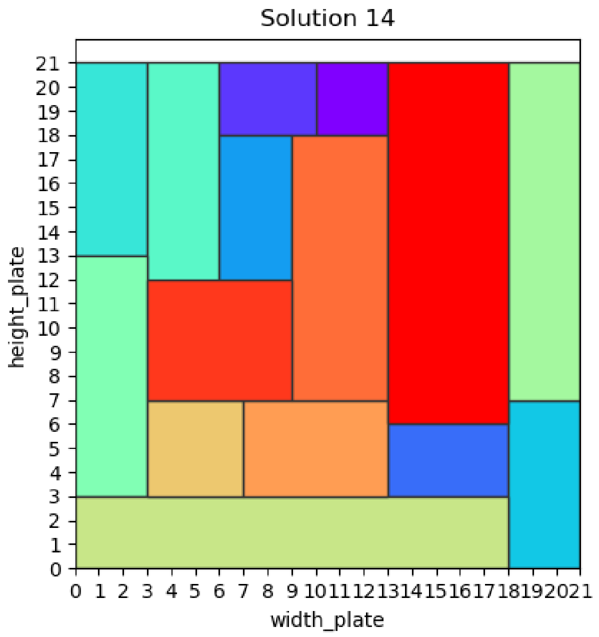
VLSI CDMO

Sumit Kumar Mishra sumitkumar.mishra@studio.unibo.it
Davide Perozzi davide.perozzi@studio.unibo.it

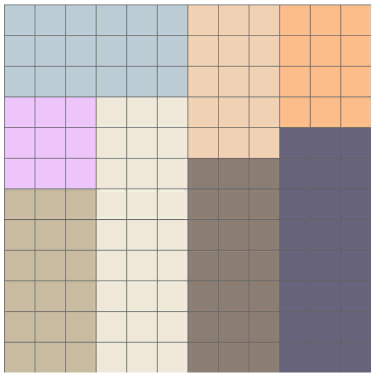
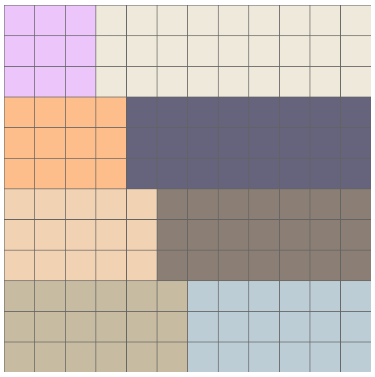
May 2023



Contents

1 Introduction

Very Large Scale Integration (VLSI) is the process of integrating electronic circuits within silicon chips. The primary objective of this project is to efficiently place a specified quantity of circuits onto a single chip with a fixed width, while minimizing the chip's overall height.
This report delves into the development and analysis of four distinct implementations. We present a Constrain Programming (CP) Model, a Satisfiability (SAT) Model, a Satisfiability Modulo Theories (SMT) Model, and a Mixed-Integer Programming (MIP) Model as potential approaches to address this problem. Each of these models offers unique methods and insights for solving the circuit placement challenge effectively.
We split the work based on our development preferences and personal commitments, we were both pursuing an internship while doing this work. This has brought many difficulties to the pace of development, but in spite of everything we are happy with the work achieved.

Here is a sample of what we achieved on a same instance using rotation and without rotation.

2 CP Model

Constraint Programming is a programming technique that is focused on solving problems by specifying the set of constraints on a possible solution, rather than writing an algo to compute the solution.
It is a declarative programming approach that allows you to express a problem in terms of constraints and variables. Then, the system automatically searches for solutions that satisfy those constraints.
We utilized Minizinc as the framework for our project, where we incorporated our model. To assess its performance, we executed it with two distinct constraint solvers available in the Minizinc binary distribution: Gecode and Chuffed.

2.1 Decision variables

First, we need to decide the variables that we need for the encoding and then we will define their domains and the relations with the constants and the other unknowns.

Taking input, a .dzn file containing:
1. W, the width of the board where the circuit must be placed.
2. N, the number of circuits to be placed.
3. The dimensions of the circuit (m x n) i.e., width(m) and height(n) of the circuit.

So, till now the variables that we got are- W, N, m and n.

As VARIABLEs we decided to create 2 arrays X and Y, that contain the x and y coordinates of the circuits and the var H which represents the height of the board on which the circuits will be installed.

The variable in array X, can have the range between 0 and W – min(m), which means that the largest allowed coordinate is when the circuit with the lowest width is placed next to the right side of the plate. This is because we want to minimize the amount of empty space on the plate, so we want to place circuits as close to the right side as possible.

$$0 \leq X[i] \leq W - \min(m)$$

Similarly, the statement explains that the variables in the array Y, can assume values between 0 and H - min(n). This is because we want to minimize the height of the plate, so we want to place circuits as close to the bottom of the plate as possible. The largest allowed coordinate in the y direction is when the circuit with the lowest height is placed at the bottom of the plate.

$$0 \leq Y[i] \leq H - \min(n)$$

The variable 'H' is constrained to be within the range from 'min_height' to 'max_height'. The lower bound 'min_height' is calculated as the maximum value between the maximum height of any circuit. The upper bound 'max_height' is the sum of the heights of all the circuits.

$$\text{min\_height} = \max(\max(n), \lceil \frac{\sum_{i=1}^{N}(m[i] \times n[i])}{W} \rceil)$$

$$\text{max\_height} = \sum_{i=1}^{N} n[i]$$

2.2 Objective function

The objective is to minimize the value of the variable H, which represents the height of the plate. The solver will try to find the smallest possible value for H that satisfies all the constraints.

By minimizing the variable 'H' using the 'minimize' statement, the solver tries to find the minimum possible value for 'H', which corresponds to the minimum height of the plate required to accommodate all the circuits without overlapping.

When the solver completes, it will provide the optimal value for H, which represents the minimum height of the plate that satisfies all the constraints.

2.3 Constraints

Now, let's move towards the constraints of the given problem:

1. **Circuit placements must not be overlapping.** This constraint ensures that for any two circuits i and j, they do not overlap in either the x or y direction.

$$\forall i, j \in 1, 2, \ldots, N, i \neq j :$$
$(X[i]+W[i] \leq X[j]) \vee (X[j]+W[j] \leq X[i]) \vee (Y[i]+H[i] \leq Y[j]) \vee (Y[j]+H[j] \leq Y[i])$

2. **All circuits must fit the on the board.** This constraint ensures that for every circuit i, its placement along the x and y directions does not exceed the dimensions of the board.

$$\begin{aligned} &\forall i \in 1, 2, \ldots, N : \\ &X[i] + W[i] \leq W \quad \text{(Width constraint)} \\ &Y[i] + H[i] \leq H \quad \text{(Height constraint)} \end{aligned}$$

Together, these constraints ensure that the circuit placements are feasible and do not overlap, while also ensuring that all circuits fit within the dimensions of the board.

2.3.1 Symmetry Breaking

VLSI designs can have multiple symmetries:

1. **Reflection over the X-axis:** For any two circuits 'i' and 'j', if their y-coordinates are the same, then we can break the symmetry by requiring that of x-coordinate of i is less than the x-coordinate of j. In easy words, A reflection over the x-axis involves flipping an image or object upside down, such that the points above the x-axis become points below the x-axis, and vice versa. In the context of the VLSI design problem, a reflection over the x-axis can occur when the circuits are flipped vertically, which will result in the same circuit placement, but with the height of the plate inverted.

$$\forall i, j \in \{1, 2, \ldots, N\}, i \neq j : \quad (Y[i] = Y[j]) \Rightarrow (X[i] < X[j])$$

2. **Reflection over the Y-axis:** For any two circuits 'i' and 'j', if their x-coordinates are the same, then we can break the symmetry by requiring that the y-coordinate of 'i' is less than the y-coordinate of 'j'. In easy words, A reflection over the y-axis involves flipping an image or object horizontally, such

that the points to the left of the y-axis become points to the right of the y-axis, and vice versa. In the context of the VLSI design problem, a reflection over the y-axis can occur when the circuits are flipped horizontally, which will result in the same circuit placement, but with the width of the plate inverted.

$$\forall i, j \in \{1, 2, \ldots, N\}, i \neq j : \quad (X[i] = X[j]) \Rightarrow (Y[i] < Y[j])$$

3. **Circuits with the same dimensions:** We can break the symmetry by requiring that the circuits are placed in lexicographic order, i.e., the circuits are ordered first by their widths, and then by their heights. In easy words, If two or more circuits have the same dimensions, they can be interchanged with each other without affecting the overall layout. For example, if there are two circuits with width 2 and height 3, they can be swapped with each other without any impact on the layout. Therefore, this symmetry can be exploited to reduce the search space and improve the efficiency of the search algorithm.

$$\forall i, j \in \{1, 2, \ldots, N\}, i \neq j : \quad (W[i] < W[j]) \vee ((W[i] = W[j]) \wedge (H[i] < H[j]))$$

2.4 Rotation

In the previous model where the circuits were not rotated, we assumed that the width and height of each circuit were fixed as given in the input. However, when we allow circuit rotation, the width, and height of each circuit can change depending on whether the circuit is rotated or not.

2.4.1 Variables when Rotation is Allowed

1. The variable 'H' that was constrained within the range from 'min_height' to 'max_height'. The Lower bound would 'min_height' will be the same but the upper bound 'max_height' will between the maximum value between the sum of the heights of circuits and the sum of the width of the circuit.

int: max_height = sum([max(n[i], m[i]) — i in 1..N]);

Wwe need to introduce some additional arrays to handle this situation

2. The first array is a Boolean array called **'rot'**, which stores 'true' for each circuit that is rotated and 'false' for each circuit that is not rotated. This array will help us keep track of which circuits are rotated.

$$\text{rot}[i] = \begin{cases} \text{true}, & \text{if circuit } i \text{ is rotated;} \\ \text{false}, & \text{if circuit } i \text{ is not rotated.} \end{cases}$$

3. And **'w_real'** and **'h_real'** arrays contain the actual width and height of each circuit after rotation, respectively. If a circuit is rotated, then its width becomes equal to its original height, and its height becomes equal to its original

width. These arrays will help us update the width and height of the circuits when they are rotated.

$$w_{\text{real}}[i] = \begin{cases} h[i], & \text{if circuit } i \text{ is rotated;} \\ w[i], & \text{if circuit } i \text{ is not rotated.} \end{cases}$$

$$h_{\text{real}}[i] = \begin{cases} w[i], & \text{if circuit } i \text{ is rotated;} \\ h[i], & \text{if circuit } i \text{ is not rotated.} \end{cases}$$

These additional arrays allow us to consider circuit rotation in our model and find the most efficient placement of circuits on the chip.

2.4.2 Constraints when Rotation is Allowed

New Constraints will be added because of the rotation of circuits.

1. **Constraints to avoid infeasible rotations:** This constraint aims to avoid infeasible rotations by preventing a circuit from being rotated if its width exceeds the maximum width (W). It uses the logical condition "if n[i] ¿ W" to check if the width of circuit i is greater than W. If this condition is true, it sets the rotation variable (rot[i]) to false, indicating that the circuit cannot be rotated. Otherwise, if the width is less than or equal to W, it sets the rotation variable to true, allowing the circuit to be rotated.

$$\forall i \in \{1, 2, \ldots, N\} : (\text{if} \;\; n[i] > W \;\; \text{then} \quad \text{rot}[i] = \text{false} \;\; \text{else} \quad \text{true} \;\; \text{endif})$$

2. **Constraints when the circuit is square:** constraint applies when the circuit is square, meaning its width (m[i]) is equal to its height (n[i]). It ensures that if a circuit i is already square, it cannot be rotated. This constraint is expressed using the logical condition "(m[i] == n[i])", which checks if the width and height are equal. If this condition is true, it further checks if the rotation variable (rot[i]) is false, indicating that the circuit cannot be rotated. If both conditions are satisfied, it ensures that the circuit remains unrotated.

$$\forall i \in \{1, 2, \ldots, N\} : ((m[i] = n[i]) \rightarrow (rot[i] = \text{false}))$$

2.4.3 Symmetry Breaking when Rotation is Allowed

The first predicate symm_breaking_same enforces a condition that ensures that no two circuits have the same width and height and also have the same X-Y coordinates.

$$\forall i, j \in \{1, 2, \ldots, n\}, i \neq j :$$
$$(\neg (X[i] = X[j] \wedge Y[i] = Y[j]) \vee \neg (w_{\text{real}}[i] = w_{\text{real}}[j] \wedge h_{\text{real}}[i] = h_{\text{real}}[j]))$$

The second predicate symm_breaking_axes computes the symmetry breaking with respect to the x or y-axis by ensuring that the (X[i] + w_real[i], Y[i] +

h_real[i]) coordinates of each circuit are less than or equal to the maximum width W and maximum height H of the layout.

$$\forall i \in \{1, 2, \ldots, n\} : (X[i] + w_{\text{real}}[i] \leq W \land Y[i] + h_{\text{real}}[i] \leq H)$$

The final constraint combines the two predicates and ensures that both the predicates hold true for the constraint to be satisfied.

$$\text{symm\_breaking\_same}(X, Y, w_{\text{real}}, h_{\text{real}}) \land \text{symm\_breaking\_axes}(X, Y, w_{\text{real}}, h_{\text{real}})$$

Overall, these constraints and predicates are used to optimize the layout of a set of circuits to minimize their total area while satisfying the given constraints. The goal is to find the optimal layout by assigning each circuit a position (X[i], Y[i]) and a rotation angle rot[i].

2.5 Validation

In this section, we will discuss the results obtained from our experimentation with various Constraint Programming models. Each test was conducted with a time constraint of 300 seconds, which encompassed both the execution time of the MiniZinc code and the time utilized by the solver. The experiments were conducted using the MiniZinc command-line interface on a MacBook Pro equipped with an Nvidia GeForce GTX 1650 graphics card and 16GB of RAM.

The choice of a search strategy plays a significant role in the performance and the number of solutions found, as demonstrated in the subsequent comparisons. In our experiments, we opted for a sequential search that takes into account different variables: 'H', the 'X' array, and the 'Y' array. We prioritized the 'H' variable as we believe it is crucial to find a suitable set of values for the height, which would subsequently enhance the solver's ability to search for appropriate coordinates.

Throughout our tests, we compared several variable choice annotations, including:

1. **First Fail**: This annotation selects the variable with the smallest domain size as the initial choice.
2. **Input Order**: With this annotation, the variables are chosen in the order they appear in the array.

To further constrain the variables, we employed the "indomain min" approach, which assigns each variable its smallest value from its domain. This restriction helps guide the search process towards exploring the lower end of the variable's possible values.

As mentioned in the introduction, we executed our model using both Gecode and Chuffed solvers.

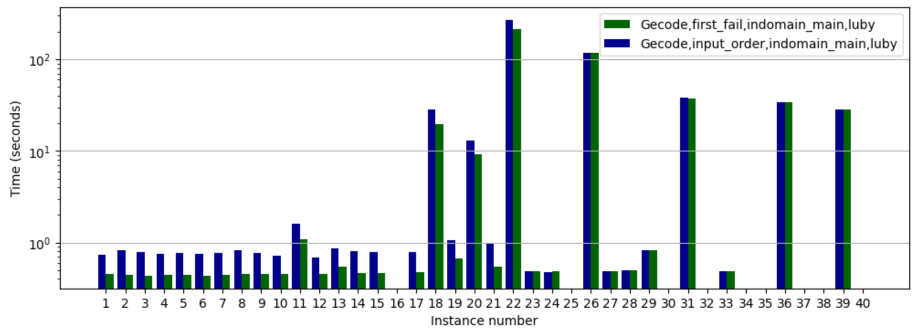


Figure 1: Comparison of the performances between first_fail and input_order using Gecode Solver on the no-rotation model

2.5.1 Without Rotation

Initially, let us examine the varied execution times of Gecode when utilizing different Variable Order Heuristics. In the figure, "im" denotes indomain_min, and "luby" represents luby restart with a value of 180. For each distinct run (blue, green), we only display a bar in the chart if the solver managed to discover the optimal solution within the imposed time constraint of 300 seconds.

We used both the variable choice annotations first_fail and input_order, we got 31 optimal results out of 40 with mean time of 11.41 seconds and 17.73 seconds respectively.

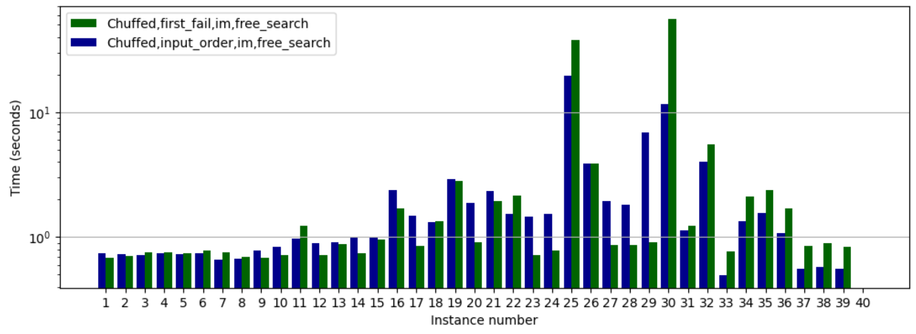


Figure 2: Comparison of the performances between first_fail and input_order using Chuffed Solver on the no-rotation model

By utilizing the **Chuffed** solver with **free_search**, employing the **first_fail** choice for variables and the **indomain_min** strategy for value selection, we were able to obtain optimal solutions for 39 out of 40 instances with mean time of 3.43 seconds.

And similarly using **Chuffed, input_order, indomain_main, and free_search** we were able to obtain optimal solutions for 39 out of 40 instances with mean time of 2.12 seconds.

Right from the start, we observed that Chuffed consistently outperformed Gecode. Additionally, we found that enabling the free search option improved the performance of both solvers.

2.5.2 With Rotation

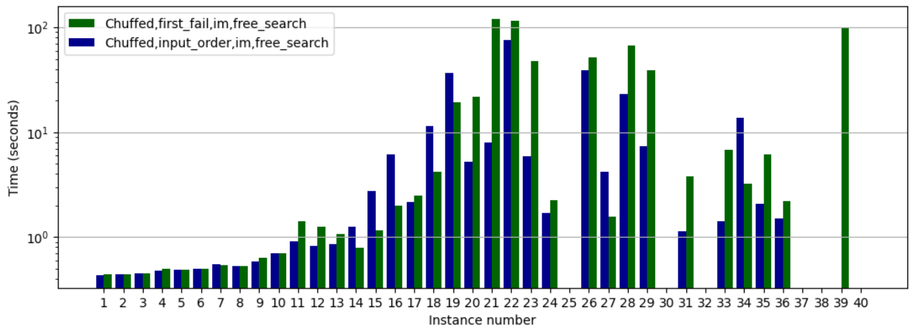


Figure 3: Comparison of the performances between first_fail and input_order using Chuffed Solver on the rotation model

We started with the Gecode solver and as we go ahead we were only able to get optimal solution for first few instances. So, we stopped using Gecode solver and started testing with Chuffed.

By utilizing the **Chuffed** solver with **free_search**, employing the **first_fail** choice for variables and the **indomain_min** strategy for value selection, we were able to obtain optimal solutions for 34 out of 40 instances with mean time of 16.21 seconds.

And with **Chuffed, input_order, indomain_min, free_search** strategy, we were able to obtain optimal solutions for 33 out of 40 instances with mean time of 7.71 seconds.

3 SAT Model

The SAT problem involves determining whether a given Boolean formula is satisfiable, which means there exists an assignment of truth values (true or false) to its variables such that the entire formula evaluates to true.
We utilized the Python interface provided by the Z3 solver to represent and formulate our problem as a SAT (Satisfiability) instance.

3.1 Decision variables

- **px**: $n \cdot w$ literals, where $px_{i,j}$ = True means block i can have x-coordinate j
- **py**: $n \cdot h$ literals, where $py_{i,j}$ = True means block i can have y-coordinate j
- **left**: $n \cdot n$ literals, where lefti, j = True means block i is at the left of block j
- **under**: $n \cdot n$ literals, where underi, j = True means block i is under block j

3.2 Objective function

In SAT we can't use an objective function to minimize h. Instead, we have to try different h values and check if the resulting formula is satisfiable. We start with the lower bound, if satisfiable we stop founding the optimal. If unsatisfiable, we increment h and try again.

3.3 Constraints

3.3.1 Axiom clauses for order encoding

For each rectangle r_i and integers e and f such that: $0 \leq e \leq w - \text{widths}_i$ and $0 \leq f \leq h - \text{heights}_i$ we have the 2-literal axiom clauses due to order encoding:

$$\neg px_{i,e} \lor px_{i,e+1}$$
$$\neg py_{i,f} \lor py_{i,f+1}$$

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:

$$left_{i,j} \lor left_{j,i} \lor under_{i,j} \lor under_{j,i}$$

We also have the following non-overlapping constraints, which hold for each e, j such that $0 \leq e \leq w - \text{widths}_i$ and $0 \leq f \leq h - \text{heights}_i$:

$$\begin{aligned}
\neg \text{left} i, j \lor pxi, e \lor \neg px_{j,e+\text{widths}i} \\
\neg \text{left} j, i \lor px_{j,e} \lor \neg px_{i,e+\text{widths}j} \\
\neg \text{under} i, j \lor py_{i,f} \lor \neg py_{j,f+\text{heights}i} \\
\neg \text{under} j, i \lor py_{j,f} \lor \neg py_{i,f+\text{heights}_j}
\end{aligned}$$

3.4 Encoding of the optimization constraints

We encoded some other constraints to prune the search tree and optimize our model.

3.4.1 Domain reduction technique

In order to remove potential symmetries we decided 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. This constraint is expressed using the following clauses:

$$\begin{aligned}
\forall i \in [0..w_{lim}] : \neg px_{max,i} \\
\forall j \in [0..h_{lim}] : \neg py_{max,j}
\end{aligned}$$

$$where \quad w_{lim} = \left\lfloor \frac{w - \text{widths} max}{2} \right\rfloor \quad hlim = \left\lfloor \frac{h - \text{heights}_{max}}{2} \right\rfloor$$

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:

$$\begin{aligned} widths_i > \left\lceil \frac{w - widths_{max}}{2} \right\rceil &\implies \neg left_{i,max} \\ heights_i > \left\lceil \frac{h - heights_{max}}{2} \right\rceil &\implies \neg under_{i,max} \end{aligned}$$

3.4.2 Same size rectangle constraint

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:

$$\begin{aligned} &\forall i, j s.t. i < j : \\ (widths_i = widths_j \land heights_i = heights_j) &\implies \\ \implies (\neg left_{i,j}) &\land (\neg under_{j,i} \lor left_{j,i}) \end{aligned}$$

3.4.3 Large Rectangle Constraint

$$\begin{aligned} \forall i, j i < j : (widths_i + widths_j > w) &\implies (\neg left_{i,j} \land \neg left_{j,i}) \\ \forall i, j i < j : (heights_i + heights_j > h) &\implies (\neg \text{under} i, j \land \neg under_{j,i}) \end{aligned}$$

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 and this would be in contrast with some other constraints.

3.5 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:

$$(\neg rotated_i \wedge originalclause) \vee (rotated_i \wedge rotatedclause)$$

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 $0 \leq e \leq w - \text{widths}_i$ and $0 \leq f \leq w - \text{heights}_i$:

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

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:

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

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:

$$(\neg rotated_i \wedge originalclause) \vee (rotated_i \wedge rotatedclause)$$

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 $0 \leq e \leq w - \text{widths}_i$ and $0 \leq f \leq w - \text{heights}_i$:

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

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:

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

which holds for each i, j such that:

$$0 < i < \left\lfloor \frac{w - widths_{max}}{2} \right\rfloor \quad and \quad 0 < j < \left\lfloor \frac{h - heights_{max}}{2} \right\rfloor .$$

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. which holds for each i, j such that:

$$0 < i < \left\lfloor \frac{w - widths_{max}}{2} \right\rfloor \quad and \quad 0 < j < \left\lfloor \frac{h - heights_{max}}{2} \right\rfloor .$$

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.

3.6 Validation

3.6.1 Without Rotation

with-rotation,no-symmetry-breaking we were able to solve 33 out of 40 instances with mean time of 31.99666666666667 seconds and with-rotation,with-symmetry-breaking 33 out of 40 instances with mean time of 16.10741935483871 seconds.

3.6.2 With Rotation

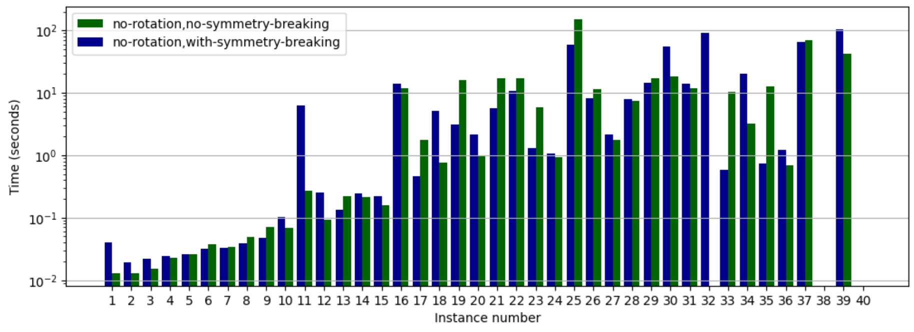


Figure 4: Comparison of the performances between no-rotation,no-symmetry-breaking and no-rotation,with-symmetry-breaking

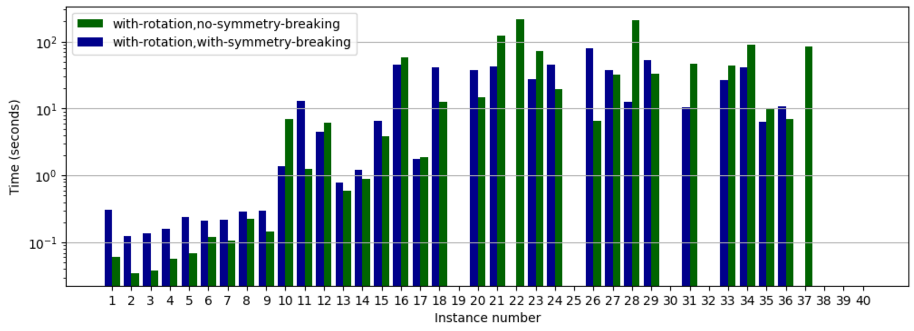


Figure 5: Comparison of the performances between with-rotation,no-symmetry-breaking and with-rotation,with-symmetry-breaking

4 SMT Model

4.1 Decision variables

The decision variables of the model are:

- X_i for $i \in 0, \ldots, N-1$: the x-coordinate of circuit i
- Y_i for $i \in 0, \ldots, N-1$: the y-coordinate of circuit i
- H: the height of the plate

All decision variables are integers. The problem is modeled in the integer theory using the Z3 SMT solver.

4.2 Objective function

Since we want to minimize the total area used and the width W is fixed by the instance, the objective is to minimize the height H of the plate. We first used a more lassive bounds, then we preceded to use a more strict to improve performance.
The old lower bound of H was defined by the constraint

$$H \geq \max(heights) \tag{1}$$

that ensure H to be greater than or equal to the maximum circuits height.
The upper bound was defined by the constraint

$$H \leq \sum_{i=0}^{n} height_i \tag{2}$$

which enforces H to be lower than or equal to the sum of all circuits heights.
The new lower bound of H we used is defined by the constraint

$$H \geq \max\left(\max(heights), \frac{\sum_{i=1}^{n} width_i height_i}{W}\right) \tag{3}$$

which enforces H must be greater than or equal to the maximum of the height of the tallest circuit and the minimum height needed to fit the total area of all circuits.
The new upper bound remained unchanged.

4.3 Constraints

The constraints encode:

Domain constraints on Coordinates

$$0 \leq X_i \leq W - \min(widths), \quad 0 \leq Y_i \leq H - \min(heights)$$

Boundary Constraints

$$X_i + \text{width}_i \leq W, \quad Y_i + \text{height}_i \leq H$$

Non-overlapping Constraints for each pair of circuits i, j where $i < j$, at least one of the following must hold:

$$X_i + \text{width}_i \leq X_j, \quad X_j + \text{width}_j \leq X_i, \quad Y_i + \text{height}_i \leq Y_j, \quad Y_j + \text{height}_j \leq Y_i$$

4.4 Rotation

4.4.1 Decision Variables

In addition to the original X_i, Y_i, H variables, this model introduces:

- rot_i: a binary variable indicating if circuit i is rotated
- $w\text{real}_i$: the actual width of circuit i after applying any rotation
- $h\text{real}_i$: the actual height of circuit i after applying any rotation

4.4.2 Constraints

Rotation Constraints

- Circuits that are wider than the plate width cannot rotate:

$$(w_i > W) \implies (r_i = \text{False})$$

- w_{real} and h_{real} are set based on rot using if-then assignments:

$$w_{real_i}, h_{real_i} = \begin{cases} h_i, w_i & \text{if } rot_i = \text{True} \\ w_i, h_i & \text{if } rot_i = \text{False} \end{cases}$$

Non-overlapping Constraints Non-overlapping constraints are modified to use the actual width and height values after rotation.

Boundary constraints The boundary constraints now ensure the circuits fit within the container dimensions using the right dimensions for rotated ones.

4.4.3 Symmetry Breaking constraints:

Squares cannot rotate if their width equals height:

$$(w_i = h_i) \implies (r_i = \text{False})$$

4.4.4 Objective

The objective remains the same, minimize height H. Also for rotation model we updated the bounds. The older lower bound was

$$H \geq h_{real_i} \tag{4}$$

that constraints H to be greater than or equal to every circuits heights. The old upper was the same of the no-rotation model bound (2).
The lower bound was updated as

$$H \geq \left(\frac{\sum_{i=1}^{n} width_i height_i}{W} \right) \tag{5}$$

where H is greater than or equal to the minimum height needed to fit the total area of all circuits.
The upper bound was updated as

$$H \geq \left(\sum_{i=1}^{n} \max(width_i, height_i) \right) \tag{6}$$

where H is lower than or equal to the sum of the maximum between height and width of all circuits, basically the highest stack of circuits possible.

4.5 Validation

Experimental design The models are implemented and solved using the SMT solver (Optimize) provided by the Z3py python library. All the trials were performed on a Samsung GalaxyBook Pro 360 with these specifications: CPU: 11th Gen Intel Core i7-1165G7 @ 2.80GHz, RAM: 16GB, GPU: Intel Xe Graphics. For each run the computation time limit was set as 5 minutes (300000 ms).

Experimental results Computation time was tracked to evaluate solver performance on both models. The two bar plots shown represent the computation with different bounds for H.
In the first one, where we used bounds (1) (2) for no-rotation model and (4) (2) for rotation model, we had poor results; no-rotation model achieved the optimum solution for 24 instances and the rotation-allowed model only for 9. In the second one we used bounds (3) (2) for no-rotation model and (5) (6) for rotation model and the performance were enhanced by a lot; no-rotation model achieved 32 solutions and the rotation-allowed model achieved 25 solutions. Comparing the two plots we can also notice that the spread between the computation times for each instance of the two models narrowed. Despite the good results, the number of non solved instances suggests that, expecially for the rotation model, there is a wide margin of improving and optimization of the constraints.

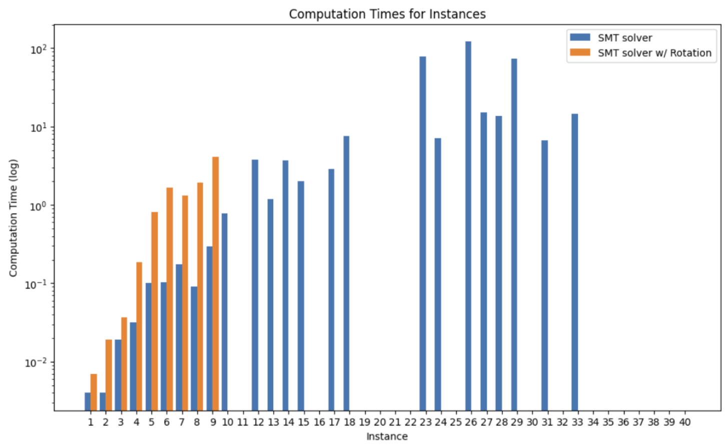


Figure 6: Bar plot showing computation time with old bounds

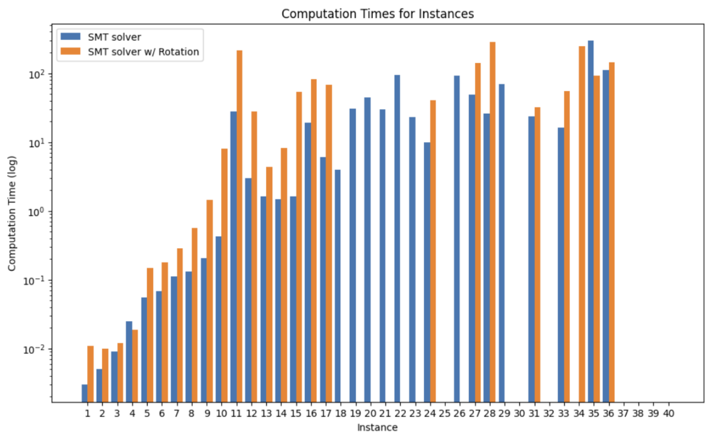


Figure 7: Bar plot showing computation time with updated bounds

5 MIP Model

MIP stands for Mixed-Integer Programming, which is a mathematical optimization technique used to solve a wide range of decision-making problems. In a MIP model, you have both continuous and discrete decision variables, where some variables can take on integer values (i.e., they are integers), while others can take on continuous values (i.e., they are real numbers).

5.1 Decision variables

1.x_cord: This is a set of decision variables representing the x-coordinates (horizontal positions) of the circuits. It is defined as a Gurobi decision variable array (addVars) for each circuit, with lower and upper bounds specified. Each x_cord[i] represents the x-coordinate of the i-th circuit.

2.y_cord: Similarly, this is a set of decision variables representing the y-coordinates (vertical positions) of the circuits. Like x_cord, it is defined as a Gurobi decision variable array for each circuit, with lower and upper bounds. Each y_cord[i] represents the y-coordinate of the i-th circuit.

3. h: This is another decision variable representing the height of the silicon plate. It is defined as a Gurobi integer variable with lower and upper bounds. The optimization aims to find the optimal value for h that minimizes the height of the plate while satisfying the constraints.

5.2 Objective function

The objective function is simple and consists of a single decision variable, **h**, which represents the height of the silicon plate. The goal is to minimize the value of **h**. The **GRB.MINIMIZE** parameter indicates that this is a minimization problem. The optimization algorithm will adjust the values of the decision variables (including h) to find the smallest possible value of h while satisfying all the defined constraints.

$$\text{Minimize } h \quad \text{(Minimize the height of the silicon plate)}$$

The objective function in this code aims to minimize the height of the silicon plate while arranging the circuits on it, subject to various constraints.

5.3 Constraints

5.3.1 Inside Plate Constraints

These constraints ensure that the horizontal position (x_cord[i]) of each circuit does not exceed the width of the silicon plate (w). In other words, it ensures that no circuit goes beyond the right edge of the plate.

$$\begin{aligned}&\text{For } i = 1, 2, \ldots, n :\\ &\quad x\_cord[i] + x[i] \leq w \quad \text{(Horizontal position within plate)}\end{aligned}$$

These constraints ensure that the vertical position (y_cord[i]) of each circuit does not exceed the height of the silicon plate (h). It prevents circuits from going beyond the top edge of the plate.

$$\begin{aligned}&\text{For } i = 1, 2, \ldots, n :\\ &\quad y\_cord[i] + y[i] \leq h \quad \text{(Vertical position within plate)}\end{aligned}$$

5.3.2 No Overlap Constraints

$$\begin{aligned}&\text{For } i, j = 1, 2, \ldots, n \text{ with } i \neq j :\\ &\quad x\_cord[i] + x[i] \leq x\_cord[j] + w \cdot s[i, j, 0] \quad \text{(No overlap to the right)}\\ &\quad y\_cord[i] + y[i] \leq y\_cord[j] + h \cdot s[i, j, 1] \quad \text{(No overlap below)}\\ &\quad x\_cord[j] + x[j] \leq x\_cord[i] + w \cdot s[i, j, 2] \quad \text{(No overlap to the right)}\\ &\quad y\_cord[j] + y[j] \leq y\_cord[i] + h \cdot s[i, j, 3] \quad \text{(No overlap below)}\\ &\text{For } i, j = 1, 2, \ldots, n \text{ with } i \neq j :\\ &\quad \sum_{k=0}^{3} s[i, j, k] \leq 3 \quad \text{(At most three directions active)}\end{aligned}$$

5.3.3 Area Constraints

$$\begin{aligned}&\text{Total Area} \leq \text{area\_max} \quad \text{(Upper bound on total area)}\\ &\text{Total Area} \geq \text{area\_min} \quad \text{(Lower bound on total area)}\end{aligned}$$

These constraints collectively ensure that the circuits are placed on the silicon plate in a non-overlapping manner, and the height of the plate (h) is minimized while satisfying area constraints.

The objective function, as mentioned earlier, aims to minimize the value of h, which represents the height of the silicon plate. The optimization process adjusts the values of decision variables (x_cord, y_cord, and h) to achieve this minimization while adhering to the defined constraints.

5.4 Rotation

In contrast to the model without rotation, we introduce a crucial boolean variable known as "rotation," which signifies whether a particular chip can be rotated or not. The inclusion of this new variable necessitated adjustments to all constraints that originally depended on the heights and widths of the chips.

For any given chip, denoted as "i," if the rotation is allowed (i.e., rotation[i] = True), it implies that the chip's width becomes its new height, and conversely, its height becomes its new width.

5.5 Validation

The model was implemented with Python through the Gurobi Optimizer, a mathematical optimization software library. Each test was done using a 300 seconds time out.

We also tried solving using Pulp but were not as successful as we wanted then we shifted to Gurobipy we got the first few good results so we stuck with it.

We need a license to run the code and to save the output correctly. Without license i was able to get the output of the first 3 instances with the timing of 0.001 sec, 0.02 sec, and 0.06 sec respectively. We applied for the student licence but as we were not using a student network while coding as away from the university so were not able to use the student-free version.

6 Conclusion

Through the course of this project, we successfully accomplished our primary objective: solving the proposed problem using four distinct approaches. For each of these approaches, we also developed a model that allows for the rotation of circuits.

Upon reviewing our results, it becomes evident that Constraint Programming (CP) stands out as the most effective approach. It can solve the first 39 instances without the need for rotation and handles 34 instances when rotation is permitted. While the best SAT rotation model can handle one additional instance, The CP model strikes a better balance between speed and the number of instances solved.

Moving on to the models that incorporate rotation, if the goal is to solve as many instances as possible, it's worth considering the idea of using all available approaches. This approach ensures that even if one model struggles with a particular instance, another model might be able to solve it, even if it is generally less efficient. Consequently, when combining SAT and CP, we are able to solve 38 instances. The three instances that remain unsolved are the 38th, and 40th. This outcome underscores the fact that certain instances inherently pose greater challenges than others. It also emphasizes that the complexity of solving an instance is not solely determined by the number of circuits to be placed within the chip.