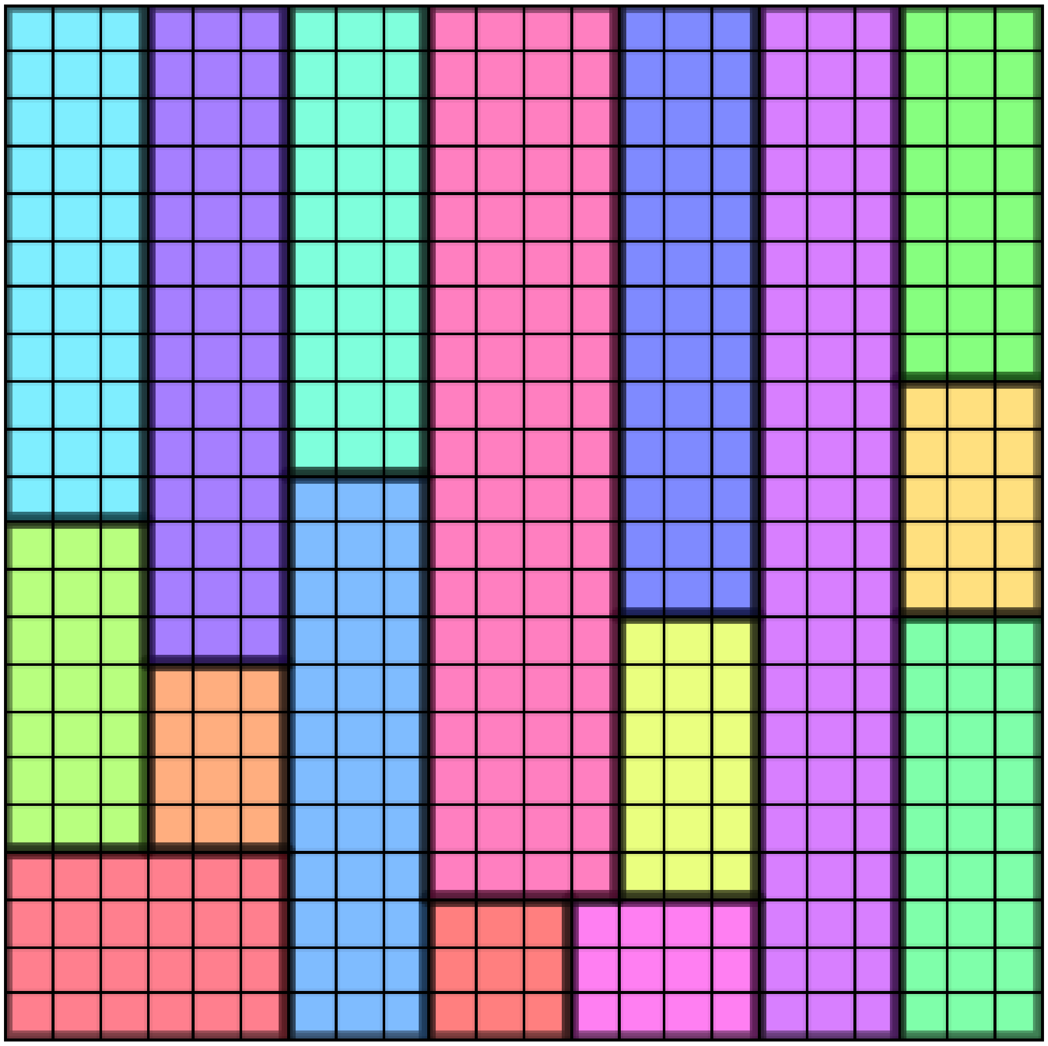
Combinatorial Decision Making and Optimization

Very Large Scale Integration Problem



Antonio Politano (antonio.politano2@studio.unibo.it), Enrico Pittini (enrico.pittini@studio.unibo.it) and Riccardo Spolaor (riccardo.spolaor@studio.unibo.it)

Contents

1 Introduction

1.1 Description of the problem

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. A typical example is the smartphone. The modern trend of shrinking transistor sizes, allowing engineers to fit more and more transistors into the same area of silicon, has pushed the integration of more and more functions of cellphone circuitry into a single silicon die (i.e. plate).
The VLSI problem taken into account is the following: given a fixed-width plate and a list of rectangular circuits, the arrangement of the circuits must be decided in order to minimize the length of the plate.
Two variants of the problem are considered. In the first, each circuit must be placed in a fixed orientation with respect to the others. This means that, an $n \times m$ circuit cannot be positioned as an $m \times n$ circuit in the silicon plate. In literature, this problem is known as *Two-dimensional Strip Packing Problem (2SPP)*.
In the second case, the rotation is allowed, which means that an $n \times m$ circuit can be positioned either as it is or as $m \times n$.

This problem has been tackled using four different combinatorial optimization approaches, namely:

- **Constraint Programming (CP)**
- **Boolean Satisfiability (SAT)**
- **Satisfiability Modulo Theory (SMT)**
- **Linear Programming (LP)**

A total of 40 instances has been provided for testing purposes, setting a time limit of 300 seconds.

1.2 General approach

In developing this project the following general approaches have been taken into account.

- No assumptions regarding the provided instances have been considered, but a general non-biased methodology has always been followed.
- A strictly incremental methodology has been followed, consisting in starting from a basic model and then trying to improve it by making it more sophisticated, through a sequence of steps. At each step, empirical measures of quality have been used for comparing the different choices and then selecting the best one.
- As quality measures for the models, the total number of solved instances and the average solving time have been considered, prioritizing the former over the latter.

Remark: the results obtained through our experiments may not be completely reproducible, due to the random behavior involved in some of these algorithms.

1.3 Notation

In the report, the following notation is used to identify the entities of the problem.

- w, the width of the plate
- n, the number of circuits
- x_i, the x-coordinate of left-bottom corner of the i-th circuit
- y_i, the y-coordinate of left-bottom corner of the i-th circuit
- w_i, the width of the i-th circuit
- h_i, the height of the i-th circuit
- l, the length of the plate

Given w, n and all the w_i, h_i as parameters, then x_i, y_i are computed, in order to minimize l.

Remark: the notation used in the code is slightly different. The arrays `coordsX`, `coordsY` *are used instead of x_i, y_i, while the arrays* `dimsX`, `dimsY` *are used instead of w_i, h_i.*

1.4 Common preliminaries

Regarding the variant without rotation, the following bounds for the length of the plate have been used.

- $l_{\min}$, the lower bound for the height of the plate, is computed as the maximum between the integer division of the sum of the areas of all the circuits and the width of the plate (guaranteeing that all of the circuits fit in the plate) and the maximum of all the heights of the circuits (guaranteeing that the highest circuit fits in):

$$l_{\min} = \max\left(\left\lfloor \frac{\sum_{i=1}^{n} w_i \cdot h_i}{w} \right\rfloor,\ \max_{i \in 1..n}(h_i)\right)$$

- $l_{\max}$, the upper bound for the height of the plate is computed as follows. First of all, the minimum number of rectangles per row is computed:

$$k = \left\lfloor \frac{w}{\max_{i \in 1..n}(w_i)} \right\rfloor$$

  Let $h_1^{'}, ..., h_n^{'}$ be the heights of the circuits sorted in descending order, and let I be the following set $I = \{i \mid i \in [1..n] \ \wedge \ i \mod k = 1\}$. Finally, $l_{\max}$ is computed as:

$$l_{\max} = \begin{cases} \sum_{i=1}^{n} h_i, & \text{if } k = 1 \\ \sum_{i \in I} h_i^{'}, & \text{otherwise} \end{cases}$$

  Since k is the minimum number of circuits per row, the second case proceeds by taking the highest rectangle and then the $(k+1)$-th highest rectangle and then the $(2 \cdot k + 1)$-th highest rectangle, and so on. On each row, we can for sure fit the remaining rectangles, therefore obtaining our upper bound for l by summing the heights of the taken circuits. The first case is added in order to sum all the heights if we cannot position more than one circuit per row.

- $w_{\min}$, the minimum width among all circuits.

$$w_{\min} = \min_{i \,\in\, 1..n} w_i$$

- $w_{\max}$, the maximum width among all circuits.

$$w_{\max} = \max_{i \,\in\, 1..n} w_i$$

- $h_{\min}$, the minimum height among all circuits.

$$h_{\min} = \min_{i \,\in\, 1..n} h_i$$

- $h_{\max}$, the maximum height among all circuits.

$$h_{\max} = \max_{i \,\in\, 1..n} h_i$$

Regarding the rotation variant, the lower and upper bounds are computed in a slightly different way.

- $l_{\min}$ considers just the first term, regarding the areas, since the height and width of each circuit may be exchanged through rotation:

$$l_{\min} = \left\lfloor \frac{\sum_{i=1}^{n} w_i \cdot h_i}{w} \right\rfloor$$

- For computing $l_{\max}$ we consider, for each circuit i, d_i being the maximum between its dimensions (i.e. maximum between w_i and h_i). First of all, the minimum number of rectangles per row is computed:

$$k = \left\lfloor \frac{w}{\max\limits_{i \in 1..n} (d_i)} \right\rfloor$$

  Let $d_1^{'}, ..., d_n^{'}$ be the max dimensions of the circuits sorted in descending order, and let I be the following set $I = \{i \mid i \in [1..n] \ \wedge \ i \mod k = 1\}$. Finally, $l_{\max}$ is computed following a similar process as the one without rotations:

$$l_{\max} = \begin{cases} \sum_{i=1}^{n} d_i, & \text{if } k < 2 \\ \sum_{i \in I} d_i^{'}, & \text{otherwise} \end{cases}$$

2 CP

Constraint Programming (CP) is a combinatorial decision making paradigm to find feasible or optimal solutions for a series of variables of a problem while guaranteeing a series of constraints on its feasible solutions are respected.

The models to test the solution have been implemented in *MiniZinc* [1], using as a base solver *Chuffed* [2].

Some tests in section 2.5 have been carried out testing as well the solver *Gecode* [3], although the obtained performances are significatively worse. Furthermore, the *OR-Tools* solver [4] was initially compared with the other ones. Although the obtained solutions were almost comparable with the one of *Chuffed*, we decided to discard it because it was not reliable in informing whether the solutions were obtained within time limit constraints.

This section solves the VLSI problem through this approach. We incrementally implemented a series of models to solve the VLSI task through the CP notation. Each model was defined as a minimization task for the variable l.

2.1 Basic model

This section describes the initial naive basic model which imposes the optimality and the correctness of the solution through a series of declared variables and necessary constraints.

2.1.1 Variables and parameters

Variables and parameters refer to the ones declared in the sections 1.3 and 1.4.
For this model, the domains for x, y are defined as:

$$\forall i \in i..n \; 0 \leq x_i \leq w$$

$$\forall i \in i..n \; 0 \leq y_i \leq \infty$$

2.1.2 Constraints

The following constraints are imposed:

- Non overlapping between pairs of circuits (i, j) must be guaranteed on either the horizontal or the vertical dimensions.

$$\forall \; i, \; j \in 1..n \land i < j, \; (x_i + w_i \leq x_j \lor x_j + w_j \leq x_i) \; \lor \; (y_i + h_i \leq y_j \lor y_j + h_j \leq y_i) \qquad \text{(CP 1)}$$

- Each circuit i must be placed inside the plate along the horizontal dimension.

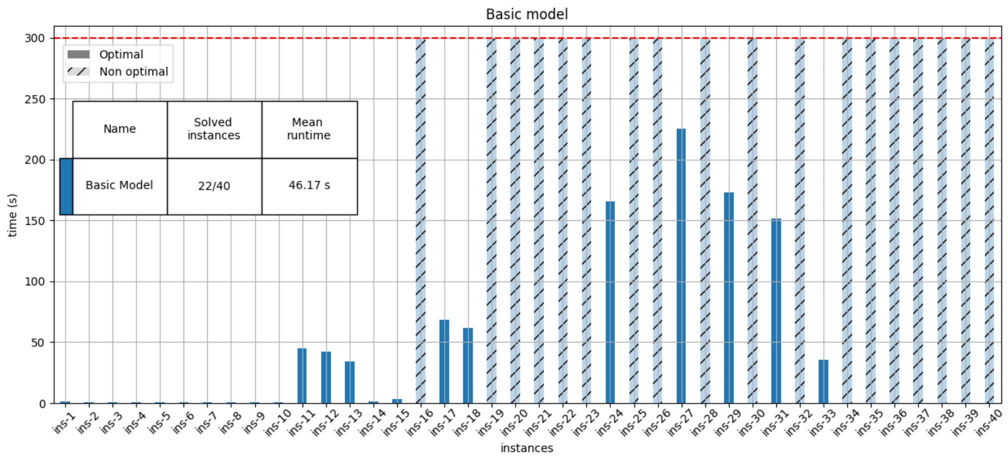
$$\forall \; i \in 1..n, \; x_i + w_i \leq w \qquad \text{(CP 2)}$$

- Each circuit i must be placed inside the plate along the vertical dimension, therefore the highest upper-edge of the circuits must reach the length of the plate l.

$$l = \max_{i \in 1..n} (y_i + h_i) \qquad \text{(CP 3)}$$

2.1.3 Results

As expected, the results for this naive model are pretty bad as shown in figure 1.



	Name	Solved instances	Mean runtime
■	Basic Model	22/40	46.17 s

Figure 1: Results of the basic model

2.2 Model with global constraints

Global constraints are high-level modelling abstractions, for which many solvers implement special, efficient inference algorithms. Thus, we decided to introduce some of them in order to try to improve the performances of our model.

2.2.1 Constraints

The following global constraints have been introduced.

- The `cumulative` constraint.

```
predicate cumulative(array [int] of var int: s,
                     array [int] of var int: d,
                     array [int] of var int: r,
                     var int: b)
```

  It is used to guarantee a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.

  In particular it was applied in these two different cases:

$$\texttt{cumulative}([x_1, ..x_n], [w_1, ..w_n], [h_1, ..h_n], l) \tag{CP 4}$$

  where:

  – The start times s is the array of horizontal coordiantes of each circuit i, x_i.
  – The duration of the tasks d is the array of widths of each circuit i, w_i.
  – The resource requirements of the tasks r is the array of heights of each circuit i, h_i.
  – The global maximum resource bound b is the length of the plate l.

$$\texttt{cumulative}([y_1, ..y_n], [h_1, ..h_n], [w_1, ..w_n], w) \qquad \text{(CP 5)}$$

where:

- The start times s is the array of vertical coordinates of each circuit i, y_i.
- The duration of the tasks d is the array of heigths of each circuit i, h_i.
- The resource requirements of the tasks r is the array of widths of each circuit i, w_i.
- The global maximum resource bound b is the width of the plate w.

- The `diffn` constraint

  ```
  predicate diffn(array [int] of var int: x,
                  array [int] of var int: y,
                  array [int] of var int: dx,
                  array [int] of var int: dy)
  ```

  It constrains rectangles i, given by their origins (x_i, y_i) and sizes (dx_i, dy_i), to be non-overlapping. It is applied in the case:

  $$\texttt{diffn}([x_1, ..x_n], [y_1, ..y_n], [w_1, ..w_n], [h_1, ..h_n]) \qquad \text{(CP 6)}$$

  It grants non-overlapping between circuits and better performances through wider propagation. Therefore it was placed in substitution of the constraint (CP 1) of the previous model.

2.2.2 Removing the non-necessary constraints

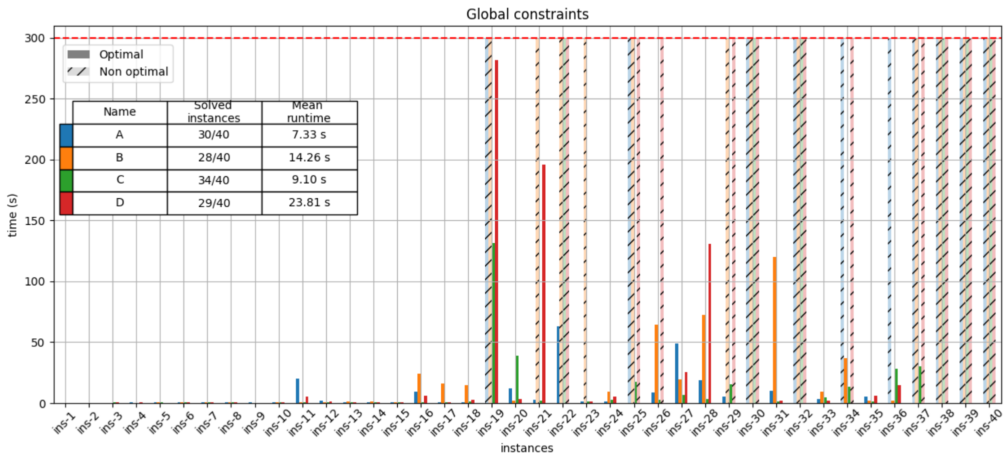
It shall be noted that the cumulative constraints $\texttt{cumulative}([x_1, ..x_n], [w_1, ..w_n], [h_1, ..h_n], l)$ and $\texttt{cumulative}([y_1, ..y_n], [h_1, ..h_n], [w_1, ..w_n], w)$ are implied by the constraints (CP 2) and (CP 3) respectively. We tested four models containing all previous constraints and different possible combinations of these implied constraints, namely:

1. **Model A**: containing all cumulative constraints.
2. **Model B**: containing solely, $\texttt{cumulative}([y_1, ..y_n], [h_1, ..h_n], [w_1, ..w_n], w)$.
3. **Model C**: containing solely, $\texttt{cumulative}([x_1, ..x_n], [w_1, ..w_n], [h_1, ..h_n], l)$.
4. **Model D**: containing no cumulative constraints.

It was finally observed that the best performances can be obtained by using **Model C**.

2.2.3 Results

Figure 2 illustrates the results for the different models. It can be clearly observed how the performances are significantly better than the naive model in all cases in both terms of number of solved instances and solving time. Furthermore **Model C** outperforms the others by solving a total of 34 instances with a mean time of 9.10 seconds to reach a solution for an instance.



	Name	Solved instances	Mean runtime
🟦	A	30/40	7.33 s
🟧	B	28/40	14.26 s
🟩	C	34/40	9.10 s
🟥	D	29/40	23.81 s

Figure 2: Global constraints models results

2.3 Model with symmetry breaking constraints

The problem taken in exam presents different symmetric solutions that can be considered as equivalent possible solutions of the task. In order to reduce the search space of the solving algorithms and get to the optimal goal with less effort and time, just one among these symmetric solutions can be considered through a process called *symmetry breaking*.

The problem presents the following symmetries:

- An horizontal symmetry. since, from a generic solution (x, y, l), we can obtain the horizontally symmetric solution x', y, l as follows:

$$\forall \ i \in 1..n \ x'_i = w - x_i - w_i$$

- Very similarly, a vertical symmetric solution can be obtained from the mapping $(x, y, l) \mapsto (x, y', l)$ where:

$$\forall \ i \in 1..n \ y'_i = l - y_i - h_i$$

- When two or more circuits happen to have the same dimensions these can be swapped on the plate to obtain again a new equivalent solution.
- Different symmetric solutions can be obtained by changing the position of the biggest circuit (in terms of height or area), so it can be enforced in a particular position to reduce the search space.
- When some rectangles are stacked on top of each other to cover a column of the plate, or next to each other to cover a row, they can be rearranged to obtain a new solution, which is symmetric to the first.

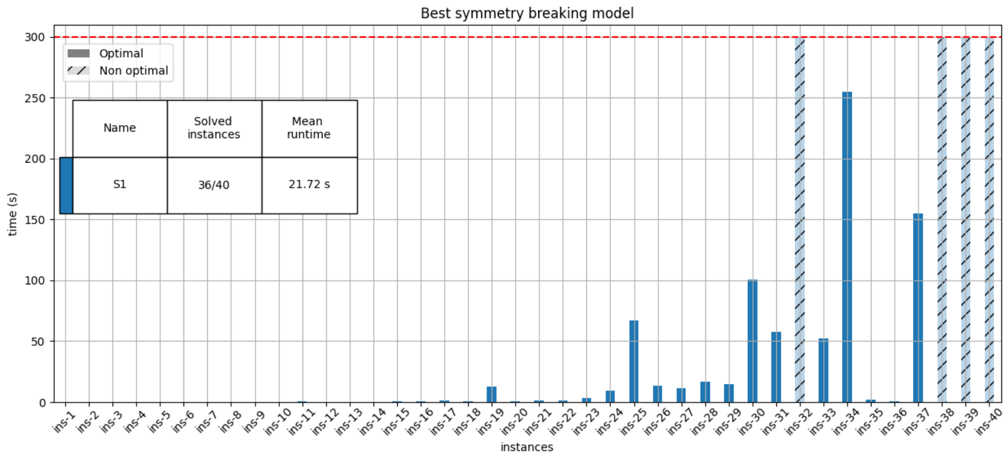
2.3.1 Constraints

In order to break the vertical and horizontal symmetries, **Model S1** was implemented. It adds two more constraints so that the sum of the areas of all the circuits in the left half of the plate is greater or equal to the sum of the areas of the circuits in the right half of the plate (same goes for the lower and upper halves). More precisely, we enforced that the sum of the areas of the circuits with the bottom left corner in the right (or upper) half of the plate is less or equal than half of the total area of the circuits. It is formalised as follows:

$$\sum_{i \in R} w_i \cdot h_i \leq \frac{\sum_{i=1}^{n} w_i \cdot h_i}{2} \tag{CP 7}$$

$$\sum_{i \in U} w_i \cdot h_i \leq \frac{\sum_{i=1}^{n} w_i \cdot h_i}{2} \tag{CP 8}$$

Where $R = \{i \mid i \in 1..n \land x_i \geq \lceil \frac{w}{2} \rceil\}$ and $U = \{i \mid i \in 1..n \land y_i > \lfloor \frac{l}{2} \rfloor\}$.



Name	Solved instances	Mean runtime
S1	36/40	21.72 s

Figure 3: Best symmetry breaking model results

2.3.2 Discarded constraints

All possible combinations of a series of other constraints that exploit the symmetries illustrated in section 2.3 were then tested in a series of different models. Although, none of these managed to improve the performances obtained through the vertical and horizontal symmetry breaking introduced in the previous section.

- **Model S2**. Imposes a lexicographic ordering between circuits of the same dimensions in order to avoid them being swapped uselessly.

$$\forall\ i,\ j \in 1..n \land i < j \land w_i = w_j \land h_i = h_j,\ x_i \leq x_j \land (x_i < x_j \lor y_i \leq y_j)$$

The constraint forces the first rectangle to be placed before the second one and denying possible swaps.

- More attempts were made to eliminate symmetric solutions and get to the optimal one in the quickest way possible by fixing a specific circuit k with particular characteristics in the bottom left corner.

$$x_k = 0 \land y_k = 0$$

In particular three new models that consider a different kind of circuit k were built such that:

  - **Model S3**. Considers k as the highest among the circuits.

$$k = \underset{i \in 1..n}{\arg\max}(h_i)$$

  - **Model S4**. Considers k as the circuit with the largest area.

$$k = \underset{i \in 1..n}{\arg\max}(w_i \cdot h_i)$$

  - **Model S5**. Considers k as the widest among the circuits.

$$k = \underset{i \in 1..n}{\arg\max}(w_i)$$

- **Model S6**. One last symmetry breaking constraint was inspired by the paper *Symmetries in Rectangular Block-Packing* [5]. It imposes to consider only one of the possible permutations for adjacent circuits. In particular, for both horizontal and vertical directions, it can be formalised as follows:

$$\forall\ i,\ j \in 1..n \land i < j \land x_i = x_j \implies y_i \neq y_j + h_j$$

$$\forall\ i,\ j \in 1..n \land i < j \land y_i = y_j \implies x_i \neq x_j + w_j$$

So, if a certain column is composed by circuits starting at the same x coordinate, they will be placed from bottom to top in ascending index order (same goes for the horizontal constraints active on the rows).

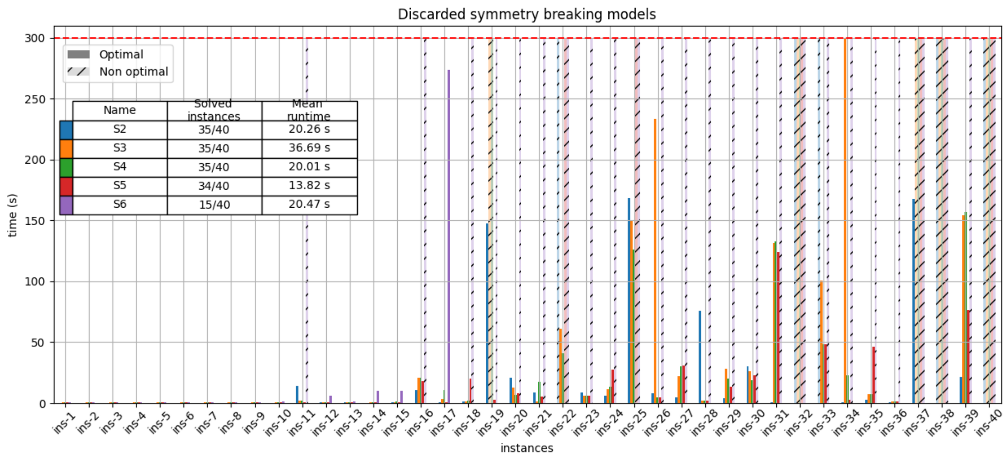
2.3.3 Variables domains improvements

Several other improvements of the model were obtained in this phase. More precisely, heuristics to estimate lower and upper bounds for l described in section 1.4 were added. Also, the domains for x and y were restricted, in particular:

$$\forall i \in i..n\ 0 \leq x_i \leq w - w_{\min}$$

$$\forall i \in i..n\ 0 \leq y_i \leq l_{\max} - h_{\min}$$

Where $w_{\min} = \min_{i\ \in\ 1..n} w_i$ and $h_{\min} = \min_{i\ \in\ 1..n} h_i$.



	Name	Solved instances	Mean runtime
🟦	S2	35/40	20.26 s
🟧	S3	35/40	36.69 s
🟩	S4	35/40	20.01 s
🟥	S5	34/40	13.82 s
🟪	S6	15/40	20.47 s

Figure 4: Comparison of discarded symmetry breaking models results

2.3.4 Results

As seen in figure 3 the two constraints of Model S1 improve the performances in terms of solved instances, therefore they were considered in all the successive implementations. However, as expected, the average time to obtain the solutions slightly increases.

Figure 4 shows the results of the models considering the discarded symmetry breaking constraints. It can be observed how Model S6 obtains the worst performances. Instead, the other models provide results comparable to the ones of model S1, although not introducing important gains in terms of time and generally solving one or two less instances.

2.4 Dual model attempt

A different view of the problem, namely a dual model, has been added to the current best model for trying to improve it. It considers the plate as a grid of cells C of dimensions $l_{\max} \times w$.

2.4.1 Variables and constraints

Each cell of the aforementioned grid is a variable of the dual model, having as a domain $0..n$. Each cell $C_{i,j}$ has value 0 if and only if it is empty and it has value $k \in 1..n$ if and only if it is covered by the k-th circuit.

The constraints for this model grant that each circuit either fits entirely or it is absent in each row and column. Let $C^k_{i,j}$ be a new matrix with the same dimensions of C such that

$$C^k_{i,j} = 1 \iff C_{i,j} = k,\ C^k_{i,j} = 0 \text{ otherwise}$$

Then, for each circuit k, the constraints can be formalised as follows:

$$\forall j \in 1..w, \sum_{i=1}^{l_{\max}} C_{i,j}^{k} = 0 \vee \sum_{i=1}^{l_{\max}} C_{i,j} = h_k$$

$$\forall\ i \in 1..l_{\max}, \sum_{j=1}^{w} C_{i,j}^{k} = 0 \vee \sum_{j=1}^{w} C_{i,j} = w_k$$

The fact that the circuits are still placed as rectangles and not cut or reshaped is guaranteed by the original model.

To obtain the channeling constraint that connects the dual model to the previous best model, it is required that each cell has value k in the grid if and only if that cell belongs to the k-th circuit according to the original model. Again we can formalise the constraint as:

$$\forall\ i \in 1..l_{\max},\ \forall j \in 1..w,\ \forall k \in 1..n,\ C_{i,j} = k \iff (x_k \leq j \leq x_k + w_k) \wedge (y_k \leq i \leq y_k + h_k)$$

The model performs very poorly, therefore it was obviously discarded.

2.5 Heuristics

MiniZinc leaves by default the search process to the underling solver. In some cases, specifying heuristics on how to proceed through the search could improve the performance of the model. These heuristics can be communicated to the solver through *search annotations.*

Following the suggestions of the official *MiniZinc* documentation page [6], the *free search* option of the solver *Chuffed* has been exploited. This feature guarantees better performances allowing the solver to switch between its predefined activity based search and the heuristics defined in the model.

All the implementations described in this section use, as well as previous ones, *bound consistency* to handle the reductions of the domains. A *generalised arc consistency* approach based on the domains was tested as well, although it was discarded since it didn't bring any sensible improvement.

2.5.1 Selected heuristics

The following search heuristics have been selected for the final model (**Model H1**):

- The first considered search annotation concerns the variable l. It forces the solver to perform a complete search for its assignment in its domain by firstly considering the lowest possible value. The solver considers successive values of l in its domain in increasing order only if the search fails.

  In other words, a solution for the problem is searched by always considering the lowest possible value of l and by then eventually discarding it if a result is not reached:

  ```
  ann: search_ann_l_from_min = int_search([l], input_order, indomain_min, complete);
  ```

- The idea behind the second heuristic (and some of the discarded ones in section 2.5.2), is inspired from the *bottom-up left-justified* algorithm, presented in the paper *Orthogonal packings in two dimensions*

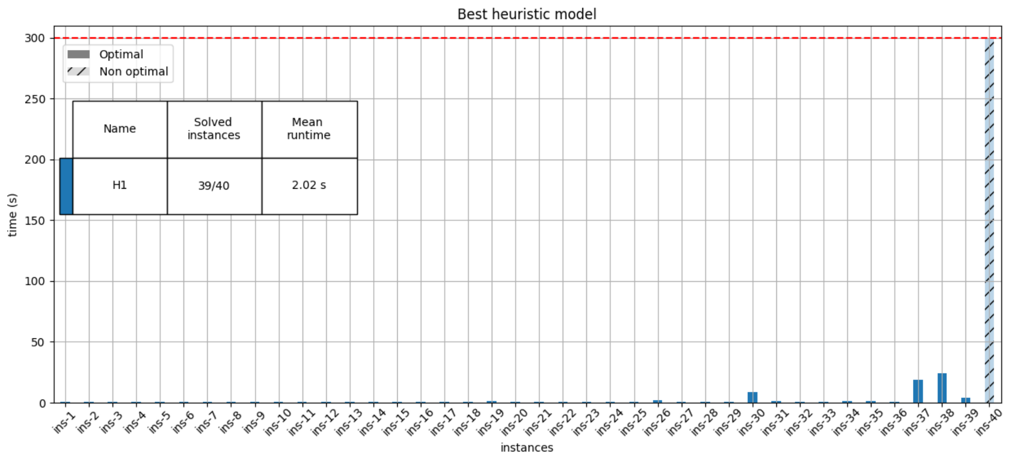
[7]. The idea behind that algorithm is to consider rectangles from a given list and sequentially put them in the lowest available position and finally justifying them on the left as much as possible. Given the nature of the problem, we implemented our variation to sequentially fix the circuit as far left as possible, then to justify the circuit on the lowest available position.

Considering that wider circuits are harder to place, the circuits are sorted based on their dimensions. In particular, in this case, for each circuit i sorted by width w_i with decreasing order, the solver searches for its coordinates $(x_i,\ y_i)$ according to our variation of the *bottom-up left-justified* algorithm. In mathematical terms, the instantiations proceed as follows:

$$\forall\ k \in 1..n\ x_k = \min(x'_k),\ \ y_k = \min(y'_k)$$

where the index k indicates the k-th widest circuit and $x'_k,\ y'_k$ represent the possible positions for that circuit.

```
array[1..n] of var int: sorted_coordsX = reverse(sort_by(coordsX,
    dimsX));
array[1..n] of var int: sorted_coordsY = reverse(sort_by(coordsY,
    dimsX));
array[1..2*n] of var int: variables = [if (i mod 2 = 0) then
    sorted_coordsY[i div 2] else sorted_coordsX[(i+1) div 2] endif |
    i in 1..2*n];
ann: search_ann_dec_width = int_search(variables, input_order,
    indomain_min);
```



Name	Solved instances	Mean runtime
H1	39/40	2.02 s

Figure 5: Results of the model with the best search heuristics (Model H1)

2.5.2 Discarded heuristics

Other models were built in order to test different search heuristics, though these were discarded due to their slightly worse performances. All of the heuristics that are going to be mentioned were tested in combination with the first heuristic presented in section 2.5.1 regarding the selection of l starting from the lowest possible value.

- **Model H2**. An heuristic very similar to the second one described in section 2.5.1 has been introduced. It interleaves the coordinates of the circuits sorting them in decreasing order by their areas, placing them with the usual variation of the *bottom-left strategy*. So the instantiation would proceed as follows:

  $$\forall\ k \in 1..n\ x_k = \min(x'_k),\ y_k = \min(y'_k)$$

  Where the index k indicates the k-th largest circuit by area, while x'_k, y'_k represent the possible positions for that circuit.

  ```
  array[1..n] of int: indices = reverse(arg_sort(areas));
  array[1..2*n] of var int: variables = [if (i mod 2 = 0) then coordsY[
      indices[i div 2]] else coordsX[indices[(i+1) div 2]] endif
    | i in 1..2*n];
  ann: search_ann_dec_areas = int_search(variables, input_order,
      indomain_min);
  ```

- **Model H3**. It uses an heuristic similar to the second one described in section 2.5.1. This time interleaving the coordinates of the circuits and placing them with the actual *bottom-left strategy*, this time ordered by their vertical dimension. The instantiation would proceed as follows:

  $$\forall\ k \in 1..n\ x_k = \min(x'_k),\ y_k = \min(y'_k)$$

  Where the index k indicates the k-th highest circuit, while x'_k, y'_k represent the possible positions for that circuit.

  ```
  array[1..n] of var int: sorted_coordsX = reverse(sort_by(coordsX,
      dimsY));
  array[1..n] of var int: sorted_coordsY = reverse(sort_by(coordsY,
      dimsY));
  array[1..2*n] of var int: variables = [if (i mod 2 = 0) then
      sorted_coordsX[i div 2] else sorted_coordsY[(i+1) div 2] endif
    | i in 1..2*n];
  ann: search_ann_dec_height = int_search(variables, input_order,
      indomain_min);
  ```

- **Model H4**. It defines the search in a slightly different way. For all the rectangles ordered by height, it firstly decides the value for the y coordinate. Then, the x coordinate of all circuits is instantiated following the same ordering. The value selection would proceed as follows:

  $$\forall\ k \in 1..n\ y_k = \min(y'_k)$$

  $$\forall\ k \in 1..n\ x_k = \min(x'_k)$$

  Where the index k indicates the k-th highest circuit, while x'_k, y'_k represent the possible positions for that circuit.

  ```
  ann: search_ann_l_from_min = int_search([l], input_order,
      indomain_min, complete);
  ann: search_ann_y_by_heigth = int_search(reverse(sort_by(coordsY,
      dimsY)), input_order, indomain_min);
  ann: search_ann_x_by_heigth = int_search(reverse(sort_by(coordsX,
      dimsY)), input_order, indomain_min);
  ```

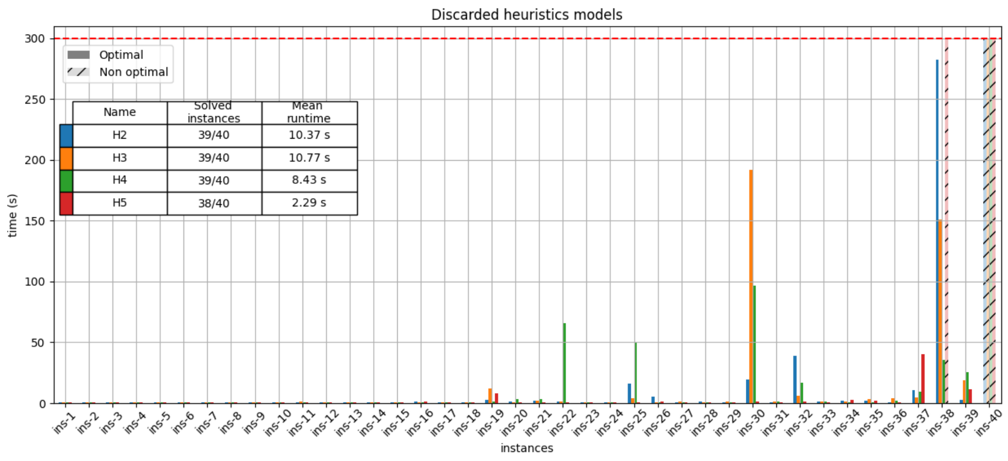
- **Model H5**. It uses an heuristic similar to the previous one. The very same procedure is carried out, though the circuits are ordered by their width. So the instantiation proceeds as follows:

$$\forall\ k \in 1..n\ y_k = \min(y'_k)$$

$$\forall\ k \in 1..n\ x_k = \min(x'_k)$$

  Where the index k indicates the k-th widest circuit, while x'_k, y'_k represent the possible positions for that circuit.

```
ann: search_ann_l_from_min = int_search([l], input_order,
    indomain_min, complete);
ann: search_ann_y_by_width = int_search(reverse(sort_by(coordsY,
    dimsX)), input_order, indomain_min);
ann: search_ann_x_by_width = int_search(reverse(sort_by(coordsX,
    dimsX)), input_order, indomain_min);
```



	Name	Solved instances	Mean runtime
🟦	H2	39/40	10.37 s
🟧	H3	39/40	10.77 s
🟩	H4	39/40	8.43 s
🟥	H5	38/40	2.29 s

Figure 6: Comparison of the results concerning models with discarded heuristics

2.5.3 Gecode heuristics

Since the solver *Chuffed* still doesn't support some value choice heuristics such as picking a random value in the domain, some other tests were performed trying out different heuristics, exploiting randomness and restarting, through the solver *Gecode*. More specifically, we imposed a search heuristic over the y_i and x_i of each circuit i by imposing:

- A `dom_w_deg` selection over the variables, which chooses the variable with the smallest value of domain size divided by weighted degree, which is the number of times it has been in a constraint that caused failure earlier in the search.
- The option `indomain_random`, which assigns the variable a random value from its domain.

- A `restart_luby` heuristic function to decide how frequently the search of a solution has to restart. A `scale` parameter is assigned to the function to determine after how many visited nodes of the tree search the restart has to be triggered. At each iteration i, the search is restarted after $L_i \cdot scale$ nodes have been visited, where L_i is the i-th element of the luby sequence.

The results though turned out to be quite mediocre.

2.5.4 Results

As shown in figure 5, the performances with the use of solver *Chuffed* applying the free search over the heuristics of Model H1 improve dramatically. A total of 39 out 40 instances are optimally soloved with an average time of 2.02 seconds.

Figure 6 expresses that similar but moderately worse results are obtained with the discarded heuristics of section 2.5.2.

Given the results, **Model H1** has finally been selected to solve the VLSI problem without rotations through a CP methodology.

2.6 Rotation variant

In order to solve the rotation variant of the problem, allowing the circuits to be rotated of 90°, the best model (Model H1) of the original variant was slightly modified.

Many tests were performed on different general and symmetry breaking constraints. In the following sections just the newly introduced variables and constraints will be described. Moreover, the edits applied to the constraints of the previous model that solves the no-rotation variant will be shown as well.

2.6.1 Variables

A few more variables were introduced in order to take into account the orientation of the circuits.

- o_i, representing whether the i-th circuit is rotated of 90° ($o_i = 1$) or not ($o_i = 0$)
- w'_i, is the actual width of the circuit, based on its orientation determined by o_i
- h'_i, is the actual height of the circuit, based on its orientation determined by o_i

2.6.2 Constraints

Some new constraints are introduced to handle the new degree of freedom of the problem. On the other hand, some others, also present in the original variant, have been slightly modified.

The newly introduced constraints impose that:

- The actual horizontal and vertical dimensions of each circuit i (w'_i and h'_i) are either equal to the original width (w_i) or to the original height (h_i). This constraint is used to reduce the domains of the new variables w'_i and h'_i:
$$\forall i \in 1..n,\ w'_i = w_i \lor w'_i = h_i$$
$$\forall i \in 1..n,\ h'_i = w_i \lor h'_i = h_i$$
- A circuit i is not rotated if and only if its actual horizontal dimension w'_i is equal to its original width w_i and its actual vertical dimension h'_i is equal to its original height h_i:
$$\forall i \in 1..n,\ o_i = 0 \iff w'_i = w_i \land h'_i = h_i$$
- If a circuit i is rotated its actual horizontal and vertical dimensions (w'_i and h'_i) are respectively equal to its vertical and horizontal original ones (h_i and w_i). The one way implication is used otherwise square circuits would need to be both rotated and not rotated, which is obviously unsatisfiable:
$$\forall i \in 1..n,\ o_i = 1 \implies w'_i = h_i \land h'_i = w_i$$

Regarding the modified constraints:

- The global constraints (CP 4) and (CP 6) are respectively modified to consider the real dimensions as follows:
$$\texttt{cumulative}(y, h', w', w)$$
$$\texttt{diffn}(x, y, w', h')$$
- The in-bounds constraints (CP 2) and (CP 3) are modified as well:
$$\forall\ i \in 1..n,\ x_i + w'_i \leq w$$
$$l = \max_{i \in 1..n}(y_i + h'_i)$$

Many symmetry breaking constraint specific for the rotation variant were tested such as:

- Square circuits shall not be rotated:
$$\forall i \in 1..n,\ w_i = h_i \implies o_i = 0$$
- Circuits of the same actual dimension shall not be swapped:
$$\forall\ i,\ j \in 1..n \land i < j \land w'_i = w'_j \land h'_i = h'_j,\ x_i \leq x_j \land (x_i < x_j \lor y_i \leq y_j)$$

None of these constraints improved the performances of the model, so just the symmetry breaking constraints of the non-rotation variant were kept (CP 7 and CP 8).

Many other general constraints were tested too, but not reported here. All the implementations are available in the source code.

2.6.3 Heuristics

The following heuristics were selected for the final model.

- The heuristic to search for the length of the plate l from the lowest possible value expressed in section 2.5.1 was kept.

  ```
  ann: search_ann_l_from_min = int_search([l], input_order, indomain_min, complete);
  ```

- A new heuristic was added in order to instantiate the orientation of the circuits before placing them. In particular, the rotation of the circuits with the largest area is decided first.

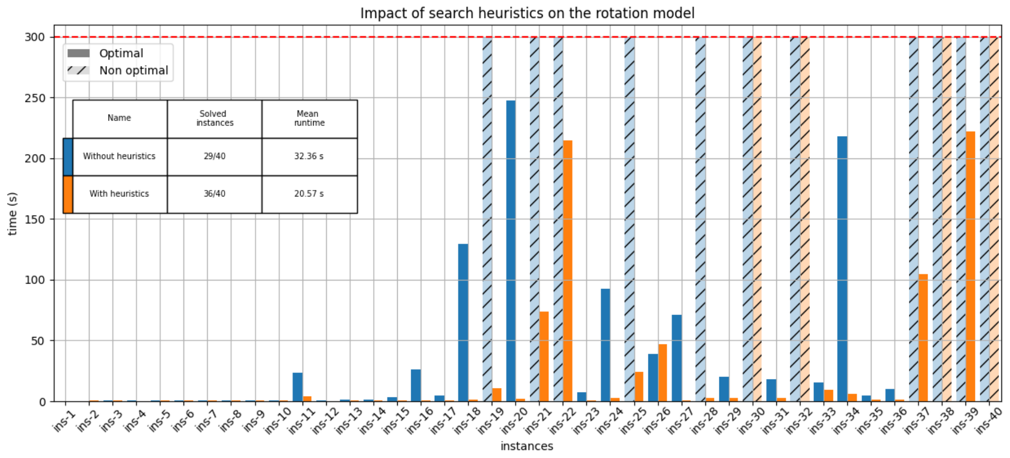
  ```
  ann: search_ann_rotations = bool_search(reverse(sort_by(rotations, areas)), input_order
  , indomain_min);
  ```

  Where `rotations` is the orientation vector o and `areas` is the vector containing the areas of the circuits, so $\texttt{areas}_i = w_i \cdot h_i$ is the area of the i-th circuit.

- Finally, the heuristic to place the circuits in the plate is similar to the one that was selected for the best model of the non-rotation variant. Again, the coordinates are interleaved and the circuits are ordered by their maximum dimension.

  ```
  array[1..n] of var int: sorted_coordsX = reverse(sort_by(coordsX,
      maxDims));
  array[1..n] of var int: sorted_coordsY = reverse(sort_by(coordsY,
      maxDims));
  array[1..2*n] of var int: variables = [if (i mod 2 = 0) then
      sorted_coordsY[i div 2] else sorted_coordsX[(i+1) div 2] endif |
      i in 1..2*n];
  ```

Some other heuristics were tested too (again available in the source code) though they were discarded due to their worse performances.



	Name	Solved instances	Mean runtime
🟦	Without heuristics	29/40	32.36 s
🟧	With heuristics	36/40	20.57 s

Figure 7: Comparison between the rotation model using symmetry breaking and lacking search heuristics with the final model that introduces them.

2.6.4 Results

As shown in figure 7 the base model with symmetry breaking constraints is outperformed by the final model also performing search heuristics, which solves 36 out of the 40 instances in an average solve time of 20.57 seconds.

3 SAT

In this section, the VLSI problem is solved using the Boolean SATisfiability approach. It consists in writing a SAT encoding for the problem and then giving it to a SAT solver for finding a satisfiable assignment, i.e. model, if any.

The *z3* [8] solver has been used. In particular, the *z3 Python library* [9] has been employed for writing and solving the encodings.

As mentioned in the introduction, an incremental approach has been followed, starting from the basic encoding, and then writing more and more complex encodings.

Before describing this step-by-step journey, it is important to point out that optimization is not natively supported in SAT: it has to be implemented by hand over the simple satisfaction encoding. This means that also the choices regarding the optimization procedure have an important role in finding the overall best alternative.

3.1 Preliminaries

This section describes some useful constraints.

- Constraint $at\_least\_one(X)$, where X is a list of boolean variables with length n (indices from 0 to $n-1$). It ensures that at least one variable in X is True. It is simply encoded as a disjunction of all the variables.
- Constraint $at\_most\_one(X)$, where X is a list of boolean variables with length n. It ensures that at most one variable in X is True. It is encoded using the *sequential encoding* [10].
- Constraint $exactly\_one(X)$, where X is a list of boolean variables with length n. It ensures that exactly one variable in X is True. It is implemented as the conjunction between $at\_least\_one(X)$ and $at\_most\_one(X)$.

3.2 Basic encoding

This section describes the basic encoding (BE).

3.2.1 Variables

The following SAT variables are used.

- c_{ijk}, where $i \in [0..w - w_{\min}]$, $j \in [0..l_{\max} - h_{\min}]$, $k \in [0..n-1]$. (i,j) represents two coordinates of the plate, k represents a circuit. c_{ijk} is True iff the circuit k is present in the cell (i,j) of the plate.
- q_{ijk}, where $i \in [0..w - w_{\min}]$, $j \in [0..l_{\max} - h_{\min}]$, $k \in [0..n-1]$. (i,j) represents two coordinates of the plate, k represents a circuit. q_{ijk} is True iff the left-bottom corner of the circuit k is put in the cell (i,j) of the plate.
- len_{kl}, where $k \in [0..n-1]$ and $l \in [0..l_{\max} - l_{\min}]$. k represents a circuit, l represents a length of the plate. len_{kl} is True iff the circuit k uses the length l of the plate. "A circuit uses a length" means that the circuit reaches or is above that length. Remark: actually, the length corresponding to a variable len_{kl} is not l, but $l + l_{\min}$. For going from an index l of len_{kl} to the actual length we have to add $l_{\min}$: $l + l_{\min}$. For going from an actual length l to an index of len_{kl} we have to subtract $l_{\min}$: $l - l_{\min}$. Example. $l_{\min} = 3$, $l_{\max} = 9$. For each circuit k, the variables len_{kl} are:
  - len_{k0} (corresponding to the actual length 3)
  - len_{k1} (corresponding to the actual length 4)
  - ...
  - len_{k6} (corresponding to the actual length 9)

3.2.2 Constraints

The following constraints are imposed.

- The left-bottom corner of each circuit k must be present exactly one time across the whole plate.

$$\bigwedge_{k=1}^{n} \textit{exactly\_one}(\{q_{ijk} \mid i \in [0..w - w_{\min}], j \in [0..l_{\max} - h_{\min}]\}) \tag{BE 1}$$

- In each cell (i,j) of the plate, at most one circuit is present. This reflects on both variables c_{ijk} and q_{ijk}. Denoting with C the set of all the possible cells (i,j) of the plate, the constraints are the following.

$$\bigwedge_{(i,j)\in C} \textit{at\_most\_one}(\{c_{ijk} \mid k \in [0..n-1]\}) \quad \bigwedge_{(i,j)\in C} \textit{at\_most\_one}(\{q_{ijk} \mid k \in [0..n-1]\}) \tag{BE 2}$$

  It is important to notice that these two constraints are non-necessary, since they are implied from the constraint BE 4: they could be removed.

- For each circuit k, its left-bottom corner can't be put on the cell (i,j) if the circuit would go out the plate, namely if $i + w_k - 1 \geq w$ or $j + h_k - 1 \geq l_{\max}$. Let C_k the set of cells (i,j) in which the left-bottom corner of k can't be put, then the constraint is the following.

$$\bigwedge_{k=1}^{n} \bigwedge_{(i,j)\in C_k} \neg q_{ijk} \tag{BE 3}$$

- For each circuit k and for each cell (i,j), let C_{ijk} be the set of cells $(i^{'},j^{'})$ covered by k if its left-bottom corner was put in (i,j). Then, the following equivalence must be ensured: for each circuit k and for each cell (i,j), the lower left-bottom corner of k is put in (i,j) iff all the cells in C_{ijk} are covered by k and no cell in C_{ijk} is covered by a circuit different from k. Denoting with C the set of all the possible cells (i,j) of the plate, the constraint is the following.

$$\bigwedge_{k=1}^{n} \bigwedge_{(i,j)\in C} (q_{ijk} \iff (\bigwedge_{(i^{'},j^{'})\in C_{ijk}} c_{i^{'}j^{'}k}) \wedge (\bigwedge_{(i^{'},j^{'})\in C_{ijk},k^{'}\neq k} \neg c_{i^{'}j^{'}k^{'}})) \tag{BE 4}$$

  It is important to point out that the equivalence is non-necessary: a simple implication would be enough.

- For each circuit k and for each cell (i,j), if the left-bottom corner of k was positioned in (i,j), then its reached length would be $j+h_k$. So, we must ensure the following constraint: if the left-bottom corner of k is positioned in (i,j), then all the lengths up to $j+h_k$ are reached by k and all the lengths above $j+h_k$ are not reached by k. This constraint is ensured by means of the variables len_{kl}, remembering that the index of len_{kl} corresponding to the actual length l is $l-l_{\min}$.

$$\bigwedge_{k\in[0..n-1]} \bigwedge_{(i,j)\in C} (q_{ijk} \implies (\bigwedge_{l\in[0..j+h_k-l_{\min}]} \text{len}_{kl}) \wedge (\bigwedge_{l\in[j+h_k-l_{\min}+1..l_{\max}-l_{\min}]} \neg\text{len}_{kl})) \tag{BE 5}$$

  It is important to point out that the second part of the right-hand side of the implication, namely $\bigwedge_{l\in[j+h_k-l_{\min}+1..l_{\max}-l_{\min}]} \neg\text{len}_{kl}$, is non-necessary: it can be removed.

3.2.3 Optimization procedure

At the beginning, a very naive optimization procedure has been used: simple cycle, looping through all the possible solutions, keeping the best one. At each iteration, the solver is created and run from scratch, with additional constraints imposing to search for a solution different from the ones already found (the already found solutions are not feasible anymore). This optimization procedure can't be used in practice, because too slow.
Then, a better optimization procedure has been built: non-incremental linear search starting from the top (i.e. $l_{\max}$). Same structure: cycle in which at each iteration the solver is created and run from scratch. But now, at each iteration, the current best length of the plate (i.e. l) is given to the solver as upper bound for the length of the plate (i.e. $l_{\max}$). (Actually, $l-1$ is given as $l_{\max}$). Then, at each iteration, l is decreased by one, until finding UNSAT. At the beginning, $l_{\max}$ is computed as described in 1.4.
It is important to point out that these first two optimization procedures don't perform incremental solving: at each iteration, the solver is created and run from scratch. It is also important to remark that, with these first two optimization procedures, the SAT variables len_{kl} and the corresponding constraints are not used. These variables are introduced exactly for having an incremental optimization procedure.

Incremental linear search starting from the top(i.e. starting from $l_{\max}$). The optimization procedure has been further improved, thanks to the variables len_{kl}: they allow to perform incremental solving. Indeed, the solver is created only one time, at the beginning. Then, a cycle is performed, in which, at each iteration, new constraints are injected into the solver. These constraints are the following.

- Constraints imposing that the lengths of the plate bigger or equal than the current best length are not feasible anymore: in this way, the next found solution, if any, is for sure better than the previous one. These constraints are built using the len_{kl} variables. Let l be the length of the plate in the new found solution, and let l_{old} be the length of the plate in the previous found solution (if only a single solution has been found since now, then l_{old} is equal to $l_{\max}$). The following equation for sure holds: $l < l_{\text{old}}$. Then, this constraint consists in ensuring that all the variables $\text{len}_{kl'}$, with $l^{'}$ from l (included) to l_{old} (excluded), are False. In doing so, we have to carefully compute the indices corresponding to the actual lengths l and l_{old}: indices $l - l_{\min}$ and $l_{\text{old}} - l_{\min}$.

$$\bigwedge_{k\in[0..n-1],l'\in[l-l_{\min}..l_{\text{old}}-l_{\min}-1]} \neg\text{len}_{kl'} \tag{BE 6}$$

- Constraints imposing that the already found solutions are not feasible anymore (like in the previous optimization procedures). In particular, let M be the set of variables q_{ijk} which are True in a certain solution (basically, M represents that model/solution). Then, the constraint representing that solution is the following:

$$\bigwedge_{q_{ijk}\in M} q_{ijk} \wedge \bigwedge_{q_{ijk}\notin M} \neg q_{ijk} \tag{BE 7}$$

  For imposing that a solution is not feasible anymore, this constraint is built and its negated version is given to the solver.
  These constraints are actually non-necessary, since BE 6 implies BE 7.

As in the previous optimization procedure, the search starts from $l_{\max}$ computed as described in 1.4. The cycle ends when UNSAT is found.

3.2.4 Removing the non-necessary constraints

In the current encoding, several constraints are non-necessary, meaning that they can be removed. Therefore, it is interesting to see if these constraints are redundant, which means that their removal improves the performances of the encoding. In particular, we have tried to remove several possible combinations of non-necessary constraints, and then we have kept the best alternative.

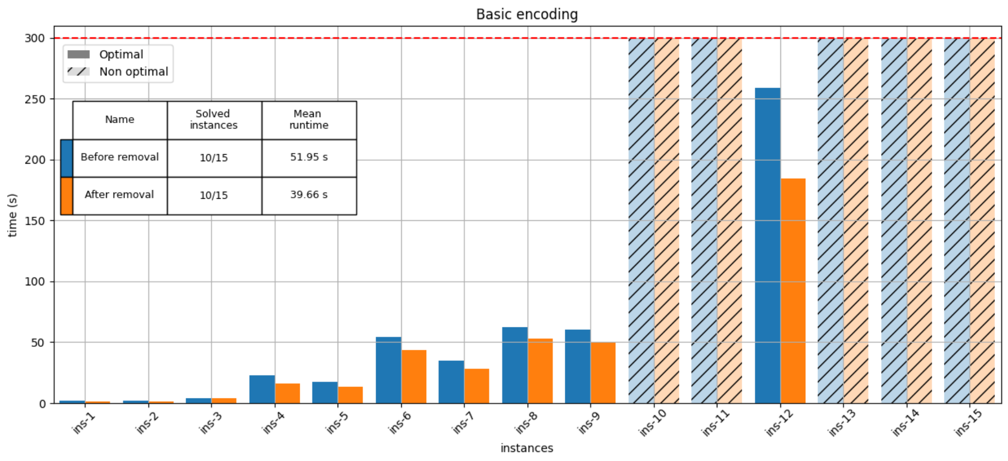
In the end, removing some of the non-necessary constraints seems to improve the situation. The best combination of non-necessary constraints to remove is the following: BE 2, BE 7.

3.2.5 Results

In figure 8, the results of the basic encoding, before and after the removal of the non-necessary constraints, are shown. As it can be seen, removing some non-necessary constraints slightly improves the performances. Nonetheless, the results are pretty bad.

3.3 Improving the basic encoding

In this section, several improvements of the basic encoding are described.



	Name	Solved instances	Mean runtime
🟦	Before removal	10/15	51.95 s
🟧	After removal	10/15	39.66 s

Figure 8: Results of the basic encoding, before and after the removal of the non-necessary constraints

3.3.1 Reducing the number of variables

The aim of these improvements is to reduce the number of variables. Indeed, in the current encoding, the number of variables is huge.

- There are two grid of variables: c_{ijk} and q_{ijk}. The number of variables is: $2 * n * (w - w_{\text{min}} + 1) * (l_{\text{max}} - h_{\text{min}} + 1)$.
- The variables len_{kl} are defined for each circuit k. The number of variables is: $n * (l_{\text{max}} - l_{\text{min}+1})$.

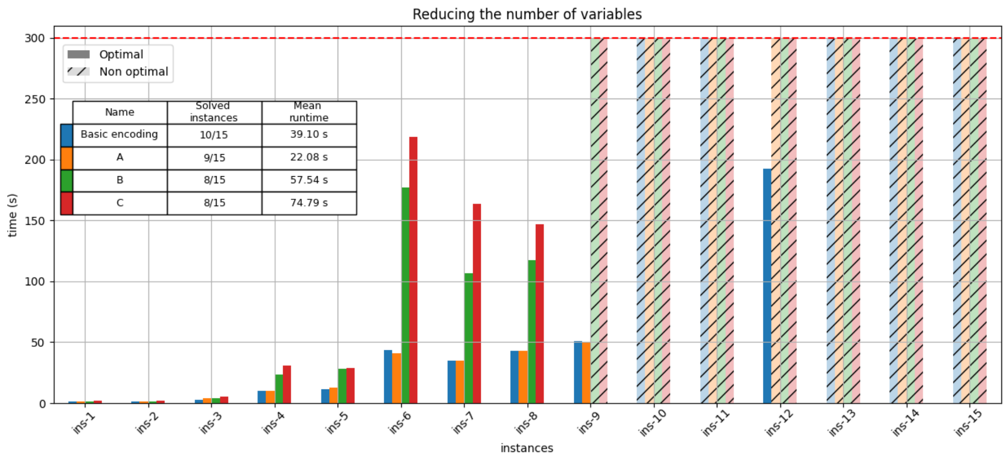
For reducing the number of variables, three new encodings have been designed.

- Encoding A. The variables c_{ijk}, q_{ijk} and the corresponding constraints are kept. Instead, the variables len_{kl} are changed into the variables len_l, where $l \in [0..l_{\text{max}} - l_{\text{min}}]$. The semantic of these variables is different: len_l is True iff the length l is the maximum length of the plate, meaning that at least one circuit reaches it and no circuit goes above it. Actually, the actual length corresponding to the variable len_l is not l, but $l + l_{\text{min}}$: l is just an index on the variables len_l. Of course, different constraints must be ensured: these constraints are now briefly described.
  - Exactly one variable len_l must be True.
  - For each possible length l, len_l is True iff there is at least one circuit reaching that length and no circuit goes above that length.
  - In the incremental solving optimization procedure, the constraints imposing that the lengths of the plate bigger or equal than the current best length l are not feasible anymore must be changed, since the variables are changed. This is implemented by ensuring that at least one of the variables $\text{len}_{l'}$, with $l^{'} < l$, is True. In doing so, we have to carefully compute the index corresponding to the actual length l: $l - l_{\text{min}}$.
- Encodings B and C. The variables q_{ijk} are removed: only one grid of variables is kept, which is c_{ijk}. The aim is to try to eliminate the redundancy of having two grid of variables. Of course, different constraints must be ensured: they are now briefly described.

- In each cell (i, j), at most one variable c_{ijk} must be True.
- For each circuit k, all the possible configurations in which that circuit can be are taken, and then it is imposed that exactly one of these configurations is True. A "configuration" is an admissible position of that circuit on the plate, consisting in a conjunction of two formulas: formula saying that all the positions c_{ijk} covered by k in that configuration are True; formula saying that all the positions c_{ijk} not covered by k in that configuration are False. For each circuit k, all its possible configurations are generated, which are formulas, and then the *exactly_one* constraint is imposed on these formulas.

The encoding B consists in the usage of the only variables c_{ijk} plus the variables len_{kl}. Instead, the encoding C consists in the usage of the only variables c_{ijk} plus the variables len_l, as described in the encoding A.

Figure 9 shows the results of these attempts. As it can be seen, the results are not improved. This



	Name	Solved instances	Mean runtime
🟦	Basic encoding	10/15	39.10 s
🟧	A	9/15	22.08 s
🟩	B	8/15	57.54 s
🟥	C	8/15	74.79 s

Figure 9: Results of reducing the number of variables of the basic encoding

probably means that, even if the number of variables is smaller, the number of constraints is bigger: on the whole, the encoding is heavier.

3.3.2 Symmetry breaking constraints

In this section, some symmetry breaking constraints are ensured. In particular, we want to break the following three symmetries:

- Horizontal flip of the plate
- Vertical flip of the plate
- 180° flip of the plate

For ensuring these symmetry breaking constraints, the lexicographic ordering between lists of lists of boolean variables is used. So, let's define some auxiliary constraints.

1. Constraint $eq(X, Y)$, where X, Y are two lists of boolean variables, with same length n (indices from 0 to $n-1$). Constraint ensuring that the lists of boolean variables X and Y are exactly equal. This means that each pair of corresponding variables $(X[i], Y[i])$ are equivalent: $X[i]$ iff $Y[i]$.

$$\bigwedge_{i \in [0..n-1]} X[i] \;\Leftrightarrow Y[i]$$

2. Constraint $lex\_lesseq(X, Y)$, where X, Y are two lists of boolean variables, with same length n (indices from 0 to $n-1$). Constraint ensuring that X is lexicographically less or equal than Y, meaning that either X and Y are equal or all the variables $X[i]$ are equal to the corresponding variables $Y[i]$ up to a certain index j, and in j we have that $X[j]$ is True while $Y[j]$ is False. In practice, the constraint consists in the conjunction of the following constraints.
    - $X[0] \vee \neg Y[0]$
    - $X[0] \equiv Y[0] \implies X[1] \vee \neg Y[1]$
    - ...
    - $\bigwedge_{i \in [0..n-1]} X[i] \equiv Y[i] \implies X[n-1] \vee \neg Y[n-1]$

3. Constraint $lex\_lesseq\_compound(X, Y)$, where X, Y are two lists of lists of boolean variables, with same length n (indices from 0 to $n-1$). Constraint ensuring that X is lexicographically less or equal than Y, meaning that either X and Y are perfectly equal or all the lists $X[i]$ are equal to the corresponding lists $Y[i]$ up to a certain index j, and in j we have that the list $X[j]$ is lexicographically less or equal than the list $Y[j]$. In practice, the constraint consists in the conjunction of the following constraints.
    - $lex\_lesseq(X[0], Y[0])$
    - $eq(X[0], Y[0]) \implies lex\_lesseq(X[1], Y[1])$
    - ...
    - $\bigwedge_{i \in [0..n-1]} eq(X[i], Y[i]) \implies lex\_lesseq(X[n-1], Y[n-1])$

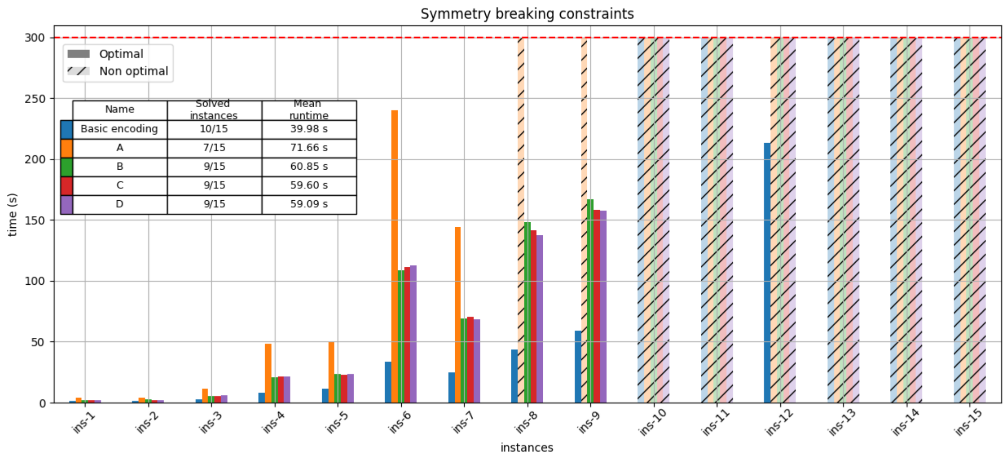
With this defined constraint, the symmetry breaking constraints can be defined as follows.

- First of all, a list of lists of boolean variables, containing the only allowed permutation of circuits, is defined.
$$correct\_positions = [q[i][j] \mid i \in [0..w - w_{\min}], j \in [0..l_{\max} - h_{\min}]]$$
where $q[i][j]$ is a compact notation for denoting all variables q_{ijk} with fixed i, j and all possible k.
- Breaking the horizontal flip symmetry.
$$lex\_lesseq\_compound(correct\_positions, [q[w - w_{\min} - i][j] \mid i \in [0..w - w_{\min}], j \in [0..l_{\max} - h_{\min}]])$$
- Breaking the vertical flip symmetry.
$$lex\_lesseq\_compound(correct\_positions, [q[i][l_{\max} - h_{\min} - j] \mid i \in [0..w - w_{\min}], j \in [0..l_{\max} - h_{\min}]])$$
- Breaking the 180° flip symmetry.
$$lex\_lesseq\_compound(correct\_positions, [q[w - w_{\min} - i][l_{\max} - h_{\min} - j] \mid i \in [0..w - w_{\min}], j \in [0..l_{\max} - h_{\min}]])$$

So, different combinations of these symmetry breaking constraints are tried.

- Combination A: all the symmetry breaking constraints.
- Combination B: only horizontal flip symmetry breaking.
- Combination C: only vertical flip symmetry breaking.
- Combination D: only 180° flip symmetry breaking.

Figure 10 shows the results. As it can be seen, adding these symmetry breaking constraints makes the



	Name	Solved instances	Mean runtime
🟦	Basic encoding	10/15	39.98 s
🟧	A	7/15	71.66 s
🟩	B	9/15	60.85 s
🟥	C	9/15	59.60 s
🟪	D	9/15	59.09 s

Figure 10: Results of adding the symmetry breaking constraints to the basic encoding

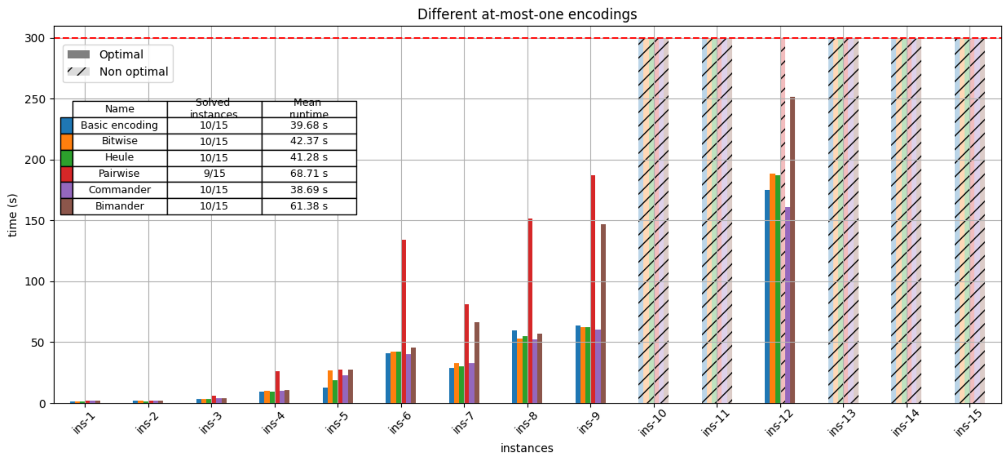
performances worse. An interpretation of this could be that these symmetry breaking constraints are big and complex relations, and they make heavier the overall encoding. Which was already pretty huge (due to two big grids of variables).

3.3.3 Different at-most-one encodings

In this next step, different encodings of the *at_most_one*(X) constraint are tried.

- *bitwise encoding* [10].
- *heule encoding* [10].
- *pairwise encoding* [10].
- *commander encoding* [10].
- *bimander encoding* [10].

Figure 11 shows the results. In the figure, "basic model" refers to the usage of the *sequential encoding*. As it can be seen, the usage of the *commander encoding* seems to slightly improve the current situation. Nonetheless, the results are still pretty poor.



	Name	Solved instances	Mean runtime
🟦	Basic encoding	10/15	39.68 s
🟧	Bitwise	10/15	42.37 s
🟩	Heule	10/15	41.28 s
🟥	Pairwise	9/15	68.71 s
🟪	Commander	10/15	38.69 s
🟫	Bimander	10/15	61.38 s

Figure 11: Results of using different encodings for the at-most-one constraint in the basic model

3.3.4 Improving the optimization procedure

As last attempt of improving the basic model, different optimization procedures are tried. Since now, the optimization procedure which has been used is the incremental linear search starting from the top (i.e. starting from $l_{\max}$). Now, two more sophisticated optimization procedures are tried.

- **Incremental binary search.** The solver is created only one time, at the beginning. Cycle in which, at each iteration, there are a certain lower bound (i.e. lb) and a certain upper bound (i.e. ub) for the length of the plate. In addition, there is also a length of interest l, computed as the middle between lb and ub: $l = \lceil \frac{lb+ub}{2} \rceil$. At each iteration, a new constraint is injected into the solver, ensuring that the actual length of the plate is smaller or equal than l: the actual length of the plate is in the interval $[lb..l]$. More precisely, this constraint is about putting to False all variables $\text{len}_{kl'}$ with $l^{'}$ from $l+1$ to the upper bound ub. As usual, the actual lengths l and ub must be transformed into the corresponding indices, by subtracting by $l_{\min}$.

$$\bigwedge_{k \in [0..n-1], l' \in [l+1-l_{\min}..ub-l_{\min}]} \neg \text{len}_{kl'}$$

  Then, after injecting this constraint, this current solver instance is solved.

  - If SAT, then the found solution is saved as the new best solution. Moreover, ub is updated as follows: $ub \longleftarrow l$.
  - If UNSAT, the last injected constraint (i.e. the one ensuring that the actual length of the plate is smaller or equal than l) is retracted from the solver, and a new constraint is instead added: constraint ensuring that the actual length of the plate is strictly bigger than l. More precisely, this constraint consists in ensuring that, for each length $l^{'}$ in $[lb..l+1]$, at least one variable among the n variables $\text{len}_{kl'}$ is True. As usual, the actual lengths lb and $l+1$ must be transformed into the corresponding indices, by subtracting by $l_{\min}$.

$$\bigwedge_{l' \in [lb-l_{\min}..l+1-l_{\min}]} \mathit{at\_least\_one}(\{\text{len}_{kl'} \mid k \in [0..n-1]\})$$

Finally, lb is updated as follows: $lb \longleftarrow l+1$.

These steps are performed at each iteration. At the beginning, lb is $l_{\min}$ and ub is $l_{\max}$. The cycle ends when lb becomes bigger than ub: in such case, the last found solution is the best solution (if no solution has been found, that problem instance is UNSAT).

- **Incremental linear search starting from the bottom** (i.e. starting from $l_{\min}$). The solver is created only one time, at the beginning. Cycle in which, at each iteration, there is a certain current l of interest. All the lengths below l have already been tested, with a failure: thus, if this l is SAT, then this l is the best possible length of the plate. At each iteration, new constraints are injected into the solver, ensuring that the actual length of the plate is smaller or equal than l. More precisely, this constraint is about putting to False all variables $\text{len}_{kl'}$, with $l^{'}$ from $l+1$ to the max length $l_{\max}$. As usual, the actual lengths $l+1$ and $l_{\max}$ must be transformed into the corresponding indices, by subtracting by $l_{\min}$.

$$\bigwedge_{k\in[0..n-1],l'\in[l+1-l_{\min}..l_{\max}-l_{\min}]} \neg\text{len}_{kl'}$$

  Then, after injecting this constraint, this current solver instance is solved.

  - If SAT, then the best possible solution has been found: the solving procedure exits.
  - If UNSAT, then the last injected constraint (i.e. the one ensuring that the actual length of the plate is smaller or equal than l) is retracted, and a new constraint is added: constraint ensuring that the actual length of the plate is strictly bigger than l. More precisely, this constraint consists in ensuring that at least one variable among the n variables $\text{len}_{k(l+1)}$ is True. As usual, the actual length $l+1$ is transformed into the corresponding index $l+1-l_{\min}$. Therefore, the constraint is the following:

$$at\_least\_one(\{\text{len}_{k(l+1-l_{\min})} \mid k \in [0..n-1]\})$$

    Finally, l is updated as follows: $l \leftarrow l+1$.

  These steps are performed at each iteration. At the beginning, l is $l_{\min}$. If, at some point, l becomes bigger than $l_{\max}$, this means that the problem instance is UNSAT.

It is important to notice that, in both cases, since some constraints must be added and retracted dynamically, the assertion stack has been used: in this way, assertion levels can be pushed into and popped out the stack.

Figure 12 shows the results of using these different optimization procedures, in combination with the *sequential encoding*. In the figure, "basic model" refers to the usage of the linear search from the top.

As it can be seen, the performances are improved, by using the linear search from the bottom: all the first 12 instances are solved.

Finally, these different optimization procedures are also tried with different encodings of the at-most-one constraint. The following combinations are tried.

- Combination A1: binary search with bitwise encoding.
- Combination A2: binary search with the commander encoding.
- Combination B1: linear search from the bottom with the bitwise encoding.

- Combination B2: linear search from the bottom with the commander encoding.

Figure 13 shows the results.

The performances are further improved, by using the linear search from the bottom with the bitwise encoding.

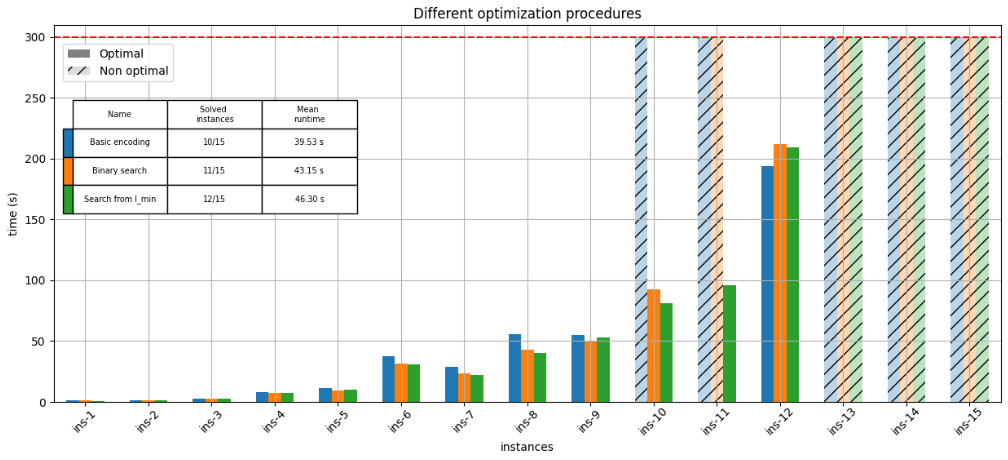


Figure 12: Results of using different optimization procedures for the basic model

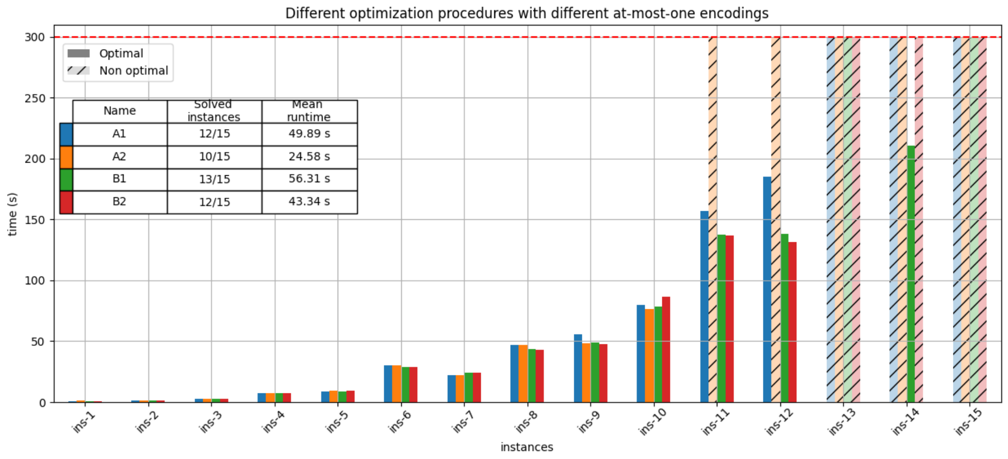


Figure 13: Results of using different optimization procedures in combination with different at-most-one encodings for the basic model

3.3.5 Final comments

Since now, the basic encoding has been exploited, and then several improvements have been carried out. Even if the results have been slightly improved, the performances are still pretty poor, and they are not satisfactory. For dramatically improving the current situation, a complete change of the current encoding must be carried out: this leads us to the **order encoding**.

3.4 Order encoding

This is a completely different encoding from the previous ones, and it dramatically boosts the performances. It has been taken from the paper *A SAT-based Method for Solving the Two-dimensional Strip Packing Problem* [11].

The main idea is the following. The basic CP model is considered. Namely, the one in which the non-overlapping of each couple of rectangles (i, j) is ensured by means of a disjunction of four constraints:

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

Then, these constraints are encoded into SAT using the **order encoding** (OE).

The notation used in this section follows the notation described in the introduction, which is the same notation of the paper.

3.4.1 Order encoding: general description

First of all, order encoding is described in general.

Let's suppose to have a general model, like a CP model. Order encoding consists in two steps.

1. First of all, the constraints are reduced into combinations of constraints of the form $x_i \leq c$, where x_i is a variable and c is a constant.
2. Then, each constraint $x_i \leq c$ is encoded as a SAT variable px_{ic}.

$$px_{ic} \leftrightarrow x_i \leq c$$

Let's consider the following example. Model consisting in a single constraint $x_1 + 1 \leq x_2$, where x_1 and x_2 are two variables in $\{0, 1, 2, 3\}$.

1. The given constraint is equivalent to the following one:

$$(x_2 > 0) \wedge (x_2 \leq 1 \implies x_1 \leq 0) \wedge (x_2 \leq 2 \implies x_1 \leq 1) \wedge (x_1 \leq 2)$$

Which is equivalent to:

$$\neg(x_2 \leq 0) \wedge (\neg(x_2 \leq 1) \vee x_1 \leq 0) \wedge (\neg(x_2 \leq 2) \vee x_1 \leq 1) \wedge x_1 \leq 2$$

2. Now, this constraint is encoded using SAT variables.

$$\neg px_{2,0} \land (px_{1,0} \lor \neg px_{2,1}) \land (px_{1,1} \lor \neg px_{2,2}) \land px_{1,2}$$

Actually, these constraints are not enough. Other constraints about the variables px_{ic} must be specified: the so-called *axiom constraints*. These constraints are about the implicit semantic of these variables. We have to encode the fact that, if px_{ic} is True, then all px_{id} with $d > c$ are True.

$$\forall c \quad (px_{ic} \implies px_{i(c+1)})$$

Which is equivalent to:

$$\forall c \quad (\neg px_{ic} \lor px_{i(c+1)})$$

In the example, the following constraints are added:

$$(\neg px_{1,0} \lor px_{1,1}) \land (\neg px_{1,1} \lor px_{1,2}) \land (\neg px_{2,0} \lor px_{2,1}) \land (\neg px_{2,1} \lor px_{2,2})$$

Given a model for our problem, consisting in an assignment of truth values to the boolean variables px_{ic}, how can the value assigned to each variable x_i be deduced? For each variable x_i, all the associated boolean variables px_{ic} are taken, which are scanned from the smallest c to the biggest c, and we stop when we find the first True boolean variable px_{id} : d is the value associated to the variable x_i. Why? Because all px_{ic} with $c < d$ are False, and all px_{ic} with $c \geq d$ are True.
In the example, if in the model we have:

$$px_{1,0} = \text{False}, \quad px_{1,1} = \text{True}, \quad px_{1,2} = \text{True}$$

then the value assigned to x_1 is 1.

3.4.2 Application of the order encoding for the VLSI problem

The following SAT variables are used.

1. px_{ie}, where i represents a circuit and e represents a value in $[0..w-1]$. $i \in [0..n-1]$, $e \in [0..w-1]$.
   px_{ie} is True iff $x_i \leq e$. Which means that the x coordinate of the left-bottom corner of the circuit i has been placed less or equal than e.
2. py_{if}, where i represents a circuit and f represents a value in $[0..l_{\max}-1]$. $i \in [0..n-1]$, $f \in [0..l_{\max}-1]$.
   py_{if} is True iff $y_i \leq f$. Which means that the y coordinate of the left-bottom corner of the circuit i has been placed less or equal than f.
3. lr_{ij}, where i and j are two circuits. $i, j \in [0..n-1]$.
   lr_{ij} is True iff the circuit i has been placed on the left of the circuit j. Namely, $x_i + w_i \leq x_j$.
4. ud_{ij}, where i and j are two circuits. $i, j \in [0..n-1]$.
   ud_{ij} is True iff the circuit i has been placed at the downward to the circuit j. Namely, $y_i + h_i \leq y_j$.

The fact that two circuits can't overlap is encoded by imposing that, for each pair of circuits i and j $(i < j)$, at least one among $\{lr_{ij}, lr_{ji}, ud_{ij}, ud_{ji}\}$ is True. So, the constraint is the following:

$$\forall i, j \in [0..n-1] \quad lr_{ij} \lor lr_{ji} \lor ud_{ij} \lor ud_{ji} \qquad \text{(OE 1)}$$

Now, for each pair of circuits i and j $(i < j)$, the "meaning" of each variable $lr_{ij}, lr_{ji}, ud_{ij}, ud_{ji}$ must be encoded.

1. If lr_{ij} is True, then we have that:
$$x_j \le e + w_i \implies x_i \le e$$
for each possible e.
Which is equivalent to:
$$\forall e \quad \neg px_{j(e+w_i)} \vee px_{ie}$$
On the whole, the following constraint must be added:
$$\forall i,j \in [0..n-1] \quad \forall e \in [0..w-1] \quad (\neg lr_{ij} \vee \neg px_{j(e+w_i)} \vee px_{ie}) \tag{OE 2}$$
The exact same reasoning is applied also to lr_{ji}, ud_{ij}, ud_{ji}.
2. For the variable lr_{ji}, the following constraint is added:
$$\forall i,j \in [0..n-1] \quad \forall e \in [0..w-1] \quad (\neg lr_{ji} \vee \neg px_{i(e+w_j)} \vee px_{je}) \tag{OE 3}$$
3. For the variable ud_{ij}, the following constraint is added:
$$\forall i,j \in [0..n-1] \quad \forall f \in [0..l_{\max}-1] \quad (\neg ud_{ij} \vee \neg py_{j(f+h_i)} \vee py_{if}) \tag{OE 4}$$
4. For the variable ud_{ji}, the following constraint is added:
$$\forall i,j \in [0..n-1] \quad \forall f \in [0..l_{\max}-1] \quad (\neg ud_{ji} \vee \neg py_{i(f+h_j)} \vee py_{jf}) \tag{OE 5}$$

Finally, the axiom constraints about px_{ie} and py_{if} for the ordering encoding must be enforced. For each circuit i, the following constraint must be ensured:
$$\forall i \in [0..n-1] \quad \forall e \in [0..w-2] \quad (\neg px_{ie} \vee px_{i(e+1)}) \tag{OE 6}$$
and also the following constraint:
$$\forall i \in [0..n-1] \quad \forall f \in [0..l_{\max}-2] \quad (\neg py_{if} \vee py_{i(f+1)}) \tag{OE 7}$$

3.4.3 Reducing the domains

The constraints described in 3.4.2 are modified, for reducing the domains and boosting the performances. The only constraint which is not touched is the constraint OE 1.

Group A of constraints. Group of constraints about the horizontal overlapping: overlapping along the width (i.e. lr_{ij}, lr_{ji}). The constraints OE 2 and OE 3 are taken into account and then modified.
If $w - w_i - w_j < 0$, this means that the circuits i, j can't be placed one on the left of the other (the sum of their widths exceeds the total width, they don't fit into the plate). In this case, the constraints OE 2 and OE 3 are not put, but simply this constraint is added: $\neg lr_{ij} \wedge \neg lr_{ji}$. So, we have:
$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j < 0, \quad \neg lr_{ij} \wedge \neg lr_{ji} \tag{OE A1}$$
Instead, if $w - w_i - w_j \ge 0$, the constraints OE 2 and OE 3 are put. But they are put only for $e \in [0..w - w_i - w_j]$, since from $w - w_i - w_j + 1$ up to w, the circuits i, j can't be placed one on the left of the other, because they don't fit into the plate.
$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j \ge 0, \quad \forall e \in [0..w - w_i - w_j] \quad (\neg lr_{ij} \vee \neg px_{j(e+w_i)} \vee px_{ie}) \tag{OE A2}$$

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j \geq 0, \quad \forall e \in [0..w - w_i - w_j] \quad (\neg lr_{ji} \vee \neg px_{i(e+w_j)} \vee px_{je}) \quad \text{(OE A3)}$$

If $w - w_i - w_j \geq 0$, also other additional constraints are enforced.
For the constraint OE 2, also the following constraint is explicitly ensured:

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j \geq 0, \quad \forall e \in [0..w_i - 1] \quad (\neg lr_{ij} \vee \neg px_{je}) \qquad \text{(OE A4)}$$

The meaning is: when $e < w_i$, if i is placed on the left of j, then $x_j > e$.
This same reasoning is applied for the constraint OE 3:

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j \geq 0, \quad \forall e \in [0..w_j - 1] \quad (\neg lr_{ji} \vee \neg px_{ie}) \qquad \text{(OE A5)}$$

In all these constraints, $i \neq j$ is assumed.

Group B of constraints. Group of constraints about the vertical overlapping: overlapping along the height (i.e. lr_{ij}, lr_{ji}). The constraints OE 4 and OE 5 are taken into account and then modified.
Exact same reasoning applied for the group A. The following constraints are obtained.

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad l_{\max} - h_i - h_j < 0, \quad \neg ud_{ij} \wedge \neg ud_{ji} \qquad \text{(OE B1)}$$

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad l_{\max} - h_i - h_j \geq 0, \quad \forall f \in [0..l_{\max} - h_i - h_j] \quad (\neg ud_{ij} \vee \neg py_{j(f+h_i)} \vee py_{if}) \quad \text{(OE B2)}$$

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad l_{\max} - h_i - h_j \geq 0, \quad \forall f \in [0..l_{\max} - h_i - h_j] \quad (\neg ud_{ji} \vee \neg py_{i(f+h_j)} \vee py_{jf}) \quad \text{(OE B3)}$$

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad l_{\max} - h_i - h_j \geq 0, \quad \forall f \in [0..h_i - 1] \quad (\neg ud_{ij} \vee \neg py_{jf}) \qquad \text{(OE B4)}$$

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad l_{\max} - h_i - h_j \geq 0, \quad \forall f \in [0..h_j - 1] \quad (\neg ud_{ji} \vee \neg py_{if}) \qquad \text{(OE B5)}$$

In all these constraints, $i \neq j$ is assumed.

Group C of constraints. Group of general axiom constraints about the order encoding variables. The constraints OE 6 and OE 7 are taken into account and then modified.
Regarding the constraint OE 6, if $e \in [w - w_i..w - 1]$, than for sure $x_i \leq e$ holds (i.e. px_{ie}), because that circuit i is for sure placed before e. Therefore, the following constraint is put:

$$\forall i \in [0..n-1] \quad \forall e \in [w - w_i..w - 1] \quad px_{ie} \qquad \text{(OE C1)}$$

At the same time, the constraint OE 6 is enforced only for $e \in [0..w - w_i - 1]$:

$$\forall i \in [0..n-1] \quad \forall e \in [0..w - w_i - 1] \quad (\neg px_{ie} \vee px_{i(e+1)}) \qquad \text{(OE C2)}$$

The exact same reasoning is applied to the constraint OE 7: two new constraints:

$$\forall i \in [0..n-1] \quad \forall f \in [l_{\max} - h_i..l_{\max} - 1] \quad py_{if} \qquad \text{(OE C3)}$$

$$\forall i \in [0..n-1] \quad \forall f \in [0..l_{\max} - h_i - 1] \quad (\neg py_{if} \vee py_{i(f+1)}) \qquad \text{(OE C4)}$$

3.4.4 Variables for building the optimization procedure

For building the optimization procedure, the new variables ph_o are introduced, as described in the same paper [11].

Variables ph_o, where o represents a length. $o \in [0..l_{\max} - l_{\min}]$.
Actually, o does not represent an actual length, but an index (on $[l_{\min}..l_{\max}]$). The corresponding actual

length is $l_o = o + l_{\min}$. So: for going from actual length l_o to index o we do $o = l_o - l_{\min}$; for going from index o to actual length l_o we do $l_o = o + l_{\min}$.
ph_o is True iff each circuit is below or at the same level of the length o. More precisely, ph_o is True iff each circuit is below or at the same level of the length $l_o = o + l_{\min}$. Formally: ph_o is True iff $\forall i \quad y_i + h_i \leq l_o$.

The constraints on these new variables are the following.
For each circuit i and for each $o \in [0..l_{\max} - l_{\min}]$, we must impose that, if ph_o is True, then the circuit i is below or at most the same level of the length $l_o = o + l_{\min}$: formally, $y_i + h_i \leq l_o$. The constraint is the following.

$$\forall i \in [0..n-1] \quad \forall o \in [0..l_{\max} - l_{\min}] \quad \neg ph_o \vee py_{i(o+l_{\min}-h_i)} \tag{OE O1}$$

Then, the usual axiom constraints for the order encoding are put.

$$\forall o \in [0..l_{\max} - l_{\min} - 1] \quad (\neg ph_o \vee ph_{(o+1)}) \tag{OE O2}$$

3.4.5 Optimization

Two different optimization procedures have been developed.

1) Incremental binary search. Same optimization procedure described in 3.3.4. At each iteration, there are lb, ub, l, and the new constraint which is injected into the solver, ensuring that the actual length of the plate is smaller or equal than l, is the following.

$$ph_{(l-l_{\min})} \tag{OE O3}$$

If UNSAT, that constraint is retracted and its negated version is injected into the solver.

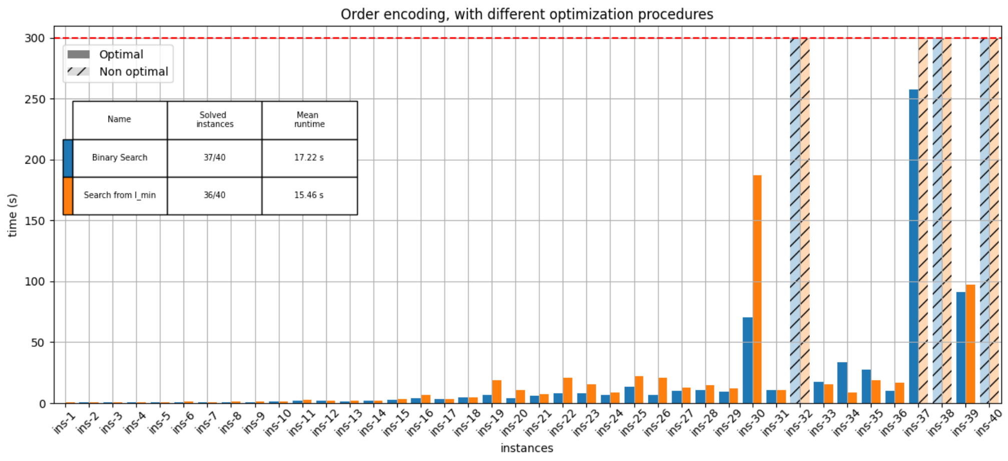
2) Incremental linear search starting from the bottom (i.e. $l_{\min}$). Same optimization procedure described in 3.3.4. At each iteration, the new constraint injected into the solver, ensuring that the actual length of the plate is smaller or equal than l, is the constraint OE O3. If UNSAT, that constraint is retracted and its negated version is injected into the solver.

3.4.6 Results

Figure 14 shows the results achieved with order encoding. As it can be seen, the results have been dramatically improved: order encoding is a complete success. The strength of the order encoding relies on the fact that is a very light encoding. Despite the shape of the constraints written in the section 3.4.3, the encoding is pretty small and simple.
The final best combination is order encoding with incremental binary search: it solves 37 instances, with an average solving time of 17.22 seconds.
These are the final best results regarding the non-rotation VLSI problem in SAT. In the next section, the rotation variant is considered.

3.5 Rotation variant

Since now, the non-rotation variant of the VLSI problem as been considered. Instead, in this section, the rotation variant is addressed. Rotation variant: each circuit can be rotated by 90°, swapping the width



	Name	Solved instances	Mean runtime
🟦	Binary Search	37/40	17.22 s
🟧	Search from l_min	36/40	15.46 s

Figure 14: Results of using different optimization procedures with the order encoding

(i.e. w_i) with the height (i.e. h_i).
For solving this variant, the order encoding described in the previous section has been considered and modified.

3.5.1 Modification of the order encoding

First of all, a new set of variables is defined. Variables r_i, where i represents a circuit. $i \in [0..n-1]$.
r_i is True iff the circuit i has been rotated, meaning that w_i and h_i have been swapped.

Using these new variables, the constraints summed up in section 3.4.3 are modified (i.e. order encoding constraints with also the reduction techniques), taking into account the possibility of rotating the circuits.
Actually, the first constraint is not modified. The constraint

$$\forall i, j \in [0..n-1] \quad lr_{ij} \vee lr_{ji} \vee ud_{ij} \vee ud_{ji}$$

remains the same.
Instead, the remaining constraints are modified. Basically, the main idea is the following.

- For each constraint involving a single circuit i, two different alternatives must be generated: alternative in which the circuit is not rotated; alternative in which the circuit is rotated.
- For each constraint involving a pair of circuits (i, j), four different alternative must be generated: circuits i, j are both non-rotated; circuit j is rotated, while i is not; circuit j is rotated, while i is not; circuits i, j are both rotated.

So, all three groups of constraints A, B, C are modified according to this idea. Only the modifications on the constraints OE A1 and OE A2 are described.

- Constraint A1)

$$\forall i, j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j < 0, \quad \neg lr_{ij} \wedge \neg lr_{ji}$$

The constraint $\neg lr_{ij} \land \neg lr_{ji}$ must be modified, by imposing that this is True only if both rectangles have not been rotated. Therefore, the following constraint is obtained:

$$(\neg r_i \land \neg r_j \implies \neg lr_{ij}) \land (\neg r_i \land \neg r_j \implies \neg lr_{ji})$$

Which is equivalent to:

$$(r_i \lor r_j \lor \neg lr_{ij}) \land (r_i \lor r_j \lor \neg lr_{ji})$$

On the whole:

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j < 0, \quad (r_i \lor r_j \lor \neg lr_{ij}) \land (r_i \lor r_j \lor \neg lr_{ji})$$

This is only about a single possibility: both the circuits i and j are not rotated. There are three other combinations.

– j is rotated, while i is not.

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - h_j < 0, \quad (r_i \lor \neg r_j \lor \neg lr_{ij}) \land (r_i \lor \neg r_j \lor \neg lr_{ji})$$

– i is rotated, while j is not.

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - h_i - w_j < 0, \quad (\neg r_i \lor r_j \lor \neg lr_{ij}) \land (\neg r_i \lor r_j \lor \neg lr_{ji})$$

– i and j are both rotated.

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - h_i - h_j < 0, \quad (\neg r_i \lor \neg r_j \lor \neg lr_{ij}) \land (\neg r_i \lor \neg r_j \lor \neg lr_{ji})$$

It is important to point out that the rotation/non-rotation of the circuits have also affected the inequality $w - w_i - w_j < 0$.

- Constraint A2)

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j \geq 0, \quad \forall e \in [0..w - w_i - w_j] \quad (\neg lr_{ij} \lor \neg px_{j(e+w_i)} \lor px_{ie})$$

The constraint $(\neg lr_{ij} \lor \neg px_{j(e+w_i)} \lor px_{ie})$ must be modified, by imposing that this is True only if both rectangles have not been rotated.

$$(r_i \lor r_j \lor \neg lr_{ij} \lor \neg px_{j(e+w_i)} \lor px_{ie})$$

(this is obtained from the implication $(\neg r_i \land \neg r_j \implies ...)$, as seen before).
So:

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - w_j \geq 0, \quad \forall e \in [0..w - w_i - w_j] \quad (r_i \lor r_j \lor \neg lr_{ij} \lor \neg px_{j(e+w_i)} \lor px_{ie})$$

This is only about a single possibility: both the circuits i and j are not rotated. There are three other combinations.

– j is rotated, while i is not.

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - w_i - h_j \geq 0, \quad \forall e \in [0..w - w_i - h_j] \quad (r_i \lor \neg r_j \lor \neg lr_{ij} \lor \neg px_{j(e+w_i)} \lor px_{ie})$$

– i is rotated, while j is not.

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - h_i - w_j \geq 0, \quad \forall e \in [0..w - h_i - w_j] \quad (\neg r_i \lor r_j \lor \neg lr_{ij} \lor \neg px_{j(e+h_i)} \lor px_{ie})$$

– i and j are both rotated.

$$\forall i,j \in [0..n-1] \quad \text{, if} \quad w - h_i - h_j \geq 0, \quad \forall e \in [0..w - h_i - h_j] \quad (\neg r_i \lor \neg r_j \lor \neg lr_{ij} \lor \neg px_{j(e+h_i)} \lor px_{ie})$$

It is important to point out that the rotation/non-rotation of the circuits have also affected the inequality $w - w_i - w_j < 0$, the domain $e \in [0..w - w_i - w_j]$ and the variable $px_{j(e+w_i)}$.

3.5.2 Modification of the constraints about the optimization variables

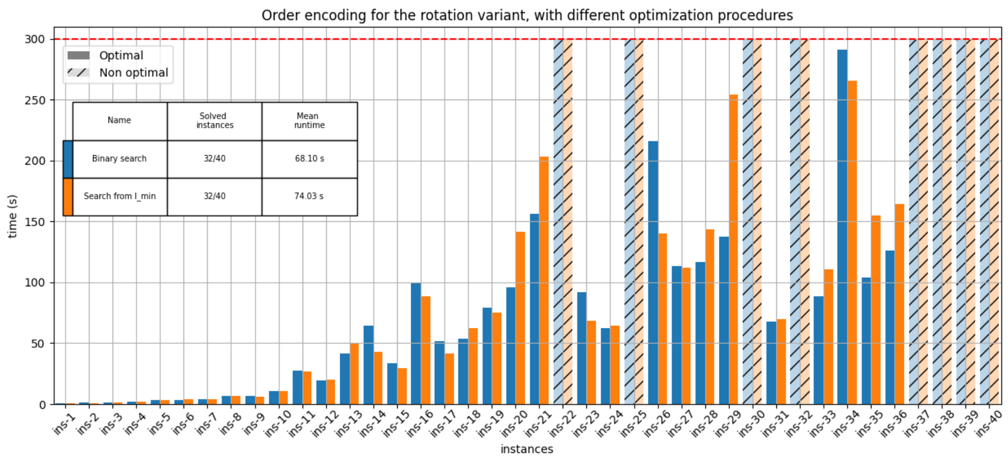
In this section, the constraints about the optimization variables ph_o (see section 3.4.4 and section 3.4.5) are modified. Actually, only the constraint OE O1 is modified: the constraints OE O2 and OE O3 remain the same.

Constraint O1)
$$\forall i \in [0..n-1] \quad \forall o \in [0..l_{\max} - l_{\min}] \quad \neg ph_o \lor py_{i(o+l_{\min}-h_i)}$$
This becomes the following two constraints:

- The circuit i has not been rotated: $\forall i \in [0..n-1] \quad \forall o \in [0..l_{\max} - l_{\min}] \quad (r_i \lor \neg ph_o \lor py_{i(o+l_{\min}-h_i)})$
- The circuit i has been rotated: $\forall i \in [0..n-1] \quad \forall o \in [0..l_{\max} - l_{\min}] \quad (\neg r_i \lor \neg ph_o \lor py_{i(o+l_{\min}-w_i)})$

3.5.3 Results

The same optimization procedures described in section 3.4.5 are performed. Figure 15 shows the results. As it can be seen, the best alternative is order encoding with binary search, which solves 32 instances,



Name	Solved instances	Mean runtime
Binary search	32/40	68.10 s
Search from l_min	32/40	74.03 s

Figure 15: Results of using different optimization procedures with the order encoding for the rotation variant

with an average solving time of 68.10 seconds. So, the rotation variant is more difficult to solve. Since more degrees of freedom are present, the encoding is bigger, containing more constraints.

4 SMT

In this section, the VLSI problem is solved using the Satisfiability Modulo Theory approach, which is an extension of SAT. It consists in writing an SMT encoding for the problem and then giving it to a SMT solver for finding a satisfiable assignment, i.e. model, if any.

The SMT-LIB language has been used for writing the encodings. The strong benefit of SMT-LIB is that it is a solver-independent standard, therefore different solvers can be exploited. In paricular, the following solvers have been taken into account: *z3* [8]; *cvc5* [12]; *yices2* [13]. It is important to point out that, differently from*z3* and *cvc5*, *yices2* can be used only if a specific logic has been fixed in the encoding.

A drawback of SMT-LIB is that it is non-parametric, meaning that parameters can't be defined and that each encoding is instance-specific. For this reason, the Python programming language has been used for automatically generating SMT-LIB encodings, depending on the specific instance and parameters.

Another drawback of SMT-LIB is that it does not have an explicit support for optimization. This means that optimization must be implemented by hand over the simple satisfaction encoding. This means that also the choices regarding the optimization procedure have an important role in finding the overall best alternative.

4.1 Basic encoding

In this section, the basic encoding is described.

4.1.1 Variables

The following SMT variables are used.

- Integer variables x_i, where $i \in [0..n-1]$. x_i is the x coordinate of the bottom-left vertex of the i-th circuit.
- Integer variables y_i, where $i \in [0..n-1]$. y_i is the y coordinate of the bottom-left vertex of the i-th circuit.
- Integer variables l. It represents the length of the plate.

Note: both groups of variables x_i and y_i are implemented as functions in the SMT-LIB code: `(declare-fun x (Int) Int)` and `(declare-fun y (Int) Int)`. More precisely, these are uninterpreted functions.

4.1.2 Constraints

The following constraints are enforced.

- Constraints defining the domains of the variables.

$$\begin{aligned} &\forall i \in [0..n-1] \quad 0 \le x_i \le w - w_i \\ &\forall i \in [0..n-1] \quad 0 \le y_i \le l_{\max} - h_i \\ &l_{\min} \le l \le l_{\max} \end{aligned} \tag{SMT 1}$$

- Constraint defining the non-overlapping of the rectangles

$$\forall i, j \in [0..n-1] \quad , \quad i \neq j \quad , \quad (x_i+w_i \leq x_j) \lor (x_j+w_j \leq x_i) \lor (y_i+h_i \leq y_j) \lor (y_j+h_j \leq y_i) \quad \text{(SMT 2)}$$

- Constraint about the length of the plate

$$\forall i \in [0..n-1] \quad y_i + h_i \leq l \qquad \text{(SMT 3)}$$

4.1.3 Optimization

The applied optimization procedures are very similar to the ones exploited in SAT.

Non-incremental linear search starting from the top (i.e. $l_{\max}$). Cycle in which, at each iteration, the encoding is generated from scratch and the solver is created and run from scratch on that encoding. At each iteration, the current best length of the plate (i.e. l_{curr}) is given to the solver as upper bound for the length of the plate (i.e. $l_{\max}$). (Actually, $l_{curr} - 1$ is given as $l_{\max}$). Then, at each iteration, l_{curr} is decreased by one, until finding UNSAT. At the beginning, $l_{\max}$ is computed as described in 1.4. No incremental solving: at each iteration, the solver is created and run from scratch. It is important to point out that the SMT variable l and the corresponding constraints are not used. This variable is introduced exactly for having an incremental optimization procedure.

Incremental linear search starting from the top (i.e. $l_{\max}$) Incremental solving: the solver is created only one time, at the beginning. A cycle is performed, in which, at each iteration, a new constraint is injected into the solver, namely the constraint imposing that the length of the plate must be strictly smaller than the current best length (i.e. l_{curr}). This is enforced by means of the variable l, in the following way: $l < l_{curr}$. As in the previous optimization procedure, the search starts from $l_{\max}$ computed as described in 1.4. The cycle ends when UNSAT is found.

Incremental binary search. The solver is created only one time, at the beginning. Cycle in which, at each iteration, there are a certain lower bound (i.e. lb) and a certain upper bound (i.e. ub) for the length of the plate. In addition, there is also a length of interest l_{curr}, computed as the middle between lb and ub: $l_{curr} = \lceil \frac{lb+ub}{2} \rceil$. At each iteration, a new constraint is injected into the solver, ensuring that the actual length of the plate is smaller or equal than l_{curr}: the actual length of the plate is in the interval $[lb..l_{curr}]$. More precisely, this constraint is $l \leq l_{curr}$. Then, after injecting this constraint, this current solver instance is solved.

- If SAT, then the found solution is saved as the new best solution. Moreover, ub is updated as follows: $ub \longleftarrow l$.
- If UNSAT, the last injected constraint (i.e. the one ensuring that the actual length of the plate is smaller or equal than l_{curr}) is retracted from the solver, and a new constraint is instead added: constraint ensuring that the actual length of the plate is strictly bigger than l_{curr}. Constraint $l > l_{curr}$. Finally, lb is updated as follows: $lb \longleftarrow l + 1$.

These steps are performed at each iteration. At the beginning, lb is $l_{\min}$ and ub is $l_{\max}$. The cycle ends when lb becomes bigger than ub: in such case, the last found solution is the best solution (if no solution has been found, that problem instance is UNSAT).

Incremental linear search starting from the bottom (i.e. starting from $l_{\min}$). The solver is created only one time, at the beginning. Cycle in which, at each iteration, there is a certain current l_{curr} of interest. All the lengths below l_{curr} have already been tested, with a failure: thus, if this l_{curr} is SAT, then this l_{curr} is the best possible length of the plate. At each iteration, a new constraint is injected into the solver, ensuring that the actual length of the plate is smaller or equal than l_{curr}. Constraint $l \leq l_{curr}$. Then, after injecting this constraint, this current solver instance is solved.

- If SAT, then the best possible solution has been found: the solving procedure exits.
- If UNSAT, then the last injected constraint (i.e. the one ensuring that the actual length of the plate is smaller or equal than l_{curr}) is retracted, and a new constraint is added: constraint ensuring that the actual length of the plate is strictly bigger than l_{curr}. Constraint $l > l_{curr}$. Finally, l is updated as follows: $l \leftarrow l + 1$.

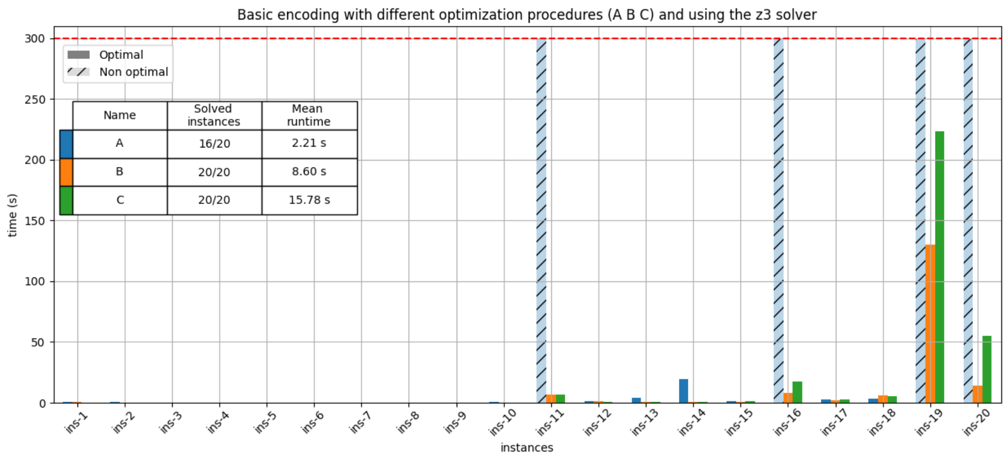
These steps are performed at each iteration. At the beginning, l_{curr} is $l_{\min}$. If, at some point, l_{curr} becomes bigger than $l_{\max}$, this means that the problem instance is UNSAT.

It is important to notice that, in both last two optimization procedures, since some constraints must be added and retracted dynamically, the assertion stack has been used: in this way, assertion levels can be pushed into and popped out the stack.

4.1.4 Results

The basic encoding has been tested, with different optimization procedures: "A" stands for incremental linear search from the top; "B" stands for incremental binary search; "C" stands for incremental linear search from the bottom.

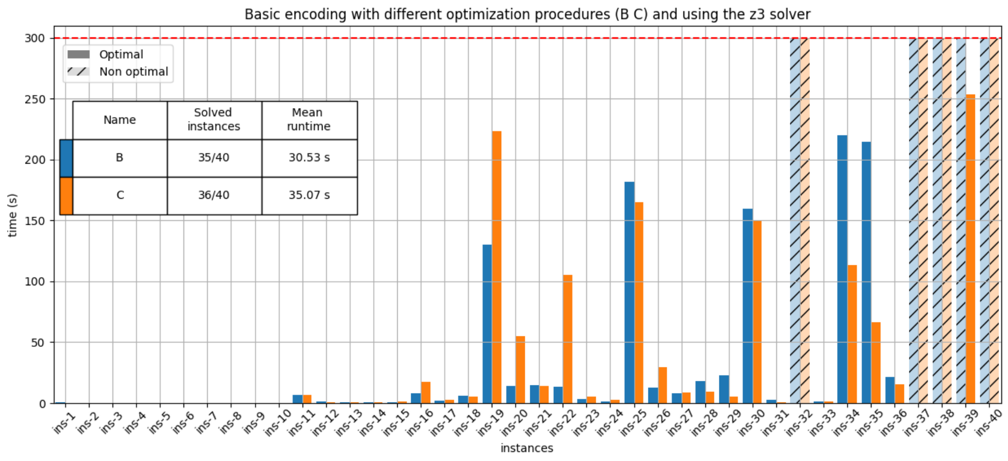
Figure 16 shows the results on the first 20 instances, using the *z3* solver. It is clear that the optimization



	Name	Solved instances	Mean runtime
🟦	A	16/20	2.21 s
🟧	B	20/20	8.60 s
🟩	C	20/20	15.78 s

Figure 16: Results of using different optimization procedures A B C with the basic encoding, *z3* solver

procedure A is less efficient. Figure 17 shows the results of B and C on all the instances, again with the *z3*

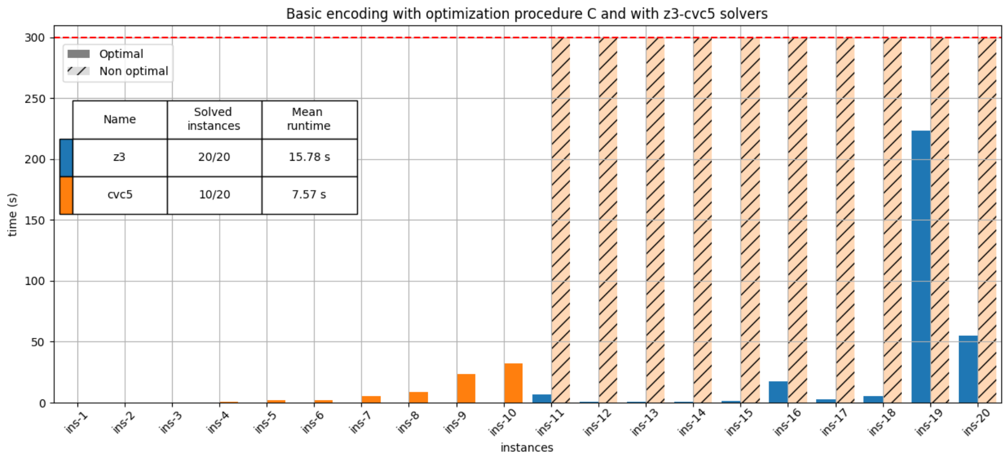


	Name	Solved instances	Mean runtime
🟦	B	35/40	30.53 s
🟧	C	36/40	35.07 s

Figure 17: Results of using different optimization procedures B C with the basic encoding, *z3* solver

solver. The results are quite good: with incremental linear search starting from the bottom, 36 instances are solved, with an average solving time of 35.07 seconds.

Lastly, the *cvc5* solver is tried. (The *yices* solver can't be used, because no specific logic has been fixed in the encoding). Figure 18 shows the results of using the *z3* and *cvc5* solvers with the optimization procedure C. It is clear that *z3* performs much better.



	Name	Solved instances	Mean runtime
🟦	z3	20/20	15.78 s
🟧	cvc5	10/20	7.57 s

Figure 18: Results of using solvers *z3* and *cvc5* with the basic encoding with the optimization procedure C

4.2 Fixing a specific logic

For improving the current basic encoding, a specific logic is explicitily fixed. Fixing a specific logic in the encoding tells to the solver the kind of variables and constraints which are admissible. Therefore, the solver can take advantage of this additional information for making the computation faster. The more specific is a logic, and the more information can be exploited by the solver.

4.2.1 Logics

The following logics are considered.

Logics with uninterpreted functions.

- A: **QF_UFLIA** [14], ”unquantified linear integer arithmetic with uninterpreted sort and function symbols”. Basically, we have only Quantifier-Free formulas, with Linear Integer Arithmetic, and with the possibility of Uninterpreted Functions. The current encoding is already compliant with this logic: no modification is needed.
- B: **QF_UFIDL** [14], ”difference Logic over the integers (in essence) but with uninterpreted sort and function symbols”. Basically, we have only Quantifier-Free formulas, with Difference Logic, and with the possibility of Uninterpreted Functions. This is a more specific logic with respect to *QF_UFLIA*. The difference logic requires each atom to be in one of the following forms:
  - q
  - $(\text{op}(-xy)n)$,
  - $(\text{op}(-xy)(-n))$, or
  - $(\text{op}xy)$

  where: q is a variable or free constant symbol of sort `Bool`; op is $<, \leq, >, \geq, =$, or distinct; x, y are free constant symbols of sort `Int`; n is a numeral.
  Even if the current encoding is not perfectly compliant with this logic, the solvers does not complain when executing such encoding with the *QF_UFLIA* logic. This means that the solvers perform some implicit translation and modification of the encoding, for making it compliant with the specified logic. (Indeed, if, in general, the solver is run with an encoding not compliant with the specified logic, an error is raised.) So, the current encoding is left as it is.
- C: still **QF_UFIDL** logic, but slightly modifying the constraints for making them more compliant with the difference logic. For instance, the constraint SMT 2 is modified as follows:

  $$\forall i, j \in [0..n-1] \quad , \quad i \neq j \quad , \quad (x_i - x_j \leq -w_i) \vee (x_j - x_i \leq -w_j) \vee (y_i - y_j \leq -h_i) \vee (y_j - y_i \leq -h_j)$$

  Or, for instance, the constraint SMT 3 is modified as follows:

  $$\forall i \in [0..n-1] \quad y_i - j \leq -h_i$$

Logics without uninterpreted functions. Now the same logics just described are taken again into account, but this time without the possibility of uninterpreted functions. These new logics are more specific then the corresponding previous ones.

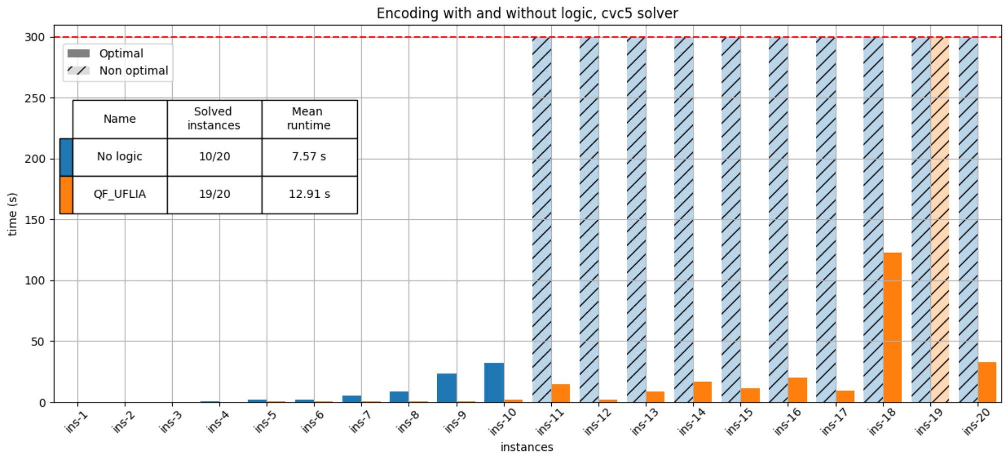
An important point is that the current encoding does contain uninterpreted functions, namely the groups of variables x_i and y_i are currently implemented as uninterpreted functions: `(declare-fun x (Int) Int)` and `(declare-fun y (Int) Int)`. Basically, the current encoding is not compliant with a logic which does not allow uninterpreted functions (if a solver is run, an error is raised). Therefore, the encoding is changed: the group of variables x_i is now implemented as n single variables, namely `(declare-const coordX_0 Int)`, ..., `(declare-const coordX_n-1 Int)`. The same for the y_i.

- D: **QF_LIA** [14], "unquantified linear integer arithmetic. In essence, Boolean combinations of inequations between linear polynomials over integer variables". Basically, we have only Quantifier-Free formulas, with Linear Integer Arithmetic. The current encoding is already compliant with this logic: no modification is needed.
- E: **QF_IDL** [14], "difference Logic over the integers. In essence, Boolean combinations of inequations of the form x - y ¡ b where x and y are integer variables and b is an integer constant". Basically, we have only Quantifier-Free formulas, with Difference Logic. This is a more specific logic with respect to *QF_LIA*. As seen before, even if the current encoding is not compliant with difference logic, the solvers don't complaint: so, the encoding is left as it is.
- F: still **QF_IDL** logic, but slightly modifying the constraints for making them more compliant to the difference logic, as seen before.

4.2.2 Results

Let's now inspect the results. The incremental linear search from the bottom is used as optimization procedure (see 4.1.3). In any case, using the incremental binary search produces the same kind of results and considerations.

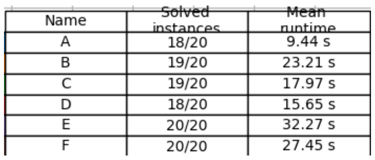
As said before, fixing a specific logic should, in theory, improve the performances. For instance, this is the case for the *cvc5* solver. Figure 19 shows the dramatic improvement of the performances if a specific logic is fixed (*QF_UFLIA* in this case).



	Name	Solved instances	Mean runtime
🟦	No logic	10/20	7.57 s
🟧	QF_UFLIA	19/20	12.91 s

Figure 19: Results of using the *cvc5* solver with and without a fixed logic

Figure 20 shows the results achieved by *cvc5* with different fixed logics. As it can be seen, the more specific

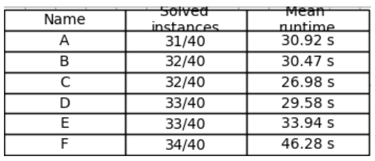


Name	Solved instances	Mean runtime
A	18/20	9.44 s
B	19/20	23.21 s
C	19/20	17.97 s
D	18/20	15.65 s
E	20/20	32.27 s
F	20/20	27.45 s

Figure 20: Results of using the *cvc5* solver with different logics

is the fixed logic, the better are the performances (more or less).

In addition, fixing a specific logic allows us to use the *yices* solver. Figure 21 shows the results achieved by *cvc5* with different fixed logics. Also here, the more specific is the fixed logic, the better are the per-

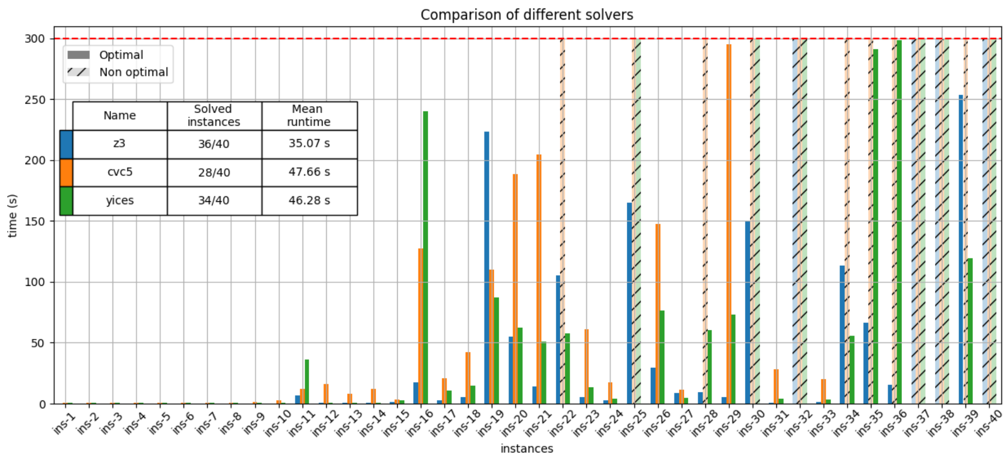


Name	Solved instances	Mean runtime
A	31/40	30.92 s
B	32/40	30.47 s
C	32/40	26.98 s
D	33/40	29.58 s
E	33/40	33.94 s
F	34/40	46.28 s

Figure 21: Results of using the *yices* solver with different logics

formances (more or less).

Nonetheless, both *cvc5* and *yices*, even if they have a fixed logic, reach worse results with respect to *z3* without a fixed logic. Figure 22 shows the comparison between *z3* without a fixed logic, *cvc5* with its more performant fixed logic (i.e. logic F) and *yices* with its more performant fixed logic (i.e. logic F).



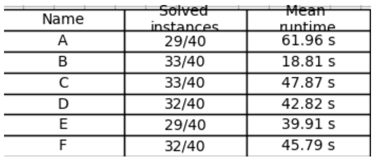
	Name	Solved instances	Mean runtime
🟦	z3	36/40	35.07 s
🟧	cvc5	28/40	47.66 s
🟩	yices	34/40	46.28 s

Figure 22: Comparison between the three solvers: *z3* without any fixed logic, while *cvc5* and *yices* with the logic F

.

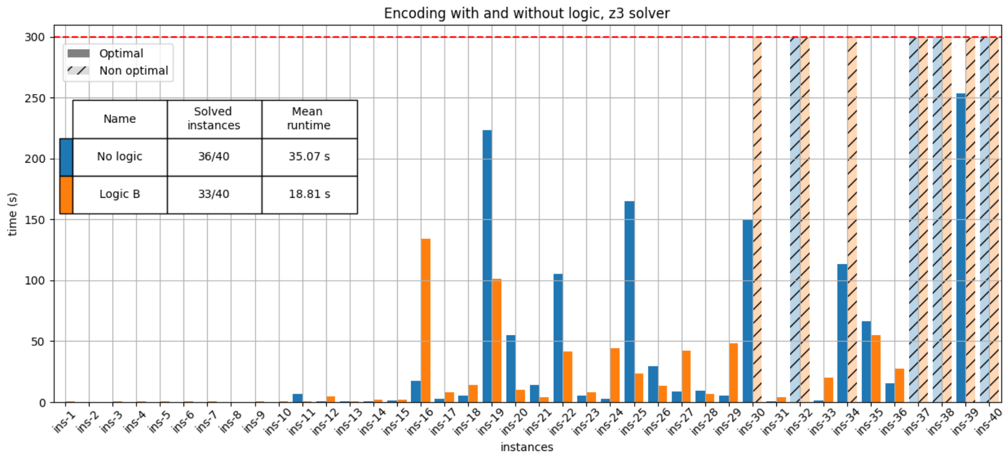
Differently from *cvc5*, fixing a logic with the *z3* solver produces results which are less intuitive. Indeed, it is not necessarily True that the more specific is the fixed logic and the better are the achieved results:

figure 23 shows the results achieved by different fixed logics using the *z3* solver.
But, more importantly, fixing a logic does not improve the performances. Figure 24 shows the comparison between using the *z3* solver without any specified logic and using the *z3* solver with its more performant fixed logic (i.e. logic B).



Name	Solved instances	Mean runtime
A	29/40	61.96 s
B	33/40	18.81 s
C	33/40	47.87 s
D	32/40	42.82 s
E	29/40	39.91 s
F	32/40	45.79 s

Figure 23: Results of using the *z3* solver with different logics



	Name	Solved instances	Mean runtime
🟦	No logic	36/40	35.07 s
🟧	Logic B	33/40	18.81 s

Figure 24: Comparison between *z3* with no logic and *z3* with the B logic

So, the best alternative remains the same: *z3* solver with incremental linear search from the bottom, with no fixed logic.

4.3 Rotation variant

Since now, the non-rotation variant of the VLSI problem as been considered. Instead, in this section, the rotation variant is addressed. Rotation variant: each circuit can be rotated by 90°, swapping the width (i.e. w_i) with the height (i.e. h_i).
For solving this variant, the basic encoding described section 4.1 has been considered and modified.

4.3.1 Modification of the basic encoding

First of all, the new group of variables r is added.
Variables r_i, where i represents a circuit. $i \in [0..n-1]$. r_i is True iff the i-th circuit has been rotated, meaning that w_i and h_i have been swapped.

Then, also two other groups of variables are added.

- $w_i^{'}$, where $i \in [0..n-1]$. $w_i^{'}$ is the actual horizontal dimension (i.e. width) of the i-th circuit.
- $h_i^{'}$, where $i \in [0..n-1]$. $h_i^{'}$ is the actual vertical dimension (i.e. heigth) of the i-th circuit.

For each circuit i, the following constraints are ensured, putting into relation the variable r_i with the variables $w_i^{'}$ and $h_i^{'}$.

$$(\neg r_i \implies w_i^{'} = w_i \wedge h_i^{'} = h_i) \wedge (r_i \implies w_i^{'} = h_i \wedge h_i^{'} = w_i)$$

Which is equivalent to:

$$(r_i \vee (w_i^{'} = w_i \wedge h_i^{'} = h_i)) \wedge (\neg r_i \vee (w_i^{'} = h_i \wedge h_i^{'} = w_i))$$

Then, the remaining of the encoding is the same of the basic encoding. The only difference is that the variables $w_i^{'}, h_i^{'}$ are used instead of the parameters w_i, h_i.

4.3.2 Alternative modification of the basic encoding

The basic encoding can be modified in an alternative way for taking into account the rotation. Different encoding with respect to the one specified in section 4.3.1.

Actually, it is very similar to the encoding described in section 4.3.1. The only difference is that the domain constraints are enforced without the usage of the variables $w_i^{'}, h_i^{'}$. They are defined using directly the variables r_i and the values w_i, h_i, in a lower-level way.

Domain of each x_i.

- $x_i \geq 0$
- If the i-circuit has not been rotated, then

  $$x_i \leq w - w_i$$

  Which is equivalent to:

  $$r_i \quad \vee \quad x_i \leq w - w_i$$

- If the i-circuit has been rotated, then

  $$x_i \leq w - h_i$$

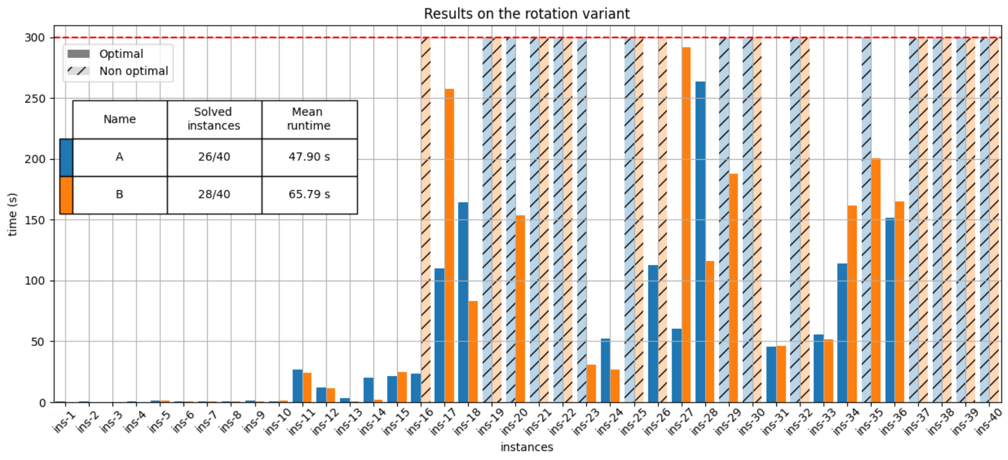
  Which is equivalent to:

  $$\neg r_i \quad \vee \quad x_i \leq w - h_i$$

The same reasoning is also applied for the variables y_i.

4.3.3 Results

Denoting wiht A the encoding described in section 4.3.1 and with B the encoding described in section 4.3.2, figure 25 shows the results achieved by these two encodings. The *z3* solver has been used, incremental linear search from the bottom has been used as optimization procedure, and no logic has been fixed (the starting point is the basic encoding).



	Name	Solved instances	Mean runtime
🟦	A	26/40	47.90 s
🟧	B	28/40	65.79 s

Figure 25: Results obtained on the rotation variant, with encodings A and B, z3 solver, incremental linear search from the bottom

As it can be seen, the best alternative is encoding B, which solves 28 instances, with an average solving time of 65.79 seconds. So, the rotation variant is more difficult to solve. Since more degrees of freedom are present, the encoding is bigger, containing more constraints.

5 LP

Linear Programming (LP) is a special case of mathematical programming whose aim is to optimize a linear objective function subject to linear equality or inequality constraints.

In order to solve the VLSI problem through a Linear Programming approach we implemented two different models with the *AMPL Modeling Language* [15] and *Python* scripts. As it will be explained in the next sections, each of the model has specific advantages over the other with respect to whether the user wants to solve a greater amount of instances or obtain feasible solutions for all instances in spite of optimality.

Furthermore, the models are solver independent letting the user specify one among three solvers: *Gurobi* [16], *CPLEX* [17] and *COIN-OR's Cbc* [18].

Regarding the solvers *Gurobi* and *CPLEX*, the possibility of demanding them to attempt to break symmetries during the pre-solve phase or to solve the dual version of the stated problem have been implemented.

In order to break symmetries during the pre-processing phase the following commands are passed to the solvers:

- Command `symmetry=5` when solver *CPLEX* is used. It demands to break symmetries with the maximum effort during the pre-processing phase.
- Command `symmetry=2` when solver *Gurobi* is used. With this command the solver will try to "aggressively" detect and break symmetries.

On the other hand, to ask the solver to consider the dual problem instead of the primal, these commands are given:

- Command `dual` when solver *CPLEX* is used.
- Command `predual=1` when solver *Gurobi* is used.

5.1 Constraint Based Approach

The first developed model takes inspiration from the technique presented in the paper *A constraint programming approach for the two-dimensional rectangular packing problem with orthogonal orientations* [19]. In the paper, the VLSI variant of the problem that allows the rotation of the circuit is presented, although the formulation without rotations could be simply derived from it.

5.1.1 Variables

Variables and parameters refer to the ones declared in the sections 1.3 and 1.4.
In addition, in the paper a variable z is introduced. It is a $n \times n \times 4$ matrix allowing just binary values 0 and 1. Its function is to take trace of the overlapping of a pair of circuit (i, j). In particular:

- $z^1_{i,j} = 1$ if and only if the circuit j does not start before the end of circuit i on the horizontal dimension
$$z^1_{i,j} = 1 \iff x_i + w_i \leq x_j$$
- $z^2_{i,j} = 1$ if and only if the circuit j does not start before the end of circuit i on the vertical dimension
$$z^2_{i,j} = 1 \iff y_i + h_i \leq y_j$$
- $z^3_{i,j} = 1$ if and only if the circuit i does not start before the end of circuit j on the horizontal dimension
$$z^3_{i,j} = 1 \iff x_j + w_j \leq x_i$$
- $z^4_{i,j} = 1$ if and only if the circuit i does not start before the end of circuit j on the vertical dimension
$$z^4_{i,j} = 1 \iff y_j + h_j \leq y_i$$

This variable was edited in our model in order to decrease its spatial complexity and ensure better performances. In particular it was observed that z as it was previously introduced could be considered as an array of 4 *lower triangular matrices* where the diagonals were unused.

$$\forall i, j \in 1..n,\ k \in \{1,2,3,4\} \wedge i \geq j,\ z^k_{i,j} = 0$$

Hence, z was rewritten as a $n \times n \times 2$ matrix such that:

- $z^1_{i,j} = 1$ if and only if the circuit j does not start before the end of circuit i on the horizontal dimension
$$\forall\ i, j \in 1..n,\ z^1_{i,j} = 1 \iff x_i + w_i \leq x_j$$
- $z^2_{i,j} = 1$ if and only if the circuit j does not start before the end of circuit i on the vertical dimension
$$\forall\ i, j \in 1..n,\ z^2_{i,j} = 1 \iff y_i + h_i \leq y_j$$

5.1.2 Constraints

A series of constraints have been imposed in order to minimize the length of the plate l. They have all been declared in such a way to respect a linear programming formulation.

For each circuit i, its coordinates (x_i, y_i) must be assigned in such a way to respect the bounds of the plate. Therefore these constraints have been introduced:

- Constraint that forces each circuit i to be inside the horizontal bound of the plate:
$$\forall i \in 1..n,\ x_i + w_i \leq w \tag{LP 1}$$
- Constraint that forces each circuit i to be inside the vertical bound of the plate:
$$\forall i \in 1..n,\ y_i + h_i \leq l \tag{LP 2}$$

Moreover constraints to impose no-overlapping between circuits have been inserted. They specify in a disjunctive manner that either a circuit i must not overlap over another circuit j in one of the two vertical and horizontal directions or vice-versa.

$$\forall\ i, j \in 1..n \wedge i < j,\ x_i + w_i \leq x_j \vee y_i + h_i \leq y_j \vee x_j + w_j \leq x_i \vee y_j + h_j \leq y_i$$

In order to represent the disjunction in linear form a *Big-M relaxation* technique has been applied and the variable z has been exploited to check which non-overlapping constrains are active among pairs of circuits.

The Big-M relaxation is used to relax a constraint by introducing a “large enough” constant M which guarantees that it is always satisfied under certain conditions.

The following constraints have been introduced:

- Given a pair of circuits (i, j), avoid overlapping of i over j on the horizontal dimension if $z^1_{i,j} = 1$
$$\forall i, j \in 1...n \wedge j < i, x_i + w_i \leq x_j + M_1 \cdot (1 - z^1_{i,j}) \tag{LP 3}$$

If $z^1_{i,j} = 0$ the constraint is not required and the inequality is always satisfied. That occurs because the left-hand side will be always less or equal than the "large enough number" M_1 in the right-hand side. This consideration can symmetrically be applied for the next three constraints.

- Given a pair of circuits (i, j), avoid overlapping of i over j on the vertical dimension if $z^2_{i,j} = 1$

$$\forall i, j \in 1...n \land j < i, y_i + h_i \le h_j + M_2 \cdot (1 - z^2_{i,j}) \tag{LP 4}$$

- Given a pair of circuits (i, j), avoid overlapping of j over i on the horizontal dimension if $z^1_{j,i} = 1$

$$\forall i, j \in 1...n \land j < i, x_j + w_j \le x_i + M_1 \cdot (1 - z^1_{j,i}) \tag{LP 5}$$

- Given a pair of circuits (i, j), avoid overlapping of j over i on the vertical dimension if $z^2_{j,i} = 1$

$$\forall i, j \in 1...n \land j < i, y_j + h_j \le y_i + M_2 \cdot (1 - z^2_{j,i}) \tag{LP 6}$$

Where M_1 and M_2 are two Big-M constants equal to w and $l_{\max}$ respectively.

To guarantee the correctness of the model three final constraints have been added:

- Constraint that guarantees that at most one horizontal non-overlapping constraint among pair of circuits (i, j) is active
$$\forall i, j \in 1...n \land j < i, z^1_{i,j} + z^1_{j,i} \le 1 \tag{LP 7}$$
- Constraint that guarantees that at most one vertical non-overlapping constraint among pair of circuits (i, j) is active
$$\forall i, j \in 1...n \land j < i, z^2_{i,j} + z^2_{j,i} \le 1 \tag{LP 8}$$
- At least one non-overlapping constraint must be not relaxed
$$\forall i, j \in 1...n \land j < i, z^1_{i,j} + z^1_{j,i} + z^2_{i,j} + z^2_{j,i} \ge 1 \tag{LP 9}$$

5.1.3 Results

Figure 26 illustrates the results obtained by the model, which will be referred as **Basic model**, in solving the set of instances. The use of symmetry breaking and the construction of the dual model of the problem have not been considered in this first case.

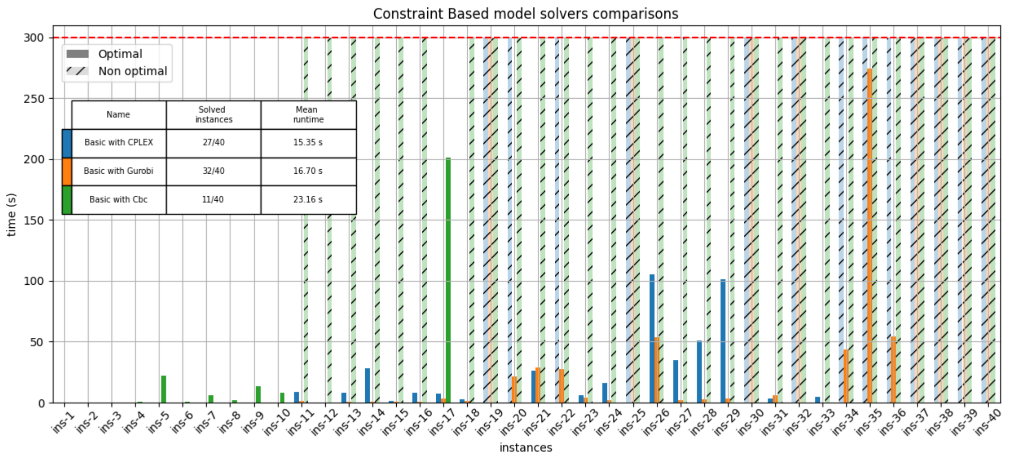
It can be observed how the solvers *CPLEX* and *Gurobi* outperform *Cbc*. Furthermore *Gurobi* solves 5 more instances than *CPLEX* (32/40) in a comparable average time per instance. For this reason, Gurobi has been selected as the solver to be used for the successive models.

Figure 27 shows the performances of the solver *Gurobi* while considering symmetry breaking and the resolution of the dual version of the problem. In particular:

- **Symmetry Model** is the variation of Basic Model that solves the problem with symmetry breaking.
- **Dual Model** is the variation of Basic Model that solves the dual problem.

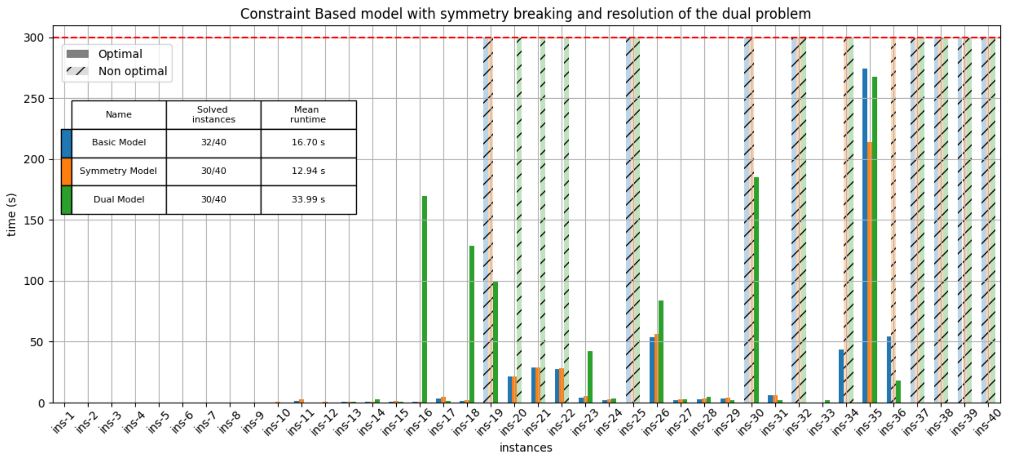
The Symmetry and Dual models perform worse than the Basic Model while solver Gurobi is used, by both increasing the average computation time and solving a lower amount of instances.

Therefore, the model giving the best results in solving the VLSI problem without considering rotations and using the *Constraint Based* approach is the **Basic Model** that uses the solver **Gurobi** as it manages to solve 32 out of 40 instances in an average time of 16.70 seconds.



	Name	Solved instances	Mean runtime
🟦	Basic with CPLEX	27/40	15.35 s
🟧	Basic with Gurobi	32/40	16.70 s
🟩	Basic with Cbc	11/40	23.16 s

Figure 26: Results for the Basic model obtained with the *Constraint Based* approach with solvers *CPLEX* and *Gurobi* and *Cbc*



	Name	Solved instances	Mean runtime
🟦	Basic Model	32/40	16.70 s
🟧	Symmetry Model	30/40	12.94 s
🟩	Dual Model	30/40	33.99 s

Figure 27: Results for solver *Gurobi* obtained with the *Constraint Based* approach considering the Basic Model, the model performing symmetry breaking (Symmetry Model) and the model solving the dual version of the problem (Dual Model)

5.1.4 Rotation variant

In order to solve the rotation variant of the problem, the previous model has been slightly edited. It can be observed that this new version corresponds to the one presented in the paper from which it takes inspiration, which allows for rotations of 90° of the circuits.

The following modifications have been applied.

Variables These variables have been introduced in addition to the previously stated ones.

- The vector o of dimension n that is composed of binary variables and for each circuit i, $o_i = 0$ if i is not rotated, otherwise $o_i = 1$.

$$o_i = \begin{cases} 1, & \text{if } i \text{ is rotated} \\ 0, & \text{otherwise} \end{cases}$$

- w'_i, where $i \in 1..n$. It is the actual horizontal dimension (i.e. width) of the i-th circuit.
- h'_i, where $i \in 1..n$. It is the actual vertical dimension (i.e. heigth) of the i-th circuit.

Constraints Some of the previous constraints have been edited in order to consider the actual dimensions of the circuits with respect to their orientation decided by o.

- Constraints (LP 1) and (LP 2) have been respectively modified as follows.

$$\forall i \in 1..n, x_i + w'_i \leq w \tag{LP 10}$$

$$\forall i \in 1..n, y_i + h'_i \leq l \tag{LP 11}$$

- Constraints (LP 3), (LP 4), (LP 5) and (LP 6) have been respectively modified as follows.

$$\forall i, j \in 1...n \land j < i, x_i + w'_i \leq x_j + M_1 \cdot (1 - z^{(1)}_{(i,j)}) \tag{LP 12}$$

$$\forall i, j \in 1...n \land j < i, y_i + h'_i \leq h_j + M_2 \cdot (1 - z^{(2)}_{(i,j)}) \tag{LP 13}$$

$$\forall i, j \in 1...n \land j < i, x_j + w'_j \leq x_i + M_1 \cdot (1 - z^{(1)}_{(j,i)}) \tag{LP 14}$$

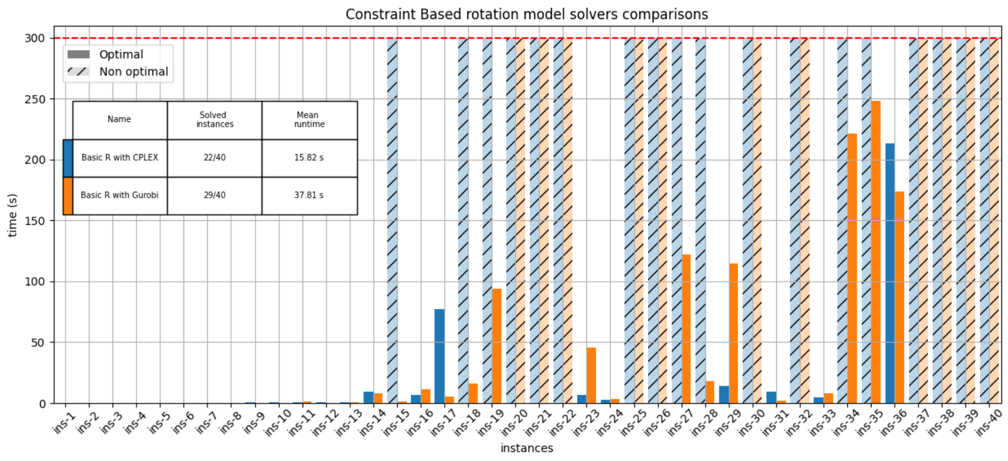
$$\forall i, j \in 1...n \land j < i, y_j + h'_j \leq y_i + M_2 \cdot (1 - z^{(2)}_{(j,i)}) \tag{LP 15}$$

New constraints have been added in order to decide the actual dimensions of each circuit i (w'_i, h'_i). In particular, if the circuit i is not rotated $w'_i = w_i$ and $h'_i = h_i$, whereas if it is rotated $w'_i = h_i$ and $h'_i = w_i$.

$$(1 - o_i) \cdot w_i + o_i \cdot h_i = w'_i \tag{LP 16}$$

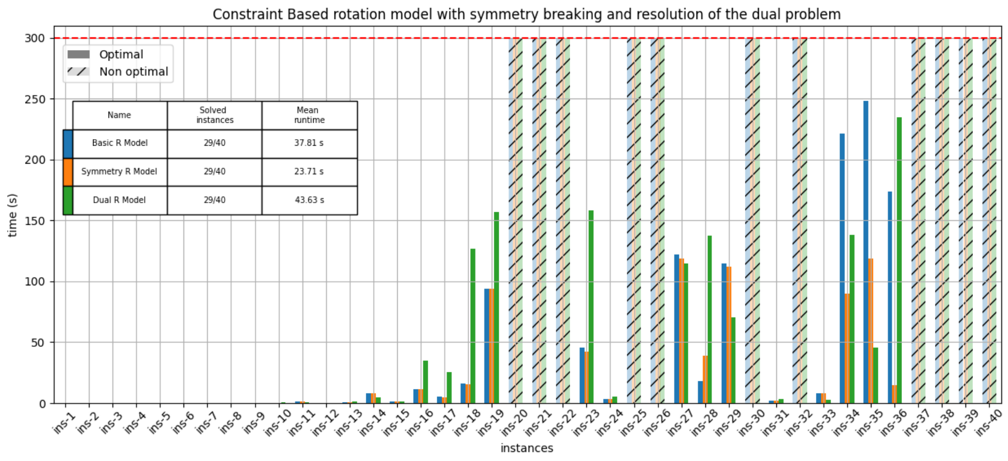
$$(1 - o_i) \cdot h_i + o_i \cdot w_i = h'_i \tag{LP 17}$$

Furthermore, constraints (LP 7), (LP 8) and (LP 9) have been left unedited in the model.



	Name	Solved instances	Mean runtime
🟦	Basic R with CPLEX	22/40	15.82 s
🟧	Basic R with Gurobi	29/40	37.81 s

Figure 28: Results for the Basic model considering the rotation variant of the problem obtained with the *Constraint Based* approach with solvers *CPLEX* and *Gurobi*



	Name	Solved instances	Mean runtime
🟦	Basic R Model	29/40	37.81 s
🟧	Symmetry R Model	29/40	23.71 s
🟩	Dual R Model	29/40	43.63 s

Figure 29: Results for solver *Gurobi* for the rotation variant of the problem obtained with the *Constraint Based* approach considering the Basic Model (Basic R Model), the model performing symmetry breaking (Symmetry R Model) and the model solving the dual version of the problem (Dual R Model)

Results Figure 28 shows how the model considering the rotation of the circuits, which will be called **Basic R Model**, behaves in solving the instances by means of solvers *CPLEX* and *Gurobi*. Once again symmetry breaking and the dual problem are not considered in the first phase. It is evident how, once again, *Gurobi* outperforms *CPLEX* by solving 7 more instances than the latter for a total of 29 out of 40.

Figure 29 illustrates comparisons of the model using symmetry breaking and the dual version of the problem with solver *Gurobi*. The notation is the following:

- **Symmetry R** is the variation of the Basic R Model that solves the rotation version of the problem with symmetry breaking.

- **Dual R** is the variation of the Basic R Model that solves the rotation version of the dual problem.

The results are similar for each variant as they all manage to solve 29 instances out of 40. Although, given the fact that Model Symmetry R spends less time in average, this version is preferred.

The best performances are obtained with model Symmetry R which applies symmetry breaking and uses solver *Gurobi* as it manages to solve optimally 29 instances out of 40 in 23.71 seconds on average.

5.2 Position and Covering Approach

The second developed model is built following the instructions presented in the article *Exact solutions for the 2d-strip packing problem using the positions-and-covering methodology* [20]. This architecture does not allow for the rotation of the circuits to occur, hence, a second model that takes this possibility in consideration has been presented by slightly editing it.

The model, after checking for the satisfiability of the instance, follows the *Position and Covering* technique, which is divided in two steps:

1. Position step: given the minimum possible value of the length of the plate l, initially computed as $l_{\min}$ in section 1.4, it generates a set of valid positions for each circuit i.
2. Covering step: an *Integer Linear Programming* formulation that satisfies the assignment of each circuit i inside one of its valid positions, making sure no circuit overlaps on another one and they all fit inside the bounds of the plate $l \times w$.

If no solution is found after the second step, the process is repeated for a value of l increased by one unit with respect to the previous one.

5.2.1 Position phase

Given the current length of the plate l, this is initially divided into a cartesian grid $l \times w$, where each cell is a 1×1 unit square. The enumeration of the grid starts from the top-left corner and proceeds row by row, stopping at the bottom-right corner as seen in figure 30.

Next, for each circuit i its set of valid positions V_i is created such that:

- Each valid position has a unique label that distinguishes it from the other ones among its set and all other sets:
$$\forall j \in V_i,\ \forall k \neq i,\ j \notin V_k$$
In particular the positions are labeled as incremental integer numbers.
- Each valid position within a set V_i maps to a specific and unique cell of the plate.
- A valid position is created if i's top left corner coincides with the cell it maps, and its width and height dimensions do not exceed the size of the plate.

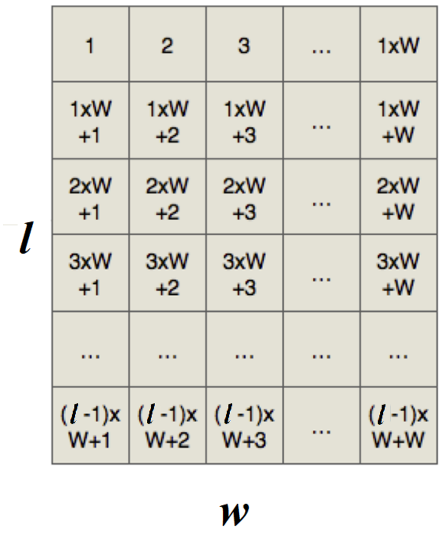


Figure 30: Division of a plate $l \times w$ into a grid with enumerated cells.

- The valid positions of the set V_i are exhaustive such that all the possible valid placing of the circuit i are considered.

Figure 31 shows two sets of valid positions, one for a circuit i of dimension 2×3 (the purple rectangle) and one for a circuit j of dimension 5×3 (the yellow rectangle). The labels of the positions are the numbers above each cartesian grid, therefore.

$$V_i = \{1, \dots, 10\}$$

$$V_j = \{11, \dots, 14\}$$

Finally, a correspondence matrix C is built. Its row correspond to the ordered labels of the sets of valid positions $\bigcup\limits_{i \in 1..n} V_i$ and its columns correspond to the cells of the plate ordinally numbered.

Matrix C is composed of 1s and 0s, where $C_{j,p} = 1$ if the circuit in the valid position j covers the cell p in the grid representing the plate, $C_{j,p} = 0$ otherwise.

5.2.2 Covering phase

For the covering phase an LP model is built in order to satisfy the VLSI problem considering the current value of l.

If the problem is unsatisfiable the value of l is increased by 1 unit and the position phase in section 5.2.1 is repeated considering this new value. Otherwise, the coordinates of the circuits according to the assigned positions are obtained and the optimal solution of the problem is achieved for the current value of l.

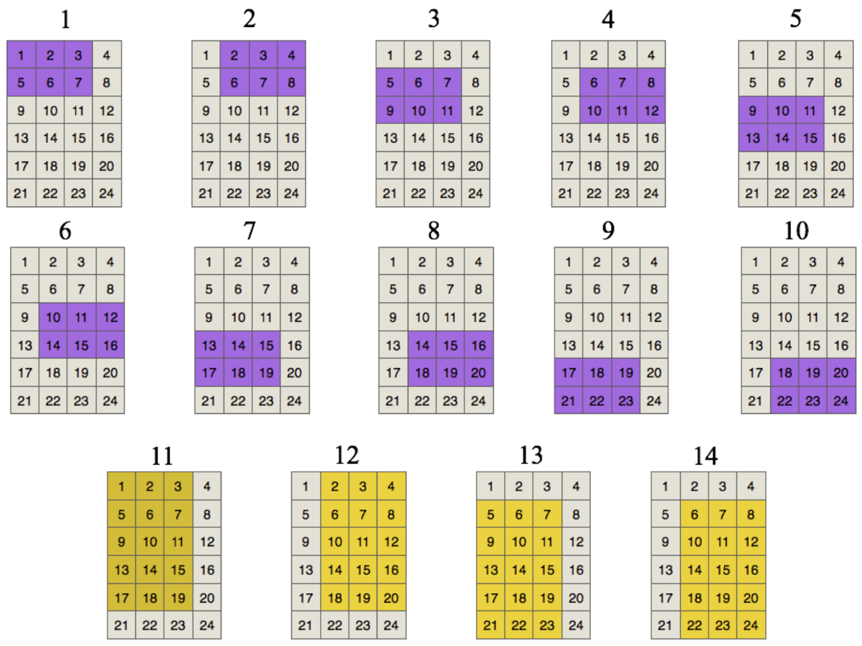


Figure 31: Sets of valid positions for circuits of dimension 2×3 and 5×3 in a plate $l \times w$ with $l = 6$ and $w = 4$.

Variables and parameters The following variables and parameters are defined for the LP model:

- n: parameter describing the number of circuits
- w: parameter describing the width of the plate
- l: parameter describing the current length of the plate for which the problem must be satisfied.
- V: parameter describing an array of dimension n containing the sets of valid positions for each circuit.
- C: parameter describing the correspondence matrix between the valid positions and the cells of the plate as computed in section 5.2.1.
- p: parameter describing the total number of available valid positions ($p = \sum\limits_{i \in 1..n} |V_i|$)
- c: parameter describing the total number of cells in the plate ($c = w \cdot l$)
- x: variable describing a binary matrix $n \times p$ where $x_{i,j} = 1$ if circuit i is placed in valid position j, 0 otherwise.

Constraints

- Constraint guaranteeing that at most one circuit is assigned to each cell.

$$\forall i \in 1..c, \sum_{j=1}^{n} \sum_{k \in V_j} C_{k,i} \cdot x_{j,k} \leq 1 \tag{LP 18}$$

- Constraint guaranteeing that each circuit is placed in the plate exactly one time.

$$\forall i \in 1..n \sum_{j \in V_i} x_{i,j} = 1 \tag{LP 19}$$

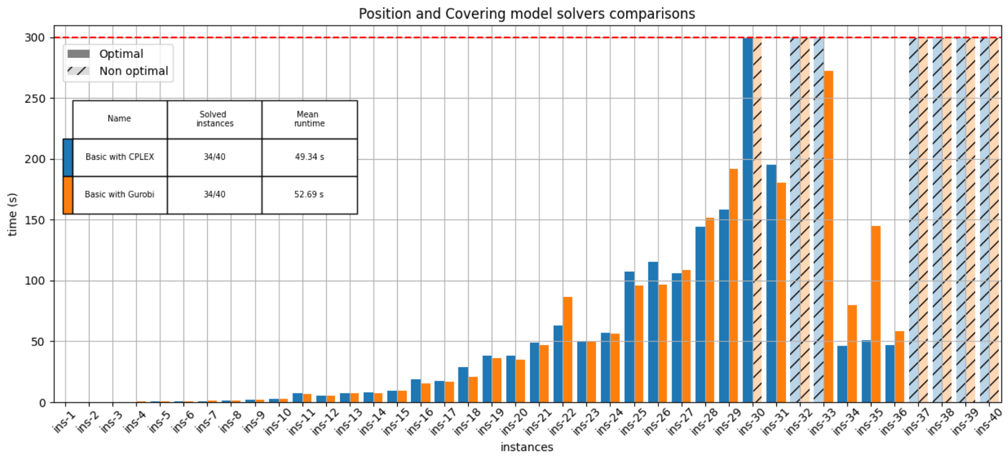
- Constraint guaranteeing that no circuit exceeds the bounds of the grid.

$$\sum_{i=1}^{n} \sum_{j \in V_i} \sum_{k=1}^{c} C_{j,k} \cdot x_{i,j} \leq w \cdot l \tag{LP 20}$$

5.2.3 Results

Figure 32 illustrates the results obtained by the model, which will be referred as **Basic model**, in solving the set of instances. The use of symmetry breaking and the construction of the dual model of the problem have not been considered in this first case.

Both the solvers *CPLEX* and *Gurobi* obtain similar results, solving the same amount of instances (34/40). However *CPLEX* takes a slightly inferior average time to solve the instances, therefore it was selected for further tests.



	Name	Solved instances	Mean runtime
🟦	Basic with CPLEX	34/40	49.34 s
🟧	Basic with Gurobi	34/40	52.69 s

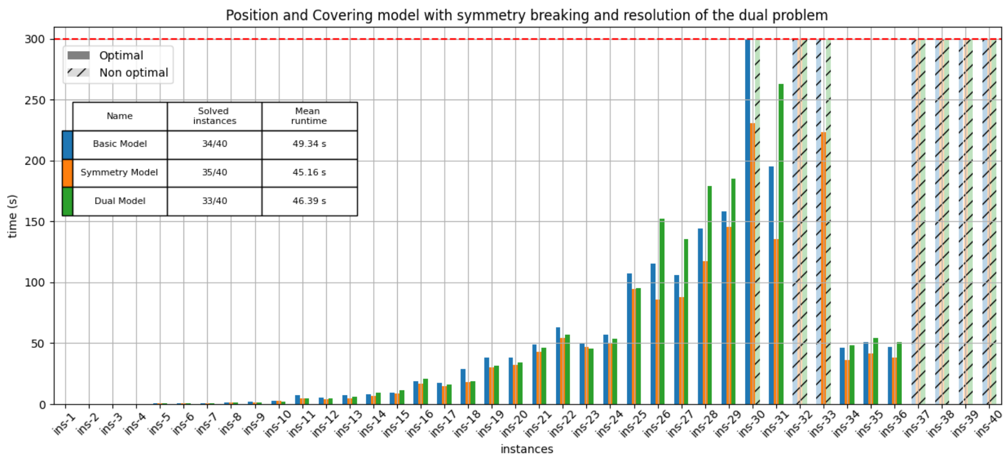
Figure 32: Results for the Basic model obtained with the *Position and Covering* approach with solvers *CPLEX* and *Gurobi*.

Figure 33 shows the performances of the solver *CPLEX* while considering symmetry breaking and the resolution of the dual problem. In particular:

- **Symmetry Model** is the variation of Basic Model that solves the problem with symmetry breaking.
- **Dual Model** is the variation of Basic Model that solves the dual version of the problem.

The Dual model is discarded since it solves one less instance than the Basic Model. On the other hand the Symmetry Model manages to solve one more instance than the Basic Model and also slightly reduces the average solving time.

Hence, the model applying symmetry breaking and using solver *CPLEX* (Symmetry Model) is the best out of the *Position and Covering* approach ones without considering rotations, solving 35 of the 40 instances in an average time of 45.16 seconds.



	Name	Solved instances	Mean runtime
🟦	Basic Model	34/40	49.34 s
🟧	Symmetry Model	35/40	45.16 s
🟩	Dual Model	33/40	46.39 s

Figure 33: Results for solver *CPLEX* obtained with the *Position and Covering* approach considering the Basic Model, the model performing symmetry breaking (Symmetry Model) and the model solving the dual version of the problem (Dual Model)

5.2.4 Rotation variant

Regarding the version of the VLSI problem allowing the rotation of 90° of the circuits, a similar procedure has been implemented.

Both the position and covering phases have been slightly modified.

Position phase The position phase proceeds as in the base variant, although the sets of valid positions V are computed firstly considering each circuit i with its normal orientation, then again for every circuit considering it rotated of 90°.

$$\forall i \in 1..n,\ V_i = \text{set of valid positions of circuit } i$$

$$\forall j \in n+1..2 \cdot n,\ V_j = \text{set of valid positions of circuit } i = j - n \text{ rotated}$$

The correspondence matrix C is built according to 5.2.1, considering the enlarged sets of valid positions in V.

Covering phase The covering phase for the rotation variant proceeds as the previous one, checking for satisfaction of the problem for the current value of l, increasing it and repeating the process if it fails.

The variables of the model are the same as the one in section 5.2.2, however, the variable x describing in which valid position each circuit is placed is now a $2 \cdot n \times p$ matrix. This occurs since the circuits may be placed with either their normal orientation or rotated.

Moreover the constraints (LP 18), (LP 19), (LP 20) are edited as follows in order to account for the increased amount of freedom in the problem

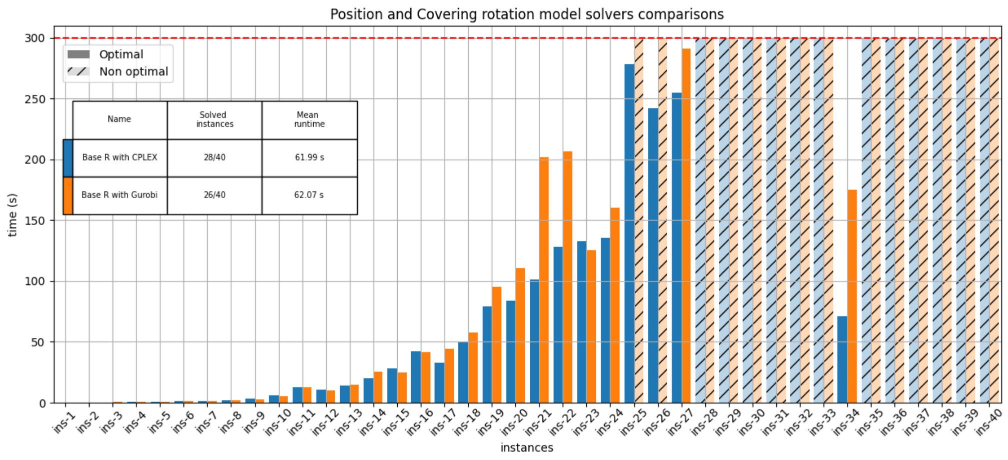
$$\forall i \in 1..c, \sum_{j=1}^{2\cdot n} \sum_{k \in V_j} C_{k,i} \cdot x_{j,k} \leq 1 \tag{LP 21}$$

$$\forall i \in 1..n,\ i' \in n+1..2\cdot n,\ \sum_{j \in V_i} x_{i,j} + \sum_{j \in V_{i'}} x_{i',j} = 1 \tag{LP 22}$$

$$\sum_{i=1}^{2\cdot n} \sum_{j \in V_i} \sum_{k=1}^{c} C_{j,k} \cdot x_{i,j} \leq w \cdot l \tag{LP 23}$$

It is important to notice how constraint LP 22 guarantees that each circuit must be placed exactly one time and it must either be placed normally oriented or rotated, not both.

Results The solver *CPLEX* was used since it obtains better performances according to the tests. As shown in figure 34 the results of the model (**Basic R Model**) are, as expected, dramatically worse than the non-rotation variant. Regarding this variation of the problem the *Position and Covering* approach seems to perform worse than the *Constraints Based* one.

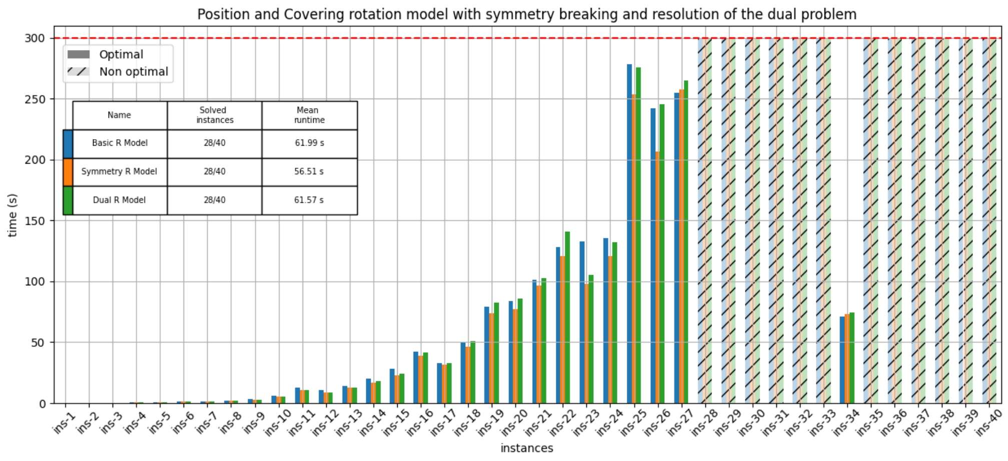


	Name	Solved instances	Mean runtime
🟦	Base R with CPLEX	28/40	61.99 s
🟧	Base R with Gurobi	26/40	62.07 s

Figure 34: Results for the Basic R model obtained with the *Position and Covering* approach considering rotations with solvers *CPLEX* and *Gurobi*.

Again some tests were performed on the model considering symmetry breaking (**Symmetry R Model**) and solving the dual version of the problem (**Dual R Model**). The results are comparable with the ones of Basic R Model, although Symmetry R Model performs slightly better in terms of average solving time.

Hence, the model applying symmetry breaking and using solver *CPLEX* (Symmetry R Model) is the

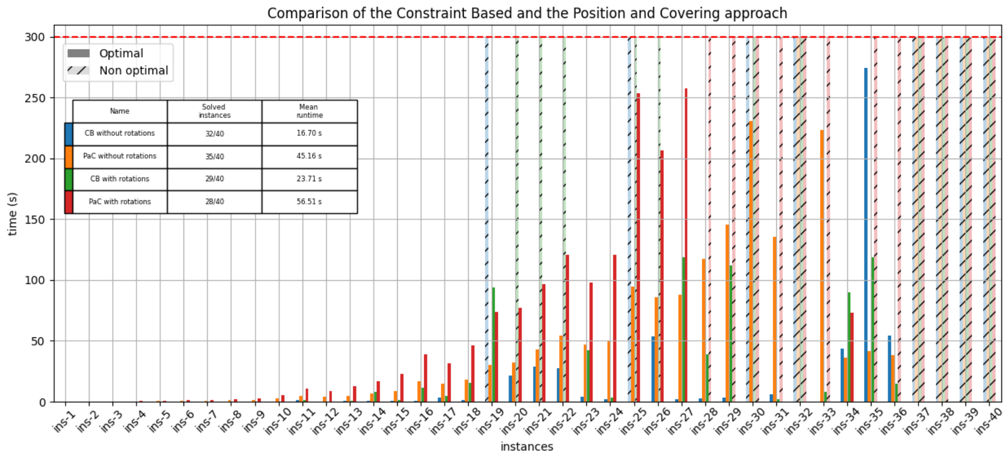


	Name	Solved instances	Mean runtime
🟦	Basic R Model	28/40	61.99 s
🟧	Symmetry R Model	28/40	56.51 s
🟩	Dual R Model	28/40	61.57 s

Figure 35: Results for solver CPLEX obtained with the Position and Covering approach accounting for rotation of the circuits considering the basic model (Basic R Model), the model performing symmetry breaking (Symmetry R Model) and the model solving the dual version of the problem (Dual R Model)

best out of the *Position and Covering* approach ones considering rotations, solving 28 of the 40 instances in an average time of 56.51 seconds.

5.3 Approaches comparison



	Name	Solved instances	Mean runtime
🟦	CB without rotations	32/40	16.70 s
🟧	PaC without rotations	35/40	45.16 s
🟩	CB with rotations	29/40	23.71 s
🟥	PaC with rotations	28/40	56.51 s

Figure 36: Comparison between the *Constraint Based (CB)* and the *Position and Covering (PaC)* approaches considering the variation of the VLSI problem with and without rotation

It is evident how, regarding the problem without rotation, the **Position and Covering** approach manages to solve more instances within the time limit than the **Constraint Based** one.

Nonetheless, the former technique doesn't manage to find a feasible solution for the instances non-optimally solved. This happens because, the first returned solution through this method is always the optimal one. Therefore if the optimal solution is not obtained within the time limit the model produces no solution at all.

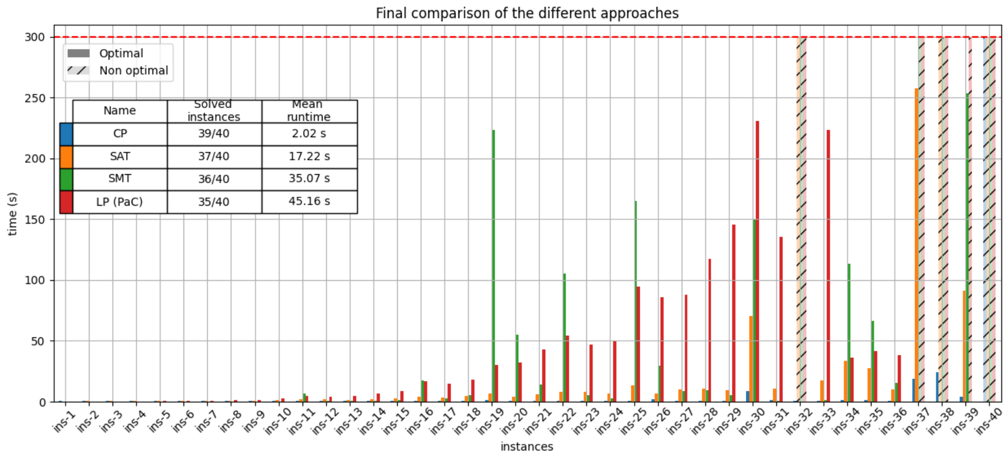
The *Constraint Based* approach instead solves optimally a lower amount of instances, but it usually manages to produce solutions which are one or two units off than the optimal value of l. Regarding the specific set of instances the technique manages to find a feasible solution for all of them.

Although, regarding the rotation variant of the VLSI problem, the **Constraint Based** approach solves a greater amount of instances and seems to be preferable in any case to the **Position and Covering** method.

The *Position and Covering* approach is apparently penalised for more difficult instances by its great spatial complexity, in particular regarding the computation of the sets of valid positions for each circuit.

6 Final conclusions

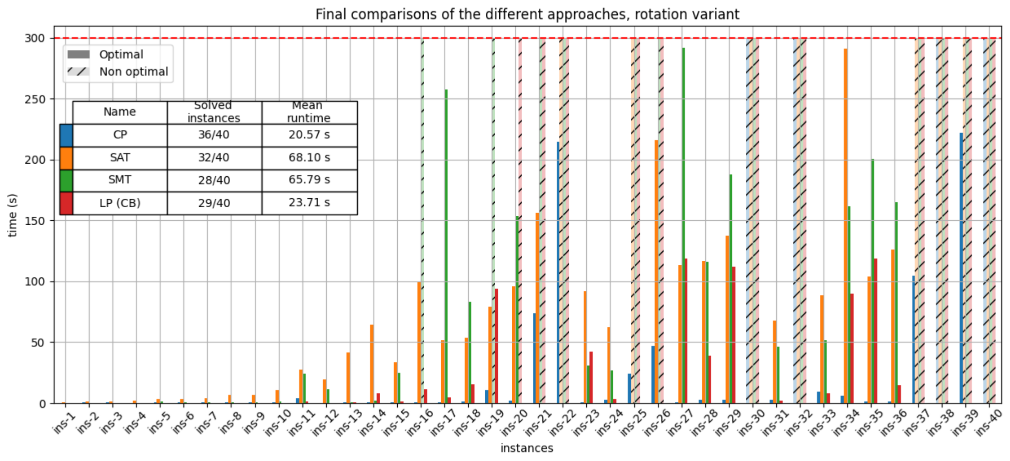
Finally, an overall comparison among all four instances (i.e. *CP*, *SAT*, *SMT* and *LP*) is shown. Figure 37 shows the results of the best models among the different approaches. As it can be seen, the *CP* approach is the best one on the available instances, while the *LP* (with *Position and Covering*) seems to be the worst one.



	Name	Solved instances	Mean runtime
🟦	CP	39/40	2.02 s
🟧	SAT	37/40	17.22 s
🟩	SMT	36/40	35.07 s
🟥	LP (PaC)	35/40	45.16 s

Figure 37: Comparison between the best models among the different approached

Regarding the rotation variant, figure 38 shows the results of the best models among the different approaches. As it can be seen, the *CP* approach is still the best one on the available instances, while the *SMT* is the worst one.



	Name	Solved instances	Mean runtime
🟦	CP	36/40	20.57 s
🟧	SAT	32/40	68.10 s
🟩	SMT	28/40	65.79 s
🟥	LP (CB)	29/40	23.71 s

Figure 38: Comparison between the best models among the different approached, rotation variant

References

[1] Nicholas Nethercote et al. "MiniZinc: Towards a Standard CP Modelling Language". In: Sept. 2007, pp. 529–543. URL: https://www.minizinc.org/.

[2] Department of Computing and Australia Information Systems University of Melbourne. *Chuffed, a lazy clause generation solver*. URL: https://github.com/chuffed/chuffed.

[3] Gecode Team. *Gecode: Generic Constraint Development Environment*. 2006. URL: https://www.gecode.org.

[4] Laurent Perron and Vincent Furnon. *OR-Tools*. Version 9.3. Google, 2022. URL: https://developers.google.com/optimization/.

[5] Hayward H. Chan and Igor L. Markov. "Symmetries in Rectangular Block-Packing". In: *In Proc. of the International Workshop on Symmetry in Constraint Satisfaction Problems*. 2003. URL: https://web.eecs.umich.edu/~imarkov/pubs/misc/symcon03-fp.pdf.

[6] The MiniZinc Team. *Solving Technologies and Solver Backends*. URL: https://www.minizinc.org/doc-2.4.3/en/solvers.html#chuffed.

[7] Brenda Baker, Ed Coffman, and Ronald Rivest. "Orthogonal Packings in Two Dimensions". In: *SIAM J. Comput.* 9 (Nov. 1980), pp. 846–855. URL: https://www.researchgate.net/publication/220617342_Orthogonal_Packings_in_Two_Dimensions.

[8] Microsoft Research. *The Z3 Theorem Prover*. URL: https://github.com/Z3Prover/z3.

[9] Microsoft Research. *The Z3 Theorem Prover, PyPI project*. URL: https://pypi.org/project/z3-solver/.

[10] Steffen Hölldobler and Hau Nguyen. "On SAT-Encodings of the At-Most-One Constraint". In: Sept. 2013.

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

[12] University of Iowa Stanford University. *cvc5*. URL: https://cvc5.github.io/.

[13] Bruno Dutertre Dejan Jovanovic Stéphane Graham-Lengrand and Ian A. Mason. *The Yices SMT Solver*. URL: https://yices.csl.sri.com/.

[14] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *SMT-LIB logics.* URL: `https://smtlib.cs.uiowa.edu/logics.shtml`.

[15] Robert Fourer. *AMPL: a modeling language for mathematical programming.* 2. ed. San Francisco, Calif.: San Francisco, Calif. : Scientific Pr., 1996. URL: `https://ampl.com/resources/the-ampl-book/`.

[16] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual.* 2022. URL: `https://www.gurobi.com`.

[17] IBM ILOG Cplex. "V12. 1: User's Manual for CPLEX". In: *International Business Machines Corporation* 46.53 (2009), p. 157. URL: `https://www.ibm.com/analytics/cplex-optimizer`.

[18] John Forrest et al. *coin-or/Cbc: Release releases/2.10.8.* Version releases/2.10.8. May 2022. URL: `https://doi.org/10.5281/zenodo.6522795`.

[19] Michael Schröder. "A constraint programming approach for the two-dimensional rectangular packing problem with orthogonal orientations". In: 2008. URL: `https://www.semanticscholar.org/paper/A-constraint-programming-approach-for-the-packing-Schr%C3%B6der/4c205b9ad7f1c396544b673063f41492c31a9c98`.

[20] Nestor M. Cid-Garcia and Yasmin A. Rios-Solis. "Exact solutions for the 2d-strip packing problem using the positions-and-covering methodology". In: *PLOS ONE* 16.1 (Jan. 2021), pp. 1–20. URL: `https://doi.org/10.1371/journal.pone.0245267`.