time limit to being solved with the optimal solution. It is also noteworthy that the model was unable to solve instances 30, 32, 38, 39, and 40 for both variants, regardless of the symmetry breaking constraints. *Table 7* shows some statistics of the model:

	SMT w/out SB w/out R	SMT + SB w/out R	SMT w/out SB + R	SMT + SB + R
Total Time [s]	2194.59	2060.039	6696.672	3870.686
Max [s]	300.0	300.0	300.0	300.0
Min [s]	0.037	0.035	0.031	0.047
Mean [s]	54.865	51.501	167.417	96.767
Std [s]	104.706	99.084	137.453	134.332
Instances Solved with H optimal	35/40	35/40	23/40	29/40

Table 7: some statistics on SMT (with and without simmetry breaking plus with and without rotation).

As indicated by the total time in *Table 7*, the model performs better for the non-rotated variant. There is no significant difference in the total time or number of instances solved optimally with h, whether symmetry breaking constraints are used or not. However, this is not true for the rotated variant, where using symmetry breaking constraints greatly reduces the total time and increases the number of optimal solution, from 23 to 29. Globally, the total time required by the rotation variant is higher than the non rotated variant, as we expected. *Figure 6* shows the time required by the model to solve the instances for both variants.

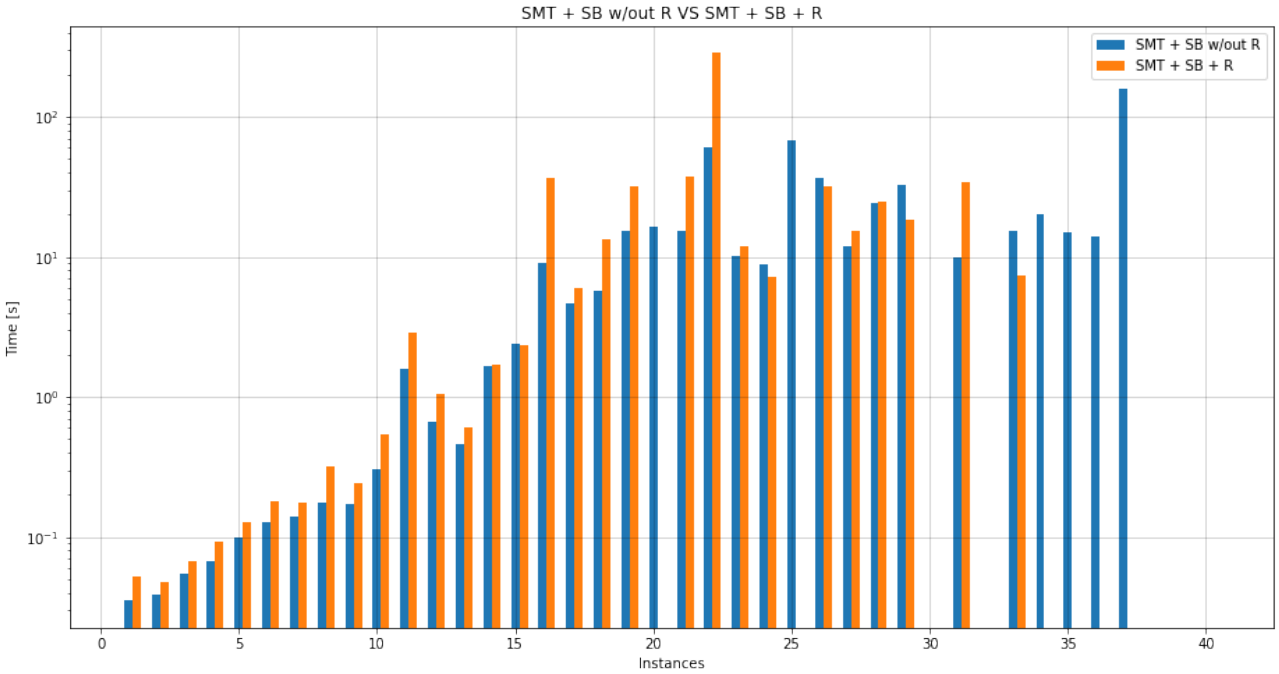

Figure 6: compared times between the two variants of the problem. The results are shown only for the variant with the simmetry breaking constraints.

5. MIP

In this chapter we illustrate our MIP approach for the VSLI design problem by using Gurobi. In particular, we adopted the big-M trick to encode the no overlapping constraint by introducing the binary variable *OR_overlap*. Finally, we compared the results for both variants of the problem, by also taking into account the Symmetry parameter of the Gurobi solver.

5.1 Decision Variables and Encodings

Since MIP solvers cannot handle logical operations, an encoding must be defined to represent the constraints discussed in *Section 1.2*. In particular, the no overlapping constraint is a logical OR between 4 inequalities: to represent this logical operation we relied on the big-M trick *[6]*. In particular, the *OR_overlap variable*, is introduced to activate or deactivate the no overlapping constraint. Its value is 0 when the constraint is active and 1 when inactive. The other variables used in the MIP model arethe ones defined in *Section 1.1*.

5.2 Objective function

The objective function is to minimize the height h of the plate, whose bounds were previously discussed in *Section 1.1*, with $h \in [h_{min}, h_{max}]$.

5.3 Constraints

As previously discussed in Section 5.1, we encode the *no overlapping constraint* (Section 1.2) with the big-M trick:

$$x\_positions_j + x\_components_j \leq x\_positions_k + M \cdot OR\_overlap[j,k,0]$$

$$y\_positions_j + y\_components_j \leq y\_positions_k + M \cdot OR\_overlap[j,k,1]$$

$$x\_positions_k + x\_components_k \leq x\_positions_j + M \cdot OR\_overlap[j,k,0]$$

$$y\_positions_k + y\_components_k \leq y\_positions_j + M \cdot OR\_overlap[j,k,3]$$

with j and k two any rectangles such that $j < k$

To ensure that at least one of the previous inequalities is active we constraint that the sum of the variables *OR_overlap* should be less than 3*:*

$$\sum_{r=0}^{4} OR\_overlap[j,k,r] \leq 3 \text{ with } i \in [0,n] \text{ and } j \in [i+1,n]$$

Then, we also considered the *position constraint* (Section 1.2), which doesn't require the big-M trick to be encoded.

5.3.1 Implied Constraints

To ensure that the x and y coordinates of the placed rectangles should be equal or greater than 0 we impose that:

$$x\_positions_j \geq 0 \text{ and } y\_positions_j \geq 0$$

5.3.2 Simmetry Breaking Constraints

Regarding the simmetry breaking constraints, we implemented the *highest rectangle constraint* (as previously discussed in *Section 1.2*) by adding:

$$x\_positions_0 == 0 \text{ and } y\_positions_0 == 0$$

We exploit the Symmetry parameter of the Gurobi solver, which controls symmetry detection. It has an integer value between -1 and 2, with higher values indicating more aggressive detection. We performed tuning of this parameter and found the optimal value to be 2.

5.4 Rotation

The variable rot_j (Section 1.1) allows us to represent the rotated component as follow:

$$y\_components\_rot_j == rot_j \cdot x\_components_j + (1 - rot_j) \cdot y\_components_j$$
$$x\_components\_rot_j == rot_j \cdot y\_components_j + (1 - rot_j) \cdot x\_components_j$$
$$\text{with } j \in [0,n]$$

These variables are now used in the no overlapping and position constraints, where $x\_components_j$ and $y\_components_j$ are now substituted by $x\_components\_rot_j$ $y\_components\_rot_j$.

We also added the following constraints to avoid rotation if the component of the rectangle exceed the dimensions of the opposite plate limits:

$$rot_j == 0 \text{ if } x\_components_j > h_{min}$$
$$rot_j == 0 \text{ if } y\_components_j > w$$
$$\text{with } j \in [1,n]$$

The constraint was imposed that the *highest block constraint* in the origin cannot be rotated (to have the highest rectangle in the origin):

$$rot_0 == 0$$

Finally, we also added the *square-constraint* (*Section 1.2*) as follow:

$$rot_j == 0 \text{ if } x\_components_j == y\_components_j \text{ with } j \in [1,n]$$

5.5 Validation

We implemented our MIP model with the Gurobi solver on Python. The results for each instance are shown in *Table 8*, considering also the *Symmetry* parameter:

Instance	h_min	MIP w/out SB w/out R	MIP + SB w/out R	MIP w/out SB + R	MIP + SB + R	Instance	h_min	MIP w/out SB w/out R	MIP + SB w/out R	MIP w/out SB + R	MIP + SB + R
1	8	**8**	**8**	**8**	**8**	**21**	28	**28**	**28**	29	29
2	9	**9**	**9**	**9**	**9**	**22**	29	**29**	**29**	30	30
3	10	**10**	**10**	**10**	**10**	**23**	30	**30**	**30**	31	31
4	11	**11**	**11**	**11**	**11**	**24**	31	**31**	**31**	**31**	**31**
5	12	**12**	**12**	**12**	**12**	**25**	32	33	33	33	33
6	13	**13**	**13**	**13**	**13**	**26**	33	**33**	**33**	**33**	**33**
7	14	**14**	**14**	**14**	**14**	**27**	34	**34**	**34**	**34**	**34**
8	15	**15**	**15**	**15**	**15**	**28**	35	**35**	**35**	36	**35**
9	16	**16**	**16**	**16**	**16**	**29**	36	**36**	**36**	**36**	**36**
10	17	**17**	**17**	**17**	**17**	**30**	37	**37**	**37**	38	38
11	18	**18**	**18**	**18**	**18**	**31**	38	**38**	**38**	**38**	**38**
12	19	**19**	**19**	**19**	**19**	**32**	39	40	40	40	40
13	20	**20**	**20**	**20**	**20**	**33**	40	**40**	**40**	**40**	**40**
14	21	**21**	**21**	**21**	**21**	**34**	40	**40**	**40**	**40**	**40**
15	22	**22**	**22**	**22**	**22**	**35**	40	**40**	41	41	**40**
16	23	**23**	**23**	**23**	**23**	**36**	40	**40**	**40**	41	**40**
17	24	**24**	**24**	**24**	**24**	**37**	60	63	62	62	61
18	25	**25**	**25**	**25**	26	**38**	60	62	62	62	61
19	26	**26**	**26**	**26**	27	**39**	60	61	61	61	61
20	27	**27**	**27**	28	28	**40**	90	99	99	100	100

Table 8: the results of each instance for MIP with and without *Symmetry parameter* and with and without rotation.

As we can see from *Table 8* our MIP solver is always able to find a solution within the time limit for both variants. Regarding the non rotated variant, for the first 24 istances the model output always the optimal solution regardless the *Symmetry parameter*, as also for instances from 26 to 31 and 33, 34 and 36. For the other instances, the solution found is always the same with and without using the simmetry breaking constraints: the values found are always very close to the optimal solution, as exception for instance 40, where the optimal solution is 90 and the one found is 99. For the rotated version, the trend is more or less the same: here the number of optimal solutions found is less, and the *Symmetry parameter* helps the model to find the optimal solution for instances 28, 35 and 36, while for instances 18 and 19 the behaviour is opposite. As observed for the non rotated variant, the non optimal solutions are always close to the optimal, as exception for instance 40. *Table 9* shows some statistics of our MIP solvers:

	MIP w/out SB w/out R	MIP + SB w/out R	MIP w/out SB + R	MIP + SB + R
Total Time [s]	2483.743	2504.815	4870.363	4838.492
Max [s]	300.0	300.0	300.0	300.0
Min [s]	0.002	0.002	0.002	0.002
Mean [s]	62.094	62.62	121.759	120.962
Std [s]	111.832	113.628	141.875	136.516
Instances Solved with H optimal	34/40	33/40	26/40	27/40

Table 9: some statistics on MIP (with and without Symmetry parameter and with and without rotation)

Table 9 shows that the best performance of the model is for the non-rotated variant without the *Symmetry parameter*, with 34 instances solved. The *Symmetry parameter* decreases the number of

solved instances to 33 for the non-rotated variant and to 26 for the rotated variant. The rotated variant has the highest total time, but the *Symmetry parameter* does not affect the runtime. Our results indicate that the *Symmetry parameter* has minimal impact on our MIP model. *Figure 7* shows the time required by the model to solve the instances for both rotation variants.

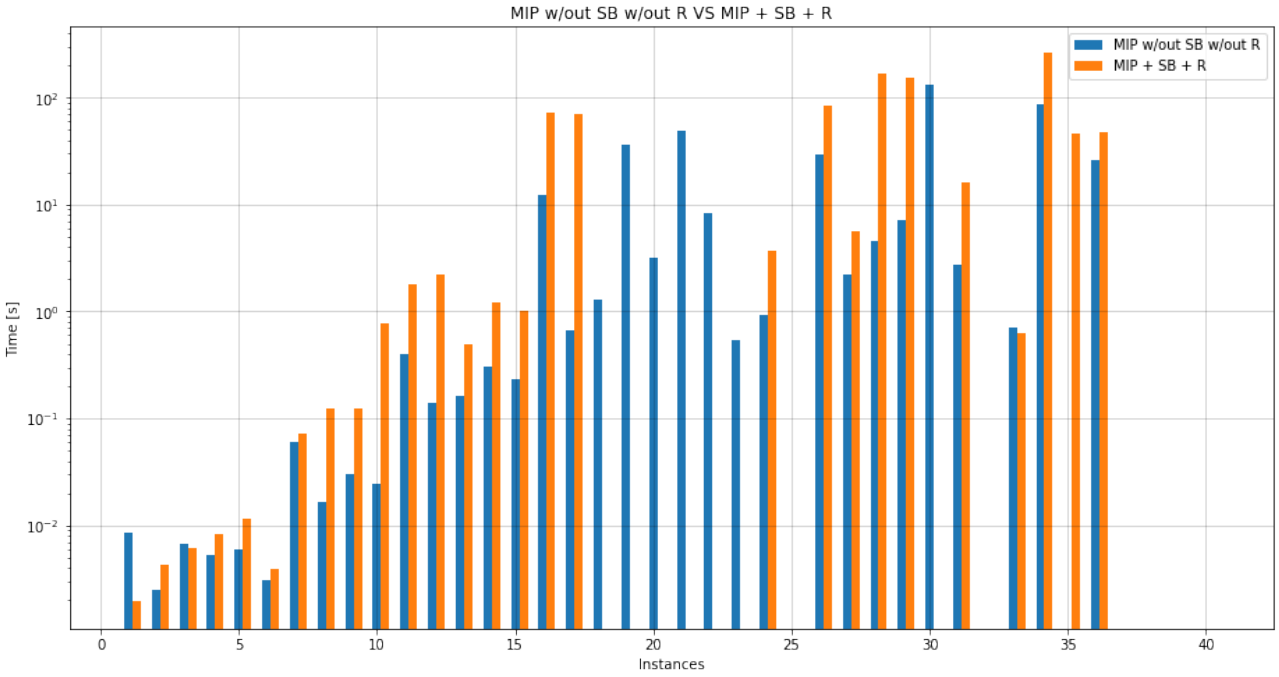

Figure 7: compared times between the two variants of the problem. The results are shown only for the variant without the *Symmetry parameter* with rotation and the variant with *Symmetry parameter*.

Conclusion

Finally, we compare all the four approaches for both the variants of the problem. In particular, for each approach we took the model that solved more instances with optimal solution. *Table 10* compares the results of the four models for the non rotation variant:

Instance	h_min	CP + SB w/out R	SAT + SB w/out R	SMT + SB w/out R	MIP w/out SB w/out R
1	8	8	8	8	8
2	9	9	9	9	9
3	10	10	10	10	10
4	11	11	11	11	11
5	12	12	12	12	12
6	13	13	13	13	13
7	14	14	14	14	14
8	15	15	15	15	15
9	16	16	16	16	16
10	17	17	17	17	17
11	18	18	18	18	18
12	19	19	19	19	19
13	20	20	20	20	20
14	21	21	21	21	21
15	22	22	22	22	22
16	23	23	23	23	23
17	24	24	24	24	24
18	25	25	25	25	25
19	26	26	26	26	26
20	27	27	27	27	27
21	28	28	28	28	28
22	29	29	29	29	29
23	30	30	30	30	30
24	31	31	31	31	31
25	32	32	32	32	33
26	33	33	33	33	33
27	34	34	34	34	34
28	35	35	35	35	35
29	36	36	36	36	36
30	37	37	37	N/A	37
31	38	38	38	38	38
32	39	39	39	N/A	40
33	40	40	40	40	40
34	40	40	40	40	40
35	40	40	40	40	40
36	40	40	40	40	40
37	60	60	60	60	63
38	60	60	N/A	N/A	62
39	60	60	60	N/A	61
40	90	91	N/A	N/A	99

Table 10 - ***No Rotation***: the results of each instance for the four models (i.e. CP, SAT, SMT and MIP) with and without simmetry.

Table 10 shows that all four models find the optimal solutions for 34 instances in the non-rotated variant. These instances include 1-29, 31, 33-36. The only unsolved instance is 40 for all models, with the CP model producing the closest solution to the optimal one. Instances 39 and 32 are difficult for both SMT and MIP solvers, while the CP model is the only one to solve instance 38 optimally. Finally, MIP fails to solve instance 25, while SMT fails to solve instance 30. *Table 11* compares some statistics of the models:

	CP + SB w/out R	SAT + SB w/out R	SMT + SB w/out R	MIP w/out SB w/out R
Total Time [s]	491.659	993.825	2060.039	2483.743
Max [s]	300.0	300.0	300.0	300.0
Min [s]	0.161	0.043	0.035	0.002
Mean [s]	12.291	24.846	51.501	62.094
Std [s]	48.287	71.622	99.084	111.832
Instances Solved with H optimal	39/40	38/40	35/40	34/40

Table 11: some statistics between the four models without rotation.

Overall, the CP algorithm performed the best, with the shortest runtime and the largest number of instances solved optimally. The SMT and MIP algorithms took significantly longer total time and and fewer instances were solved optimally with these models. The SAT algorithm had intermediate performance, with a longer total time compared to CP but shorter than SMT and MIP, and a moderate number of instances solved optimally. *Table 12* compares the results of the four models for the non rotation problem:

Instance	h_min	CP w/out SB + R	SAT + SB + R	SMT + SB + R	MIP + SB + R
1	8	**8**	**8**	**8**	**8**
2	9	**9**	**9**	**9**	**9**
3	10	**10**	**10**	**10**	**10**
4	11	**11**	**11**	**11**	**11**
5	12	**12**	**12**	**12**	**12**
6	13	**13**	**13**	**13**	**13**
7	14	**14**	**14**	**14**	**14**
8	15	**15**	**15**	**15**	**15**
9	16	**16**	**16**	**16**	**16**
10	17	**17**	**17**	**17**	**17**
11	18	**18**	**18**	**18**	**18**
12	19	**19**	**19**	**19**	**19**
13	20	**20**	**20**	**20**	**20**
14	21	**21**	**21**	**21**	**21**
15	22	**22**	**22**	**22**	**22**
16	23	**23**	**23**	**23**	**23**
17	24	**24**	**24**	**24**	**24**
18	25	**25**	**25**	**25**	26
19	26	**26**	**26**	**26**	27
20	27	**27**	**27**	N/A	28
21	28	**28**	**28**	**28**	29
22	29	**29**	**29**	**29**	30
23	30	**30**	**30**	**30**	31
24	31	**31**	**31**	**31**	**31**
25	32	33	N/A	N/A	33
26	33	**33**	**33**	**33**	**33**
27	34	**34**	**34**	**34**	**34**
28	35	**35**	**35**	**35**	**35**
29	36	**36**	**36**	**36**	**36**
30	37	38	**37**	N/A	38
31	38	**38**	**38**	**38**	**38**
32	39	**39**	N/A	N/A	40
33	40	**40**	**40**	**40**	**40**
34	40	**40**	**40**	N/A	**40**
35	40	**40**	**40**	N/A	**40**
36	40	**40**	**40**	N/A	**40**
37	60	**60**	N/A	N/A	61
38	60	**60**	**60**	N/A	61
39	60	**60**	**60**	N/A	61
40	90	91	N/A	N/A	100

Table 12 - ***Rotation***: the results of each instance for the four models (i.e. CP, SAT, SMT and MIP) with and without simmetry.

Regarding the rotation variant, *Table 12* shows that the first 17 instances are optimally solved by all four models, with instances 24, 26-29, 31, and 33 also being solved optimally. Instances 25 and 40 are the only instances that are unsolved by all four models, with the CP producing the closest solution to the optimal one. Instance 30 is more difficult to solve for the CP and SMT models, while instance 32 is challenging only for the SAT and SMT models. Finally, all instances from 34 to 39 cannot be solved by the SMT model, with instances 37-39 being particularly challenging for the MIP model, and instance 37 also being difficult for the SAT model. *Table 13* compares some statistics of the models:

	CP w/out SB + R	SAT + SB + R	SMT + SB + R	MIP + SB + R
Total Time [s]	2001.613	2433.407	3870.686	4870.363
Max [s]	300.0	300.0	300.0	300.0
Min [s]	0.163	0.134	0.047	0.002
Mean [s]	50.04	60.835	96.767	121.759
Std [s]	94.943	101.056	134.332	141.875
Instances Solved with H optimal	37/40	36/40	29/40	26/40

Table 13: some statistics between the four models with rotation.

For the non-rotation problem, the trend between the four models is the same of the previous variant, with only higher values of runtime and less instances solved for each approach.

In conclusion, the CP solver emerged as the best performer in both the rotation and non-rotation variants, taking the least amount of time. On the other hand, the MIP solver was the worst performer in both cases. The addition of *symmetry breaking constraints* significantly impacted the performance in the rotation variant (with the exception of CP), and less impact in the non-rotation variant. In addition, the rotation variant requires more times than the one without rotation, as we expected since its complexity.

References

[1] Nestor M. Cid-Garcia et al., “Exact solutions for the 2d-strip packing problem using the positions-and-covering methodology”

[2] Figure taken from *https://virtuale.unibo.it/pluginfile.php/1139783/mod_resource/content/3/PropagationGlobalConstraints.pdf*

[3] Peter J. Stuckey, Kim Marriott, Guido Tack, “The MiniZinc Handbook”, Predicates lex_lesseq, diffn and cumulative, https://www.minizinc.org/doc-2.6.4/en/predicates.html24

[4] Francesco Antici, https://github.com/francescoantici/CDMO-2021-2022-exercises

[5] Takehide Soh et al., “A SAT-based Method for Solving the Two-dimensional Strip Packing Problem”

[6] Roberto Amadini, https://virtuale.unibo.it/pluginfile.php/1230705/mod_resource/content/2/7-LP-ext.pdf, slide 24