VLSI

Federico Rullo federico.rullo@studio.unibo.it
Edoardo Conca edoardo.conca@studio.unibo.it
Fabio Zanotti fabio.zanotti@studio.unibo.it

February 1, 2024

1 Introduction

VLSI (Very Large Scale Integration) is the process of integrating circuits into silicon chips, and it is commonly used in devices such as smartphones. The process involves shrinking the size of transistors, which allows engineers to fit more and more transistors into the same area of silicon. This, in turn, has led to the integration of more and more functions of cellphone circuitry into single silicon plates, making modern cell phones powerful tools that can fit comfortably in a pocket or purse and come with advanced features such as video cameras and touchscreens.

There are two variants of the VLSI problem. The first variant involves placing each circuit in a fixed orientation, meaning that an $n * m$ circuit cannot be positioned as an $m * n$ circuit on the plate. The second variant allows for rotation, meaning that an $n * m$ circuit can be positioned either as it is or rotated. There are several techniques to solve the VLSI problem, including:

- **Constraint Programming**
- **SAT**
- **Satisfiability Modulo Theories**
- **Mixed-Integer Linear Programming**

1.1 Format of the instances

Instances: An instance of VLSI is a txt file consisting of lines of integer values. The first two lines give **w**, which is the width of the silicon plate and **n**, the number of necessary circuits to place inside the plate. The following lines, divided into columns x_i and y_i, represent the horizontal and vertical dimensions of the single circuit. For the Minizinc models, in Constraint Programming, we converted the .txt files into .dzn files to be able to read them directly.

Solution The position of the single circuit can be described by the position on the silicon plate. The solution should indicate the length of the plate l, as well as the position of each circuit by its $\hat{x}_i$ and $\hat{y}_i$, which are the coordinates of the left-bottom corner. This could be done by adding the length, **l**, next to **w**, and adding $\hat{x}_i$ and $\hat{y}_i$ next to x_i and y_i in the instance file. Hence, we create a .txt file for each solved instance and plot them.

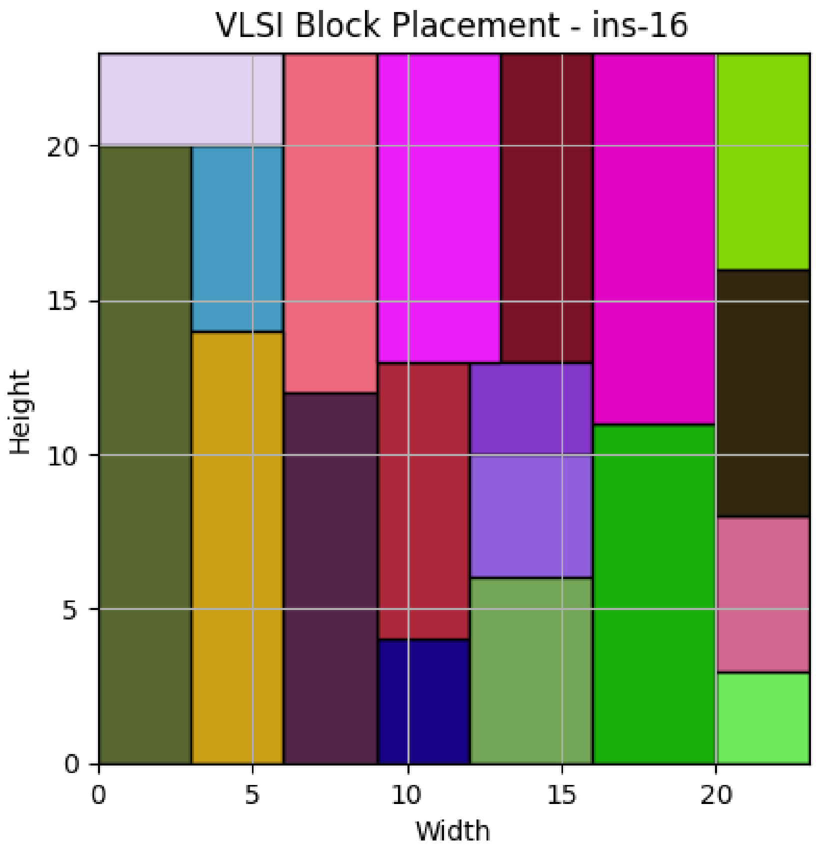


Figure 1: Plot of a solution of the VLSI problem.

1.2 Scripts and different models

For each model, implemented in the different techniques, we considered different parameters to find the best set allowing us to get the maximum number of optimal solutions. To simplify this, Python scripts have been implemented to run the different tests using the available parameters and output the results.
Plots which reproduce the circuits placed on the plate have been generated using the optimal set of parameters.

2 CP Model

Constraint Programming is a programming paradigm that is used to solve combinatorial optimization problems with a declarative approach. The model formalizes the unknowns, the possible values, and their relationships. For this particular project, the **Minizinc** tool was implemented and executed with two different constraint solvers, namely **Gecode** and **Chuffed**.

2.1 Decision Variables

We start by taking a .dzn file as input, which includes the following information:

- The width of the plate, which we call **W**;
- The number of circuits, which we call **N**;
- The width and height of each circuit, which we call X_i and Y_i.

We save this information into variables with corresponding names. For the CP model, we define the following decision variables:

- $pos_x[i]$: Represents the x-coordinate of the placement of circuit i on the rectangular plate. This variable is bounded by the range of values from 0 to the difference between the plate width and circuit width, i.e.,

$$pos_x[i] \in [0, W - X]$$

.

- $pos_y[i]$: Represents the y-coordinate of the placement of circuit i on the rectangular plate. This variable is bounded by the range of values from 0 to the difference between the plate height and circuit height, i.e.,

$$pos_y[i] \in [0, H - Y]$$

.

- l: Represents the boundaries and is bounded by the maximum value between the sum of the heights of all circuits, and the sum of the areas of all circuits divided by the plate width. We take the maximum of these two values to ensure that the formula is correct even when there is a circuit with a height much bigger than others. Therefore,

$$l \in [l_{\text{min}}, l_{\text{max}}]$$

, where

$$l_{\text{max}} = \sum_{i=1}^{N} Y_i$$

and

$$l_{\text{min}} = \max\left(\max(Y), \lceil \frac{\text{circuit\_area}}{W} \rceil\right)$$

.

- $rotation[i]$: Represents whether circuit i is rotated or not. This variable can take on values of 0 or 1, where 0 indicates no rotation, and 1 indicates rotation, this variable is also present only when rotation is used in the solver.

2.2 Objective Function

The objective function aims to minimize the total area occupied by circuits to fit them all on the plate.
The total area is calculated by:

$$Total\_Area = \sum_{i \in circuits} (pos_x[i] + circuit\_width) \cdot (pos_y[i] + circuit\_height)$$

2.3 Constraints

We have identified two key constraints to the problem of designing circuits on a plate. Firstly, the circuits cannot exceed the plate, and secondly, they cannot overlap with each other. To address this problem, we have imposed the following constraints:

- Scheduling Parallelism: We plan to use the global constraint $cumulative$ to optimize the space on the plate. We view the task as a resource usage problem, where each circuit is an activity whose duration is its height, and the number of resources used is its width. The total number of resources available is the plate width (or height, for rotation). We can apply this reasoning in the opposite direction as well, depending on the plate's dimensions. We will use the following constraints:
$$cumulative(pos_y, y, x, plate\_width)$$
$$cumulative(pos_x, x, y, plate\_length)$$

- Overlapping of circuits is another problem. To solve this, we plan to use a $diffn$ from MiniZinc. Which takes the origin points and sizes of the rectangles and imposes non-overlapping constraints. We will use the following constraint to ensure that the circuits do not overlap:
$$diffn(pos_x, pos_y, x, y)$$

2.3.1 Implied constraints

It is important also to consider one more constraint when designing such circuits. If we draw a horizontal line and add up the horizontal sides of the circuits where the line intersects, the total sum should not exceed the width of the plate. Similarly, when we draw a vertical line and add up the vertical sides of the circuits that intersect with it, the total should not exceed the length of the plate.
Assuming that the circuits are not rotated, we can solve this constraint in the following way:

$$max(pos_{x_1} + x_1, pos_{x_2} + x_2, ..., pos_{x_n} + x_b) \leq plate\_width$$
$$max(pos_{y_1} + y_1, pos_{y_2} + y_2, ..., pos_{y_n} + y_b) \leq plate\_length$$

2.3.2 Symmetry breaking constraints

VLSI design exhibits several symmetries, such as:

- Reflection over x-axis;
- Reflection over y-axis;
- circuits with equal dimensions;

One simple way to break these symmetries is to place the largest circuit at position (0,0), which forces the largest circuits to be in the same position while rejecting other configurations. This can be represented as an equation:

$$pos_x(max\_x\_index) = 0 \land pos_y(max\_y\_index) = 0$$

A constraint is implemented to ensure that equivalent solutions, including horizontal flips, are avoided.
This constraint requires circuits with coordinates pos_{x_i} in the left half of the plate to have a covered area that is greater than or equal to the ones in the right half side. This constraint is represented by the following equations:

$$\forall i \in 1..n : area_i = x_i * y_i$$

$$\sum_{i=1, pos_{x_i} \leq w \div 2}^{n} area_i \geq \sum_{i=1, pos_{x_i} \leq w \div 2}^{n} area_i$$

2.4 Rotation

In this variation of the problem, we allow the circuits to be rotated. This means that the final height and width of the circuits could be different from the dimensions provided in the input. To account for this, a new boolean array called ”rotation” was created and added to the existing model, where it was indicated if the circuit i had swapped dimensions.
From a logical perspective, the constraints were the same as the previous model but they differ in the way we used them. To incorporate this change, two new arrays were introduced to store the real height and width of each circuit relative to the rotation array.

$$\forall i \in 1..n$$

$$real\_pos_{x_i} = \begin{cases} y_i, & rotated_i \\ x_i, & otherwise \end{cases}$$

$$real\_pos_{y_i} = \begin{cases} x_i & rotated_i \\ y_i & otherwise \end{cases}$$

Here, $real\_pos_{x_i}$ and $real\_pos_{y_i}$ are the actual circuit dimensions, while x and y are the initial inputs.
We also observed that when a circuit has a height greater than the plate width, it cannot be rotated. Therefore, we had to introduce the constraint:

$$y_i > plate\_width \implies rotation_i = False$$

2.5 Validation

2.5.1 Experimental design

By default Minizinc does not specify a default method for search solutions, leaving it to the underlying solver. However, for combinatorial integer problems, we had to define how the search is carried out. A search strategy determines which choices to make and influences the number of solutions found and their performance.
In our case, we used a sequential search that took into account different variables.

- **l**: was given more importance in finding a good set of values for the height, doing so allowed the solver to be better suited to searching for the coordinates.
- **pos_x** and **pos_y** arrays.

For each **int_search** we specified variable choice annotation and constraint choice.
We used different variable choice annotations such as:

- **first fail**: which chooses the variable with the smallest domain size;
- **input order**: which chooses in order from the array;
- **indomain min**: which chooses the variable with the smallest value in the domain;

2.5.2 Experimental results

The results show that some instances consistently produce optimal solutions, while others pose challenges with high solve times. The incorporation of rotation in the models leads to lower solve times, suggesting a positive impact. However, instances 21, 25, and 26 consistently show no solutions across all models, requiring a closer examination of the constraints. Additionally, instance 11 presents a common challenge with its elevated solving time in CHUFFED and GECODE solvers.

The variations in solve times across instances highlight the need for a robust optimization strategy, as high solving times could indicate potential outliers. In conclusion, the findings provide valuable insights into the strengths and challenges of both solvers. GECODE in both tables 6 and 4 presents a more optimal solution and faster solving times compared to CHUFFED presented in tables 1 and 2.

input_name	**status**	**height**	**time_solved**
1	OPTIMAL	8	0.001
2	OPTIMAL	9	0.002
3	OPTIMAL	10	0.003
4	OPTIMAL	11	0.002
5	OPTIMAL	12	0.003
6	OPTIMAL	13	0.004
7	OPTIMAL	14	0.017
8	OPTIMAL	15	0.006
9	OPTIMAL	16	0.019
10	OPTIMAL	17	0.007
11	OPTIMAL	18	45.794
12	OPTIMAL	19	0.009999999999999
13	OPTIMAL	20	0.796
14	OPTIMAL	21	2.268
15	OPTIMAL	22	4.348
16	OPTIMAL	23	0.009
17	OPTIMAL	24	0.009
18	NO_SOLUTION		
19	NO_SOLUTION		
20	NO_SOLUTION		
21	NO_SOLUTION		
22	NO_SOLUTION		
23	OPTIMAL	30	0.062
24	OPTIMAL	31	0.012
25	NO_SOLUTION		
26	NO_SOLUTION		
27	OPTIMAL	34	0.337
28	OPTIMAL	35	0.02
29	OPTIMAL	36	0.02
30	NO_SOLUTION		
31	OPTIMAL	38	0.009
32	NO_SOLUTION		
33	OPTIMAL	40	0.011
34	NO_SOLUTION		
35	OPTIMAL	40	10.327
36	OPTIMAL	40	0.156
37	NO_SOLUTION		
38	OPTIMAL	60	248.17000000000002
39	NO_SOLUTION		
40	NO_SOLUTION		

Table 1: CHUFFED No rotation model

input_name	**status**	**height**	**time_solved**
1	OPTIMAL	8	0.0
2	OPTIMAL	9	0.001
3	OPTIMAL	10	0.001
4	OPTIMAL	11	0.002
5	OPTIMAL	12	0.003
6	OPTIMAL	13	0.004
7	OPTIMAL	14	0.003
8	OPTIMAL	15	0.005
9	OPTIMAL	16	0.005
10	OPTIMAL	17	0.007
11	OPTIMAL	18	22.629
12	OPTIMAL	19	0.011
13	OPTIMAL	20	0.478
14	OPTIMAL	21	1.184
15	OPTIMAL	22	2.289
16	OPTIMAL	23	0.023
17	OPTIMAL	24	0.0209999999999999
18	NO_SOLUTION		
19	NO_SOLUTION		
20	NO_SOLUTION		
21	NO_SOLUTION		
22	NO_SOLUTION		
23	OPTIMAL	30	0.08
24	OPTIMAL	31	0.0289999999999999
25	NO_SOLUTION		
26	NO_SOLUTION		
27	OPTIMAL	34	0.378
28	OPTIMAL	35	0.053
29	OPTIMAL	36	0.062
30	NO_SOLUTION		
31	OPTIMAL	38	0.034
32	NO_SOLUTION		
33	OPTIMAL	40	0.036
34	NO_SOLUTION		
35	OPTIMAL	40	12.911
36	OPTIMAL	40	0.214
37	NO_SOLUTION		
38	OPTIMAL	60	289.449
39	NO_SOLUTION		
40	NO_SOLUTION		

Table 2: CHUFFED rotation model

input_name	**status**	**height**	**time_solved**
1	OPTIMAL	8	0.0611349999999999
2	OPTIMAL	9	0.000497
3	OPTIMAL	10	0.000534
4	OPTIMAL	11	0.00085
5	OPTIMAL	12	0.000542
6	OPTIMAL	13	0.001382
7	OPTIMAL	14	0.000508
8	OPTIMAL	15	0.000547
9	OPTIMAL	16	0.00168
10	OPTIMAL	17	0.001483
11	OPTIMAL	18	6.400291
12	OPTIMAL	19	0.000708
13	OPTIMAL	20	0.037421
14	OPTIMAL	21	0.000931
15	OPTIMAL	22	0.002046
16	NO_SOLUTION		
17	OPTIMAL	24	56.345727
18	OPTIMAL	25	9.098912
19	OPTIMAL	26	0.302029
20	OPTIMAL	27	7.932002
21	NO_SOLUTION		
22	OPTIMAL	29	94.548793
23	NO_SOLUTION		
24	OPTIMAL	31	0.000887
25	NO_SOLUTION		
26	NO_SOLUTION		
27	OPTIMAL	34	0.001167
28	OPTIMAL	35	1.431465
29	OPTIMAL	36	0.001069
30	NO_SOLUTION		
31	OPTIMAL	38	0.0028
32	NO_SOLUTION		
33	OPTIMAL	40	0.00106
34	NO_SOLUTION		
35	OPTIMAL	40	2.271162
36	OPTIMAL	40	10.589222
37	NO_SOLUTION		
38	NO_SOLUTION		
39	NO_SOLUTION		
40	NO_SOLUTION		

Table 3: GECODE No Rotation Model

input_name	**status**	**height**	**time_solved**
1	OPTIMAL	8	0.000536
2	OPTIMAL	9	0.000609
3	OPTIMAL	10	0.000733
4	OPTIMAL	11	0.001101
5	OPTIMAL	12	0.001728
6	OPTIMAL	13	0.002988
7	OPTIMAL	14	0.000834
8	OPTIMAL	15	0.0009339999999999
9	OPTIMAL	16	0.002155
10	OPTIMAL	17	0.0012079999999999
11	OPTIMAL	18	9.543198
12	OPTIMAL	19	0.001309
13	OPTIMAL	20	0.051956
14	OPTIMAL	21	0.001195
15	OPTIMAL	22	0.002428
16	NO_SOLUTION		
17	OPTIMAL	24	69.979337
18	OPTIMAL	25	13.025828
19	OPTIMAL	26	0.391491
20	OPTIMAL	27	11.043688
21	NO_SOLUTION		
22	OPTIMAL	29	116.63728
23	NO_SOLUTION		
24	OPTIMAL	31	0.001601
25	NO_SOLUTION		
26	NO_SOLUTION		
27	OPTIMAL	34	0.00202
28	OPTIMAL	35	1.832597
29	OPTIMAL	36	0.001857
30	NO_SOLUTION		
31	OPTIMAL	38	0.003227
32	NO_SOLUTION		
33	OPTIMAL	40	0.001722
34	NO_SOLUTION		
35	OPTIMAL	40	2.428724
36	OPTIMAL	40	12.370852000000001
37	NO_SOLUTION		
38	NO_SOLUTION		
39	NO_SOLUTION		
40	NO_SOLUTION		

Table 4: GECODE Rotation Model

3 SAT Model

An SAT Solver decides the classical NP-complete problem of whether the given propositional formula F in Conjunctive Normal Form (CNF) is satisfiable. To model the SAT solution for both cases i.e. with and without rotation, we used the Z3 solver in a Python environment.

3.1 Problem Parameters

The input parameters for a specific instance of the VLSI problem are:

1. w: the width of the silicon plate;
2. n: the number of circuits to place within the plate;
3. (w_i, h_i): width and height of the i^{th} circuit.

3.2 Objective Function

As requested, given a fixed-width plate and a list of rectangular circuits, we have to decide how to place them within the plate so that the length of the final device is minimized. Thus the goal is to determine the minimum height/length of the plate that satisfies the problem. More details about the search strategy adopted are presented in Section 6.1.

3.3 Propositional Variables

The variables shared by both models are:

- $cells_{i,j,k}$ encoded as 3D array of Boolean variables, with dimensions $h \times w \times n$, where each element $cells[i][j][k]$ represents whether circuit k is placed in cell (i, j) on the plate:
    1. If $cells[i][j][k]$ is True, it indicates that circuit k occupies the cell (i, j)
    2. If it is False, it means that the cell (i, j) is empty concerning circuit k.
- $Left_{k,l}$ and $Down_{k,l}$ are two 2D array of Boolean variables of dimension $n \times n$, encoding the relative position of each circuit. In particular, they are true if, respectively, the circuit k is placed to the left concerning the circuit l; the circuit k is placed downwards concerning the circuit l.

The additional variables needed for the model which considers rotations are:

- $rotated_k$ which is true if the circuit k is rotated in the current configuration. These variables are enclosed in an array of dimension k equal to the number of circuits to place.

4 Constraints for Base model

In the following section, we elucidate the constraints employed in the model, emphasizing the restriction on circuit rotation. The subsequent presentation outlines the specific constraints that govern the circuit placement without rotation.

4.1 Main Problem Constraints

These are essentially constraints that concern the correct positioning of the circuits on the plate, guaranteeing the boundaries and their non-overlap property.

- **Unique Circuit Placement**: This is a common constraint used in many combinatorial problems, including Sudoku, N-Queens, and many scheduling problems. In the context of the VLSI problem, this constraint ensures that exactly one circuit is positioned at each cell in a grid preventing any two circuits from overlapping in the same cell.

$$\begin{gathered}\forall i \in \{0 \dots h-1\}, \forall j \in \{0 \dots w-1\} : \\ exactly\_one(\{cell_{i,j,k} | k \in \{0 \dots n-1\}\})\end{gathered} \tag{1}$$

- **Valid Circuit Positioning**: This constraint ensures that each circuit is placed in a valid position on the plate, considering all possible starting positions and ensuring that the circuit does not exceed the plate's boundaries. The steps performed are listed here in detail:

  1. For each circuit k, the solver considers all possible starting positions on the plate where the circuit could be placed.
  2. for $x \in [0, \dots, h - chips\_h[k] + 1]$ and for $y \in [0, \dots, w - chips\_w[k] + 1]$ iterate over all possible starting positions (x, y) for circuit k on the plate.
  3. $h - chips\_h[k] + 1$ and $w - chips\_w[k] + 1$ ensure that the circuit does not exceed the plate's height (h) and width (w) limits.
  4. For each possible starting position (x, y), the solver creates a condition And([cells[x + i][y + j][k] for j in range(chips_w[k]) for i in range(chips_h[k])]). This condition checks that all cells that would be occupied by circuit k (given its dimensions chips_w[k] and chips_h[k]) are indeed occupied by that circuit.
  5. The solver adds an *at_least_one(possible_cells)* constraint, ensuring that at least one of the generated conditions for circuit k is true. In other words, at least one of the potential starting positions must be selected to place circuit k on the plate.

$$\bigwedge_{k=0}^{n-1} \left(\bigvee_{x=0}^{h-chips\_h[k]+1} \bigvee_{y=0}^{w-chips\_w[k]+1} \left(\bigwedge_{i=0}^{chips\_h[k]-1} \bigwedge_{j=0}^{chips\_w[k]-1} cells[x+i][y+j][k] \right) \right) \tag{2}$$

- **Relative position Constraint**: This constraint ensures that for each pair of circuits, at least a variable representing a relative position must be true.

$$\begin{gathered}\forall i \in [0, n-1] \forall j \in [i+1, n-1] \\ (\text{left}[i][j] \vee \text{left}[j][i] \vee \text{down}[i][j] \vee \text{down}[j][i])\end{gathered} \tag{3}$$

4.2 Implication Constraints

These constraints are used to enforce the relative positions of the circuits on the grid. They are used also to break symmetry in the solution space, which can significantly reduce the search space and improve the efficiency of the solver.

- **Leftmost Constraint**: This constraint is needed because if a circuit k is at the leftmost position of the board, then any other circuit l cannot be on its left. In other words, if circuit k is at the leftmost position, then $left[l]kl]$ must be False. The same for swapped indexes i.e. if l is at the leftmost position then $left[k][l]$ must be False.

- **Rightmost Constraint**: Like for the Leftmost, this constraint is necessary because if a circuit k is at the rightmost position of the board, then it must be to the right of any other circuit l. In other words, if circuit k is at the rightmost position, then $leftkl]lk]$ must be False. The same for swapped indexes i.e. if l is at the rightmost position then $left[l][k]$ must be False since it cannot be to the left of any other circuit.

$$\begin{aligned}&\forall k \in \{0, ..., n-1\}, \forall l \in \{k+1, ..., n-1\}, \forall i \in \{0, ..., h-1\} : \\ &cells[i][w-1][k] \vee \rightarrow left[l][k] \\ &\forall k \in \{0, ..., n-1\}, \forall l \in \{k+1, ..., n-1\}, \forall i \in \{0, ..., h-1\} : \\ &cells[i][w-1][k] \vee \rightarrow left[l][k]\end{aligned} \tag{4}$$

- **Topmost Constraint**: This constraint ensures that if a circuit k is at the topmost position of the board, then it must be above any other circuit l, that is to say, if circuit k is at the topmost position, then $down[k][l]$ must be False since k cannot be placed downward concerning any other circuit l. The same must hold with swapped indexes i.e. if l is placed at the topmost border of the plate then it cannot be placed downward any other circuit k.

- **Bottommost Constraint**: This constraint ensures that if a circuit k is at the bottommost position of the board, then it cannot be above any other circuit l. In other words, if circuit k is at the bottommost position, then $down[l][k]$ must be False.

$$\begin{aligned}&\forall k \in \{0, ..., n-1\}, \forall l \in \{k+1, ..., n-1\}, \forall j \in \{0, ..., w-1\} : \\ &cells[0][j][k] \rightarrow \neg down[l][k]\end{aligned} \tag{5}$$

$$\begin{aligned}&\forall k \in \{0, ..., n-1\}, \forall l \in \{k+1, ..., n-1\}, \forall j \in \{0, ..., w-1\} : \\ &cells[h-1][j][k] \rightarrow \neg down[k][l]\end{aligned} \tag{6}$$

4.3 Symmetry Breaking Constraints

These constraints are used to simplify the problem and improve the efficiency of the solver. In particular, we defined:

- **Priority Placement for Largest Circuit**: This ensures that the largest circuit (in terms of area) is placed first. By requiring the largest circuit to occupy the first cells in the grid, we eliminate all other possible positions for this circuit, thereby reducing the number of potential solutions that the solver needs to explore. This is a strategy to simplify the problem and improve the efficiency of the solver. More formally:
  1. After computing the area of each circuit, the largest circuit is identified by its height

     largest_c = np.argmax(areas)
  2. For each cell within the dimensions of the largest circuit (i.e., for each cell (i, j) where i is in the range of the height of the largest circuit and j is in the range of the width of the largest circuit), the solver adds a constraint:
     - If the circuit index k is equal to the index of the largest circuit $(largest\_c)$, the solver adds $cells[i][j][k]$, which means that the largest circuit occupies this cell.
     - Otherwise, the solver adds $Not(cells[i][j][k])$, which means that no other circuit occupies this cell.

$$\begin{aligned}
&\forall i \in \{0, \ldots, chips\_h[largest\_c]\},\\
&\forall j \in \{0, \ldots, chips\_w[largest\_c]\},\\
&\forall k \in \{0, \ldots, n-1\} : \begin{cases} cell_{i,j,k} & \text{if k = largest\_c} \\ \neg cell_{i,j,k} & \text{otherwise} \end{cases}
\end{aligned} \tag{7}$$

- **C5 - Lexicographic Ordering Constraint**: This constraint is a way to break symmetry since it ensures that, for each row of the grid, the circuits are placed with a left-to-right order. More precisely the constraint is saying: for each cell (i, j) in the grid, if circuit k is placed at that cell, then there should be at least one circuit l (where l is not equal to k) to the right of it i.e. at cell $(i, j + 1)$.

$$\begin{aligned}
&\forall i \in [0, h), \forall j \in [0, w-1), \forall k \in [0, n) :\\
&\text{cells}[i][j][k] \rightarrow \exists l \in [0, n), l \neq k : \neg\text{cells}[i][j+1][l]
\end{aligned} \tag{8}$$

5 Constraint for Rotation model

A second variant of the VLSI Design Problem is the case where the rotation of the circuits is allowed which means that an $n \times m$ circuit can be positioned either as it is or as $m \times n$. From a practical point of view, some constraints remained the same as the model without rotation while others were changed to take into account the rotation of the circuits. In particular:

- **Valid Circuit Positioning**: This constraint is modified to consider both rotated and non-rotated orientations for each circuit k. In particular, it ensures that each circuit k is placed in a valid position within the grid considering both rotated and non-rotated states of the circuit. The constraint enforces that at least one valid position (either rotated or non-rotated) is selected for each circuit, making sure the placement adheres to grid boundaries and circuit dimensions.
- **Implication Constraints: Leftmost, Rightmost, Topmost, Bottom-most**: These constraints are modified to consider the rotation of the circuits. For each pair of circuits (k, l), it checks if circuit k (or circuit l when indexes are swapped) is at the leftmost, rightmost, topmost, or bottom-most position in both rotated and non-rotated orientations, and updates the $left[k][l], left[l][k], down[k][l]$, and $down[l][k]$ variables accordingly.

It's important to notice that adding more constraints can make the problem harder to solve, so it's important to balance the complexity of the constraints with the efficiency of the solver. For this reason, we decided to remove the Lexicographic constraint for the rotation case since when rotation is allowed the search space increases consistently.

5.1 Output

The output for this second case is the same as the base model but it was added to each row a "Rot" flag whether the circuit has been rotated and a "NoRot" flag otherwise.

$$\begin{array}{lllll}
w & h & & & \\
n & & & & \\
x_0 & y_0 & p_{x_0} & p_{y_0} & NotRot \\
x_1 & y_1 & p_{x_1} & p_{y_1} & Rot \\
\ldots & & & &
\end{array}$$

6 Validation

6.1 Experimental Design and Search Strategy

To determine the optimal length of the plate, an iterative approach is employed, systematically increasing the height h of the plate until a satisfactory solution is found. The process begins by establishing a valid range of heights, defined by a lower bound min_h and an upper bound max_h as follows: :

$$\text{min\_h} = \frac{\sum_i h_i w_i}{w}$$

$$\text{max\_h} = \sum_i h_i$$

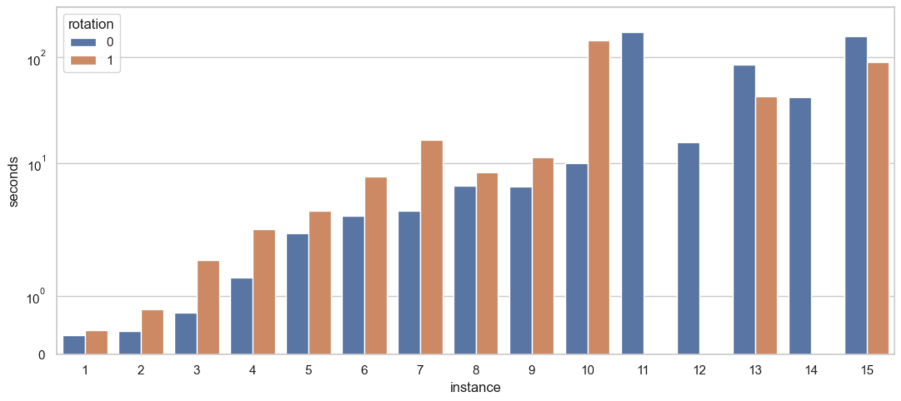
This ensures that h is constrained within the interval $[min\_h, max\_h]$. Subsequently, we start the iterative procedure by selecting $h = min\_h$ and we try to find a configuration for the selected h. If a satisfactory solution is found then h is the minimum possible length for the given instance, otherwise we increment $h = h + 1$ and repeat the process (this will lead to sub-optimal solutions). This is done $while \ \ h \leq max\_h$. Due to the increasing complexity of the instances, a timeout for the solver only, not considering the time spent in creating the constraints and adding them to the solver, was set to 300000 ms so that if the solving processes exceed this time limit are aborted. Moreover, *chips_w* and *chips_h* which represent respectively the widths and the lengths/heights of the circuits, are ordered by descending area which improves performances.

6.2 Experimental Results

The performance is summarized below and detailed further in Tables 5 and 6:

- 15/40 optimal solutions for the model which does not consider rotations.
- 12/40 optimal solutions for the model which considers rotations.

The execution times for the solver are presented in the histogram below. The overall results align with the increasing difficulty of instances, which progressively become numerically more complex leading to no optimal solutions within the set resolution time. Furthermore, employing a 3D array for problem encoding seems to yield a sub-optimal strategy for the given problem. The solver struggled to find a solution (sat or unsat) within a reasonable resolution time for the instances, indicating that the encoding approach might be causing prolonged convergence times.



input_name	**status**	**height**	**solve_time**
1	OPTIMAL	8.0	0.27937
2	OPTIMAL	9.0	1.11044
3	OPTIMAL	10.0	0.86703
4	OPTIMAL	11.0	1.58434
5	OPTIMAL	12.0	2.32738
6	OPTIMAL	13.0	4.07818
7	OPTIMAL	14.0	3.76301
8	OPTIMAL	15.0	6.10685
9	OPTIMAL	16.0	6.323
10	OPTIMAL	17.0	10.09613
11	OPTIMAL	18.0	177.32636
12	OPTIMAL	19.0	16.17026
13	OPTIMAL	20.0	88.16139
14	OPTIMAL	21.0	43.65875
15	OPTIMAL	22.0	162.99149
16	NO_SOLUTION		
17	NO_SOLUTION		
18	NO_SOLUTION		
19	NO_SOLUTION		
20	NO_SOLUTION		
21	NO_SOLUTION		
22	NO_SOLUTION		
23	NO_SOLUTION		
24	NO_SOLUTION		
25	NO_SOLUTION		
26	NO_SOLUTION		
27	NO_SOLUTION		
28	NO_SOLUTION		
29	NO_SOLUTION		
30	NO_SOLUTION		
31	NO_SOLUTION		
32	NO_SOLUTION		
33	NO_SOLUTION		
34	NO_SOLUTION		
35	NO_SOLUTION		
36	NO_SOLUTION		
37	NO_SOLUTION		
38	NO_SOLUTION		
39	NO_SOLUTION		
40	NO_SOLUTION		

Table 5: Z3 No Rotation Model

input_name	**status**	**height**	**solve_time**
1	OPTIMAL	8.0	0.42126
2	OPTIMAL	9.0	0.77162
3	OPTIMAL	10.0	1.60842
4	OPTIMAL	11.0	2.38085
5	OPTIMAL	12.0	3.57513
6	OPTIMAL	13.0	7.45374
7	OPTIMAL	14.0	16.56333
8	OPTIMAL	15.0	8.18642
9	OPTIMAL	16.0	11.46662
10	OPTIMAL	17.0	144.76537
11	NO_SOLUTION		
12	NO_SOLUTION		
13	OPTIMAL	20.0	43.20479
14	NO_SOLUTION		
15	OPTIMAL	22.0	89.81267
16	NO_SOLUTION		
17	NO_SOLUTION		
18	NO_SOLUTION		
19	NO_SOLUTION		
20	NO_SOLUTION		
21	NO_SOLUTION		
22	NO_SOLUTION		
23	NO_SOLUTION		
24	NO_SOLUTION		
25	NO_SOLUTION		
26	NO_SOLUTION		
27	NO_SOLUTION		
28	NO_SOLUTION		
29	NO_SOLUTION		
30	NO_SOLUTION		
31	NO_SOLUTION		
32	NO_SOLUTION		
33	NO_SOLUTION		
34	NO_SOLUTION		
35	NO_SOLUTION		
36	NO_SOLUTION		
37	NO_SOLUTION		
38	NO_SOLUTION		
39	NO_SOLUTION		
40	NO_SOLUTION		

Table 6: Z3 Rotation Model

7 SMT Model

Satisfiability Modulo Theory (SMT) refers to the study of the satisfiability of formulas based on certain background theories. In our case, we test the model by considering the LIA (Linear Integer Arithmetic) logic, which is a perfect fit with our implementation.

There are several solvers available that can be used to run the model, and each solver typically has its specific language in which you can write. However, we chose to implement the model using SMT-LIB, a standard library for encoding SMT problems. This allowed us to create a solver-independent model that different solvers can run without modifying the code. We tested the model of the most popular solver, namely **Z3**, using the Python package PYSMT. This package enables the installation of different solvers.

7.1 Decision variables

The first step in the implementation process is to choose the variables necessary for the encoding. Similar to the CP implementation, we must define a variable called **l**, which is the height to minimize. We also define variables x_i and y_i for each circuit, which represent the coordinates of the bottom left point. Additionally, we consider variables **N** and **W** which stand for the number of circuits and the plate width, respectively. Finally, we calculate the remaining variables:

- $pos_x[i]$: Represents the x-coordinate of the circuit i's placement on the rectangular plate. The domain is

$$pos_x[i] \in [0, plate\_width - circuit\_width]$$

- $pos_y[i]$: Represents the y-coordinate of the circuit i's placement on the rectangular plate. The domain is

$$pos_y[i] \in [0, plate_height - plate_width]$$

- $rotation[i]$: Represents whether circuit i is rotated. Domain is

$$rotation[i] \in 0, 1$$

It is important to provide a lower and upper bound to the height l since we need to minimize it. A reasonable lower bound is given by the following equation:

$$l_{low} = max(max(heights), lower(\frac{\sum_{i=1}^{N} widths_i * heights_i}{W})) \tag{9}$$

In the optimal case, there is no space between the circuits. Therefore, if we have a very large circuit, it is given the minimum lower bound, which is the height of that circuit. We also consider an upper bound, which is simply the sum of all heights since, in the worst case, all circuits need to be stacked. The equation for the upper bound is as follows:

$$l_{up} = \sum_{i=1}^{N} heights_i \tag{10}$$

7.2 Objective function

The objective function for the SMT (Surface Mount Technology) model is the same as the CP (Constraint Programming) model, which aims to minimize the total area occupied by the circuits on the plate.

7.3 Constraints

In the SMT model, we have two main constraints that prevent the circuits from exiting the available plate and overlapping with each other. The first constraint consists of the following inequalities:

- $pos_x[i] + circuit\_width <= plate\_width, \forall i \in CIRCUITS$

- $pos_y[i] + circuit\_height <= plate\_height, \forall i \in CIRCUITS$

The second constraint concerns non-overlap and includes the following inequalities:

- $pos_x[i] + circuit\_width <= pos_x[j] \vee pos_x[j] + circuits\_width <= pos_x[i]$: i is left of j, or j is left of i
- $pos_y[i] + circuit\_height <= pos_y[j] \vee pos_y[j] + circuits\_height <= pos_y[i]$: i is below j, or j is below i

We use an OR between the inequalities because we want at least one of the inequalities to be satisfied for each pair of circuits.

7.3.1 Symmetry Breaking Constraint

We have added symmetry-breaking constraints to reduce the search space and eliminate symmetric solutions. We identified three potential symmetries:

- **Horizontal symmetry**: inverting the position of the circuits from left to right or vice-versa;
- **Vertical symmetry**: we invert the position of the circuits from top to bottom or vice-versa;
- **Same size**

Although we tried using global constraints in the CP implementation, we found that it decreased model capability. As a result, we decided to place the circuit with the bigger area at position (0,0) on the plate, using the constraint

$$pos_x[i] <= pos_x[j], \forall i, j \in CIRCUITS where i < j$$

7.4 Rotation

We are considering a model that allows the different circuits to be rotated. In this case, we need additional variables for each circuit. The constraints are logically the same as in the standard model, except that the circuit dimensions provided in the input are not considered. To address this issue, we introduce two arrays that store each circuit's actual heights and widths, which are related to the rotation array. Specifically:

$$\forall i \in 1..n$$
$$rot_{x_i} = y_i if rotation_i, else x_i$$
$$rot_{y_i} = x_i is rotation_i, else y_i$$

Here, rot_x and rot_y represent the actual dimensions of the circuit, while x and y represent the dimensions of the circuit as provided in the input.

7.5 Constraints

To limit the search space, we can set the variable $rot_i = false$ for all circuits with a height greater than the maximum width. This is because the solver cannot rotate a circuit that doesn't adhere to the boundary constraints on the width.

$$(heights_i > W) \implies rot_i = false, \forall i \in [0, N]$$

Also, we can break the symmetry for circuits that have the same width and height. This is because rotating a square does not help in solving the problem.
Therefore,

$$(height_i == width_i) \implies rot_i = false, \forall i \in [0, N]$$

7.6 Validation

As mentioned earlier, the SMT-LIB does not support the optimization of a target function. Therefore, we had to implement the search for the optimal solution manually. The algorithm we chose was **Lower Bound Search**.

7.6.1 Lower Bound Search

This algorithm is simple and effective. We know that the lowest possible height is the best solution, so we start by looking for the constraint l==*lower_bound*. If this constraint exists, it is the optimal solution and we don't need to look any further. If the solver cannot find a solution, we increase the lower_bound and start the search again, removing the previous constraint from the assertion stack. This helps to ensure that we find the best possible solution.

input_name	**status**	**height**	**time_solved**
ins-1	OPTIMAL	8	0.0003385109994269
ins-2	OPTIMAL	9	0.0007289810000656
ins-3	OPTIMAL	10	0.000883511000211
ins-4	OPTIMAL	11	0.0033977370003412
ins-5	OPTIMAL	12	0.0132729270007985
ins-6	OPTIMAL	13	0.0043375690001994
ins-7	OPTIMAL	14	0.0095552099992346
ins-8	OPTIMAL	15	0.0101603910006815
ins-9	OPTIMAL	16	0.008595236999099143
ins-10	OPTIMAL	17	0.0365536970002722
ins-11	OPTIMAL	18	0.1667628340001101
ins-12	OPTIMAL	19	0.0873612460000003
ins-13	OPTIMAL	20	0.0450839540008018
ins-14	OPTIMAL	21	0.1477774750001117
ins-15	OPTIMAL	22	0.1389200879984855
ins-16	OPTIMAL	23	3.785524204999092
ins-17	OPTIMAL	24	0.759396788000231
ins-18	OPTIMAL	25	0.6179904279997572
ins-19	OPTIMAL	26	5.499112034000063
ins-20	OPTIMAL	27	0.706220387999565
ins-21	OPTIMAL	28	9.503089869998805
ins-22	OPTIMAL	29	105.6708004709999
ins-23	OPTIMAL	30	1.6778666819991486
ins-24	OPTIMAL	31	0.2031870529990556
ins-25	OPTIMAL	32	80.84033408100004
ins-26	OPTIMAL	33	3.5868398279999383
ins-27	OPTIMAL	34	6.296135294000123
ins-28	OPTIMAL	35	3.642826113999035
ins-29	OPTIMAL	36	1.47080428299887
ins-30	NO_SOLUTION		
ins-31	OPTIMAL	38	1.84785823600032
ins-32	NO_SOLUTION		
ins-33	OPTIMAL	40	0.9297758950015124
ins-34	OPTIMAL	40	4.084766218000368
ins-35	OPTIMAL	40	4.583545811001386
ins-36	OPTIMAL	40	7.4991591099988
ins-37	OPTIMAL	60	47.91743978900013
ins-38	NO_SOLUTION		
ins-39	OPTIMAL	60	47.11924432399974
ins-40	NO_SOLUTION		

Table 7: Z3 No Rotation Model

input_name	**status**	**height**	**time_solved**
ins-1	OPTIMAL	8	0.0014062779991945
ins-2	OPTIMAL	9	0.0054308139988279
ins-3	OPTIMAL	10	0.01250545700168
ins-4	OPTIMAL	11	0.019423731999268
ins-5	OPTIMAL	12	0.1851721230013936
ins-6	OPTIMAL	13	0.09892374699848
ins-7	OPTIMAL	14	0.2760268569982145
ins-8	OPTIMAL	15	0.5271182920005231
ins-9	OPTIMAL	16	0.0574279700012994
ins-10	OPTIMAL	17	0.4482730159979837
ins-11	OPTIMAL	18	21.940824602999783
ins-12	OPTIMAL	19	22.55469398800051
ins-13	OPTIMAL	20	1.0333040190016618
ins-14	OPTIMAL	21	1.7273788950005835
ins-15	OPTIMAL	22	27.500049989001127
ins-16	OPTIMAL	23	201.19836770799884
ins-17	OPTIMAL	24	64.33261134799977
ins-18	OPTIMAL	25	119.97437033899767
ins-19	NO_SOLUTION		
ins-20	OPTIMAL	27	179.69288826799675
ins-21	NO_SOLUTION		
ins-22	NO_SOLUTION		
ins-23	NO_SOLUTION		
ins-24	OPTIMAL	31	41.98867479600085
ins-25	NO_SOLUTION		
ins-26	OPTIMAL	33	81.19450930300081
ins-27	OPTIMAL	34	219.31658997799968
ins-28	NO_SOLUTION		
ins-29	OPTIMAL	36	297.99495755300086
ins-30	NO_SOLUTION		
ins-31	OPTIMAL	38	39.736316375998285
ins-32	NO_SOLUTION		
ins-33	OPTIMAL	40	69.62622167999871
ins-34	OPTIMAL	40	86.49462557199877
ins-35	OPTIMAL	40	42.87163828499979
ins-36	OPTIMAL	40	28.241105464996508
ins-37	NO_SOLUTION		
ins-38	NO_SOLUTION		
ins-39	NO_SOLUTION		
ins-40	NO_SOLUTION		

Table 8: Z3 Rotation Model

7.7 Experimental Results

In the Base Model in Table 7, we have observed that the solve times are generally low. The instances range from microseconds to a few seconds. We have noticed that the solution time gradually increases with the growth in problem size (height). However, one instance, ins-16, shows a substantial spike in solve time compared to its predecessors. This indicates a potential challenge for the model in handling larger instances efficiently.

On the other hand, the Rotation Model in table 8 demonstrates a more varied and, in some cases, significantly higher solve times. Instances ins-16, ins-19, and ins-28, in particular, show substantial increases in solve time compared to the No Rotation Model. This suggests that the introduction of rotation in the model may lead to increased computational complexity for certain instances.

Comparing the two models, it is evident that the Z3 Rotation Model introduces additional capabilities. However, it comes at the cost of increased solve times for specific instances. Therefore, the decision to use rotation in the model should be carefully considered, weighing the benefits against the potential performance trade-offs.

Unfortunately due to problems with the pysmt library, it was not possible to run the problem using the **CVC4** solver.

8 MIP Model

Mixed-integer programming (MIP) is a mathematical optimization technique used for solving optimization problems where the decision variables can also take discrete (integer) values. MIP can be suitable for VLSI problems due to its ability to handle discrete decisions (e.g., component placement, routing), resource allocation, optimization of layouts, timing constraints, and combinatorial aspects.

The solver we chose is **Gurobi**, a highly efficient optimization solver used for solving linear programming (LP), mixed-integer programming (MIP), quadratic programming (QP), and other mathematical optimization problems. It is known for its exceptional performance, advanced algorithms, and ability to handle large and complex optimization models: it's a popular choice among practitioners in diverse industries for finding optimal solutions to a wide range of optimization challenges.

It also comes with its official Python API, gurobipy, allowing us to create models, set objectives, define constraints, and interact with the solver, all within a Python environment.

8.1 Problem Parameters

We start from the usual parameters given with each instance:

- w: the width of the silicon plate;
- n: the number of circuits to place within the plate;
- (w_i, h_i): width and height of the i^{th} circuit.

from which we can derive the bounds for the total $area$ of the plate and the ones for its height (h).

8.2 Decision Variables

As done before, the first step in the modelling process is to define the decision variables that will be needed for the encoding:

- x_i: which again represents the x coordinate of the bottom left corner of each circuit i. In this case, it will be an integer variable.

$$x_i \in [0, w - \min(w_i)]$$

- y_i: represents the y coordinate of the bottom left corner of each circuit i. It's again an integer variable.

$$y_i \in [0, h\_max - \min(h_i)]$$

- h: the height of the plate, modelled as an integer variable. It's the objective of the model.

$$h \in [h\_min, h\_max]$$

- $\delta_{i,j,k}$: are binary variables that will be used to implement the non-overlapping constraint through the *big-M* method. Defined one for every tuple of circuits (i, j), with $k \in [1, 4]$, one for each of the inequalities that will be shown later.

8.3 Objective Function

As we saw before, the objective function aims at minimizing the total area of the plate. Since the width w is given in the instance, and the area of a rectangle is $w * h$, the objective reduces to minimizing the height, h, defined as a decision variable.

8.4 Constraints

These are essentially constraints that concern the correct positioning of the circuits on the plate, guaranteeing the boundaries and their non-overlap property.

- **Area Boundaries**:
  - $w * h \geq area\_min$
  - $w * h \leq area\_max$

  These two constraints ensure the total area stays between the boundaries we derived.

- **Not exceeding dimensions of the plate**:
  - $x_i + w_i \leq w$
  - $y_i + h_i \leq h$

  with $i \in [1, ..., n]$. These constraints ensure the circuits placed don't exceed the dimensions of the plate.

- **No overlap**: to implement this important constraint that ensures two blocks are not overlapping when placed on the plate, we exploited the *big-M* method. There are four cases to consider when placing two rectangles i and j: $x_i + w_i \leq x_j$, $x_j + w_j \leq x_i$, $y_i + h_i \leq y_j$ and $y_j + h_j \leq y_i$. From the above, the "no overlap" constraint can be formulated as:

  $$x_i + w_i \leq x_j \vee x_j + w_j \leq x_i \vee y_i + h_i \leq y_j \vee y_j + h_j \leq y_i$$

  The "or" condition can be modelled with the help of binary variables δ and *big-M* constraints:

  - $x_i + w_i \leq x_j + M_1 * \delta_{i,j,1}$
  - $x_j + w_j \leq x_i + M_2 * \delta_{i,j,2}$
  - $y_i + h_i \leq y_j + M_3 * \delta_{i,j,3}$
  - $y_j + h_j \leq y_i + M_4 * \delta_{i,j,4}$
  - $\sum_k \delta_{i,j,k} \leq 3$

The binary variables $\delta_{i,j,k}$ make sure at least one of the constraints is not relaxed. Moreover, to keep the constants M_k as small as possible, we can see that considering our rectangular plate of width w and height h, $M_1 = M_2 = w$ and $M_3 = M_4 = h$

8.4.1 Symmetry breaking constraints

As did for other models, to break symmetry without decreasing our MIP model performances, we imposed the circuit with the bigger area to be at position $(0, 0)$ on the plate:

- $x_B = 0$
- $y_B = 0$

with B being the largest area circuit among the ones in the instance.

8.5 Rotation

In this variation of the problem circuits can be rotated: this means that the height and width of each circuit can be swapped. ù

8.5.1 Decision Variables

To account for this, a new decision variable rot_i has been added:

- rot_i: a binary variable for each circuit i that is either 0 if the circuit is not rotated, 1 otherwise.

8.5.2 Constraint Reductions

As a constraint reduction form to reduce the search space, we impose the binary variables rot_i to be 0 for our square blocks (i.e. the blocks i where $w_i = h_i$)

- $rot_i = 0$ if $w_i = h_i$

In this way we prevent useless rotations, since the two dimensions of square blocks are the same and swapping them would be meaningless.

8.5.3 Constraints

Two of the aforementioned constraints have to be modified to take into account the rotations of the blocks. In the specific:

- **Not exceeding dimensions of the plate**:
  - $x_i + (h_i * rot_i + w_i * (1 - rot_i)) \leq w$
  - $y_i + (w_i * rot_i + h_i * (1 - rot_i)) \leq h$
- **No overlap**:

- $x_i + (h_i * rot_i + w_i * (1 - rot_i) \leq x_j + M_1 * \delta_{i,j,1}$
- $x_j + (h_i * rot_i + w_i * (1 - rot_i) \leq x_i + M_2 * \delta_{i,j,2}$
- $y_i + (w_i * rot_i + h_i * (1 - rot_i) \leq y_j + M_3 * \delta_{i,j,3}$
- $y_j + (w_i * rot_i + h_i * (1 - rot_i) \leq y_i + M_4 * \delta_{i,j,4}$
- $\sum_k \delta_{i,j,k} \leq 3$

In this way we can represent a conditional selection between w_i and h_i based on the binary variable rot_i:

$$f(\text{rot}_i) = \begin{cases} w_i & \text{if rot}_i = 1 \\ h_i & \text{if rot}_i = 0 \end{cases}$$

8.6 Validation

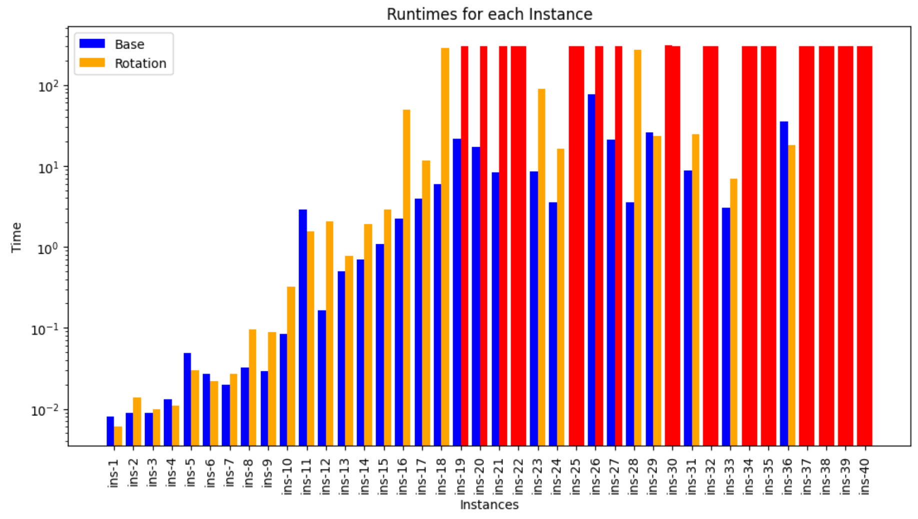
8.6.1 Experimental Design

The first implementation tried was through the *PuLP* library of Python and the *GLPK* solver, and it led to a slow solution even of the first, more simple, instances. We switched to *Gurobi* and *gurobipy* without further tests on the first implementation, and it helped since the beginning both in achieving more clarity from a coding perspective and way better runtimes of the solving process.

Adding the symmetry-breaking constraints helped a lot too, allowing us to solve more instances within the time limit (6 more not-rotated instances solved). Also, considering the rotation model, imposing the square blocks to not be rotated slightly lowered the respective instances' runtimes.

8.6.2 Experimental Results

The following histogram helps us compare the runtimes of each instance: blue bars are relative to models without rotation, and orange bars to models with rotation. Red bars highlight the instances that didn't have an optimal solution in the 5-minute limit.



The following tables instead show in detail the run of each instance, with the status of the solution, the runtime and the value of the objective found, both for the base and the rotation models.

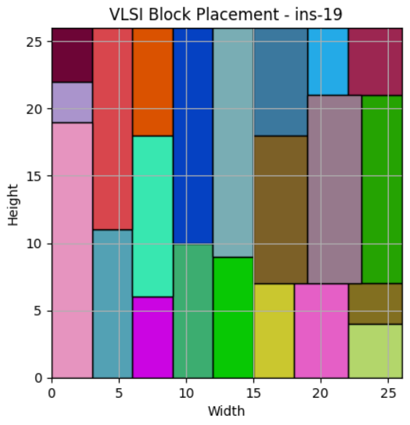
input_name	**status**	**solve_time**	**height**
ins-1	OPTIMAL	0.00800013542175293	8.0
ins-2	OPTIMAL	0.009000062942504883	9.0
ins-3	OPTIMAL	0.009000062942504883	10.0
ins-4	OPTIMAL	0.013000011444091797	11.0
ins-5	OPTIMAL	0.04900002479553223	12.0
ins-6	OPTIMAL	0.026999950408935547	13.0
ins-7	OPTIMAL	0.019999980926513672	14.0
ins-8	OPTIMAL	0.032000064849853516	15.0
ins-9	OPTIMAL	0.029000043869018555	16.0
ins-10	OPTIMAL	0.08500003814697266	17.0
ins-11	OPTIMAL	2.867000102996826	18.0
ins-12	OPTIMAL	0.1640000343322754	19.0
ins-13	OPTIMAL	0.5039999485015869	20.0
ins-14	OPTIMAL	0.6979999542236328	21.0
ins-15	OPTIMAL	1.0949997901916504	22.0
ins-16	OPTIMAL	2.2060000896453857	23.0
ins-17	OPTIMAL	3.9030001163482666	24.0
ins-18	OPTIMAL	5.978000164031982	25.0
ins-19	OPTIMAL	21.54099988937378	26.0
ins-20	OPTIMAL	17.35199999809265	27.0
ins-21	OPTIMAL	8.299999952316284	28.0
ins-22	TIME LIMIT	-	30.0
ins-23	OPTIMAL	8.462000131607056	30.0
ins-24	OPTIMAL	3.5439999103546143	31.0
ins-25	TIME LIMIT	-	33.0
ins-26	OPTIMAL	77.43099999427795	33.0
ins-27	OPTIMAL	21.14800000190735	34.0
ins-28	OPTIMAL	3.5279998779296875	35.0
ins-29	OPTIMAL	25.79099988937378	36.0
ins-30	TIME LIMIT	-	38.0
ins-31	OPTIMAL	8.775999784469604	38.0
ins-32	TIME LIMIT	-	40.0
ins-33	OPTIMAL	3.00600004196167	40.0
ins-34	TIME LIMIT	-	41.0
ins-35	TIME LIMIT	-	41.0
ins-36	OPTIMAL	35.46500015258789	40.0
ins-37	TIME LIMIT	-	62.0
ins-38	TIME LIMIT	-	62.0
ins-39	TIME LIMIT	-	61.0
ins-40	TIME LIMIT	-	106.0

Table 9: Gurobi No Rotation Model

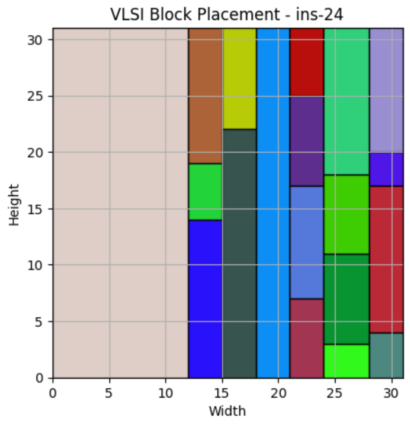
input_name	**status**	**solve_time**	**height**
ins-1	OPTIMAL	0.006000041961669922	8.0
ins-2	OPTIMAL	0.01399993896484375	9.0
ins-3	OPTIMAL	0.009999990463256836	10.0
ins-4	OPTIMAL	0.01100015640258789	11.0
ins-5	OPTIMAL	0.029999971389770508	12.0
ins-6	OPTIMAL	0.02200007438659668	13.0
ins-7	OPTIMAL	0.026999950408935547	14.0
ins-8	OPTIMAL	0.0970001220703125	15.0
ins-9	OPTIMAL	0.08899998664855957	16.0
ins-10	OPTIMAL	0.31999993324279785	17.0
ins-11	OPTIMAL	1.566999912261963	18.0
ins-12	OPTIMAL	2.0879998207092285	19.0
ins-13	OPTIMAL	0.7840001583099365	20.0
ins-14	OPTIMAL	1.8959999084472656	21.0
ins-15	OPTIMAL	2.9070000648498535	22.0
ins-16	OPTIMAL	49.69600009918213	23.0
ins-17	OPTIMAL	11.66700005531311	24.0
ins-18	OPTIMAL	288.97900009155273	25.0
ins-19	TIME LIMIT	-	27.0
ins-20	TIME LIMIT	-	28.0
ins-21	TIME LIMIT	-	29.0
ins-22	TIME LIMIT	-	30.0
ins-23	OPTIMAL	89.66200017929077	30.0
ins-24	OPTIMAL	16.131999969482422	31.0
ins-25	TIME LIMIT	-	33.0
ins-26	TIME LIMIT	-	34.0
ins-27	TIME LIMIT	-	35.0
ins-28	OPTIMAL	271.91100001335144	35.0
ins-29	OPTIMAL	23.634000062942505	36.0
ins-30	TIME LIMIT	-	38.0
ins-31	OPTIMAL	24.82699990272522	38.0
ins-32	TIME LIMIT	-	40.0
ins-33	OPTIMAL	6.876999855041504	40.0
ins-34	TIME LIMIT	-	41.0
ins-35	TIME LIMIT	-	41.0
ins-36	OPTIMAL	17.896000146865845	40.0
ins-37	TIME LIMIT	-	61.0
ins-38	TIME LIMIT	-	62.0
ins-39	TIME LIMIT	-	61.0
ins-40	TIME LIMIT	-	120.0

Table 10: Gurobi Rotation Model

By reading those results, we can confirm the trend of seeing increasing runtimes as the complexity of the problem grows. Gurobi plays a strong role in managing to solve some complex instances within the imposed 300s time limit. The additional variables in the rotation model may lead to increased complexity of the overall model and eventually to longer runtimes, as we can see when comparing the solving times of most of the instances (e.g. ins-16, ins-18, ins-28, where we encounter larger differences).



(a) Solution of instance 19



(b) Solution of instance 24

By visually examining the solutions, we can see that the presence of a block having way bigger dimensions than the other blocks helps in solving faster runtimes: by comparing for example instances 19 and 24, we go from a 21.5s runtime to a 3.5s one despite the larger number of circuits. Thanks to the symmetry-breaking constraint, the biggest block is placed first, leaving less area on the plate to position the remaining blocks and eventually reducing the search space.

9 Conclusion

At the end of each section, we have reported the best results achieved by each technique. With regards to Constraint Programming, we found it to be the easiest technique to implement and also capable of producing optimal results in a significant number of instances. To improve our implementation, we could test and compare it with other solvers such as CPLEX. As for SMT, our implementation is not solver-dependent because we used the pysmt library. Other solvers, apart from Z3, could be used to run our models as long as they are supported by the library. However, we faced some challenges while installing other solvers since pysmt is still in development. We intended to compare our implementation with CVC4 but were unable to install it on our machine. Despite this, SMT produced acceptable results and was able to solve a high number of instances. In the realm of SAT solving, we wanted to use a different approach from the SMT one. However, the encoding choice led to a sub-optimal encoding strategy for the given problem. This approach proved to be less effective in capturing the problem's nuances and constraints than the one used in the SMT problem. As for the MIP part, we didn't use a non-solver-dependent implementation, trying to fully exploit the power of the Gurobi solver. VLSI problems often involve decisions that are naturally expressed as integers, such as the number of modules and the location of components: MIP seems effective in handling integer constraints and searching for optimal integer solutions, and we reach a good amount of optimal solutions within the time limit (30 solved with the base model, 25 with the rotation ones) but still isn't the best performing approach among us: maybe due to the mainly combinatorial nature of the task, the SMT model outperforms all the other approach we tried.