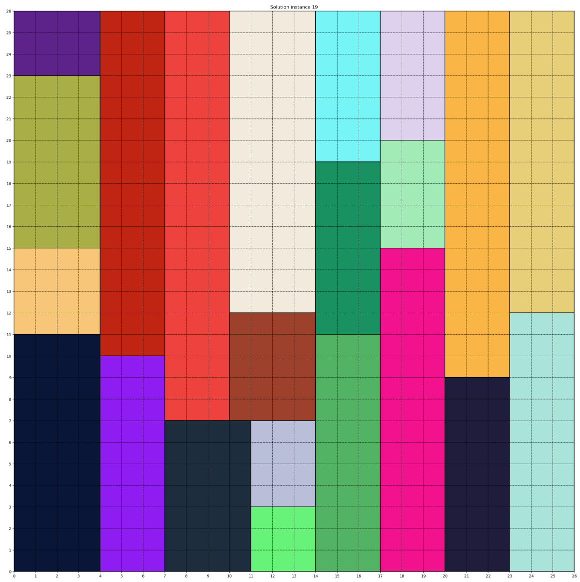
Combinatorial Decision Making and Optimization

VLSI problem

Nicola Carrassi, ID: 0001037813 (nicola.carrassi@studio.unibo.it)
Gabriele Colasuonno, ID: 0001037489 (gabriele.colasuonno@studio.unibo.it)
Antonio Guerra, ID: 0001037365 (antonio.guerra7@studio.unibo.it)



July 2022

Chapter 1

Introduction

The topic of this project is to design the VLSI (Very Large Scale Integration) of circuits into silicon chips. Given a plate with fixed width w and a list of n rectangular circuits of dimension $width_i$ and $height_i$, the objective of the problem is to place all the rectangles on the chip and minimize the height of the silicon chip.
According to Wikipedia definition, VLSI is the process of creating an integrated circuit of transistors into a single chip. A crucial step in the physical design of integrated chips is the so called **floor-planning**. Floor-planning is a method used to place all the modules of the circuit in such a way it meets some performance goals, which usually are related to the physical dimensions of the chip.
From a computational point of view, it is an NP hard problem and the search space increases exponentially with the increasing of blocks, therefore finding an optimal solution is a challenging task.
In this work, we will try to tackle the problem using different technologies such as:

- Constraint Programming;
- SATisfiability solving;
- Satisfiability Modulo Theory (SMT);
- Integer Linear Programming.

1.1 Format of the instances

An instance of VLSI is composed by integer values. The first line of the file containing the instance gives w, the width of the plate in which the chips have to be placed. The second line gives n which is the number of the rectangles and then there are n lines representing the width and height of each rectangle.
The output format contains: at the first line the width and the height of the chip, then at the second we have the number of blocks, as for the input instance. The other n lines have in order: the dimensions of the circuits followed by the horizontal and vertical coordinate of its bottom-left corner. An example is described by the following table:

Example Instance		
Input Parameter file	**Output file**	**Meaning of the line (for the output file)**
9	9 12	The plate has width 9 and height 12
5	5	We have 5 blocks
3 3	3 3 4 0	Block of dimension 3x3 with bottom-left corner at (4,0)
2 4	2 4 7 0	Block of dimension 2x4 with bottom-left corner at (7,0)
2 8	2 8 7 4	Block of dimension 2x8 with bottom-left corner at (7,4)
3 9	3 9 4 3	Block of dimension 3x9 with bottom-left corner at (4,3)
4 12	4 12 0 0	Block of dimension 4x12 with bottom-left corner at (0,0)

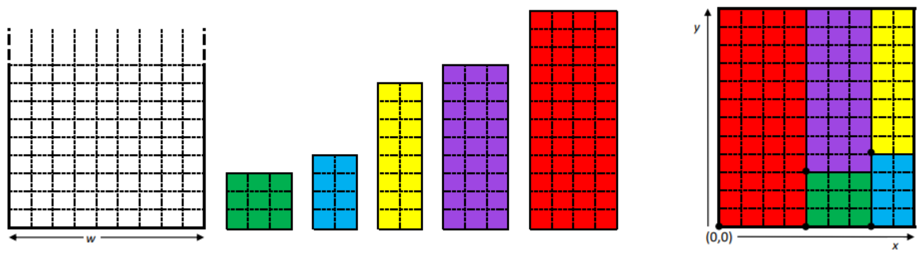


Figure 1.1: Graphical representation of a VLSI instance with its solution

1.2 Preliminaries

1.2.1 Basic formulation of the problem

The VLSI problem can be formulated in the following way. We have as input parameters:

- $w \in \mathbb{N},\ w \geq 0$ which represents the width of the plate;
- $n \in \mathbb{N},\ n \geq 0$ representing the number of the chips;
- $widths,\ heights \in \mathbb{N}$, where $widths_i, heights_i \geq 0$ representing respectively, the width and the height of the i-th chip.

In order to have all the information about the position that each rectangle has on the plate, we defined two variables for each block: x_i and y_i which refer to the position on the x and y axis of the bottom-left corner of the i-th rectangle.
The most obvious constraints needed to solve this problem are given by the fact that the rectangles must be inside the chip and that they cannot overlap. We can formalize the first constraint in the following way:

$$\forall i \in [1, n]:\ x_i + widths_i \leq w \tag{1.1}$$

The non-overlapping constraint instead can be expressed as follows:

$$\begin{aligned}\forall i, j \in [1, n]\ s.t.\ i \neq j:\ & x_i + widths_i \leq x_j \lor \\ & x_j + widths_j \leq x_i \lor \\ & y_i + heights_i \leq y_j \lor \\ & y_j + heights_j \leq y_i\end{aligned} \tag{1.2}$$

The objective is to minimize the height of the silicon chip, namely h, we do so finding the smallest value of h which satisfies:

$$h \geq y_i + heights_i,\ \forall\ i \in [1..n] \tag{1.3}$$

1.2.2 Common elements in all formulations

Upper and Lower bound In order to reduce the search space, and thus simplify the search process, we defined boundaries both for the x_i and for the total height of the chip h. First of all we will explain the lower and upper bound for our objective variable h.
The lower bound was firstly defined as the smallest $h \in \mathbb{N}$ which is greater or equal than total area of the blocks divided by the width of the plate, in formula:

$$\left\lceil \frac{\sum_{i=1}^{n} widths_i \cdot heights_i}{w} \right\rceil$$

We then realized that there are some situations in which we might get a better lower bound if we consider it equal to the height of the highest block. For instance, if we suppose that we have the following situation:

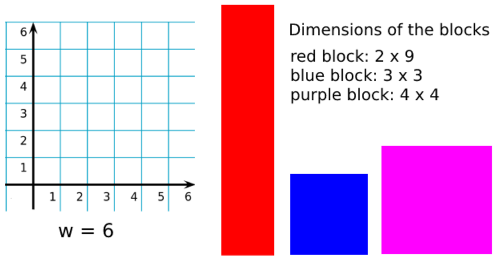


Figure 1.2: Example in which $\max(heights)$ is a better lower bound

In this case, using the definition of lower bound based on the sum of the areas of the rectangles, we would have a value of 8, instead using the highest block we would obtain 9 as lower bound.
We decided to keep both of the given definitions and use as lower bound the biggest among them:

$$min_h = \max\left(\left\lceil \frac{\sum_{i=1}^{n} widths_i \cdot heights_i}{w} \right\rceil, \ \max(heights)\right) \tag{1.4}$$

Initially, we defined the upper bound as the sum of the heights of all blocks:

$$max_h = \sum_{i=1}^{n} heights_i$$

We quickly realized that when the number of block grows this bound is too wide and the resulting search space still very huge. So we decided to change approach using the following one.
Firstly we divided all the rectangles in subsets SW_i with the following properties:

$$SW_i \subseteq widths \ s.t \ (\sum_{dim \in \ SW_i} dim) \leq w$$
$$\forall i, j : SW_i \cap SW_j = \emptyset$$

Then, for each subset SW_i, we denote with h_i the maximum height among the blocks which belong to the subset, so we imposed the following bound:

$$max_h = \sum h_i \tag{1.5}$$

After the definition of the bounds for the object value we also bounded the domain of the possible values which can be taken from the variables x_i and y_i.
For the array of x coordinates of the blocks we imposed that each block has to be placed in a range between 0 (which is the left border of the plate) and $w - min(widths)$, because we know that:

$$\forall i : x_i + widths_i \leq w \iff x_i \leq w - widths_i$$

The maximum value which can be assigned to x_i is exactly equal to $w - min(widths)$ and this is possible only if we are referring to the block with the smallest width. Otherwise we would get out of the chip and we would break a constraint. In the same way we bounded the domains of the y_i, obviously considering the heights instead of the widths and max_h instead of w.

Ordering of the blocks In order to simplify the search of the optimal solution, we decided to order, and consequently place, the blocks from the one with the biggest area to the smallest one. We did this because the biggest blocks are the ones which are more constrained, and placing the biggest block can be considered the most constrained sub-problem in each stage of the search. It is obvious that placing a bigger block is more complex than placing a small one and thus we decided to solve before the hardest sub-problems and then solve the easiest ones in a more constrained situation.

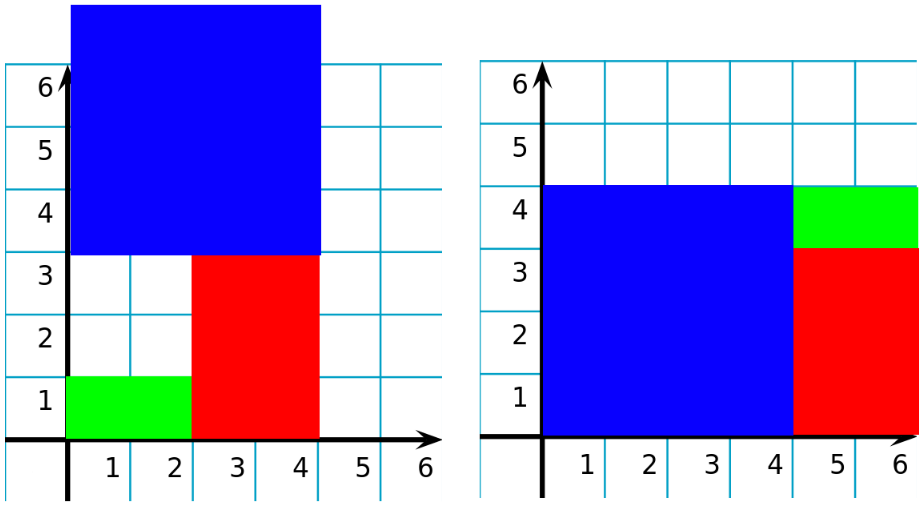


Figure 1.3: The image on the left represents a possible scenario in which the blocks are not placed in order of area, on the right there are the same blocks placed ordered by area.

Chapter 2

Constraint Programming

Constraint Programming (CP) is a declarative paradigm for solving combinatorial problems. The basic idea in CP is that the user states the constraints of the problem and then a general purpose solver is used to find a solution to the problem.
CP problems can require to find just a solution which satisfies all the constraints, in that case we refer to a **Constraint Satisfaction Problem (CSP)** or, like in our case, the problem requires to find a solution which maximizes (or minimizes) a function, we are asking to the solver to solve an optimization problem. The second class of problems is called **Constraint Optimization Problems (COP)** and our problem is of this kind, since we need to find a value which minimizes the height of the chip, satisfying all the constraints.

Encoding

The first model we realized for the problem actually was the encoding of the basic formulation of the problem. To improve its performances we introduced some modifications which now we will discuss:

2.1 Global constraints

It is known that constraint propagation plays a very important role in the process of solving complex problems. Constraint propagation is a technique which consists in removing inconsistent values from the domain of the variables. Constraining the problem allows to reduce the search tree and find the optimal solution in less time. Using global constraints is a technique which allows to efficiently cut the search tree, due to the specialized algorithms they embed, which can be more efficient than a generalized propagation. For this reason we removed the no-overlapping constraint we defined and we changed it with the procedure `diffn`. This constraint, defined in the language allows to handle the complexity of the no-overlapping constraint in a more efficient way.
Literature search helped us in noticing that we can think of the positioning of the rectangles as 2 scheduling problems, one for each axis, as proposed in [1]. For each axis we have n blocks to be placed, each of them uses a given amount of resources (in our case the space on the chip), some of the blocks can have the same value for one of the coordinates (for example when we place one above the other, they can have the same x value) but never exceeding a limit.
Thanks to this intuition found in the literature we used, for both the x and y axis, the constraint `cumulative` which is formalized as follow:

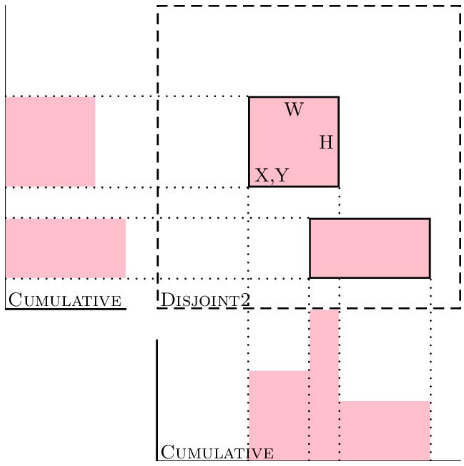


Figure 2.1: Image taken from [1]

$$\forall\ u\ \in\ widths: \sum_{i\ |\ x_i \leq u \leq x_i + widths_i} heights_i\ \leq h$$
$$\forall\ u\ \in\ heights: \sum_{i\ |\ y_i \leq u \leq y_i + heights_i} widths_i \leq w$$

2.2 Symmetry breaking constraints

In constraint optimization problems, symmetries can cause considerable difficulties for exact solver. One way to overcome this problem is the encoding of tailored constraints which aim to solve this problem, simplifying the search. It is important to remark that, when encoding these constraints we need to ensure that we always have at least a solution, namely sol_{sb} such that:

$$sol_{sb} = sol$$

where *sol* is obviously the optimal solution of the problem.
For our problem we found several symmetries which can be broken and in this section we will briefly present them.

2.2.1 Same dimensions blocks

If we consider two rectangles i and j which have the following properties:

$$widths_i = widths_j$$
$$heights_i = heights_j$$

Then we can safely say that we can place the the rectangle i whenever rectangle j is placed, and also the opposite holds. In this way for each optimal solution we have another one in which we just swap the two blocks.
This is handled imposing an ordering among the blocks, in particular we imposed that a rectangle has to be placed closer to the bottom-left corner of the plate (which has coordinates (0,0), we will refer to it also as origin of the coordinates). To express this constraint we considered the Manhattan distance of the blocks with respect to the origin. This can be expressed as:

$$\forall\ i, j\ s.t.\ i < j:\ (widths_i = widths_j \wedge heights_i = heights_j) \implies (x_i + y_i) < (x_j + y_j) \tag{2.1}$$

2.2.2 Fixing the position of a pair of rectangles

To remove more symmetries we considered only the solutions in which the relative position of the first pair of blocks is fixed. This is done because given a solution we can generate another one by flipping the board along one of the axis. This is achieved once again using the Manhattan distance of the blocks with respect to the origin of coordinates:

$$x_1 + y_1 < x_2 + y_2 \tag{2.2}$$

2.2.3 Fixed position of the biggest block

Other possible symmetries are given moving the block with the biggest area in different sections of the plate. If we impose that the block with the biggest area is placed in the bottom-left section of the plate we can effectively remove some of the solutions:

$$x_{big} \leq \frac{w}{2} \ \wedge \ y_{big} \leq \frac{h}{2}$$

$$where\ big = arg \max_i(widths_i \cdot heights_i) \tag{2.3}$$

2.2.4 Blocks with same horizontal coordinate and same widths

If we consider two blocks whose bottom left corner has the same horizontal coordinate and which have the same width, we can identify a symmetric solution where the two blocks are just swapped on the vertical axis. To broke this symmetry we imposed an ordering between the two vertical coordinates.

$$\forall\ i, j\ s.t\ i < j :\ (x_i = x_j \ \wedge\ (widths_i = widths_j)) \implies y_i < y_j \tag{2.4}$$

2.2.5 Blocks with same vertical coordinate and same heights

The same relationship can obviously be expressed also for blocks which have the same vertical coordinate and the same height.

$$\forall\ i,\ j\ s.t.\ i < j :\ (y_i = y_j \ \wedge\ (heights_i = heights_j)) \implies x_i < x_j \tag{2.5}$$

2.2.6 Three blocks constraint

In this case we considered a possible situation in which 3 blocks are in relationship, namely i, j, and k. If:

- The horizontal coordinate of two blocks are equal;
- The widths of those two blocks are equal;
- The vertical coordinate of one of the first two blocks is the same of the third one
- The height of the third block is equal to the sum of the heights of the first two blocks

Then we can impose an ordering among those two blocks.

$$\forall\ i, j, k\ s.t\ i < j < k :$$
$$\Big(x_i = x_j \ \wedge widths_i = widths_j \ \wedge\ y_i = y_k \wedge + heights_j = heights_k\Big) \implies x_k < x_i \tag{2.6}$$

All the constraints proposed have been tested with all the solvers used, and for each solver we produced the combination of them which had the best performance with our test cases.

2.3 Search

Search annotations in Minizinc are used to specify how to search, in order to find a solution to the problem. To better guide the solver through the search we tried to find out which is the best combination of the heuristics that allowed to have better results, in terms of solved instances and time necessary to find an optimal solution.
We run some tests for the objective variable and both the vertical and horizontal coordinates of the rectangles.
For the objective variable h we decided to use as heuristic `indomain_median`. This heuristic tries to assign to h the median domain value. This heuristic allowed us to find a feasible solution in the majority of the cases and reducing the search space in this way. For this variable one might think that choosing the first value would be a better choice but this actually might lead to problems, especially in case the lower bound is not a feasible solution. Our experimental result actually helped

us decide, because only using `indomain_median` we managed to solve all the instances between 10 and 20[1].
For the vertical coordinate we decided to try to place first the blocks with the `smallest` domain and try to give them the minimum value possible. In this way we are asking our solver to place first the highest blocks and then to place all the others, trying to place them towards the bottom of the plate if possible.
For the horizontal coordinate we also decided to place first the most constrained values, assigning them towards the left part first. The heuristics chosen for x are: `first_fail` and `indomain_min`.
In the end the search strategy used is the following one:

$$\begin{aligned}&\textit{seq\_search}([\\&\qquad \textit{int\_search}(x,\ first\_fail,\ \textit{indomain\_min}),\\&\qquad \textit{int\_search}([h],\ \textit{input\_order},\ \textit{indomain\_median}),\\&\qquad \textit{int\_search}(y,\ \textit{smallest},\ \textit{indomain\_min})\\&\qquad])\end{aligned} \tag{2.7}$$

2.3.1 Restarting

A problem that any kind of depth first search might suffer is making a wrong decision at the top of the search tree. This can lead to a huge amount of time to undo the mistake. This problem can be avoided introducing some restarting technique to the search procedure.
Using a restarting technique allows to avoid being stuck in a wrong path and introduces some randomness in the search process.
We decided to use luby restart to avoid getting stuck, especially because with a big number of blocks we generate a search tree which has a high depth.

2.3.2 LNS: Large Neighborhood Search

Large Neighborhood Search (LNS) is a combinatorial optimization heuristic that starts with an assignment of values for the variables to be optimized, and iteratively improves it by searching a large neighborhood around the current assignment. The heuristic is based on the idea that, given a solution, we can find a better one changing only some of the variables which contribute to it, executing a local search.
We introduced LNS using its search annotation in the code, removing half of the values of the vertical coordinates of the blocks.

2.4 Solvers

To solve the instances of the problem we decided to test our model with 3 different solvers: **Gecode** [2], **Chuffed** and **OR-Tools** [3].
We used each solver to solve the instances without any use of symmetry breaking constraint and then we tried to find the best possible set of symmetry breaking constraint. This was made testing for each solver firstly the set containing all the symmetry breaking constraint, then we started removing a constraints and made the tests once again.
Based on the number of correct instances we decided whether to keep or remove the constraint which we did not use, if the number of failure was less than the one with all symmetry breaking constraints we removed the constraint, otherwise we kept using it. We repeated the experiment with all the constraints and with all the solvers until we finally reached our configurations.

2.4.1 Gecode

Gecode, as mentioned in its website is an open source state of the art solver. It is one of the solvers provided when installing Minizinc. One of the greatest advantages of Gecode is the native support to many of Minizinc's global constraints.
The tests executed[2] with Gecode made us keep the following list of symmetry breaking constraints:

- Pair constraint;

[1] All the tests for the search heuristics are in the folder collected_data, in a file named "Search Strategy Test.xlsx"
[2] all the results of the tests made are in the folder Tests and results, in the file "Symmetry Breaking Constraint Evaluation.xlsx"

- Same dimension block constraint;
- Blocks with same height and vertical coordinate constraint;
- Three blocks constraint.

2.4.2 Chuffed

Chuffed is a state of the art lazy clause solver designed from the ground up with lazy clause generation in mind. Thanks to this approach, advantages of constraint programming are combined with some of the advantages of SAT solving.
The final configuration of our model, using Chuffed as solver, has the following list of symmetry breaking constraints:

- Fixed position of the first largest block;
- Blocks with same width and horizontal coordinate constraint;
- Blocks with same height and vertical coordinate constraint;
- Three blocks constraint

2.4.3 OR-Tools

The last solver we decided to use is OR-Tools. This solver was developed by Google. We decided to use this solver too because it won in almost all categories at the Minizinc challenge 2021, proving to be one of the best solvers to solve constraint programming problems. After all the tests executed with this solver, we saved the following list of symmetry breaking constraints:

- Blocks with the same dimensions are interchangeable;
- Fixed position of the first largest block;
- Pair constraint;
- Blocks with same height and vertical coordinate constraint;
- Three blocks constraint

2.4.4 Solver configuration

We need to highlight also the fact that we used the following configurations for all of the solvers:

- Optimization Level **-O5**
- Number of threads **7**

Those information are reported because are needed in order to make our results repeatable, even if we know that the due to the restart technique there is some randomness in the process.
We used the same configuration with all the solvers but Chuffed, which did not allow us to use more than one thread.

2.5 Results

After the definition of all the solvers and the configurations used with them, we proceed with the testing of the models just built with and without using symmetry breaking constraints. For each test we added a time-limit of 300 seconds.
All the tests were made on a personal computer with the following specifics:

- CPU: Intel(R) Core i7-8565U CPU @ 1.80GHz
- RAM: 16,0 GB

We tested the models generated with all the instances and the results produced are summarized in this table:

Solver + with/without symmetry breaking constraints	Number of solved instances	Mean time required	Mean number of propagations
Gecode without SB constraints	29	93.033	102 875 512
Chuffed without SB constraints	34	58.583	-
OR-Tools without SB constraints	36	36.522	16 457 539
Gecode with SB constraints	29	89.290	160 525 045
Chuffed with SB constraints	35	51.829	-
OR-Tools with SB constraints	39	25.055	33 133 811

As we can see from the table, we did not have information about the number of propagations which are made by Chuffed. Another important thing to say is that the number of propagations in Gecode and OR-Tools has a slightly different meaning. For OR-Tools the number of propagations is related to the propagations made in order to reach the best solution, whereas in Gecode this value is related to the number of propagations made in the time of search. In case of optimal result the value has the same meaning, whereas in the situation in which there is a timeout the obtained value will have a different meaning.
We didn't consider this difference much relevant in our choice because as it clearly shown in the table, OR-Tools is the solver which produced the best results in both cases: with and without using symmetry breaking constraints.
For this reason we decided to show the results obtained by OR-Tools.

Best model for OR-Tools - No symmetry breaking constraints				
Instance number	**Time**	**Best solution found**	**Number of failures**	**Propagations**
1	0,821	8	0	0
2	0,722	9	1	4.030
3	0,847	10	0	3.221
4	0,872	11	6	8.402
5	0,906	12	3	12.922
6	0,940	13	20	18.327
7	0,948	14	5	18.575
8	0,969	15	79	45.463
9	0,892	16	22	31.622
10	1,070	17	202	107.509
11	1,141	18	348	187.451
12	1,674	19	561	221.597
13	1,186	20	4	73.759
14	1,244	21	58	101.450
15	1,317	22	376	236.063
16	1,416	23	779	449.433
17	1,458	24	1.100	555.920
18	1,754	25	225	233.084
19	4,998	26	25.280	9.967.062
20	1,701	27	1.589	901.687
21	2,169	28	413	378.941
22	3,638	29	1.871	600.346
23	1,772	30	957	677.359
24	1,559	31	161	329.409
25	6,578	32	3.665	1.975.383
26	11,096	33	58.104	24.870.238
27	2,049	34	4	369.097
28	4,884	35	20.046	8.925.978
29	2,083	36	13	460.620
30	TIMED OUT	38	35.771	22.267.469
31	2,205	38	6.608	1.973.489
32	1:29.000	39	362.284	146.527.172
33	3,628	40	16.457	5.812.589
34	1:19.000	40	59.554	15.772.400
35	10,761	40	64.670	20.729.758
36	2,116	40	1.619	999.311
37	TIMED OUT	61	27.875	18.732.440
38	TIMED OUT	61	1.692	2.447.511
39	11,473	60	41.844	22.410.643
40	TIMED OUT	91	339.809	348.863.817

Best model for OR-Tools - Using symmetry breaking constraints				
Instance number	**Time**	**Best solution found**	**Number of failures**	**Propagations**
1	0,825	8	0	0
2	0,862	9	1	3.907
3	0,884	10	1	3.332
4	0,930	11	2	9.794
5	0,930	12	24	16.340
6	0,983	13	12	17.341
7	0,922	14	4	18.647
8	1,028	15	69	41.111
9	1,006	16	16	30.457
10	1,070	17	12	58.896
11	1,389	18	60	101.334
12	1,215	19	167	126.788
13	1,268	20	111	112.527
14	1,396	21	534	267.217
15	1,573	22	3	119.621
16	1,701	23	935	478.836
17	2,328	24	87	250.548
18	2,509	25	12.295	2.899.203
19	2,677	26	9.165	2.886.781
20	2,241	27	314	261.268
21	2,484	28	250	324.257
22	2,676	29	7.679	2.481.733
23	2,065	30	0	273.440
24	1,828	31	493	407.233
25	5,082	32	2.016	1.230.238
26	4,080	33	935	683.157
27	2,228	34	4	372.998
28	2,313	35	2.203	1.282.729
29	2,616	36	19	475.800
30	1:22.000	37	1.228.300	384.222.923
31	2,070	38	37	399.904
32	4:39.000	39	658.243	421.982.125
33	2,327	40	2.050	1.357.287
34	5,978	40	31.093	10.724.284
35	4,020	40	10.529	4.904.259
36	2,344	40	474	759.864
37	3:57.000	60	728.219	370.157.825
38	15,924	60	58.429	29.412.809
39	12,438	60	37.199	19.441.784
40	TIMED OUT	91	37.196	66.753.846

To give a better visual understanding, we also decided to generate some plots for the results obtained, comparing for each solver the model with and without symmetry breaking constraints. As already seen from the table, the use of symmetry breaking constraints helped pruning the search tree and it lowered the time needed to give an optimal solution.

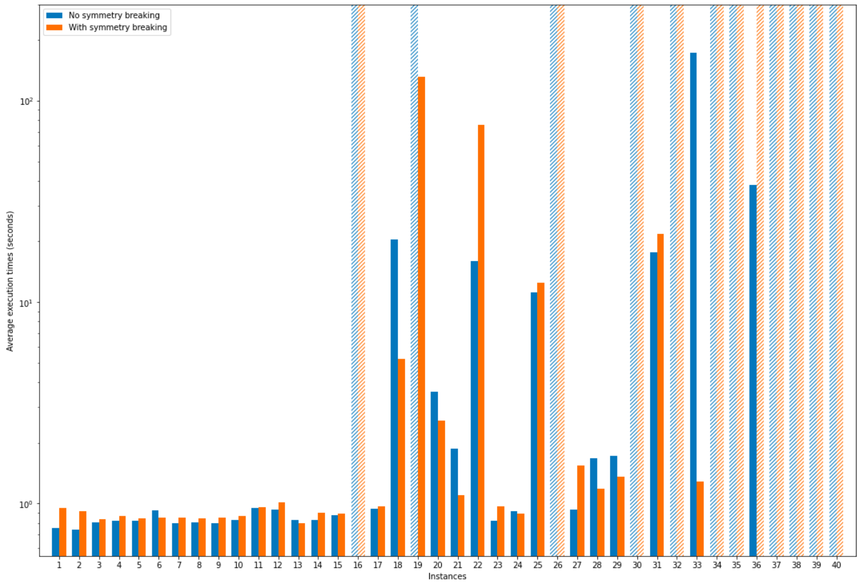


Figure 2.2: Results obtained using Gecode as solver

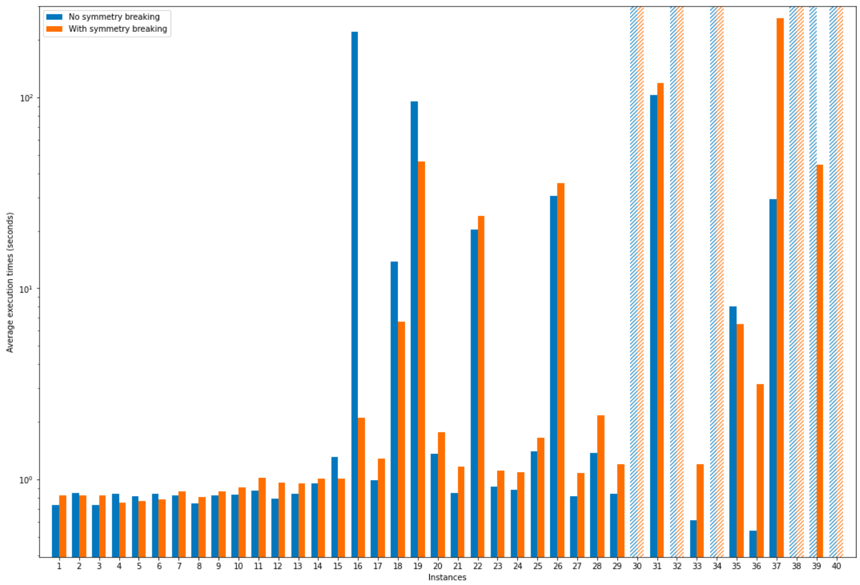


Figure 2.3: Results obtained using Chuffed as solver

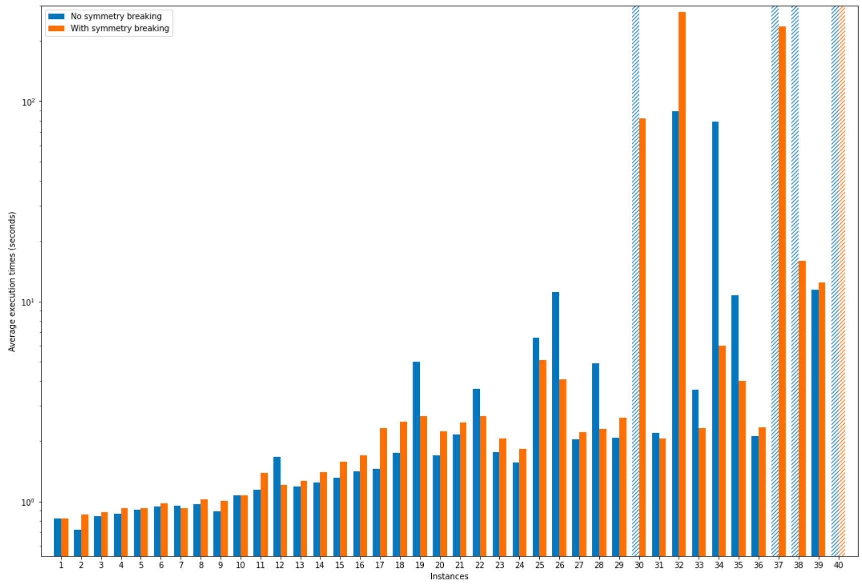


Figure 2.4: Results obtained using OR-TOOLS as solver

2.6 Rotation

To allow rotation we need to make some simple changes to our model, adding new variables and constraints. The first thing done is to create two new arrays of variables, namely $effective\_widths$ and $effective\_heights$, which are used to represent the dimensions of a block. We quickly constrained those variables in the following way:

$$\begin{aligned}\forall\ i\ \in [1..n] :&(effective\_widths_i = widths_i\ \wedge effective\_heights_i = heights_i)\vee\\ &(effective\_widths_i = heights_i\ \wedge effective\_heights_i = widths_i)\end{aligned} \tag{2.8}$$

The use of an exclusive or guarantees that the blocks preserve their aspect ratio (they can have only dimension $m \times n$ or $n \times m$). All the constraints described were now modified, using the just defined variables instead of $widths$ or $heights$.
For each used solver we defined the same symmetry breaking constraints used for the situation in which rotation was not allowed, obviously modified as above. The results of the experiments are summarized, as before, by the following tables and graphs:

Solver + with/without symmetry breaking constraints	Number of solved instances	Mean time required	Mean number of propagations
Gecode without SB constraints	24	133.390	2 646 625 685
Chuffed without SB constraints	21	151.656	-
OR-Tools without SB constraints	34	77.721	4 785 280
Gecode with SB constraints	24	128.704	2 115 564 454
Chuffed with SB constraints	22	151.671	-
OR-Tools with SB constraints	32	77.721	7 060 735

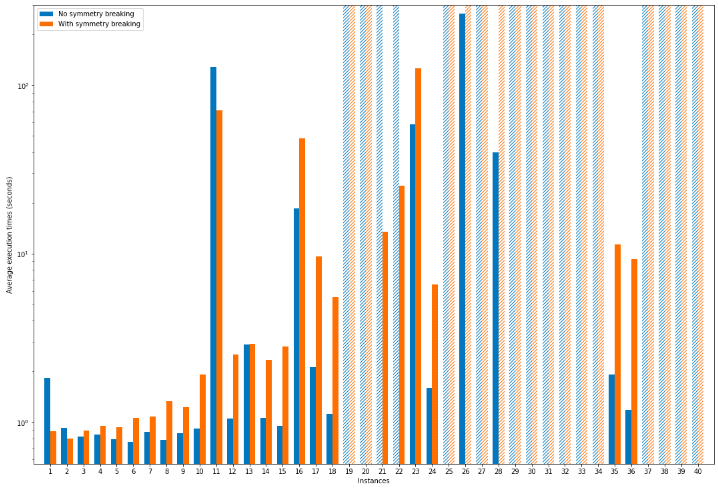


Figure 2.5: Results obtained using Gecode as solver

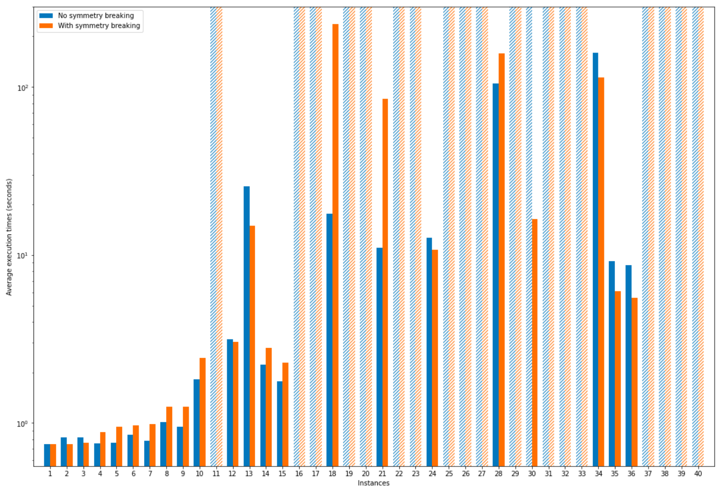


Figure 2.6: Results obtained using Chuffed as solver

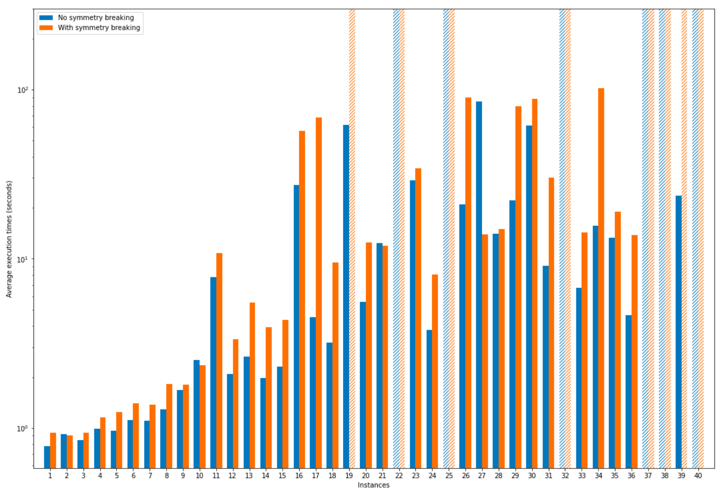


Figure 2.7: Results obtained using OR-TOOLS as solver

As before, we attach the results obtained using OR-Tools, both with and without the use of symmetry breaking constraints.

Best model for OR-Tools with rotation - No symmetry breaking constraints				
Instance number	**Time**	**Best solution found**	**Number of failures**	**Propagations**
1	0,781	8	0	2.116
2	0,919	9	0	14.107
3	0,851	10	0	12.029
4	0,992	11	11	26.239
5	0,965	12	0	36.877
6	1,113	13	34	47.733
7	1,107	14	6	58.572
8	1,286	15	30	106.549
9	1,683	16	57	99.245
10	2,522	17	12	180.319
11	7,781	18	21.339	8.536.935
12	2,093	19	562	484.174
13	2,645	20	2.145	1.082.043
14	1,976	21	124	296.233
15	2,294	22	61	404.436
16	27,27	23	1.985	2.064.160
17	4,532	24	3.502	2.242.555
18	3,197	25	295	748.365
19	1:02,000	26	312	687.149
20	5,551	27	5.192	1.650.551
21	12,387	28	9.788	7.848.944
22	TIMED OUT	30	284	946.234
23	28,943	30	108	891.049
24	3,787	31	111	905.492
25	TIMED OUT	33	7.123	7.631.325
26	20,967	33	329	1.640.975
27	1:25.000	34	52.469	31.527.281
28	14,032	35	13.554	3.442.167
29	22,127	36	9.499	9.325.995
30	1:01.000	37	39.250	27.866.303
31	9,055	38	4.243	4.071.892
32	TIMED OUT	40	4.378	5.543.607
33	6,709	40	320	1.870.959
34	15,605	40	8.946	6.770.594
35	13,279	40	2	2.031.906
36	4,629	40	62	1.489.486
37	TIMED OUT	61	11.454	15.816.110
38	TIMED OUT	61	2.646	7.887.685
39	23,562	60	278	5.657.302
40	TIMED OUT	112	8.587	29.465.501

Best model for OR-Tools with rotation - Using symmetry breaking constraints				
Instance number	**Time**	**Best solution found**	**Number of failures**	**Propagations**
1	0,939	8	0	3.153
2	0,905	9	0	14.997
3	0,940	10	32	19.646
4	1,159	11	3	27.367
5	1,242	12	0	40.623
6	1,393	13	34	52.303
7	1,375	14	6	62.416
8	1,811	15	34	114.306
9	1,797	16	57	106.473
10	2,342	17	137	244.874
11	10,754	18	11.320	5.248.807
12	3,351	19	656	589.367
13	5,521	20	3.461	1.980.439
14	3,929	21	124	305.351
15	4,350	22	2	402.767
16	56,690	23	59.726	40.935.041
17	1:08.000	24	42.175	27.695.165
18	9,518	25	659	1.067.163
19	TIMED OUT	27	62	624.536
20	12,533	27	4.805	1.893.954
21	11,890	28	238	836.376
22	TIMED OUT	30	2.590	2.683.760
23	34,107	30	16.351	10.727.636
24	8,073	31	111	933.319
25	TIMED OUT	33	5.146	6.194.464
26	1:30.000	33	5.151	7.674.236
27	13,971	34	3.736	4.165.390
28	14,961	35	1.985	3.150.678
29	1:20.000	36	33.729	22.962.807
30	1:28.000	37	31.094	27.953.945
31	30,253	38	22.044	16.226.949
32	TIMED OUT	40	4.683	6.818.794
33	14,257	40	3.484	4.489.692
34	1:42.000	40	62.299	53.974.780
35	19,016	40	2.881	4.362.111
36	13,771	40	37	1.506.560
37	TIMED OUT	61	2.535	9.404.286
38	TIMED OUT	61	6.603	11.479.403
39	TIMED OUT	61	90	5.455.455
40	TIMED OUT	-	-	-

Chapter 3

SAT solving

In computer science, **SAT solving** (SATisfiablity solving) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. We say that a Boolean formula is satisfied if there is an assignment of values which makes the formula true. If no such assignment exists then we say the formula is unsatisfiable.
SAT is the first problem proven to be NP-complete [4]. This means that all problems which are in NP are at most as difficult as SAT. Despite it being in the NP complexity class, solvers which have good performances have been developed.

3.1 Z3

A SAT solver is a program which aims to decide, given a SAT formula, whether it is SAT or not. A famous Solver is **Z3**. It was developed at Microsoft research, and it is a solver for symbolic logic. It allows to solve really complex logical formulas and provides a lot of interfaces which allow it to be used with different programming languages (in our case we used the python interface).

3.2 Encoding of the problem in SAT

To encode our problem we first tried to encode all the constraints and our data using a tri-dimensional matrix M of dimensions $n \times w \times h$ defined in the following way:

$$M_{i,j,k} = \begin{cases} True & block\ i\ is\ using\ the\ position\ (j,k) \\ False & otherwise \end{cases} \tag{3.1}$$

We quickly realized that this encoding would not lead us to an acceptable result, so we started looking into the literature for another solution to the problem. Soh et Al. in [5] solved the problem using **order-encoding**.
In order encoding [6], we use boolean variables to represent a comparison of the kind $x \leq c$, where x is a variable and c is a constant. In our model we have two type of literals to express this kind of relationship, namely px and py. We used $n \cdot w$ literals for each px_i to encode the relationship $px_i \leq j,\ j \in [1..w]$. The same can be said to the py but considering h instead of w.
Differently from the CP formulation, in this case we cannot use an objective function to find the minimum value for h but we have to inject it directly and check if the resulting formula is satisfiable. We start from the lower bound found in the corresponding section and in case of positive result we stop, since we found the optimal solution. If the algorithm produces `unsat`, we increment the value of h and try again to find a solution. To produce our model we also used some other literals, which are used to express the relationship between the blocks, i.e. to know whether a block is below or at the left of another block. Those literals are called $left_{i,j}$ and $under_{i,j}$. So we can recap all the literals of the model with the following table:

Literals for SAT encoding		
Literal	Number of literals	Meaning of the True literal
px	$n \cdot w$	$px_{i,j}$ = block i can have x-coordinate j
py	$n \cdot h$	$py_{i,j}$ = block i can have y-coordinate j
left	$n \cdot n$	$left_{i,j}$ = block i is at the left of block j
under	$n \cdot n$	$under_{i,j}$ = block i is under block j

3.3 Encoding of the constraints

Once defined all the literals that will be used for the SAT encoding, we need to define all the constraints we used to encode our problem.

3.3.1 Axiom clauses due to order encoding

For each rectangle r_i and integers e and f such that: $0 \le e \le w - widths_i$ and $0 \le f \le h - heights_i$ we have the 2-literal axiom clauses due to order encoding:

$$\neg px_{i,\ e} \ \vee \ px_{i,\ e+1} \tag{3.2}$$

$$\neg py_{i,\ f} \ \vee \ py_{i,\ f+i} \tag{3.3}$$

3.3.2 Non-overlapping constraints

For each pair of blocks r_i and r_j in which $i < j$ we have two kinds of non-overlapping constraints. The first non-overlapping constraint is a 4-literal constraint:

$$left_{i,\ j} \ \vee left_{j,\ i} \ \vee under_{i,\ j} \ \vee under_{j,\ i} \tag{3.4}$$

We also have the following non-overlapping constraints, which hold for each $e,\ j$ such that $0 \le e \le w - widths_i$ and $0 \le f \le h - heights_i$:

$$\neg\ left_{i,\ j} \ \vee \ px_{i,\ e} \ \vee \ \neg\ px_{j,\ e+widths_i} \tag{3.5}$$

$$\neg\ left_{j,\ i} \ \vee \ px_{j,\ e} \ \vee \ \neg\ px_{i,\ e+widths_j} \tag{3.6}$$

$$\neg\ under_{i,\ j} \ \vee \ py_{i,\ f} \ \vee \ \neg\ py_{j,\ f+heights_i} \tag{3.7}$$

$$\neg\ under_{j,\ i} \ \vee \ py_{j,\ f} \ \vee \ \neg\ py_{i,\ f+heights_j} \tag{3.8}$$

3.4 Encoding of the optimization constraints

We encoded some other constraints which aim to prune the search tree in order to optimize our model. We decided to use several techniques to prune the search space, hoping to improve our model and the performances we could get.

3.4.1 Domain reduction technique

In order to remove potential symmetries we decided, as we did in CP, to constrain the block with the biggest area to be in a specific quadrant of the plate. In this case we decided to place the block in the top-right quadrant, which is equivalent to the choice we made in CP. This constraint is expressed using the following clauses:

$$\forall\ i\ \in [0..w_{lim}]\ :\ \neg\ px_{max,i} \tag{3.9}$$

$$\forall\ j\ \in [0..h_{lim}]\ :\ \neg\ py_{max,i} \tag{3.10}$$

$$where\ w_{lim} = \left\lfloor \frac{w - widths_{max}}{2} \right\rfloor \ h_{lim} = \left\lfloor \frac{h - heights_{max}}{2} \right\rfloor$$

Applying this reduction, if a block's width, namely $widths_i$, satisfies $widths_i > \left\lceil \frac{w - widths_{max}}{2} \right\rceil$ we can assign $left_{i,max}$ to false. The same thing can be said for the height of each rectangle.

In formula:

$$widths_i \ > \left\lceil \frac{w - widths_{max}}{2} \right\rceil \implies \neg left_{i,max} \tag{3.11}$$

$$heights_i > \left\lceil \frac{h - heights_{max}}{2} \right\rceil \implies \neg under_{i,max} \tag{3.12}$$

3.4.2 Same size rectangle constraint

As we did for CP, in order to reduce the symmetries we can impose an ordering for the blocks which have the same dimensions. We encoded this constraint using the literals which refer to the relationship among blocks, which are $left$ and $under$. The constraints can be expressed in the following way:

$$\forall i, j \ s.t. \ i < j \ : \ (widths_i = widths_j \ \wedge \ heights_i = heights_j) \implies (\neg left_{i,j}) \wedge (\neg under_{j,i} \vee left_{j,i}) \tag{3.13}$$

3.4.3 Large Rectangle Constraint

Differently from the aforementioned constraints, this constraint does not remove any symmetry. With this constraint we are enforcing the fact that if the sum of the widths (resp. heights) of two blocks is greater than the width (resp. height) of the plate, then they cannot be placed one at the left (resp. under) the other. We can express those constraints as follows:

$$\forall i, j \ i < j \ :(widths_i + widths_j > w) \implies (\neg left_{i,j} \wedge \neg left_{j,i}) \tag{3.14}$$

$$\forall i, j \ i < j \ :(heights_i + heights_j > h) \implies (\neg under_{i,j} \wedge \neg under_{j,i}) \tag{3.15}$$

Obviously this situation cannot happen because in the case the implication is false we would have two blocks placed one above the other which summed height is greater than the total height of the plate (the same holds for the width) but this would be in contrast with some other constraints.

3.5 Results

Although the results of Z3 are not deterministic due to some random decisions it makes, we confronted both versions of the model: the one with and without symmetry breaking constraints. To make to comparison more reliable we repeated all the tests on the instances 5 times and then we averaged the produced results.
All the tests were run on `Google Colaboratory Environment`. Both models showed a good behavior in the majority of instances, in fact during the five execution of the model we managed to solve all but 2 instances (38, 40), as summarized from the table:

	Number of solved instances	Mean time for the solving	Mean number of propagation
Solver with no optimization constraints	38	27.829	80617233
Solver with optimization constraints	38	29.168	62206784

Despite from what we can see from the table, the model with optimization constraints helped saving some time for the most complex instances. The higher value of mean time is due to the fact that in the smaller instances it is slower than the model without those constraints. Since the loss in performance in smaller instances was not excessive we chose to present the results of the model with the optimization constraints illustrated before[1].

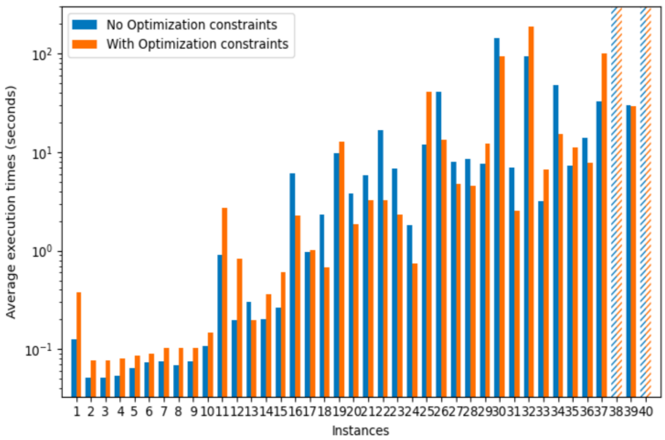


Figure 3.1: Results obtained both with and without optimization constraintd

[1] all the other results are in the folder collected_data

SAT model with optimization constraints		
Instance number	**Elapsed time**	**Number of propagations**
1	0,125	68
2	0,051	408
3	0,051	1133
4	0,054	2704
5	0,064	4731
6	0,073	10595
7	0,076	14145
8	0,069	18487
9	0,076	25540
10	0,108	38323
11	0,914	454412
12	0,196	590649
13	0,299	856651
14	0,203	952326
15	0,266	1110765
16	6,133	5676643
17	0,979	6666379
18	2,339	9101480
19	9,797	17837742
20	3,774	21467800
21	5,777	27544360
22	16,7	41341897
23	6,855	47816577
24	1,823	49809627
25	11,871	61045208
26	40,723	97722657
27	8,018	105626333
28	8,555	113759093
29	7,577	122132724
30	145,208	189673880
31	6,918	196076766
32	93,384	211745812
33	3,192	215158712
34	47,571	228128901
35	7,263	230513658
36	13,913	234311319
37	32,599	242862902
38	TIMED OUT	242862902
39	29,598	250862509
40	TIMED OUT	250862509

3.6 Rotation

Allowing rotation increases exponentially the number of possible combinations of rectangle we can place on the plate. To encode this situation we had to tweak our model and consider only one of the optimization constraints we mentioned before.
First of all, we need to introduce a new literal for each rectangle, *rotated*. This literal is true in case the rectangle is rotated, false otherwise. It was fundamental to consider also rotation in all clauses, that now will be of the form:

$$(\neg rotated_i \wedge original\_clause) \vee (rotated_i \wedge rotated\_clause)$$

where *original_clause* refers to the clauses we discussed before and *rotated_clause* is their equivalent version in case the block is rotated.
An example of this is given by the order encoding constraint of the px. For each rectangle i and for each e, f which satisfy $0 \leq e \leq w - widths_i$ and $0 \leq f \leq w - heights_i$:

$$\Big(\neg rotated_i \wedge (\neg px_{i,e} \vee px_{i,e+1})\Big) \vee \Big(rotated_i \wedge (\neg px_{i,f} \vee px_{i,f+1})\Big) \tag{3.16}$$

As regards symmetry breaking constraints, we considered only the placement of the biggest block in the top-right quadrant in this case, which is implemented with the following constraint:

$$(\neg rotated_{max} \wedge \neg px_{max,i} \wedge \neg py_{max,j}) \vee (rotated_{max} \wedge \neg px_{max,j} \wedge \neg py_{max,i}) \tag{3.17}$$

which holds for each $i, \ j$ such that: $0 < i < \left\lfloor \frac{w - widths_{max}}{2} \right\rfloor$ and $0 < j < \left\lfloor \frac{h - heights_{max}}{2} \right\rfloor$.

The last aspect we need to remark is that we used all those constraints which include the rotated clause only in case the height of the block is smaller than the total width of the plate, because otherwise we will exceed the horizontal bound.
The results obtained from the model which considers rotation are summarized by the following table:

	Number of solved instances	Mean time for the solving	Mean number of propagation
Solver with no optimization constraints	37	60.998	245296
Solver with optimization constraints	36	61.893	232070

The model which did not use the optimization constraint performed better, being able to solve one instance more than the other onw during the five execution of the test. For this reason we consider this as our best model in case we can also rotate the blocks. To conclude this section we add the results obtained from the tests of the solver in the tests: Also in this case we produced the plot of the time used to solve the instances both with and without optimization constraints:

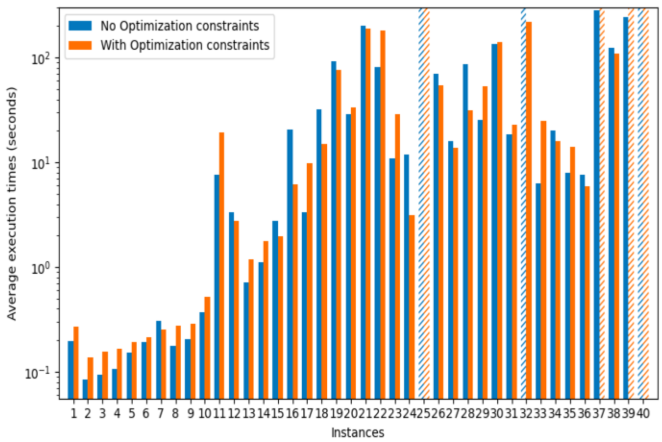


Figure 3.2: Results obtained both with and without optimization constraintd

SAT model with optimization constraints		
Instance number	**Elapsed time**	**Number of propagations**
1	0,27	376
2	0,137	845
3	0,155	1905
4	0,166	2880
5	0,191	4402
6	0,213	6220
7	0,253	8396
8	0,274	11661
9	0,286	14921
10	0,522	19477
11	19,347	26796
12	2,762	34279
13	1,189	42269
14	1,771	52140
15	1,968	63767
16	6,114	78207
17	9,878	94470
18	15,108	112872
19	75,441	135179
20	33,318	158831
21	187,457	169321
22	180,271	187508
23	28,853	216527
24	3,168	245456
25	TIMED OUT	245456
26	54,065	287375
27	13,873	327842
28	31,691	371370
29	52,697	419131
30	140,744	442636
31	22,704	484348
32	220,192	498537
33	24,837	551128
34	16,036	587763
35	14,164	627625
36	5,918	667147
37	TIMED OUT	667147
38	109,698	708303
39	TIMED OUT	708303
40	TIMED OUT	-

Chapter 4

Satisfiability Modulo Theory

Satisfiability Modulo Theory (SMT) concerns the study of the satisfiability of formulas with respect to some backgrounds theories. SMT extends the problem of Boolean satisfiability (SAT) with convex optimization and term-manipulating symbolic systems. In particular, SMT formulation allows a much richer vocabulary than Boolean operations and variables when creating formulas. Obviously, this takes into account some loss of efficiency, but the increase in expressiveness of the model encoding might actually make it easier to improve solutions.
There are two main approaches for SMT solving:

- *Eager approach*, which consists in translating the SMT formula in an equisatisfiable SAT formula and solves it with a SAT solver;
- *Lazy approach*, defines and uses ad-hoc procedures for the background theories.

Nowadays, SMT solvers are much more complex tools that integrate DPPL-style boolean reasoning with theory-specific solvers.

4.1 Z3 solver

Z3 [7] is an efficient SMT solver with specialized algorithms for solving background theories. A theory is defined by a signature, which defines the domains, functions, and relations of the theory, and a set of interpretations of the relations and functions.
Moreover, Z3 provides capabilities to work with linear real and integer arithmetic. It provides APIs for the most common programming languages like Python, Java, C++ and .Net.
Z3 is used in a wide range of software engineering applications, ranging from program verification, compiler validation, testing, fuzzing using dynamic symbolic execution, model-based software development, network verification and optimization.

4.2 Encoding of the problem in SMT

4.2.1 Variables and parameters

The input parameters are exactly the one used in CP:

- $w \in \mathbb{N},\ w \geq 0$ which represents the width of the plate;
- $n \in \mathbb{N},\ n \geq 0$ representing the number of the chips;
- $width,\ height \in \mathbb{N}$, where $width_i, height_i \geq 0$ representing respectively, the width and the height of the i-th chip.

Then, to represent the information about the position that each rectangle has on the plate we defined two variables for each block: x_i and y_i, which refer to the position on the x and y axis of the bottom-left corner of the i-th rectangle.
Furthermore, we have an additional variable $h \in \mathbb{N}$ which encodes the height of the plate. This is necessary due to the strategy chosen to solve the problem. In fact, unlike CP, with SMT we are not trying to find the optimal solution using an optimization approach, but we are using binary search to prove if the encoded formula is satisfaible for the given value of h. This strategy will be discussed in more detail in section 4.3.

4.2.2 Encoding of constraints

The constraints are practically the same as for CP modeling. As stated in [8] we have two main types of constraints:

- *Boundary Constraints*, to ensure that each block is within the boundary of the plate. This can be represented as $\forall i \in \{1, ..., n\}$:

$$x_i + width_i \leq w$$

$$y_i + height_i \leq h$$

- *Non-Overlap Constraints*, which states that for each pair of blocks, $block_i$ and $block_j$, at least one of the following cases must be satisfied:

$$block_i \textbf{ to the left of } block_j\textbf{:} \qquad x_i + width_i \leq x_j$$

$$block_i \textbf{ to the right of } block_j\textbf{:} \qquad x_i - width_j \geq x_j$$

$$block_i \textbf{ below } block_j\textbf{:} \qquad y_i + height_i \leq y_j$$

$$block_i \textbf{ above } block_j\textbf{:} \qquad y_i - height_j \geq y_j$$

Therefore, a first possible model will be the one that satisfies both Boundary Constraints and Non-overlap Constraints.
In order to improve our model both the constraints have been optimized, in particular we have defined two different formulation for the *Non-Overlap Constraints*:

- In the first formulation we have that:

$$\forall i, j \in \{1, .., n\} \text{ with } i \neq j$$

$$(x_i - x_j \leq -\ width_i) \vee (x_j - x_i \leq -\ width_j) \vee (y_i - y_j \leq -\ height_i) \vee (y_j - y_i \leq -\ height_j)$$

  For the sake of simplicity, from here we will refer to this formulation as *Or formulation*:

- In the second one, knowing that thanks to De Morgan's laws: $A \vee B = \neg(\neg A \wedge \neg B)$, we first computed the inverse inequalities and then we expressed the constraints in the form:

$$\forall i, j \in \{1, .., n\} \text{ with } i \neq j$$

$$\neg((x_i - x_j \leq width_j - 1) \wedge (x_j - x_i \leq width_i - 1) \wedge (y_i - y_j \leq height_j - 1) \wedge (y_j - y_i \leq height_i - 1))$$

  We will refer to this formulation as *NotAnd formulation.*

In both the introduced formulations all the inequalities have the form $x - y \leq k$, where x and y are integer variables and k is an integer constant. Boolean combinations of this type of inequalities are in a restricted fragment of the Linear Integer Arithmetic (LIA) called Integer Difference Logic (IDL). IDL supports more efficient decision procedures then LIA. in particular, Z3 provides a dedicated solver for IDL problems, in the section 4.4.2 we will test the difference between different theory solvers provided by Z3.

4.2.3 Encoding of Symmetry Breaking constraints

With the aim of pruning the search tree and reducing symmetries,we have encoded some of the symmetry breaking constraint used in CP also for SMT. The implemented constraints are the following:

- *Fixed position of the biggest block*;
- *Blocks with same horizontal coordinate and same widths*;
- *Blocks with same horizontal coordinate and same widths.*

For more information on how these constraints have been implemented, refer to the section 2.2.

4.3 Search strategy

As previously mentioned, with SMT we are not trying to find the optimal solution using an optimization approach.
In particular, we are computing the upper and lower bound for h as described in 1.2.2. Then, knowing that:

$$h_{min} \leq h_{optimal} \leq h_{max}$$

We can try to find the optimal solution using the *binary search*.
Binary Search is a search algorithm used to find the position of $h_{optimal}$ in the range $[h_{min}, ..., h_{max}]$. In this approach, $h_{optimal}$ is always searched in the middle of the range. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element of the range to compare to the target value, and repeating this until $h_{optimal}$ is found.
We have chosen this approach since in Z3 determine if a formula is satisfiable is faster then finding the optimal value for an objective function.

4.4 Experiments and results

4.4.1 Z3 configuration

Before talking about the different tested configurations of Z3, we need to introduce what tactics are in Z3.
In contrast to solvers that ultimately check the satisfiability of a set of assertions, tactics transform assertions to sets of simplified assertions. In particular, many useful pre-processing steps can be formulated in this way, and it is possible to combine different tactics with different theory solvers.
In order to test whether the changes made in the encoding and the use of ad-hoc tactics for integer problems led to improvements, we tested Z3 with three different configurations:

1. **Default configuration**, the first configuration is the standard Z3 configuration, so no specialized *tactic* or *theory solve*r of Z3 were used.

2. **LIA configuration**, in the second configuration the following pre-processing tactics were used:

   - *simplify*: used to apply simplification rules;
   - *solve-eqs*: try to eliminate variables by solving equations;
   - *propagate-ineqs*: propagate inequalities and bounds, remove subsumed inequalities.
   - *propagate-values*: propagate the values of the constants;
   - *symmetry-reduce*: try to apply symmetry reduction;
   - *purify-arith*: eliminate unnecessary operators.

   The last tactic used is called *qflia*, which is a specialized solver for linear integer arithmetic problems.

3. **IDL configuration**, the third configuration uses the same pre-processing tactics of the second one, the only difference is that it uses a specialized solver for integer difference logic problems called *qfidl*.

In addition, each configuration was tested with both the *Or formulation* and *NotAnd formulation* for the *No-Overlap Constraints*.

4.4.2 Results analysis

Due to the nondeterminism of Z3 we decided to repeat all tests for each configuration 5 times and then averaged the results. All tests were run on Google Colaboratory environment.
In the following tables it is shown the number of instances solved in 300 seconds and the total number of timeouts without finding the optimal solution for each configurations.

	No symmetry breaking		With symmetry breaking	
Solver	**Solved instances in 5 runs**	**num. of timeouts in 5 runs**	**Solved instances in 5 runs**	**num. of timeouts in 5 runs**
Default	38	19	38	16
LIA	37	23	37	23
IDL	38	18	37	19

Table 4.1: Collected data for 5 runs using Or formulation for No-overlap constraints

	No symmetry breaking		With symmetry breaking	
Solver	**Solved instances in 5 runs**	**num. of timeouts in 5 runs**	**Solved instances in 5 runs**	**num. of timeouts in 5 runs**
Default	37	16	38	14
LIA	38	10	36	30
IDL	39	9	38	12

Table 4.2: Collected data for 5 runs using NotAnd formulation for No-overlap constraints

As we can see from the tables, in all the configurations and formulations the symmetry breaking constraints have worsen the situation, this may be due to the increase in complexity of the model. The configuration which produces the better results is the IDL configuration with the *NotAnd formulaton* for the *No-overlap constraints*.
For this reason we decided to show the results obtained by this configuration, both with and without the use of symmetry breaking constraints.

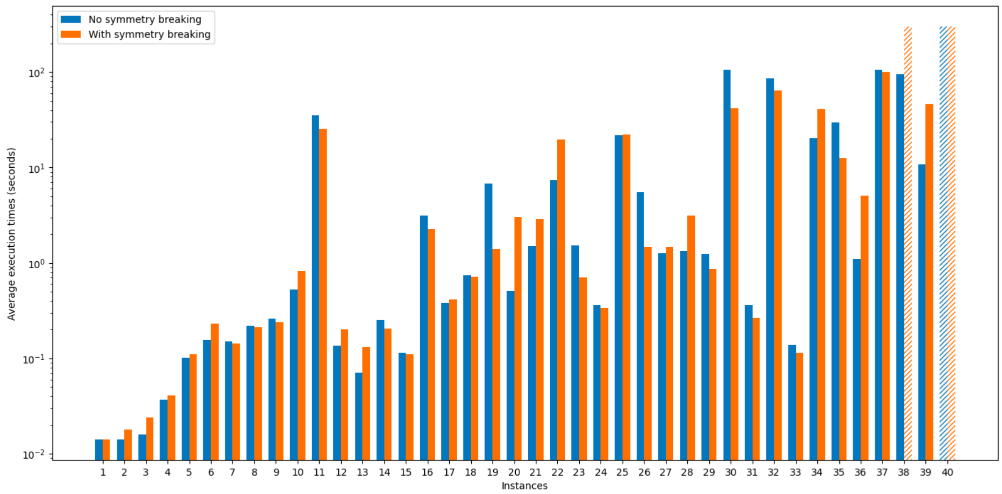


Figure 4.1: Results obtained using IDL with NotAnd formulation

Results with IDL configurations and NotAnd formulation no Symmetry breaking constraints				
Instance number	Time	Propagations	Best solution found	Num. of timeouts in 5 runs
1	0,014	-	8	0
2	0,014	-	9	0
3	0,016	-	10	0
4	0,037	-	11	0
5	0,102	-	12	0
6	0,155	-	13	0
7	0,151	-	14	0
8	0,218	-	15	0
9	0,262	-	16	0
10	0,53	-	17	0
11	35,354	-	18	0
12	0,136	24018	19	0
13	0,071	9757	20	0
14	0,25	46448	21	0
15	0,115	17224	22	0
16	3,122	509266	23	0
17	0,378	51230	24	0
18	0,743	118204	25	0
19	6,804	888741	26	0
20	0,512	63680	27	0
21	1,501	180456	28	0
22	7,423	906835	29	0
23	1,514	209403	30	0
24	0,358	47992	31	0
25	21,809	2210403	32	0
26	5,567	673440	33	0
27	1,268	173584	34	0
28	1,331	148089	35	0
29	1,249	143783	36	0
30	106,355	11061459	37	0
31	0,363	45958	38	0
32	86,653	7337343	39	0
33	0,137	13357	40	0
34	20,531	1963477	40	0
35	29,85	2686408	40	0
36	1,106	102792	40	0
37	105,487	8668455	60	0
38	94,759	1578177	60	4
39	10,749	879254	60	0
40	TIMED OUT	719862	93	5

Results with IDL configurations and NotAnd formulation with Symmetry breaking constraints				
Instance number	Time	Propagations	Best solution found	Num. of timeouts in 5 runs
1	0,014	-	8	0
2	0,018	-	9	0
3	0,024	-	10	0
4	0,041	-	11	0
5	0,111	-	12	0
6	0,229	-	13	0
7	0,142	-	14	0
8	0,211	-	15	0
9	0,239	-	16	0
10	0,825	-	17	0
11	25,593	-	18	0
12	0,202	34814	19	0
13	0,131	20073	20	0
14	0,206	34178	21	0
15	0,11	13827	22	0
16	2,283	362993	23	0
17	0,411	66200	24	0
18	0,714	106326	25	0
19	1,395	172682	26	0
20	3,009	398119	27	0
21	2,878	382816	28	0
22	19,583	2459728	29	0
23	0,702	91478	30	0
24	0,339	45194	31	0
25	22,314	2330174	32	0
26	1,485	186185	33	0
27	1,469	199604	34	0
28	3,133	419772	35	0
29	0,872	85599	36	0
30	41,928	4416211	37	0
31	0,263	30958	38	0
32	64,009	3612790	39	2
33	0,114	10879	40	0
34	41,031	4361895	40	0
35	12,573	1241223	40	0
36	5,091	519793	40	0
37	100,587	9001994	60	0
38	Time-out	37724	62	5
39	46,731	4253235	60	0
40	Time-out	499728	93	5

4.5 Rotation

To allow blocks rotation we need to make some changes to our encoding. First, we need to introduce a boolean variables called *rotated* for each block, which indicates whether the i-th block has been rotated. Then, we need to add the following constraint:

$$\forall i \in \{1, ..., n\}$$

$$(\neg rotated_i \wedge width_i = original\_width_i \wedge height_i = original\_height_i)$$

$$\vee$$

$$(rotated_i \wedge width_i = original\_height_i \wedge height_i = original\_width_i)$$

Where *original_width* and *original_height* respectively represent the dimensions of the i-th block not rotated.
Thanks to this encoding it is not necessary to modify the *Non-Overlap* and *Boundary* constraints, for this reason also in this case we have the same two different formulations for the *Non-overlap* constraints and the same Symmetry breaking constraints of the model without rotation.
Furthermore, in order to improve the model, the following auxiliary constraints were added:

$$\forall i \in \{1, ..., n\}\ (original\_width_i = original\_height_i) \implies \neg rotated_i$$

$$\forall i \in \{1, ..., n\}\ (original\_height_i > w) \implies \neg rotated_i$$

$$\forall i \in \{1, ..., n\}\ (original\_width_i > h) \implies \neg rotated_i$$

For optimization reasons, knowing that thanks to equivalence rules and De Morgan's laws: $A \implies B \equiv \neg(A \wedge \neg B)$ we can rewrite the auxiliary constraints as:

$$\forall i \in \{1, ..., n\}\ \neg((original\_width_i = original\_height_i) \wedge rotated_i)$$

$$\forall i \in \{1, ..., n\}\ \neg((original\_height_i > w) \wedge rotated_i)$$

$$\forall i \in \{1, ..., n\}\ \neg((original\_width_i > h) \wedge rotated_i)$$

4.5.1 Results analysis

As the case without rotation we repeated all tests for each configuration 5 times. Also in this case all tests were run on Google Colaboratory environment. The following tables show the number of solved instances in 300 seconds and the total number of timeouts without finding the optimal solution for each configurations.

	No symmetry breaking		**With symmetry breaking**	
Solver	**Solved instances in 5 runs**	**num. of timeouts in 5 runs**	**Solved instances in 5 runs**	**num. of timeouts in 5 runs**
Default	29	87	30	83
LIA	30	88	32	77
IDl	31	85	30	86

Table 4.3: Collected data for 5 runs using Or formulation for No-overlap constraints with rotation enabled

	No symmetry breaking		**With symmetry breaking**	
Solver	**Solved instances in 5 runs**	**num. of timeouts in 5 runs**	**Solved instances in 5 runs**	**num. of timeouts in 5 runs**
Default	31	79	35	71
LIA	29	71	32	74
IDl	33	70	33	65

Table 4.4: Collected data for 5 runs using NotAnd formulation for No-overlap constraints with rotation enabled

Allowing rotation increases exponentially the number of possible combinations of rectangle we can place on the plate. For this reason, as we can see from the tables, allowing rotation has made the situation worse in all the configurations and formulations.
It is interesting to note that unlike before, in this case the symmetry breaking constraints have helped to improve the situation. Moreover, the configuration that solved the most instances in 5 runs is *Default configuration* with *NotAnd formulation* and symmetry breaking constraints, this could be due to the increase of logical operators in the encoding. Anyway, we decided to not use this formulation to compare the model with and without rotation because the total number of timeouts was higher than the *IDL* configuration with *NotAnd formulation*. Instead, to allow a better comparison with the model without rotation, we decided to use the latter.

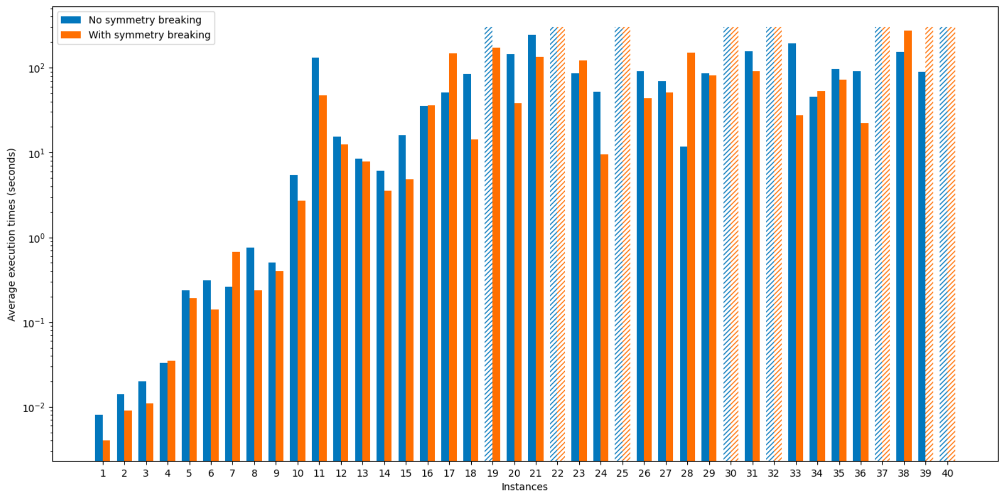


Figure 4.2: Results obtained using IDL with NotAnd formulation

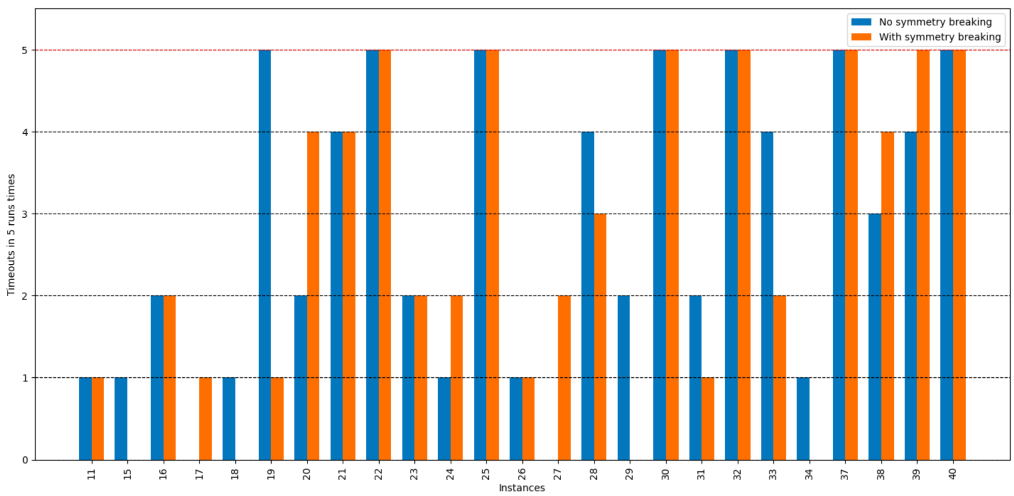


Figure 4.3: Number of timeouts in 5 runs using IDL with NotAnd formulation

Results with IDL configurations and NotAnd formulation no Symmetry breaking constraints with Rotation enabled				
Instance number	Time	Propagations	Best solution found	Num. of timeouts in 5 runs
1	0,008	67	8	0
2	0,014	767	9	0
3	0,02	939	10	0
4	0,033	3694	11	0
5	0,236	38936	12	0
6	0,312	61774	13	0
7	0,26	42957	14	0
8	0,762	115437	15	0
9	0,501	84076	16	0
10	5,432	924358	17	0
11	131,995	18989836	18	1
12	15,411	2010221	19	0
13	8,522	1220090	20	0
14	6,086	934423	21	0
15	16,107	1781916	22	1
16	35,637	2776156	23	2
17	51,02	6217822	24	0
18	83,942	8179729	25	1
19	TIMED OUT	467019	28	5
20	143,391	11335676	27	2
21	242,168	6917820	28	4
22	TIMED OUT	2969935	30	5
23	86,218	6531572	30	2
24	52,41	4668084	31	1
25	TIMED OUT	7281129	33	5
26	90,28	8515266	33	1
27	69,26	8944773	34	0
28	11,816	3356854	35	4
29	85,512	6593068	36	2
30	TIMED OUT	5349000	38	5
31	156,835	9809282	38	2
32	TIMED OUT	4387339	41	5
33	192,343	5432451	40	4
34	45,465	4112994	40	1
35	95,757	11120547	40	0
36	91,036	12387969	40	0
37	TIMED OUT	9893805	61	5
38	154,048	7780871	60	3
39	89,05	5130726	60	4
40	TIMED OUT	-	-	5

Results with IDL configurations and NotAnd formulation with Symmetry breaking constraints with Rotation enabled				
Instance number	Time	Propagations	Best solution found	Num. of timeouts in 5 runs
1	0,004	84	8	0
2	0,009	727	9	0
3	0,011	755	10	0
4	0,035	3415	11	0
5	0,19	30820	12	0
6	0,14	18864	13	0
7	0,677	165379	14	0
8	0,238	36865	15	0
9	0,403	86718	16	0
10	2,705	678517	17	0
11	47,491	9320043	18	1
12	12,339	2324692	19	0
13	7,752	1482198	20	0
14	3,547	729533	21	0
15	4,806	853543	22	0
16	35,746	4169062	23	2
17	146,505	17718588	24	1
18	14,308	3087060	25	0
19	171,861	29196825	26	1
20	38,321	6527576	27	4
21	134,928	4361018	28	4
22	TIMED OUT	1793028	30	5
23	120,479	13113868	30	2
24	9,578	863510	31	2
25	TIMED OUT	3411070	33	5
26	43,938	5113647	33	1
27	50,713	5790530	34	2
28	149,574	12319092	35	3
29	81,066	12021584	36	0
30	TIMED OUT	7321552	38	5
31	90,176	10581365	38	1
32	TIMED OUT	9089736	41	5
33	27,461	2482492	40	2
34	53,067	7678047	40	0
35	72,756	12473125	40	0
36	22,339	4256802	40	0
37	TIMED OUT	10036104	61	5
38	271,657	10594663	60	4
39	TIMED OUT	2804617	61	5
40	TIMED OUT	-	-	5

Chapter 5

Integer Linear Programming

Linear Programming deals with the problem of optimizing a linear objective function, subject to linear equality and inequality constraints on the decision variables.
It could be very useful for many problems that require an optimization of resources.
A linear program can take many different forms. First, we have a minimization or a maximization problem, depending on whether the objective function is to be minimized or maximized. The constraints can either be inequalities ($\leq$ or $\geq$) or equalities. Some variables might be unrestricted in sign (i.e. they can take positive or negative values; this is denoted by $\gtrless 0$) while others might be restricted to be non-negative.

5.1 Encoding

Starting from the basic formulation of the problem already presented in the "Preliminaries", the following modification have been introduced.

5.1.1 Big-M Method

The Big-M method for constraints is typically used to provide a way for binary variables to turn constraints on or off only when a certain binary variable takes on one value.
They are so named because they typically involve a large coefficient M that is chosen to be larger than any reasonable/possible value that a variable or expression may take.
In the case of VLSI problem it could be used to make sure that in the non-overlapping constraints at most one horizontal and at most one vertical geometric relation is implied.

So we extended our basic formulation of the problem in order to apply the Big-M method.
First of all we need four auxiliary binary variables, one for each non-overlapping constraint:

$$Z^k_{i,j} \in \{0,1\},\ k \in [1,4]$$

Then we need to define our Big-M constants, we will use the plate width W for the horizontal constraints and the max height H computed in the "Upper and Lower bound" paragraph.
So, the tailored non-overlapping constraints for this formulation are expressed by the following inequalities:

$$\forall i,j\ s.t\ i<j:\ x_i + width_i \leq x_j + W(1 - Z^1_{i,j}), \tag{5.1}$$
$$x_j + width_j \leq x_i + W(1 - Z^2_{i,j}), \tag{5.2}$$
$$y_i + height_i \leq y_j + H(1 - Z^3_{i,j}), \tag{5.3}$$
$$y_j + height_j \leq y_i + H(1 - Z^4_{i,j}) \tag{5.4}$$

In order to ensure that at most one horizontal geometric relation between (5.1) and (5.2) and at most one vertical geometric relation between (5.3) and (5.4) is implied, we add the following inequalities:

$$1 \geq Z^1_{i,j} + Z^2_{i,j} \tag{5.5}$$
$$1 \geq Z^1_{i,j} + Z^2_{i,j} \tag{5.6}$$

Finally, as proposed in [9], another inequality was added in order to ensure that at least one geometric relation between any rectangle pair is implied:

$$1 \leq Z^1_{i,j} + Z^2_{i,j} + Z^3_{i,j} + Z^4_{i,j} \tag{5.7}$$

5.1.2 Symmetry Breaking Constraints

To effectively reduce the search space size and the wasted time visiting new solutions which are symmetric to the already visited one, the following symmetry breaking constraint were added to the encoding:

- Fixed position of the first biggest block;
- Fixing the position of a pair of rectangles.

We refer to the same implementation already presented in the "Symmetry Breaking Constraint" chapter in CP.

5.2 AMPL

In order to test more than one solver and to facilitate experimenting with different formulations, we decided to exploit a modeling language, our choice fell on AMPL.
AMPL[10] is a modeling tool that uses a notation close to familiar mathematical notation to state variables, objectives, constraints and parameters that may be involved in an optimization problem. AMPL does not solve problems by itself, but instead writes files with full details of the problem instances to be solved and invokes separate solvers.

5.2.1 Preprocessing of the Instances

The original formulation of the instances to solve went trough some slight modification to make them compatible with AMPL. We wrote a Python script to automatically convert the original instances into the format accepted by AMPL. Moreover this script add two additional parameter, the maximum and the minimum height of the instance that we already seen in the "Upper and Lower bound" paragraph.
This is how an instance is done:

```
data;
param w := 8;
param n := 4;
param min_h := 8;
param max_h := 10;
set BLOCKS := 1 2 3 4 ;
param:   width   height :=
      1   5        5
      2   5        3
      3   3        5
      4   3        3;
```

5.2.2 Solvers

A solver is a mathematical software that 'solves' a mathematical problem. A solver takes problem descriptions in some sort of generic form and calculates their solution. The emphasis is on creating a program or library that can easily be applied to other problems of similar type.
AMPL offers a selection of linear solvers that can be used in linear optimization problems, among these we decided to exploit Gurobi[11] and CPLEX[12].

Solver's Parameters With the aim of improving the performance, we also tried to play with the various options and parameters that the two solvers provide.
Both of them offer an automatic tuning tool that performs multiple solves, choosing different parameter settings for each solve, in a search for settings that improve runtime. After running several tuning on a representative set of the instances and after trying to manually adjust some parameters, we came up with the following set of parameters.

Gurobi Parameter's Set

- **method=1** : Set "Dual-Simplex" as method for the root node of the problem;
- **nodemethod=0** : Set "Primal-Simplex" as method to solve relaxed node problems;
- **mipfocus=2** : Set the solution strategy in favor to prove optimality;
- **presolve=2** : Apply Gurobi aggressive presolve;
- **cuts=2** : Apply Gurobi aggressive global cut generation;

CPLEX Parameter's Set

- **dualopt** : Set "Dual-Simplex" as solution algorithm.
- **mipstartalg=1** : Set "Primal-Simplex" as method to solve the initial MIP subproblem;
- **predual=1** : CPLEX's presolve phase presents the CPLEX solution algorithm with the dual problem;
- **mipemphasis=2** : Set the solution strategy in favor to prove optimality;
- **mipcuts=2** : Apply CPLEX aggressive global cut generation;

5.3 Results

The presented model was tested on a personal notebook with the following specifics:

- **Processor** : AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
- **Ram** : 8,00 GB (5,93 GB available)

The limit for the solving time of each instance was set to 300 seconds. We decided to run three different tests for each solver:

1. Without Symmetry Breaking Constraint;
2. With Symmetry Breaking Constraint;
3. With Symmetry Breaking Constraint and Solver Parameters;

The obtained results are summarized in the following table.

	Number of Solved Instances		
Solver	**No Symmetry Breaking**	**With Symmetry Breaking**	**With Symmetry Breaking and Solver's Parameter**
Gurobi	27	31	32
CPLEX	26	25	27

As we can clearly see, Gurobi generally outperforms CPLEX in every configuration. The use of symmetry breaking constraints improves the performance only for Gurobi with four more instances solved, while with CPLEX we solve one less. A general behaviour that is emerged is that symmetry breaking constraints have increased the solve time for the simpler instances (with fewer blocks) but they have helped to solve the more difficult one that timed out in the first configuration. The addition of the Solver's Parameter slightly improved the performance for both. Another aspect that can be noticed is that CPLEX's performance had a slight variation across all configurations. We decided to show in detail the differences between the two solvers in their best configuration, i.e. the one with symmetry breaking constraints and solver's parameters.

Gurobi- With Symmetry Breaking Constraint and Solver's Parameters				
Instance Number	**Time**	**Best Solution Found**	**Simplex Iteration**	**Branch and Bound Nodes**
1	0.047	8	4	1
2	0.047	9	14	1
3	0.032	10	32	1
4	0.047	11	94	1
5	0.063	12	174	1
6	0.093	13	268	1
7	0.031	14	177	1
8	0.078	15	378	1
9	0.047	16	244	1
10	0.125	17	540	1
11	74.406	18	2032740	42889
12	1.578	19	13516	1200
13	1.204	20	3435	11
14	4.265	21	28565	2010
15	1.485	22	11357	897
16	5.609	23	43776	2695
17	13.828	24	259661	10649
18	8.531	25	119626	5109
19	275.734	26	5280165	116795
20	11.187	27	124900	5514
21	180.656	28	3536774	85714
22	42.093	29	740155	22484
23	11.594	30	157226	7229
24	8.281	31	56199	3345
25	Timed Out	33	5930914	139787
26	219.407	33	4617356	112523
27	5.797	34	94286	3761
28	9.781	35	101183	3262
29	23.468	36	440898	14952
30	Timed Out	38	5233347	102986
31	9.032	38	106685	4735
32	Timed Out	40	5315246	87571
33	32.422	40	828594	21217
34	264.438	40	4806778	74688
35	Timed Out	41	4822133	82721
36	50.734	40	568025	23638
37	Timed Out	62	4303244	73344
38	Timed Out	62	5236068	37290
39	Timed Out	61	4697590	53011
40	Timed Out	102	1572244	12507

CPLEX - With Symmetry Breaking Constraint and Solver's Parameters				
Instance Number	**Time**	**Best Solution Found**	**Simplex Iteration**	**Branch and Bound Nodes**
1	0.063	8	17	0
2	0.047	9	44	0
3	0.062	10	47	0
4	0.094	11	109	0
5	0.125	12	139	0
6	0.11	13	227	0
7	0.094	14	154	0
8	0.14	15	190	0
9	0.172	16	169	0
10	0.36	17	1520	102
11	9.704	18	360121	14481
12	1.656	19	96465	5539
13	4.609	20	265506	6761
14	4.188	21	179417	7415
15	3.625	22	141024	8297
16	Timed out	24	6035097	95304
17	13.75	24	365838	15790
18	118.359	25	3316964	55786
19	112.343	26	2660902	50788
20	269.719	27	5346425	99222
21	5.469	28	60021	5106
22	Timed out	30	4335372	63859
23	33.703	30	930969	23827
24	43.953	31	1296679	36814
25	Timed Out	34	5684486	75501
26	168.156	33	4479170	92205
27	20.187	34	444872	19250
28	Timed Out	36	4677795	74520
29	Timed Out	37	4890201	122131
30	Timed Out	38	4631751	67592
31	5.781	38	149209	5480
32	Timed Out	41	3745609	33890
33	34.297	40	1208438	23996
34	Timed Out	42	3977725	39284
35	Timed Out	41	4423245	54886
36	181.969	40	2345864	36144
37	Timed Out	63	3770419	45812
38	Timed Out	64	3622716	39380
39	Timed Out	63	3565783	30798
40	Timed Out	124	984603	13375

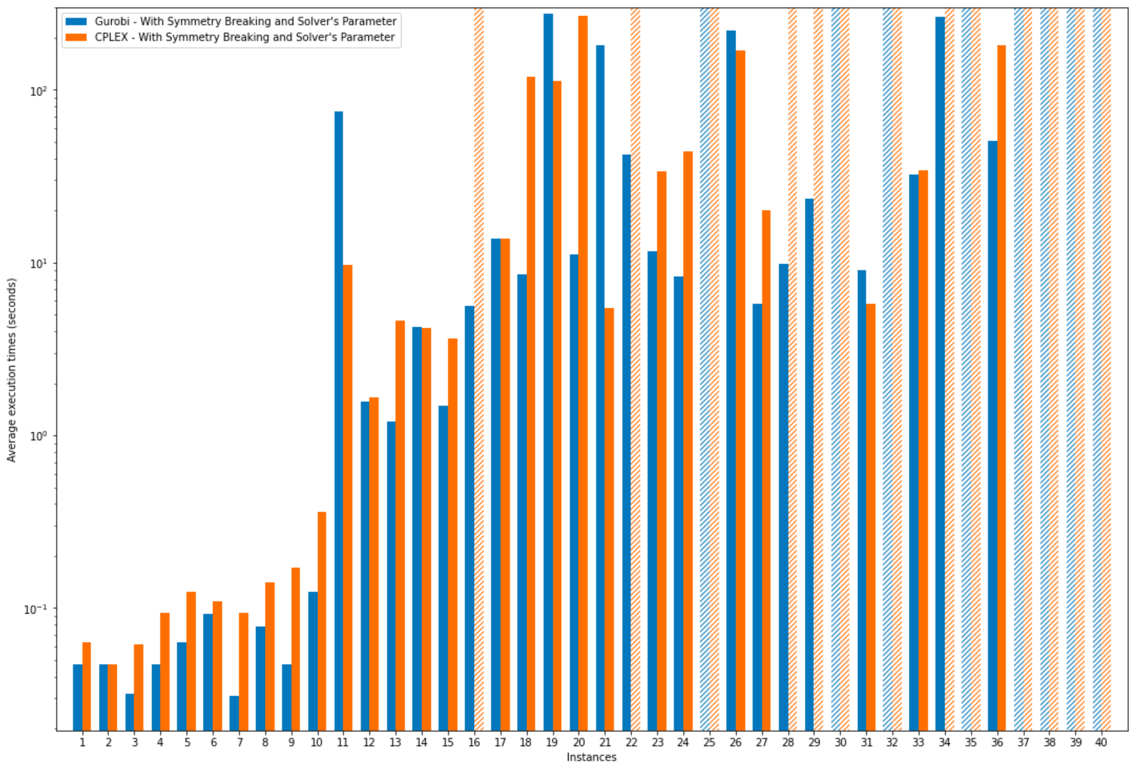


Figure 5.1: Results obtained by both solvers using Symmetry Breaking and Solver's Parameters

5.4 Rotation

In order to handle the possibility of rotation for each block, we need to slightly modify the encoding of our model. First we need to add one additional Boolean variable:

$$R_i \in \{0, 1\},\ \forall i \in \{1, number\ of\ blocks\}$$

that is used to enable the change in orientation of the block, setting $R_i = 1$ when the block is rotated by 90° and $R_i = 0$ when placed in its initial orientation.

Now we need to modify the bound constraints for the coordinates of the blocks, in fact if the block is rotated we need to swap its width with its height and viceversa:

$$\forall i:\ X_i + (1 - R_i) * (width_i) + R_i * height_i \leq W, \tag{5.8}$$

$$Y_i + (1 - R_i) * (height_i) + R_i * width_i \leq H \tag{5.9}$$

Lastly, we need to update the non overlap constraints in order to handle the swap between width and height of a block if it is rotated:

$$\forall i, j\ s.t\ i < j:\ x_i + (1 - R_i) * (width_i) + R_i * height_i \leq x_j + W(1 - Z^1_{i,j}), \tag{5.10}$$

$$x_j + (1 - R_j) * (width_j) + R_j * height_j \leq x_i + W(1 - Z^2_{i,j}), \tag{5.11}$$

$$y_i + (1 - R_i) * (height_i) + R_i * width_i \leq y_j + H(1 - Z^3_{i,j}), \tag{5.12}$$

$$y_j + (1 - R_j) * (height_j) + R_i * width_j \leq y_i + H(1 - Z^4_{i,j}) \tag{5.13}$$

The new model was tested on the same machine with the time limit for the solving time of each instance setted again to 300 seconds.
The obtained results for each solver, with the same test types already done in the case without rotation, are summarized in the following table.

	Number of Solved Instances, Rotation Enabled		
Solver	**No Symmetry Breaking**	**With Symmetry Breaking**	**With Symmetry Breaking and Solver's Parameter**
Gurobi	26	24	23
CPLEX	21	21	22

Again, Gurobi outperforms CPLEX in every configuration. The use of the symmetry breaking constraint in this case worsen the performance for Gurobi while in the case of fixed block it made a significantly difference. This could be due to the fact that we introduced a whole new set of variables and weighed down the non overlap constraint. The addition of the Solver's Parameter slightly increased the performance only for Gurobi. Even in this case the performance of CPLEX are not heavily effected when changing the configuration.
We decided to show in detail the differences between the two solvers in their best configuration, i.e. the one with symmetry breaking constraints and solver's parameters for CPLEX and the one without Symmetry Breaking for Gurobi.

Gurobi - Without Symmetry Breaking, Rotation Enabled				
Instance Number	**Time**	**Best Solution Found**	**Simplex Iteration**	**Branch and Bound Nodes**
1	0.031	8	13	1
2	0.047	9	19	1
3	0.046	10	32	1
4	0.047	11	142	1
5	0.094	12	371	1
6	0.047	13	237	1
7	0.125	14	2022	375
8	0.344	15	9471	699
9	0.078	16	454	1
10	0.281	17	2330	13
11	7.265	18	286832	19745
12	1.25	19	38898	3158
13	0.782	20	24457	1194
14	4.047	21	216176	18635
15	5.719	22	232545	19047
16	17.234	23	654745	36198
17	87.703	24	3345818	239215
18	Timed out	26	11168581	648549
19	Timed out	27	10245658	575745
20	Timed out	28	9889456	628991
21	Timed out	29	9324017	522806
22	Timed out	30	8950723	339186
23	261.718	30	9345006	599818
24	72	31	2519644	213449
25	Timed out	33	11333385	218602
26	Timed out	34	9379812	651792
27	214.813	34	6701532	476100
28	15.141	35	532079	17539
29	55.343	36	1436006	90129
30	Timed out	38	7440803	358199
31	2.187	38	50798	2677
32	Timed out	40	6853222	336509
33	7.235	40	164929	8043
34	Timed out	41	12273867	342425
35	287.406	40	7926308	463243
36	127.015	40	3461285	196845
37	Timed out	61	11288735	198160
38	Timed out	62	11588654	174517
39	Timed out	61	10973683	250696
40	Timed out	105	1960158	14849

CPLEX - With Symmetry Breaking Constraint and Solver's Parameters, Rotation Enabled				
Instance Number	**Time**	**Best Solution Found**	**Simplex Iteration**	**Branch and Bound Nodes**
1	0.141	8	0	0
2	0.047	9	56	0
3	0.047	10	32	0
4	0.078	11	105	0
5	0.11	12	1059	161
6	0.094	13	150	0
7	0.296	14	17077	2458
8	0.125	15	252	0
9	0.297	16	2538	614
10	0.578	17	19951	2048
11	Timed out	19	8777732	207194
12	0.703	19	2488	190
13	12.031	20	412719	19001
14	21.828	21	756001	22957
15	4.218	22	88148	7046
16	79.156	23	1611361	32039
17	36.797	24	839556	27926
18	Timed out	26	5899579	187218
19	Timed out	27	3704837	50724
20	Timed out	28	4635394	93854
21	Timed out	29	4596208	88869
22	Timed out	31	4190540	42793
23	Timed out	31	5739393	100509
24	5.14	31	133935	4901
25	301.219	34	2890838	36239
26	Timed out	34	4909761	100885
27	Timed out	35	5259342	111392
28	51.969	35	1153207	27585
29	81.813	36	1589970	41305
30	Timed out	39	4094628	40109
31	4.781	38	119131	5127
32	Timed out	41	3734556	52989
33	49.093	40	1122908	28144
34	Timed out	42	3885713	36621
35	Timed out	41	3992321	59722
36	Timed out	41	3762926	59251
37	Timed out	63	4281245	52160
38	Timed out	62	3540336	29328
39	Timed out	62	3650802	32753
40	Timed out	132	816701	22396

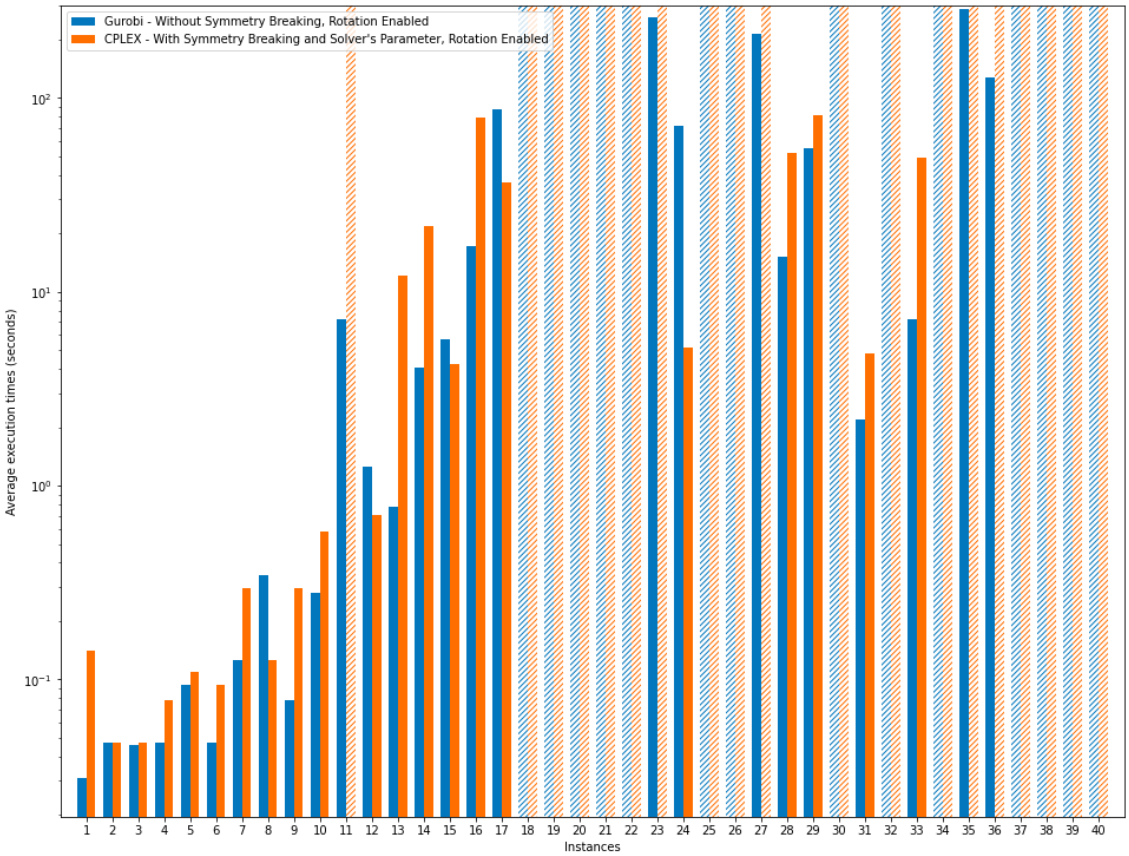


Figure 5.2: Results obtained by both solvers using their best configuration

Chapter 6

Conclusions

In this report we have presented four different optimization approaches for the VLSI problem, tackling the formulation with and without the possibility of rotating blocks.
For the case without rotation, CP and SMT produced the best result solving in both cases all instances except the number 40. With a similar result SAT managed to solve 38 instances. Finally we have ILP that proved to optimality 32 instances.
When the rotation of blocks was introduced, in general we noticed a worsening of performance for all the approaches. The one with less difference in solved instances was SAT that proved to optimality 37 instances. CP and SMT follow with, respectively, 34 and 33 instances solved and lastly ILP found the optimal solution for 26 instances.
It is interesting to notice that a sub-optimal solution was found almost for all instances in every approach except for SAT due to the search strategy adopted.
To conclude, even if VLSI Floorplanning is a known NP-Complete problem, with state-of-the-art solvers supported by a tailored model we managed to obtain satisfying results.

Bibliography

[1] Helmut Simonis and Barry O'Sullivan. *Using global constraints for rectangle packing.* Proceedings of the first Workshop on Bin Packing and Placement Constraints BPPC'08, associated to CPAIOR'08, 2008.

[2] Christian Schulte et Al. Gecode.

[3] Laurent Perron and Vincent Furnon. Or-tools.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. *Association for Computing Machinery*, 1971.

[5] Takehide Soh et Al. A sat-based method for solving the two-dimensional strip packing problem. *Fundamenta Informaticae*, 2010.

[6] Naoyuki Tamura et Al. Compiling finite linear csp into sat. *Constraints*, 2009.

[7] Microsoft. Z3 solver. `https://github.com/z3prover/z3`.

[8] Suchandra Banerjee et Al. Satisfiability modulo theory based methodology for floorplanning in vlsi circuits. *IEEEXplore*, 2017.

[9] K.-H. Küfer M. Berger, M. Schröder. *A constraint programming approach for the two-dimensional rectangular packing problem with orthogonal orientations.* Springer Berlin, Heidelberg, 2008.

[10] David M. Gay. *The AMPL Modeling Language — an Aid to Formulating and Solving Optimization Problems.* Springer, 2014.

[11] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[12] IBM ILOG Cplex. V12. 8: Ibm ilog cplex optimization studiocplex user's manual. *International Business Machines Corporation*, 2017.