VLSI-CDMO

Luca Reggiani, luca.reggiani6@studio.unibo.it 0001029787
Andrea Alfonsi, andrea.alfonsi2@studio.unibo.it, 0001030871

1 Introduction

VLSI, or Very Large Scale Integration, is all about fitting circuits onto tiny silicon chips. Think of your smartphone, a perfect example of this technology. Over time, we've shrunk transistors to pack more of them into the same silicon space. This technology has allowed the cell phone to transform from big devices to elegant, small instruments.
VLSI is similar to the 2D Strip Packing problem (2SP): you've got various rectangles and a fixed-width strip. The goal is to arrange those rectangles inside the strip so they do not overlap and, most importantly, to minimize the strip's height. It is classified as a NP-hard problem.
In this report, we delve into solving this strip packing puzzle using Combinatorial Optimization techniques. We explore four different methods:

- Constraint Programming (CP);
- Boolean SATisfiability (SAT);
- Satisfiability Modulo Theory (SMT);
- Mixed Integer (Linear) Programming (MIP);

In order to solve the problem proposed, some common approaches have been used.

1.1 Input Instances

An instance of the strip packing problem revolves around several key variables:

- The width of the plate;
- A collection of "n" rectangles, each represented with a specific width and height.

The primary objective is to determine the coordinates for the bottom-left corners of these rectangles in such a way that they don't overlap each other. Simultaneously, we aim to minimize the overall height of the strip.
For this problem, we work with integer values for both the dimensions of the rectangles and the plate itself. In essence, we're seeking a smart arrangement that optimizes space utilization while keeping things orderly.
We're confronted with a total of 40 diverse instances, categorized roughly by their varying levels of complexity. The exact details on how these instances are structured and what constitutes the desired outputs can be found in section 1.2 of the project assignment, providing clarity on the formatting requirements. The instances are contained in an folder called "input".

1.2 Input Parameters

In this project, we have implemented three essential input terminal parameters to enhance user control.

1. The folder_name parameter enables users to specify the input folder, with a default value of "input" for convenience.
2. The rotation parameter, when provided, allows for rectangle rotation to change the placement. Omitting it retains original orientations.
3. The simmetry_breaking parameter, when activated, applies symmetry-breaking constraints.

1.3 Preprocessing

In our exploration of all the techniques, we sorted the rectangles in a non-increasing order before giving them into the model. The central concept behind this approach is to take the larger rectangles first, with the hope of simplifying the placement of the remaining ones. In doing so, we aimed to enhance the efficiency of our solutions. In order to limit the execution time for instance solving, we set a timeout to 300 seconds.

1.4 Height lower and upper bound

To compute the lower bound, we calculated the sum of the areas for all the circuits, and subsequently divided this sum by the plate width. This operation provided an initial estimate of the optimum height for our packing problem. On the other hand, the upper bound was determined as the maximum possible height, which is equivalent to the sum of the heights of all the rectangles involved in the packing.
In the following formula, these the lower bound(lb) and the upper bound(ub) representation:

$$lb = \lfloor \frac{1}{PlateWidth} \sum_{i=1}^{N} w_i * h_i \rfloor$$

.

$$ub = \sum_{i=1}^{N} h_i$$

.
Where N is the number of circuits of a given instance, w_i and h_i are the width and the height of the circuit we are considering.

1.5 Constraints

In the whole project, in order to solve the problem proposed, we imposed some common constraints:

- **domain constraints:** every circuit should fit the plate (its width and its height);
- **non-overlapping constraints:** in order to avoid rectangles intersection.

1.6 Metrics

In assessing the effectiveness of each model, we employ a set of performance metrics, with priority given in the following order:

- **Number of Instances Solved:** This primary metric tells us how many problem instances were successfully solved by each model. The results are shown in a table.
- **Total Solving Time:** We measure the time spent by each model to solve every instance. This data is represented visually as a histogram.
- **Average Runtime:** We calculate it exclusively for instances that were successfully solved. However, it's important to note that this metric, while informative, may not provide a comprehensive evaluation, as it doesn't consider the number of instances solved. We include it mainly for illustrative purposes.

1.7 Output

The output is saved in the folder "out" in case of no rotation. Inside that folder, there are two subfolders: no_simmetry_breaking and simmetry_breaking. The same structure is kept in case of rotation. If an instance fails, or cannot be solved into the time limit set, we reported into the "failures.txt" file. Finally, all the timings required for solving instances are saved into a csv file, named timing.csv.

1.8 Additional Information

Our project took about a month of intensive work. Andrea Alfonsi took care of the 'CP' and 'MIP' approaches, while Luca Reggiani focused on the 'SAT' and 'SMT' methods. Together, we combined our skills to successfully handle the various aspects of the project.

2 CP Model

Constraint Programming (CP) consists in a declarative programming paradigm used to state and solve combinatorial optimization problems. In CP, the user has to model a decision problem (composed by a set of decision variables, their domains and the relations among them), while a solver will then find a satisfying solution by assigning a value to every variable using a search algorithm. CP is particularly powerful when dealing with complex constraints and combinatorial problems where brute-force search methods would be impractical, since it provides tools which make modelling easier and also allow control of the search strategies used by the solver.

2.1 Decision variables

In this section, the decision variables used in the CP models will be listed and explained. The ones introduced exclusively for the rotation variant of the problem will be explained in a later section. Other parameters, which are common to every approach, have already been explained in the introductory section (w, n, widths and heights).

- **X**: list corresponding to the horizontal position in the plate in which each circuit will be placed. The domain of this variable goes from 0 (y axis) to w-min(widths). This means that the upper bound is the width of the plate minus the smallest value among the widths of the circuits. This imply that if the upper bound was not set, some circuits may have overcome the width of the plate.
- **Y**: list corresponding to the vertical position in the plate in which each circuit will be placed. The domain of this variable goes from 0 (x axis) to h_max-min(heights). The reasoning is the same as before, the upper bound is set in order to not overcome the boundary of the plate, in this case the vertical one.

2.2 Objective function

As discussed in the first section, also here the objective is to minimize the *height* variable, which corresponds to the maximum value among the sums of the vertical position of each circuit with their heights. Again, the lower bound corresponds to the sum of the areas of all circuits divided by the width of the plate. On the other hand, the upper bound, which is more naive, corresponds to the sum of all the heights of the circuits.

2.3 Constraints

The first two constraints implemented are used to make it so that each circuit doesn't overcome the boundaries of the plate, both vertical and horizontal. They are encoded by enforcing that the max value of the horizontal/vertical coordinate of the circuit plus its width/height is smaller or equal than the width/height of the plate.

$$max([X[i] + widths[i]) <= w; \quad i \in \{1, ..., num\_circuits\}$$
$$max([Y[i] + heights[i]) <= h; \quad i \in \{1, ..., num\_circuits\}$$

Then, we added the **no overlap** constraint. MiniZinc allows the use of *global constraints*, which are special constraints representing high-level modelling abstractions, for which many solvers implement special, efficient inference algorithms. For the no overlap constraint we used the *diffn(x, y, dx, dy)* global constraint, which constrains rectangles i, given by their origins (x[i], y[i]) and sizes (dx[i], dy[i]), to be non-overlapping.

$$diffn(X, Y, widths, heights)$$

These previously described constraints are sufficient to model the problem, since they ensure that the boundaries of the plate are not overcome and also that the circuits do not overlap.

2.3.1 Implied Constraints

Implied constraints can be beneficial for optimization and for reducing the search space. If the solver deduces that certain combinations of variables are not valid based on the existing constraints, it can prune those possibilities from the search space, making the solving process more efficient. For this problem, we added two implied constraints using the *cumulative(s, d, r, b)* global constraint, which is usually used in resource usage problems, and which enforces that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.

$$cumulative(Y, heights, widths, w)$$

$$cumulative(X, widths, heights, h)$$

2.3.2 Symmetry breaking constraints

In order to try and reduce the set of solutions and the search space we created symmetry breaking variants of our model by adding **symmetry breaking constraints**. One common technique is to impose an order to avoid permutations which would correspond to symmetrically equivalent solutions. To apply this strategy we used the *lex_lesseq(x, y)* MiniZinc global constraint, which requires that the array x is lexicographically less than or equal to array y. We applied this global constraint for three different needs. Firstly, to get rid of symmetrical solutions generated by the switched positions of circuits with the same width and height.

$$forall(i, j\ in\ \{1..n\}\ where\ i < j\ and\ widths[i] == widths[j]\ and\ heights[i] == heights[j])$$
$$(lex\_lesseq([X[i], Y[i]], [X[j], Y[j]]))$$

Secondly, to force an order in order to avoid horizontally symmetric solutions, and thirdly for vertical symmetries.

$$lex\_lesseq([X[i] \mid i\ in\ \{1..n\}], [w - X[i] - widths[i] \mid i\ in\ \{1..n\}])$$

$$lex\_lesseq([Y[i] \mid i\ in\ \{1..n\}], [h - Y[i] - heights[i] \mid i\ in\ \{1..n\}])$$

2.4 Rotation

For the rotation version of the VLSI problem, we introduced further variables:

- **rotate**: list made of boolean elements, where each one is set to 1 if the corresponding circuit is rotated.
- **widths_upd**: list of the horizontal values of each circuit in the final representation. Each value will corresponds to the height if a circuit is rotated, otherwise it will be equal to the original width.
- **heigths_upd**: list of the vertical values of each circuit in the final representation. Each value will corresponds to the height if a circuit is rotated, otherwise it will be equal to the original width.

$$widths\_upd[i] = [if\ rotate[i]\ then\ heights[i]\ else\ widths[i]\ endif \mid i\ in\ \{1..n\}]$$
$$heights\_upd[i] = [if\ rotate[i]\ then\ widths[i]\ else\ heights[i]\ endif \mid i\ in\ \{1..n\}]$$

Furthermore, another constraint was added for the symmetry breaking variant, which was used to reduce the search space by locking the rotation to false for squared circuits (circuits with the same width and height).

$$forall(i\ in\ \{1..n\})(if\ widths[i] == heights[i]\ then\ rotate[i] = false\ endif);$$

Lastly, all the old constraints were upgraded by swapping the *width* and *height* parameters when used with the *width_upd* and *height_upd* decision variables.

2.5 Validation

The hardware in which these experiments were run was an Acer Aspire A315-23 laptop with an AMD Athlon Silver 3050U processor. The software used to design the models was MiniZinc and in order to automate the execution of all instances, a python script was written, allowing it to interact with MiniZinc via the minizinc python library. All the instances were run using two solvers (gecode and chuffed) and in the next table we will show the number of instances which the two solvers were able to optimize using all the four variants of the model (standard, standard with symmetry breaking, rotation, rotation with symmetry breaking).
For each instance we will show the optimal height found by each model or we will write N/A if no solution was found in the given time. We highlighted in bold, the values which are not optimal.
Multiple search strategies were tested, trying different annotations such as *input_order, first_fail, smallest*, and in the end the best results were found with the free search strategy and with the following search constructor:

$$\begin{aligned}
&solve :: seq\_search([\\
&int\_search([h], smallest, indomain\_min),\\
&int\_search(X, input\_order, indomain\_min),\\
&int\_search(Y, input\_order, indomain\_min),\\
&restart\_constant(3000)])\\
&minimize\ h;
\end{aligned}$$

The constant restart was crucial in order to solve some instances which would otherwise get stuck without a solution in the given time.

ID	No Rot		No Rot + SB		Rot		Rot + SB	
	gecode	chuffed	gecode	chuffed	gecode	chuffed	gecode	chuffed
1	8	8	8	8	8	8	8	8
2	9	9	9	9	9	9	9	9
3	10	10	10	10	10	10	10	10
4	11	11	11	11	11	11	11	11
5	12	12	12	12	12	12	12	12
6	13	13	13	13	13	13	13	13
7	14	14	14	14	**N/A**	14	**N/A**	14
8	15	15	15	15	**N/A**	15	**N/A**	15
9	16	16	**N/A**	16	**N/A**	16	**N/A**	16
10	17	17	**N/A**	17	**N/A**	17	**N/A**	17
11	**19**	18	**N/A**	18	**N/A**	18	**N/A**	18
12	19	19	**N/A**	19	**N/A**	19	**N/A**	19
13	**N/A**	20	**N/A**	20	**N/A**	20	**N/A**	20
14	**24**	21	**N/A**	21	**N/A**	21	**N/A**	21
15	**N/A**	22	**N/A**	22	**N/A**	22	**N/A**	22
16	**N/A**	23	**N/A**	23	**N/A**	23	**N/A**	23
17	**N/A**	24	**N/A**	24	**N/A**	24	**N/A**	24
18	**N/A**	25	**N/A**	25	**N/A**	25	**N/A**	25
19	**N/A**	26	**N/A**	26	**N/A**	26	**N/A**	26
20	**N/A**	27	**N/A**	27	**N/A**	27	**N/A**	27
21	**N/A**	28	**N/A**	28	**N/A**	28	**N/A**	28
22	**N/A**	29	**N/A**	29	**N/A**	29	**N/A**	**30**
23	**N/A**	30	**N/A**	30	**N/A**	30	**N/A**	30
24	**N/A**	31	**N/A**	31	**N/A**	31	**N/A**	31
25	**N/A**	32	**N/A**	32	**N/A**	**33**	**N/A**	**33**
26	**N/A**	33	**N/A**	33	**N/A**	33	**N/A**	33
27	**N/A**	34	**N/A**	34	**N/A**	34	**N/A**	34
28	**N/A**	35	**N/A**	35	**N/A**	35	**N/A**	35
29	**N/A**	36	**N/A**	36	**N/A**	36	**N/A**	36
30	**N/A**	37	**N/A**	37	**N/A**	**38**	**N/A**	**38**
31	**N/A**	38	**N/A**	38	**N/A**	38	**N/A**	38
32	**N/A**	39	**N/A**	39	**N/A**	**40**	**N/A**	**40**
33	**N/A**	40	**N/A**	40	**N/A**	40	**N/A**	40
34	**N/A**	40	**N/A**	40	**N/A**	40	**N/A**	40
35	**N/A**	40	**N/A**	40	**N/A**	40	**N/A**	40
36	**N/A**	40	**N/A**	40	**N/A**	40	**N/A**	40
37	**N/A**	60	**N/A**	60	**N/A**	**61**	**N/A**	**61**
38	**N/A**	60	**N/A**	60	**N/A**	60	**N/A**	**61**
39	**N/A**	60	**N/A**	60	**N/A**	**61**	**N/A**	**61**
40	**N/A**	**92**	**N/A**	**92**	**N/A**	**N/A**	**N/A**	**N/A**

As we can see, chuffed was able to reach far better results than gecode, being able to solve almost all instances. In particular, the no rotation version of the models was able to find the optimal height in 39 out of 40 instances, obtaining 92 instead of 90 for the last one. On the other hand, the rotation models were able to solve less instances, in particular 34/40 for the standard model and 32/40 for the symmetry-breaking version.
Because of the much better results of chuffed, we will now focus on it, exploring and displaying

the different times it took for the models to optimize each instance.

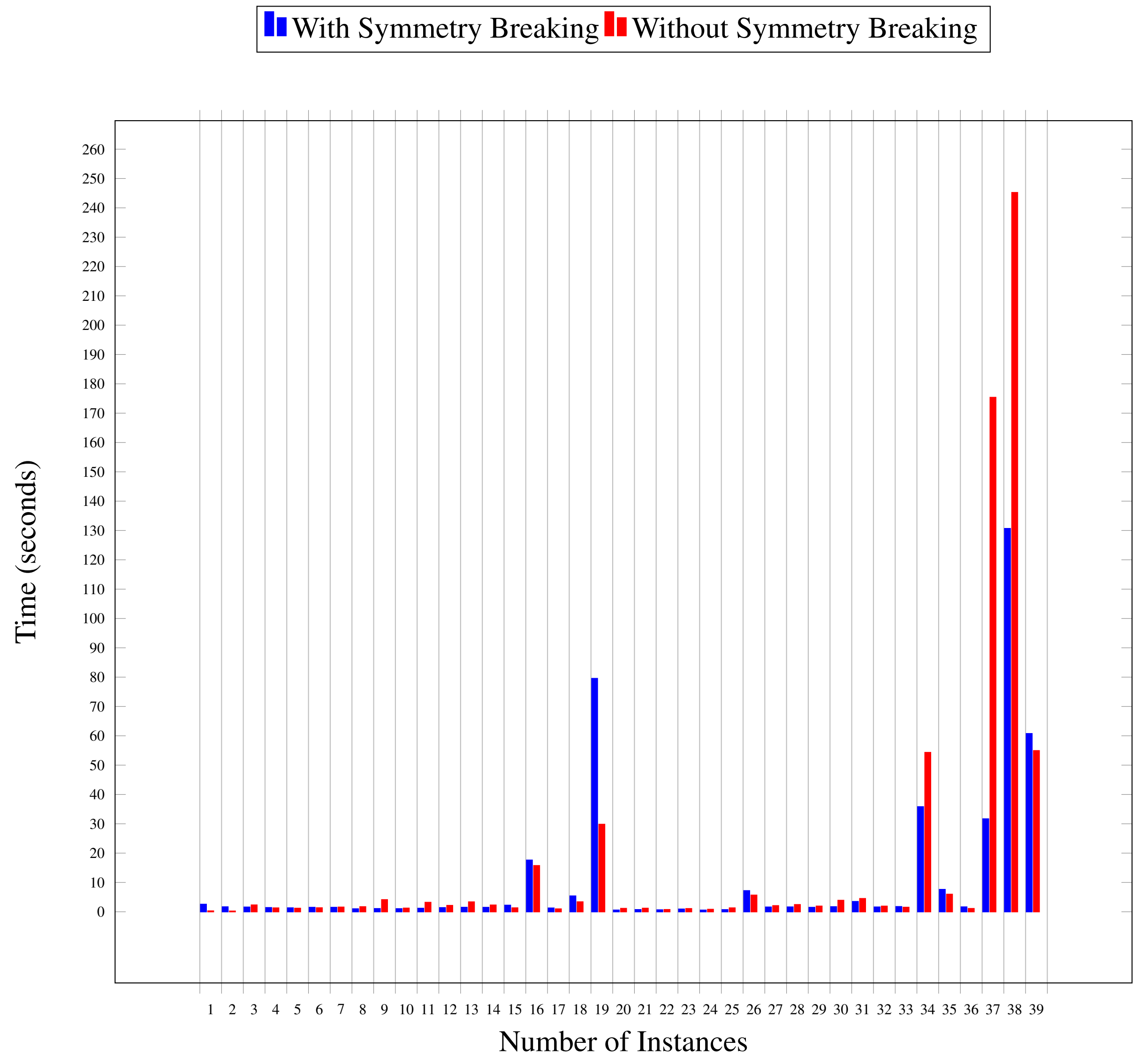

Figure 1: Results of CP without rotation. The plot displays solving times for each instance in both model configurations. Missing instances bar indicate timeouts.

As we can see from this graph, the symmetry breaking model is usually faster or close to the standard one, with the only exception being instance 19, where the symmetry breaking model is much worse. On average, the standard model takes 23.656364 seconds per instance, while the symmetry breaking model takes 18.016191 seconds. In total, the standard model takes 644.959898 seconds, while the more strict one takes 419.305748 seconds.

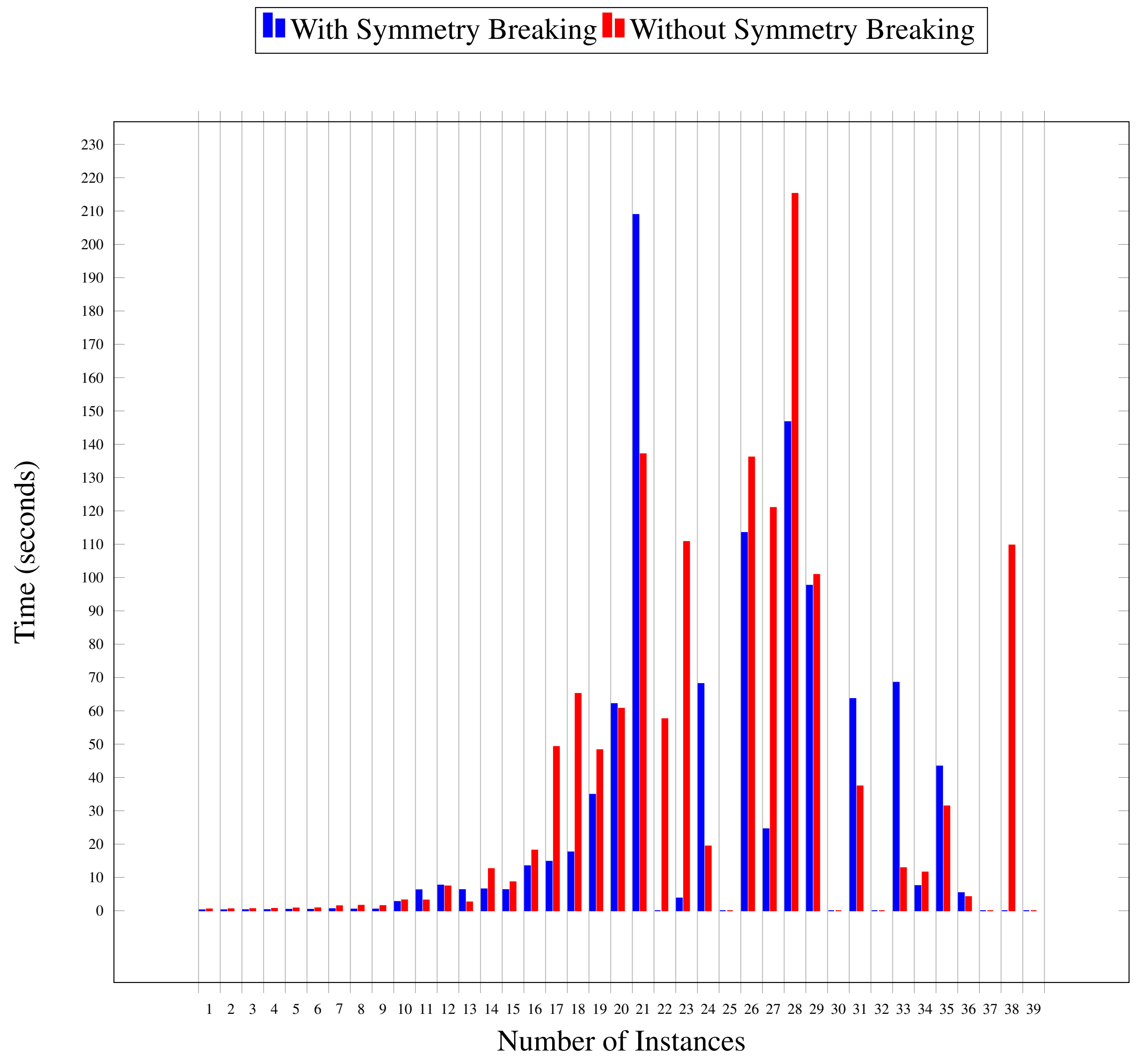

Figure 2: Results of CP with rotation. The plot displays solving times for each instance in both model configurations. Missing instances bar indicate timeouts.

Although not for all instances, symmetry breaking constraints seems to speed up the solution of most instances. In fact we have a mean execution time of 38.32 for the no symmetry breaking variant versus a 32.32 for the better variant (although we have to keep in mind the number of solved instances is lower this way). The total runtime for the symmetry breaking model is of 1034.4 seconds, against the 1226.34 needed for the standard version. To compute this values we only counted instances where both models were able to find the optimal solution.

3 SAT Model

In the context of Boolean logic, the SAT problem is a central topic focused on evaluating the feasibility of a set of propositional formulas. It manages the fundamental question of whether there exists an interpretation that satisfies all the provided formulas. In pursuit of this goal, we've exploited the capabilities of the Z3 theorem prover, integrated through its Python library [3]. In the context of VLSI (Very Large Scale Integration) domain, the primary aim is to

optimize the height of a layout while adhering to defined width and height constraints. It's imperative to recognize that SAT solvers excel at verifying the satisfiability of problems but do not inherently yield optimal solutions.
To overcome this limitation, an innovative strategy has been devised. This strategy involves decomposing the problem into a series of two-dimensional orthogonal packing problems (2OPPs) [2]. These 2OPPs are decision problems similar to the provided problem but come with a fixed height dimension specified. In our case, we must predefine both a maximum width and a maximum height, setting the stage for subsequent optimizations.
Within the broader context of translating the above **C**onstraint **S**atisfaction **P**roblem (CSP) into a SAT problem, a range of techniques has been explored. Notably, order encoding emerges as an exceptional choice due to its ability to provide an intuitively comprehensible representation of integer order relations. Consequently, order encoding has been thoughtfully selected as the encoding technique to address the complex VLSI problem, ensuring a clear and natural depiction of order relations within the SAT-based problem-solving framework.

3.1 Decision variables

In order to model the problem in SAT encoding, the following boolean variables were defined for each instance to solve:

- **x_positions**: List of Boolean z3 encoded variables, as long as the width of the plate of the current instance. Every literal px_i_w = True iff the circuit i is positioned before the width w.
- **y_positions**: List of Boolean z3 encoded variables, as long as the height of the plate of the current instance. Every literal py_i_h = True iff the circuit i is positioned before the height h.
- **rot_flags**: List of Boolean z3 encoded flags, as long as the number of circuits available. It is used in order to understand if a specific circuit is rotated or not. Every literal i_rotation = True iff the circuit i must be rotated. This is used only in case of rotation.
- **lr**: list of Boolean z3 encoded variables that indicate if two rectangles are positioned alongside (left-right). Every literal LeftRight_i_j = True iff the circuit i is placed on the left of the circuit j.
- **ud**: list of Boolean z3 encoded variables that indicate if two rectangles are positioned one under the other (up-down). Every literal DownUp_i_j = True iff the circuit i is placed up-down of the circuit j.

3.2 Objective function

In the domain of SAT, the concept of a conventional objective function, as commonly encountered in optimization problems, does not apply directly. Instead, when striving to determine an optimal solution, it becomes essential to establish a lower bound and an upper bound as described in subsection 1.4, in order to facilitate the process of finding the optimal height.
Considering these bounds, we started the task of optimizing the packing height. Commencing from the lower bound, we diligently sought to identify a solution that adhered to this height constraint. In cases where no satisfactory solution could be found within the time limit, regardless of whether the maximum height was reached or not, the instance was deemed to have failed.

This stringent time limit was enforced to ensure computational efficiency and avoid protracted search efforts. However, when a solution was successfully identified within the allocated time frame, the instance was considered solved.

3.3 Constraints

There have been several studies on translation methods that encode a CSP into a SAT problem, namely: direct encoding, log encoding, support encoding, order encoding, and log support encoding. Among them, order encoding has been chosen. Order encoding is a powerful technique used in constraint satisfaction problems (CSP) and optimization tasks to represent and manage the order relationships among integers or elements. It plays a key role in translating complex problems into a format that can be efficiently solved using SAT solvers. In order encoding, the objective is to express the specific order or sequence in which elements should be arranged or processed to satisfy a set of constraints or achieve optimization goals. This encoding method transforms these order-related constraints into Boolean variables and logical clauses that are comprehensible to SAT solvers. In the context of the Very Large Scale Integration (VLSI), order encoding can be applied to establish the sequence in which components or rectangles are placed, ensuring efficient space utilization while adhering to width and height constraints.
In the context of order encoding, a crucial two-step process is employed to effectively represent constraints involving integer variables and values. In the first step, the initial constraint undergoes a transformation into a set of simpler primitive comparisons. These primitive comparisons are expressed in the form of $x \leq c$, where x is the integer variable, and c is the integer value. In the subsequent step, these simplified primitive comparisons are further encoded into a set of Boolean variables, denoted as $p_{x,c}$. This encoding strategy allows for the conversion of intricate integer constraints into a format that can be efficiently processed by SAT solvers. It not only simplifies the representation of the problem but also ensures that the vital relationships between integer variables and values are preserved.

3.3.1 Main Constraints

There are three main constraints:

1. **Ordering Constraints**: For each circuit i, and integer e and f such that $0 \leq e < W - 1$ and $0 \leq f < H - 1$, we have the 2-literal axiom clauses due to order encoding:

$$\neg posx_{i,e} \vee posx_{i,e+1}, \qquad e \in \{0,..,W-1\}$$
$$\neg posy_{i,f} \vee posy_{i,f+1}, \qquad f \in \{0,..,H-1\}$$

   Where W is the plate width, and H is the height.

2. **Plate Boundaries Constraints**: To ensure that the placement of a circuit, denoted as i with a width w_i, remains within the confines of the given plate or container, it is imperative to limit the domain of possible positions. Specifically, we must enforce the constraint that the x coordinate of the rectangle, represented as x_i, does not exceed the maximum allowable value, which is given by $W - w_i$. This constraint serves a vital purpose; without it, the rightmost border of the rectangle could extend beyond the right edge of the plate, resulting in an infeasible placement.

   Similarly, the y coordinate of the rectangle, denoted as y_i, must also adhere to a similar constraint. It should not exceed the maximum allowable y coordinate, which is determined by the height of the plate.

In essence, these constraints act as safeguards to prevent the placement of rectangles from violating the boundaries of the plate, ensuring that they remain entirely within the allowed spatial region. This is a fundamental requirement to ensure the feasibility and validity of any placement solution in this context.

$$posx_{i,e}, \qquad e \in \{W - w_i, ..., W-1\}$$
$$posy_{i,f}, \qquad f \in \{H - h_i, ..., H-1\}$$

for each circuit i.

3. **non-overlapping constraints** Useful constraints to ensure that components or circuits do not occupy the same physical space, preventing intersection between circuits and ensuring proper functionality. we can distinguish two constraints under this wider category:

   - **4-literal clause constraint**: for each circuit cir_i and cir_j (i<j), we have the following 4-literal clauses;
   $$lr_{i,j} \vee lr_{j,i} \vee ud_{i,j}, \vee ud_{j,i}$$
   . The meaning of this clause is that circuit j must not be positioned inside circuit i, because they can be, according to these flags, one alongside the other or up-down.
   - **Circuits size constraints**: Merely employing one constraint isn't adequate as it doesn't consider the dimensions of rectangle j. Additional constraints are necessary. For instance, if we place rectangle i to the left of j, we must ensure non-overlapping placement, meaning that the sum of the x coordinate of i and its width w_i should be less than or equal to the x coordinate of j. The same logic applies to the y coordinate, and this condition also holds when the positions of rectangles i and j are swapped. This concept can be represented in the following way:
   $$lr_{i,j} \Rightarrow x_i + w_i \leq x_j$$
   $$lr_{j,i} \Rightarrow x_j + w_j \leq x_i$$
   $$ud_{i,j} \Rightarrow y_i + h_i \leq y_j$$
   $$lr_{i,j} \Rightarrow y_i + h_i \leq y_j$$
   As mentioned before, we need to encode these inequalities using order encoding. The non-overlapping constraints therefore become the following sets of clauses, that are put in disjunction; e.g. the disjunction set of clauses of the first implication is the following:
   $$\neg lr_{i,j} \vee \neg px_{j,e}, \qquad e \in \{0, .., w_i - 1\}$$
   $$\neg lr_{i,j} \vee posx_{i,e} \vee \neg px_{j,e+w_i}, \qquad e \in \{0, .., W - w_i - 1\}$$

In essence, these constraints guarantee that the rectangles are positioned in a manner that prevents overlap and respects their dimensions, fitting the plate's dimension.

3.3.2 Symmetry Breaking Constraints

Like in CP, Symmetry Breaking constraints are useful to improve model performance; By applying these constraints, we make the problem more manageable for solvers and improve their efficiency in finding optimal solutions.
We implemented 3 of the 4 symmetries described in [2]:

1. **Large Rectangles:** if we consider two circuits r_i and r_j that respect the condition w_i + w_j > W, where W is the plate width, we can not pack these rectangles in the horizontal direction. It is therefore always the case that $ud_{i,j}$ or $ud_{j,i}$ is true, therefore we can remove the non-overlapping constraints containing $lr_{i,j}$ and $lr_{j,i}$ by avoiding to add the following constraints from the list of constraints presented without symmetry breaking:

$$lr_{i,j} \vee lr_{j,i}$$

$$\neg lr_{i,j} \vee \neg px_{j,e}, \qquad e \in \{0,..,w_i - 1\}$$
$$\neg lr_{j,i} \vee \neg posx_{i,e}, \qquad e \in \{0,..,w_j - 1\}$$

$$\neg lr_{i,j} \vee posx_{i,e} \vee \neg px_{j,e+w_i}, \qquad e \in \{0,..,W - w_i - 1\}$$
$$\neg lr_{j,i} \vee px_{j,e} \vee \neg posx_{i,e+w_j}, \qquad e \in \{0,..,W - w_j - 1\}$$

The same can be done in the vertical direction.

2. **Same Rectangles:** For each pair of circuits r_i and r_j having identical dimensions (w_i, h_i) = (w_j, h_j), we have the opportunity to impose a fixed positional relationship between them, with rectangle r_i being positioned at the bottom-left corner of rectangle r_j. Consequently, we can avoid to insert the non-overlapping constraints here below:

$$lr_{j,i}$$

$$\neg lr_{j,i} \vee \neg posx_{i,e}, \qquad e \in \{0,..,w_j - 1\}$$

$$\neg lr_{j,i} \vee px_{j,e} \vee \neg posx_{i,e+w_j}, \qquad e \in \{0,..,W - w_j - 1\}$$

The same is done in the vertical direction.

3. **Domain Reduction:** In order to prune the search space, we can place the largest rectangle r_{max} on the bottom left part of the grid. The maximum rectangle could mean maximum width, maximum height, and maximum area. The maximum area has been chosen here. The domain-reducing constraints for r_{max} need to be modified accordingly:

$$px_{max,e}, \quad e \in \{\lfloor \frac{W - w_{max}}{2} \rfloor, ..., W - 1\}$$

$$py_{max,f}, \quad f \in \{\lfloor \frac{H - h_{max}}{2} \rfloor, ..., H - 1\}$$

In addition, any circuit denoted as r_i that satisfies the condition $w_i > \lfloor \frac{W - w_{max}}{2} \rfloor$ (meaning a rectangle wider than half the strip's width) must not be positioned to the left of r_{max}. As a result, it becomes necessary to adjust the non-overlapping constraints accordingly, by excluding the following from the constraint presented above:

$$lr_{i,max}$$

$$\neg lr_{i,max} \vee \neg px_{max,e}, \qquad e \in \{0,..,w_i - 1\}$$

$$\neg lr_{i,max} \vee posx_{i,e} \vee \neg px_{max,e+w_i}, \qquad e \in \{0,..,W - w_i - 1\}$$

Where max represents, as we mentioned above, the largest circuit in the instance. The same can be done in the vertical direction.

3.4 Rotation

Previously, we made an assumption that all rectangles had a fixed orientation. To expand our model to include the possibility of 90-degree rotation for each rectangle, we introduce a new binary variable, denoted as i_rotation, for each rectangle i. When i_rotation is True, it indicates that rectangle i has been rotated; when it's False, the rectangle remains in its original orientation. When a rectangle is rotated, its height and width effectively switch places. Integrating rotation into our model primarily involves updating constraints related to i_rotation and ¬i_rotation, as most constraints remain unchanged. Only the domain-reducing constraints and the non-overlapping constraints require modification.

The domain reduction constraints are adapted as follows:

$$i\_rotation \implies posx_{i,j} \qquad j \in \{W - h_i, ..., W - 1\}$$
$$i\_rotation \implies posy_{i,j} \qquad j \in \{H - w_i, ..., H - 1\}$$

$$\neg i\_rotation \implies posx_{i,j} \qquad j \in \{W - w_i, ..., W - 1\}$$
$$\neg i\_rotation \implies posy_{i,j} \qquad j \in \{H - h_i, ..., H - 1\}$$

When a circuit is rotated, its width and its height exchange the values. This change applies to non-overlapping constraints as well:

$$i\_rotation \implies \neg lr_{i,j} \vee \neg px_{j,e} \qquad e \in \{0, ..., h_i - 1\}$$
$$i\_rotation \implies \neg lr_{i,j} \vee px_{i,e} \vee \neg px_{j,e+h_i} \qquad e \in \{0, ..., W - h_i - 1\}$$

$$\neg i\_rotation \implies \neg lr_{i,j} \vee \neg px_{j,e} \qquad e \in \{0, ..., w_i - 1\}$$
$$\neg i\_rotation \implies \neg lr_{i,j} \vee px_{i,e} \vee \neg px_{j,e+w_i} \qquad e \in \{0, ..., W - w_i - 1\}$$

The same rules are true for the other variables $lr_{j,i}$, $ud_{i,j}$, and $ud_{j,i}$.

3.5 Validation

Our hardware setup for getting the following results consisted of an Asus N552VW - FY136T laptop powered by an Intel Core i7 - 6700HQ processor. To evaluate the effectiveness of our approach, we conducted a comparative analysis between the baseline model (as detailed in Section 3.3.1) and our model enhanced with symmetry-breaking constraints (outlined in Section 3.3.2). This evaluation was carried out both in scenarios with and without rotation, allowing for a comprehensive assessment of the impact of symmetry breaking on the model's performance. In our evaluation, we present a comprehensive analysis of four distinct cases: no rotation with symmetry breaking, no rotation without symmetry breaking, rotation with symmetry breaking, and rotation without symmetry breaking. For each case, we report the optimum height achieved.

ID	No Rotation + SB	No Rotation w/out SB	Rotation + SB	Rotation w/out SB
1	8	8	8	8
2	9	9	9	9
3	10	10	10	10
4	11	11	11	11
5	12	12	12	12
6	13	13	13	13
7	14	14	14	14
8	15	15	15	15
9	16	16	16	16
10	17	17	17	17
11	18	18	18	18
12	19	19	19	19
13	20	20	20	20
14	21	21	21	21
15	22	22	22	22
16	23	23	23	23
17	24	24	24	24
18	25	25	25	25
19	26	26	26	26
20	27	27	27	27
21	28	28	28	28
22	29	29	N/A	29
23	30	30	30	30
24	31	31	31	31
25	32	32	32	N/A
26	33	33	33	33
27	34	34	34	34
28	35	35	35	35
29	36	36	36	36
30	37	37	37	N/A
31	38	38	38	38
32	39	39	N/A	N/A
33	40	40	40	40
34	40	40	40	40
35	40	40	40	40
36	40	40	40	40
37	60	60	N/A	N/A
38	N/A	N/A	60	N/A
39	60	60	60	60
40	N/A	N/A	N/A	N/A

3.5.1 No Rotation

As we can see in the above table, instances 38 and 40 posed particularly challenging problems as both the baseline model and the model enhanced with symmetry breaking were unable to find solutions for these cases.
An interesting observation is the difference in runtime between the two configurations. In sce-

narios where symmetry breaking was employed, the average runtime was shorter, averaging around 14.57 seconds. Conversely, in instances without symmetry breaking, the average runtime extended to approximately 21.93 seconds. This contrast underscores the effectiveness of symmetry-breaking techniques in enhancing computational efficiency, particularly when dealing with complex problem instances.
To provide a more visual representation of our findings, we have encapsulated all the results in the comprehensive figure below.

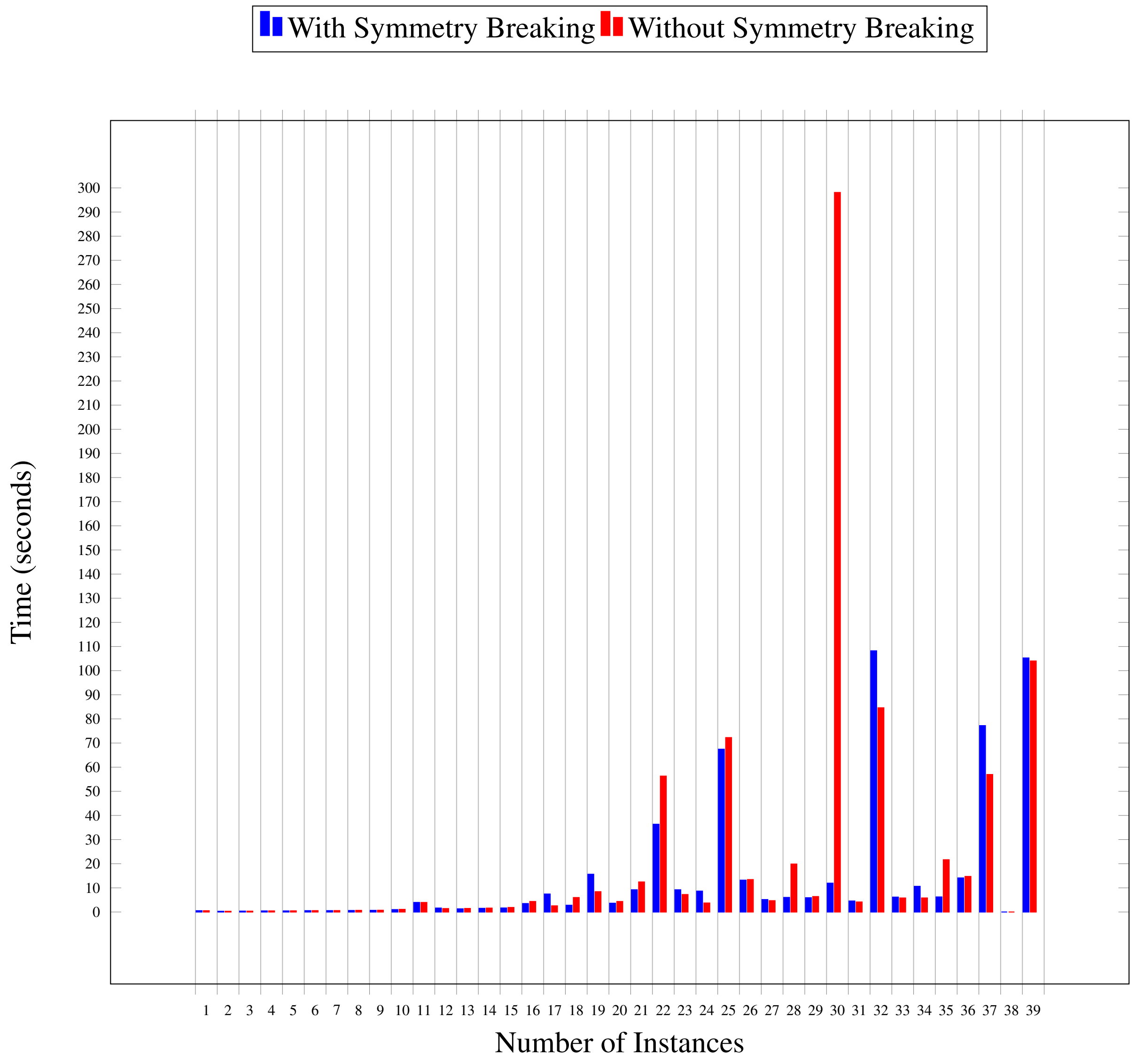

Figure 3: Results of SAT without rotation. The plot displays solving times for each instance in both model configurations. Missing instances bar indicates timeouts or failures.

3.5.2 Rotation

Instances 32, 37, and 40 were quite difficult, since both the models had no way to solve them. The baseline model solved 34 of 40 situations. 35 examples were solved by the model with symmetry-breaking.
The average runtime was reduced in instances where symmetry breaking was used, averaging roughly 31.25 seconds. In the absence of symmetry breaking, the average runtime increased to

around 38.05 seconds.

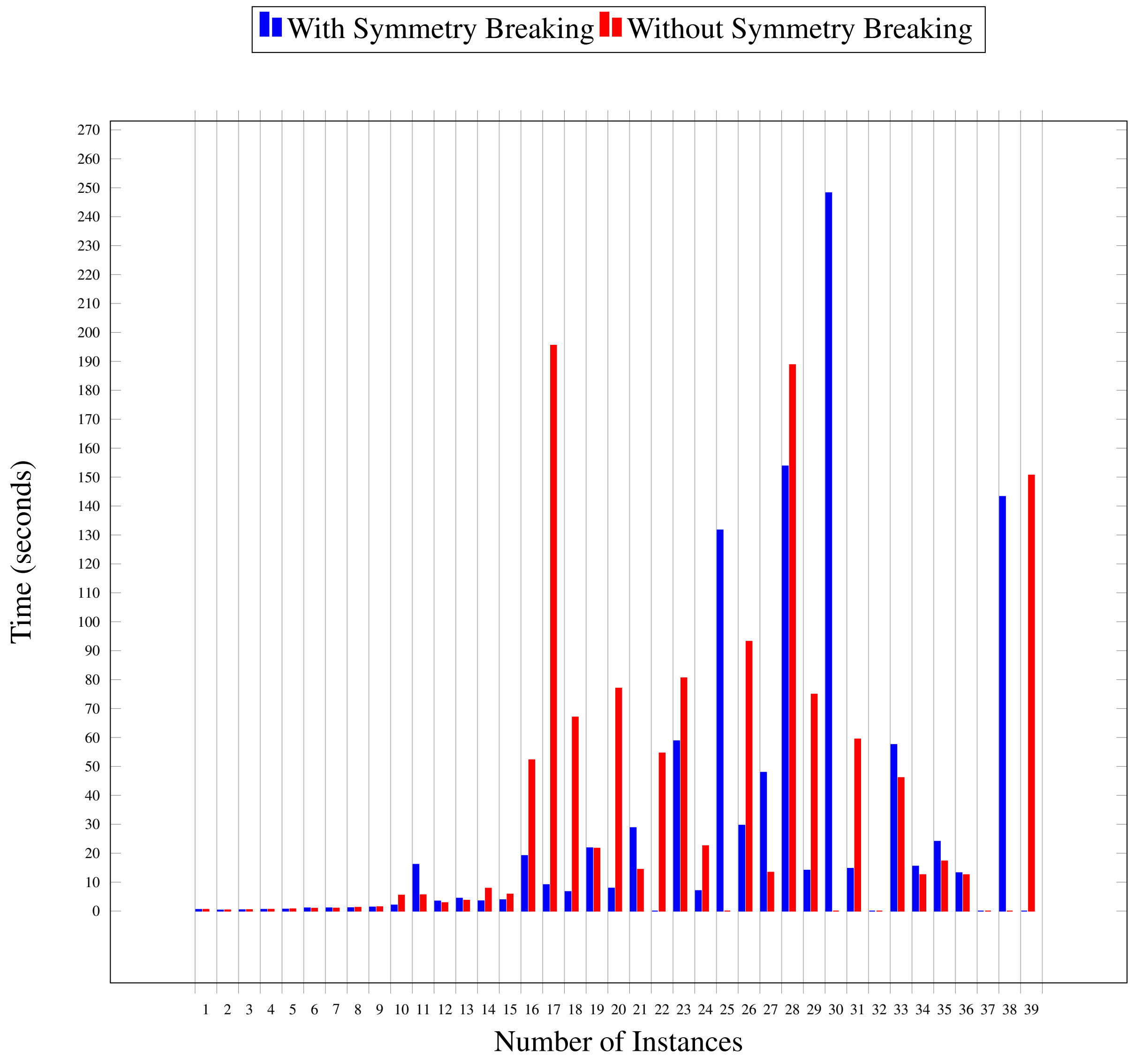

Figure 4: Results of SAT with rotation. The plot displays solving times for each instance in both model configurations. Missing instances bar indicates timeouts or failures.

4 SMT Model

Satisfiability Modulo Theory (SMT) extends the capabilities of the SAT problem by accommodating first-order logic formulas. This expansion grants it the power to represent a wider variety of elements, including real numbers, arrays, and more. Consequently, SMT offers a more versatile means of expression.
In our project, we took the decision to employ Z3 as our SMT solver (like in SAT). Specifically, we leveraged the Z3 Python API to interact with and utilize its robust solving capabilities.

4.1 Decision variables

The decision variables encoded in SMT are the following:

- circuit_i_X: Integer value representing the x coordinate of where the i-th circuit should be placed. $i \in \{1,...,num\_circuits\}$

- circuit_i_Y: Integer value representing the y coordinate of where the i-th circuit should be placed. $i \in \{1,...,num\_circuits\}$

- height: It defines the height of the entire board.

- i_w: Integer value representing the width of the circuit i. It is represented as an encoded variable only in case of rotation because if the circuit is rotated, width and height switch. $i \in \{1,...,num\_circuits\}$

- i_h: Integer value representing the height of the circuit i. It is represented as an encoded variable only in case of rotation, like before. $i \in \{1,...,num\_circuits\}$

Other variables used are rot_flags, lr, and ud, already explained in section 3.1.

4.2 Objective function

The z3 Optimize class provides several methods; among them, we chose to use the minimize() function. The aim of the given problem is to minimize the height. So, we encoded the height variable, and then we called the above method of Optimize class. In this way, by adding two constraints for reducing the height domain, we could find the best height possible with the restrictions imposed (described in the section below).

4.3 Constraints

4.3.1 Main Constraints

In SMT, the main constraints are a bit different, because we do not need anymore of ordering constraint, but we have a new important constraint:

- **Height Constraints:** as long as the height is an encoded variable, we need to reduce its domain. In detail, we impose that the height should be $\geq$ the lower bound, and $\leq$ the upper bound (see section 1.4 for lower bound / upper bound definition).

- **Plate Boundaries Constraints:** as in SAT, thanks to these constraints we can ensure that the circuits are not going to exceed the plate. In SMT, however, it is easier to impose it, because it is possible to handle integers. As a consequence, we can just write that the encoded variable circuit_i_X is greater greater or equal to 0, and lower or equal to the plate width. The same can be said about the height. In formula:

$$circuit\_i\_X \geq 0 \wedge circuit\_i\_X \leq Plate\_Width \qquad i \in \{1,...,num\_circuits\}$$

- **Non-Overlapping constraints:** the main goal of these constraints, as in SAT, is to avoid that two or more circuits overlap each other. Handling these constraints is quite easier than in SAT, considering we have integer encoded variable.

  - **4-literal clause constraints:** they are explained here 3

- **Circuits size constraints:** the main idea, is equal to SAT (here 3. The main difference, is that we do not need order encoding anymore; basically, given two circuits i and j, it is needed to check that the (x, y) coordinates of the rectangle i (integer encoded variables) summed respectively by the width and the height of i, are $\leq$ the (x,y) coordinates of the other rectangle j. Of course, it is also true the opposite. In formula:

  $$circuit\_i\_X + width\_i \leq circuit\_j\_X \quad i \in \{1,...,num\_circuits\} \;\; j \in \{1,...,num\_circuits\}$$

  $$circuit\_j\_X + width\_j \leq circuit\_i\_X \quad i \in \{1,...,num\_circuits\} \;\; j \in \{1,...,num\_circuits\}$$

  The same can be done with the height.

4.3.2 Simmetry Breaking Constraints

1. **Large Rectangles:** if two rectangles are wider then the plate widht(or height), then they cannot be placed one on the left (or top) of the other (and vice versa). In formula:

   $$width\_i + width\_j > plate\_Width \implies lr_{i,j} = False$$
   $$i \in \{1,...,num\_circuits\} \;\; j \in \{1,...,num\_circuits\}$$

   This is done also in the opposite case (rectangles j and i), and in case of height.

2. **Domain Reduction:** The basic idea is similar to the one exposed in SAT. The main difference is that in this case we can handle numeric integer variables, so with a simple equation we can check if a circuit can be placed before the widest rectangle, (up-down or left-right) depending on the numeric size of the rectangles itself.

4.4 Rotation

Based on the explanation provided in section 3.4, we need to apply a 90-degree orientation to rectangles. To achieve this, we have introduced two newly encoded variables, namely i_w and i_h. These variables allow us to apply similar constraints as mentioned earlier regarding the width and height of the rectangles, but with values that are not fixed. Additionally, we have defined an array of rotation flags, as outlined in the SAT part of our approach.
One significant difference is the inclusion of two additional constraints. The first constraint determines the values of the rotation flags based on the size of the circuit and the dimensions of the plate. Specifically, if a circuit's width exceeds the plate's width, the rotation flag for that circuit is set to True. Conversely, if the circuit's height is greater than to the plate's width, the rotation flag is set to False. In formula:

$$(i\_h > Plate\_Width) \implies \neg rotation\_flag\_i \qquad i \in \{1,...,num\_circuits\}$$

$$(i\_w > Plate\_Width) \implies rotation\_flag\_i \qquad i \in \{1,...,num\_circuits\}$$

The same principle is applied to the height of the circuits in relation to the plate's dimensions.
Another constraint applied, the most important one, is checking if, for each circuit, its rotation flag is set to True or False. If yes, it is rotated, and the width and the height of that circuit are swtched. Otherwise, they remain unchanged.

4.5 Validation

All the configuration and the metrics explained in section 3.5 are applied also in SMT. Here below, the table with all the optimum heights achieved:

ID	No Rotation + SB	No Rotation w/out SB	Rotation + SB	Rotation w/out SB
1	8	8	8	8
2	9	9	9	9
3	10	10	10	10
4	11	11	11	11
5	12	12	12	12
6	13	13	13	13
7	14	14	14	14
8	15	15	15	15
9	16	16	16	16
10	17	17	17	17
11	18	18	18	N/A
12	19	19	19	19
13	20	20	20	20
14	21	21	21	21
15	22	22	22	22
16	23	23	N/A	N/A
17	24	24	24	24
18	25	25	N/A	N/A
19	26	26	N/A	N/A
20	27	27	N/A	N/A
21	28	28	N/A	N/A
22	N/A	N/A	N/A	N/A
23	30	30	N/A	N/A
24	31	31	N/A	31
25	N/A	32	N/A	N/A
26	33	33	N/A	N/A
27	34	34	N/A	N/A
28	35	35	N/A	N/A
29	36	36	N/A	N/A
30	N/A	N/A	N/A	N/A
31	38	38	38	38
32	N/A	N/A	N/A	N/A
33	40	40	N/A	N/A
34	N/A	N/A	N/A	40
35	40	40	N/A	N/A
36	40	40	N/A	N/A
37	N/A	N/A	N/A	N/A
38	N/A	N/A	N/A	N/A
39	N/A	N/A	N/A	N/A
40	N/A	N/A	N/A	N/A

4.5.1 No Rotation

As we can see from the histogram below, a lot of instances failed, more than SAT. The reason is that here we have to consider more encoded variables, and an arbitrary objective function as well. This means that in order to find the right solution it requires much more time, and considering the time limit we set, it is normal so many instances fail. The average time here is 38.13 seconds without symmetry breaking and 26.12 seconds with symmetry breaking constraints.

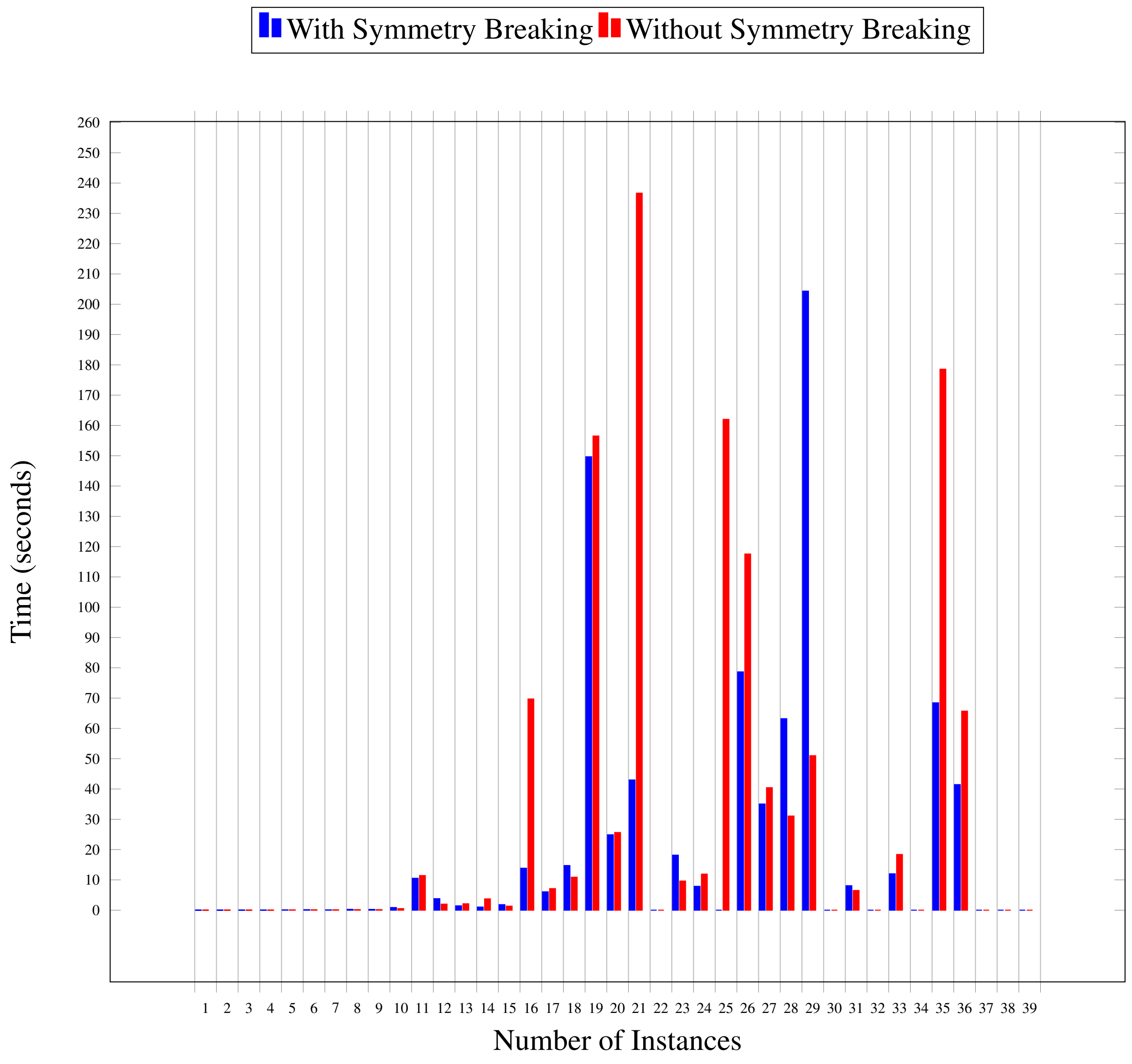

Figure 5: Results of SMT without rotation. The plot displays solving times for each instance in both model configurations. Missing instances bar indicates timeouts or failures.

4.5.2 Rotation

The average time here is 32.60 seconds without symmetry breaking and 60.1 seconds with symmetry breaking constraints.

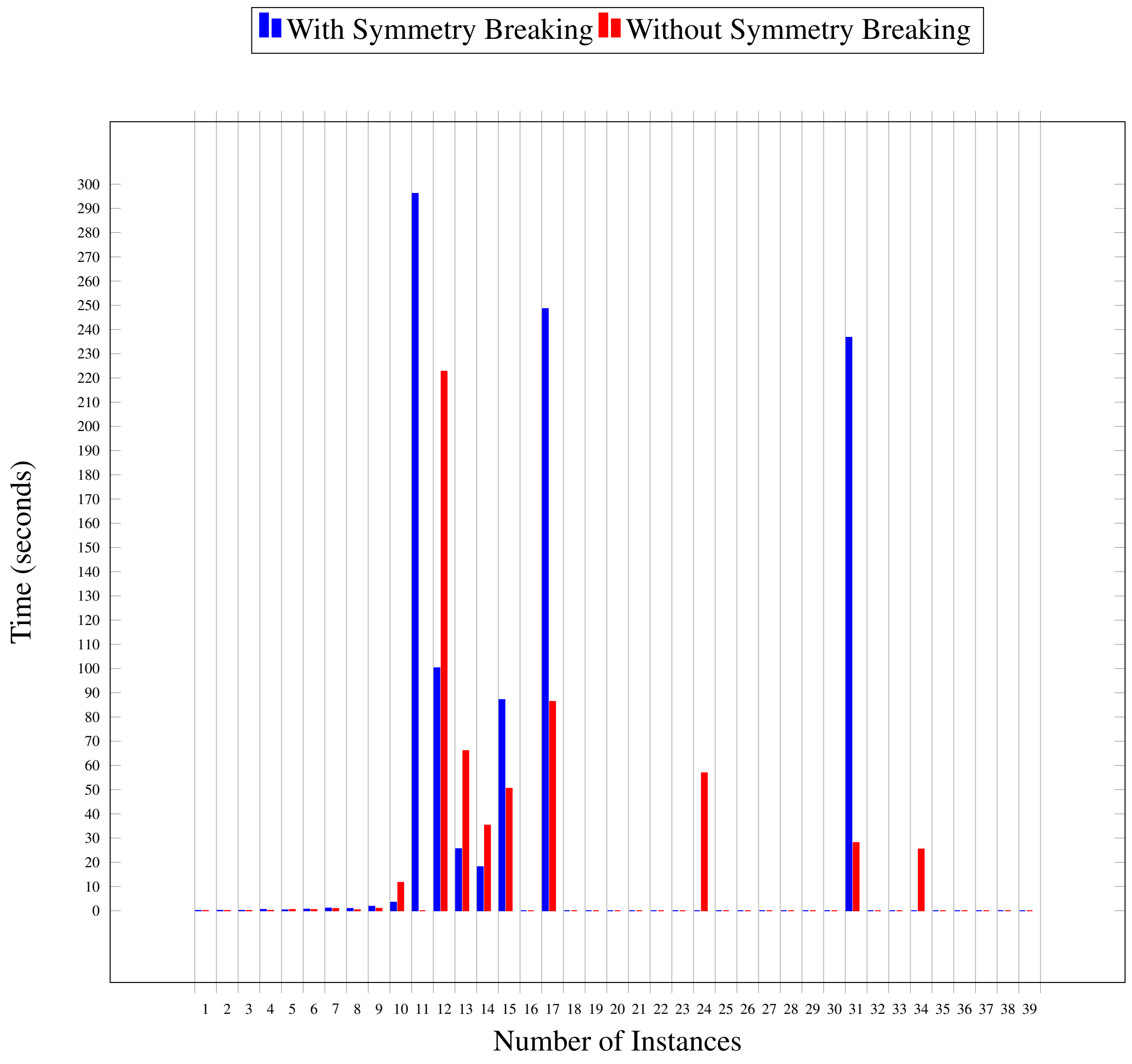

Figure 6: Results of SMT with rotation. The plot displays solving times for each instance in both model configurations. Missing instances bar indicates timeouts or failures.

Due to the bad performances, we tried to solve the SMT approach by fixing the height equal to the lower bound, and increasing it as we did in SAT. The results are much more better, we guess because the height is not an encoded variable anymore, so computationally it is cheaper. Check the "SMT_FH" folder in the project for results comparison.

5 MIP Model

Mixed Integer (Linear) Programming (MIP) is a mathematical optimization technique used to solve optimization problems in which the objective function and constraints are represented as linear equations or inequalities, and some or all of the decision variables are required to take integer values. It is particularly useful for addressing complex decision-making problems where discrete decisions are involved and its goal is to find the values of the decision variables that optimize (minimize or maximize) a linear objective function while satisfying a set of linear constraints.

5.1 Decision variables

The decision variables encoded in MIP are the following:

- X: array of variables containing the horizontal coordinate of each circuit. Their lower and upper bounds are the same as the ones discussed in the CP section.
- Y: array of variables containing the vertical coordinate of each circuit. Their lower and upper bounds are the same as the ones discussed in the CP section.
- b: matrix of binary elements, used for the Big-M trick applied in the no overlap constraint.

5.2 Objective function

As with the previous methods, the objective is still to minimize the *height* variable. The upper and lower bounds of its domain are also the same. Gurobi allows to do so by using the *model.setObjective()* function, which will take as arguments two elements: the variable we want to optimize (height) and the sense in which to optimize (in this case GRB.MINIMIZE for minimization).

5.3 Constraints

The main constraints used in MIP are the same as usual. We implemented two constraints to make it so that the circuits can't overcome the size limits of the plate (both height and width). And then we implemented the *no overlap* constraint. To implement it MIP we had to make use of the big-M trick[1]. In particular we knew that there are four cases to consider when dealing two circuits (or rectangles in general) i and j. Indicating by (x_i, y_i) the lower-left corner of a rectangle and indicating their height and width as (h_i, w_i) we can formulate the *no overlap* constraint as:

$$x_i + w_i <= x_j \ \ or$$

$$x_j + w_j <= x_i \ \ or$$

$$y_i + h_i <= y_j \ \ or$$

$$y_j + h_j <= y_i$$

In order to model the *or* condition (needed to make sure at least one constraint is active) we will use the previously described binary variable *b* and a big-M parameter.
We will have:

$$x_i + w_i <= x_j + M_1 * b_{i,j,1}$$

$$x_j + w_j <= x_i + M_2 * b_{i,j,2}$$

$$y_i + h_i <= y_j + M_3 * b_{i,j,3}$$

$$y_j + h_j <= y_i + M_4 * b_{i,j,4}$$

$$b_{i,j,k} <= 3$$

Since we don't want to compare a circuit with itself we looped i and j with j always greater than i. Regarding the M parameter, it's important to set it as small as possible, and in our case we set it as equal to the width of the plate for the two horizontal constraints and as the maximum height of the plate for the other two:

$$M_1 = M_2 = w$$

$$M_3 = M_4 = max\_h$$

5.3.1 Symmetry Breaking Constraints

For MIP, the symmetry breaking constraint used was the *biggest circuit* constraint, which, as described before, impose that the biggest circuit will have to be placed in the bottom-left corner of the plate.

5.4 Rotation

For the rotation variant of the problem, we introduced the same variables used and described in the CP section: rotate, widths_upd and heights_upd. The difference is that in this part, the values of *x_upd* and *y_upd* are computed in a different way. In CP a simple *if* statement was used, setting their values equal to the original one if the corresponding value in the *rotate* array was set to false, and vice versa. This time we added two constraints to reach the same purpose:

$$x\_upd[i] == rotate[i] * y[i] + (1 - rotate[i]) * x[i] \quad i \in \{1, ..., num\_circuits\}$$

$$y\_upd[i] == rotate[i] * x[i] + (1 - rotate[i]) * y[i] \quad i \in \{1, ..., num\_circuits\}$$

We also added the *square circuits* constraint which lock the rotation of a circuit to false if they have a squared shape.

5.5 Validation

The hardware in which these experiments were run was an Acer Aspire A315-23 laptop with an AMD Athlon Silver 3050U processor. The sover used was Gurobi in its interface with python (gurobipy). In the following table we displayed the number of instances solved by each variant of the models. In bold we highlighted the instances which were not optimized.

ID	No Rotation + SB	No Rotation w/out SB	Rotation + SB	Rotation w/out SB
1	8	8	8	8
2	9	9	9	9
3	10	10	10	10
4	11	11	11	11
5	12	12	12	12
6	13	13	13	13
7	14	14	14	14
8	15	15	15	15
9	16	16	16	16
10	17	17	17	17
11	18	18	18	18
12	19	19	19	19
13	20	20	20	20
14	21	21	21	21
15	22	22	22	22
16	23	23	23	23
17	24	24	24	24
18	25	25	25	25
19	**27**	26	**27**	**27**
20	27	27	**28**	**28**
21	28	28	**29**	**29**
22	**30**	**30**	**30**	**30**
23	30	30	**31**	30
24	31	31	31	31
25	**33**	**33**	**33**	**33**
26	**34**	**34**	33	**34**
27	34	34	34	**35**
28	35	35	**36**	**36**
29	36	36	**37**	36
30	**38**	**38**	**38**	**38**
31	38	38	38	38
32	**41**	**41**	**40**	**40**
33	40	40	40	40
34	40	**41**	**41**	**41**
35	**41**	**41**	**41**	40
36	40	**41**	40	40
37	**62**	**62**	**62**	**62**
38	**62**	**62**	**61**	**61**
39	**61**	**62**	**61**	**61**
40	**102**	**100**	**102**	**103**

As we can see, the standard model was able to solve 30/40 instances, the symmetry breaking variant 28/40, the rotation standard model 24/40 and the rotation with symmetry breaking model 25/50. In the following graph we'll dive deeper into the time it took for the no rotation models to solve each instances.

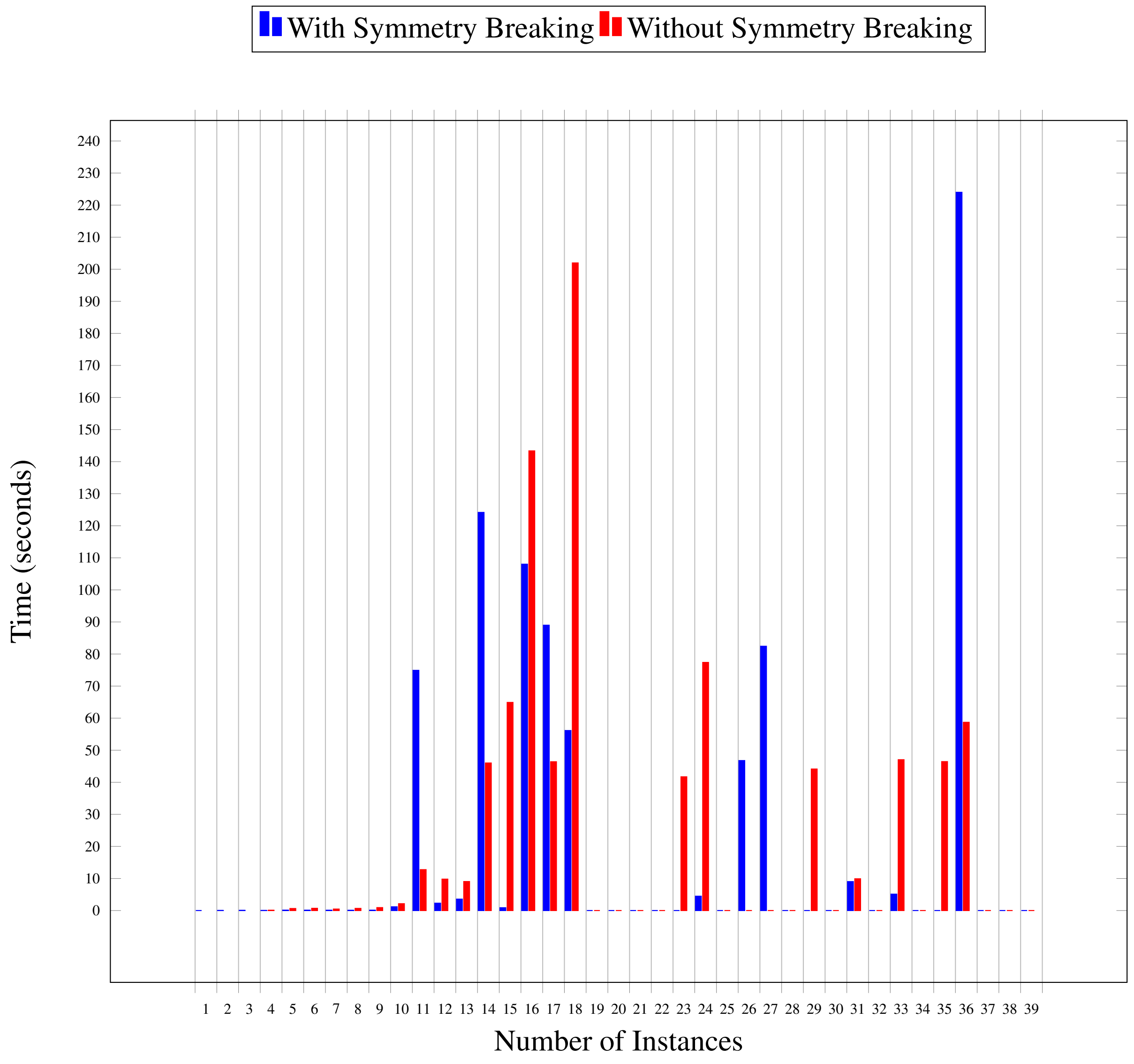

Figure 7: Results of MIP without rotation. The plot displays solving times for each instance in both model configurations. Missing instances bar indicate timeouts.

This time the runtime were closer, but in the end, the symmetry breaking model was faster. In fact, the total time needed to compute all the instances was 880,57 seconds for the standard model and 632,49 for the symmetry breaking variant. The mean values instead were 32,61 and 23,32. In the following graph we showed the time values for the rotation models.

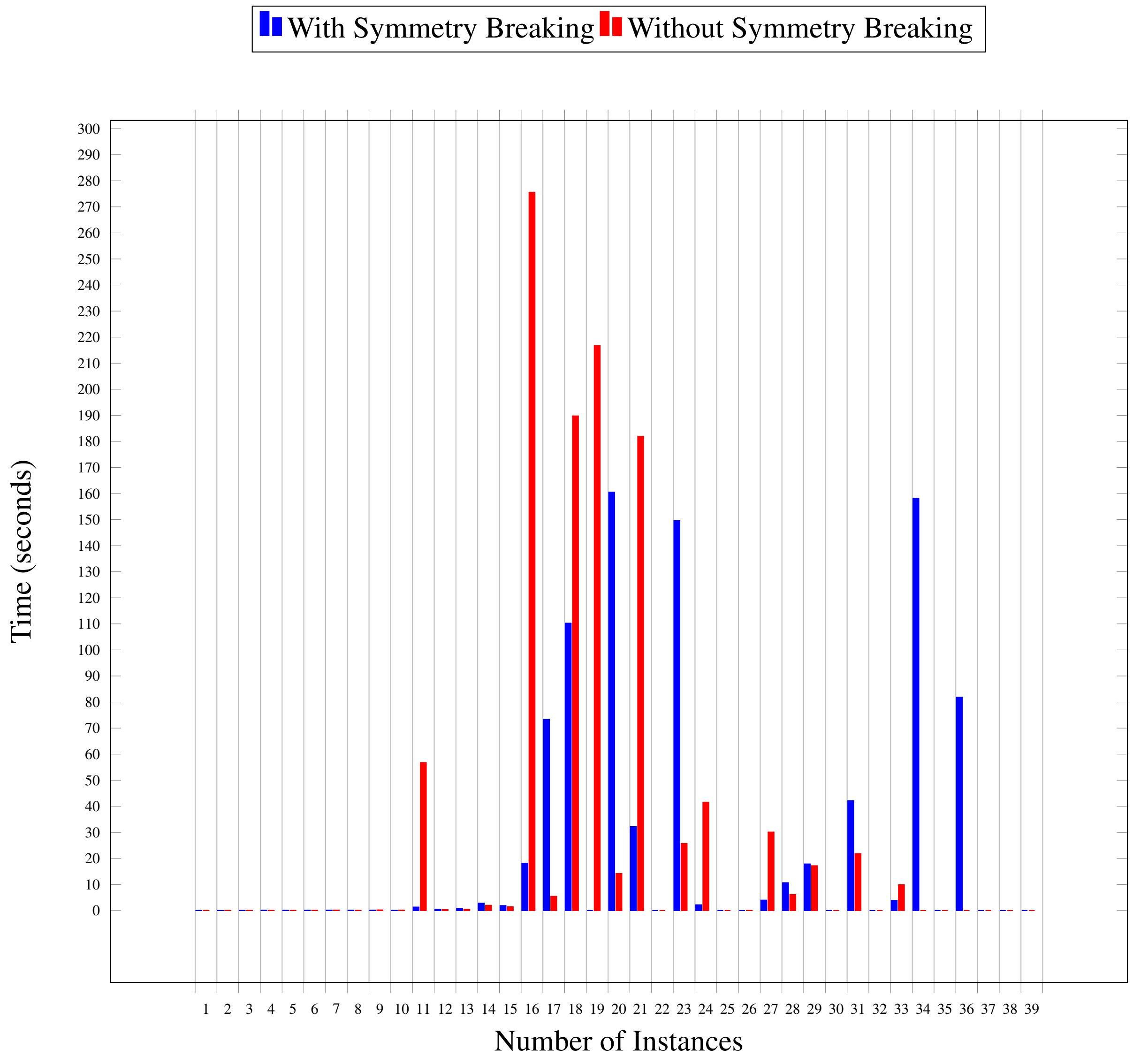

Figure 8: Results of MIP with rotation. The plot displays solving times for each instance in both model configurations. Missing instances bar indicate timeouts.

The total time needed to compute all the instances was 880,57 for the standard model and 632,49 for the symmetry breaking variant. The mean values instead were 32,61 and 23,32.

6 Conclusions

In conclusion, we discussed about the VLSI problem, tackling it using different methods. We first explained the common strategies, variables and constraints used in all methods, and then we talked about all the peculiar implementations used in each methods, while finally showing the results for each one.
The next table show for each instance, the best height reached for every method.

ID	CP	SAT	SMT	MIP
1	8	8	8	8
2	9	9	9	9
3	10	10	10	10
4	11	11	11	11
5	12	12	12	12
6	13	13	13	13
7	14	14	14	14
8	15	15	15	15
9	16	16	16	16
10	17	17	17	17
11	18	18	18	18
12	19	19	19	19
13	20	20	20	20
14	21	21	21	21
15	22	22	22	22
16	23	23	23	23
17	24	24	24	24
18	25	25	25	25
19	26	26	26	26
20	27	27	27	27
21	28	28	28	28
22	29	29	N/A	**30**
23	30	30	30	30
24	31	31	31	31
25	32	32	32	**33**
26	33	33	33	33
27	34	34	34	34
28	35	35	35	35
29	36	36	36	36
30	37	37	N/A	**38**
31	38	38	38	38
32	39	39	N/A	**40**
33	40	40	40	40
34	40	40	40	40
35	40	40	40	40
36	40	40	40	40
37	60	60	N/A	**62**
38	60	60	N/A	**62**
39	60	60	N/A	**61**
40	**92**	N/A	N/A	**100**

With CP we were able to find the optimal value for 39/40 instances, and in particular the models with this score were the ones without rotation.
For SAT, the only instance we were never able to optimize was the last one.
For SMT, we optimized 33/40 instances.
Finally, MIT models were not able to optimize 8 instances.

References

[1] *Rectangles no overlap constraint*. URL: `http://yetanothermathprogrammingconsultant.blogspot.com/2017/07/rectangles-no-overlap-constraints.html`.

[2] Takehide Soh et al. “A SAT-based Method for Solving the Two-dimensional Strip Packing Problem”. In: *Fundam. Inform.* 102 (Jan. 2010), pp. 467–487. DOI: `10.3233/FI-2010-314`.

[3] *z3-solver 4.12.2.0*. URL: `https://pypi.org/project/z3-solver/`.