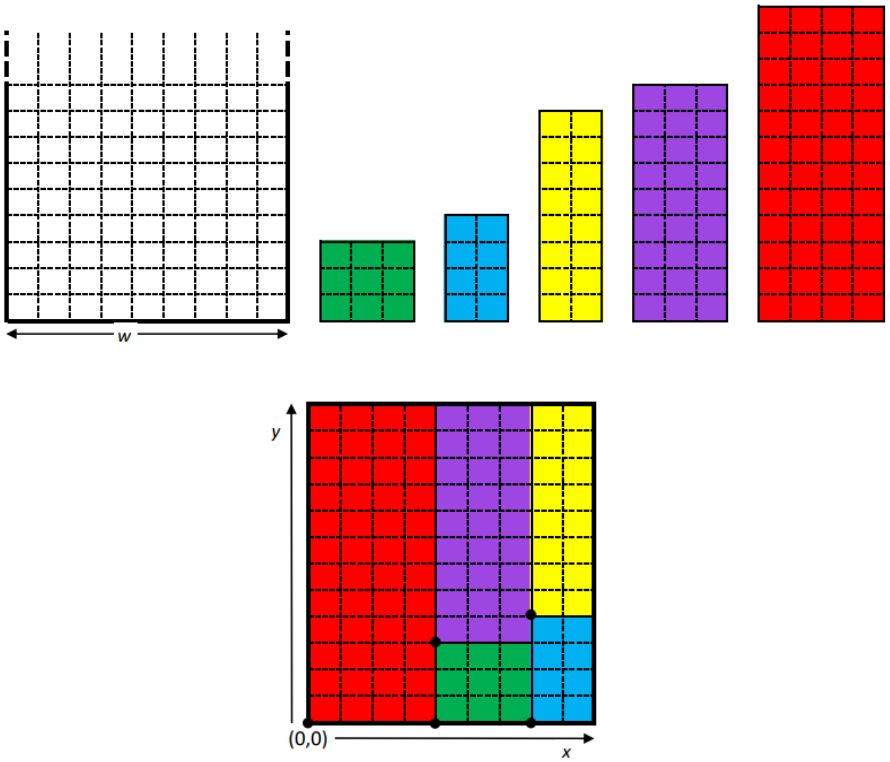
VLSI Design

Combinatorial Decision Making and Optimization

Pietro Epis (pietro.epis@studio.unibo.it)
Michele Milesi (michele.milesi@studio.unibo.it)
Anna Valanzano (anna.valanzano@studio.unibo.it)

July 11, 2022



Contents

1 Introduction

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. Given a fixed-width plate and a list of rectangular circuits, the goal is to decide how to place them on the plate so that the length of the final device is minimized. Accordingly, the problem's parameters are:

- the `width` of the silicon plate;
- the number `n` of necessary circuits to place inside the plate;
- the dimensions of the `n` rectangles.

We considered two variants of the problem: in the first, each circuit must be placed in a fixed orientation with respect to the others, while, in the second one, the rotation is allowed. We approached the problem using Constraint Programming (CP), propositional Satisfiability (SAT) and Linear Programming (LP). Our VLSI project is published on Github.

In this report we will use the following terminology:

- $x_1, ..., x_n$ for the horizontal coordinates of the bottom left vertex of the n chips;
- $y_1, ..., y_n$ for the vertical coordinates of the bottom left vertex of the n chips;
- $w_1, ..., w_n$ for the widths (or horizontal dimensions) of the n chips;
- $h_1, ..., h_n$ or the heights (or vertical dimensions) of the n chips;

Moreover, when we refer to the horizontal and vertical positions of the rectangles, we mean the horizontal and vertical positions of the bottom left vertex of the rectangles.

Preprocessing

We defined other two parameters for the problem, that are useful to reduce the domain of the decision variables:

- the maximum height of the plate `max_height`, that can be defined as the sum of all the vertical dimensions, i.e.,
$$max\_height = \sum_{i}^{n} h_i.$$
- the minimum height of the plate `min_height`, that can be defined as ratio between the total area occupied by the chips and the width, where the total area can be computed as the sum of the products between the horizontal and vertical dimension for each circuit:
$$min\_height = \left\lfloor \frac{\sum_{i}^{n} w_i \cdot h_i}{width} \right\rfloor$$

The problem consists of placing rectangles of very different sizes. After some experiments, it turned out that placing first the biggest rectangle can be convenient. Therefore, in the preprocessing steps we ordered the rectangles with decreasing areas. Accordingly, when we will refer to any two rectangles with indices i, j such that $i \leq j$ we know that the area of the former is bigger or equal to the area of the latter.

2 Constraint Programming (CP)

We tried two different MiniZinc solvers, Chuffed and Gecode and we found out that Chuffed guarantees better performance with our model.

2.1 Decision variables and domains

- the n-dimensional array `positions_x` represents the horizontal positions of the bottom left vertex of the n rectangles. Given the minimum width of the rectangles, the horizontal positions cannot overcome the width of the plate minus the minimum width, i.e.,

$$\forall i \quad x_i \in [0, width - min\{w_1, ..., w_n\}]$$

- the n-dimensional array `positions_y` represents the vertical positions of the bottom left vertex of the n rectangles. We can bound the vertical positions through the `max_height`. Given the minimum height of the rectangles, the vertical positions cannot overcome the `max_height` minus the minimum height, i.e.,

$$\forall i \quad y_i \in [0, max\_height - min\{h_1, ..., h_n\}]$$

- the height of the plate `height` that we defined as the maximum between $y_1 + h_1, ..., y_n + h_n$.

2.2 Constraints

- **Positions bounds**: The fact that both the position coordinates must be greater than 0 is imposed by the domain of the variables. Moreover, for each rectangle, the x coordinate must be less then the difference between the `width` of the plate (that is a fixed parameter of the problem) and the horizontal dimension of that rectangle. In the same way, the y coordinate must be less then the `height` of the plate minus the vertical dimension of the rectangle, i.e.,

$$\forall i \in [1, n] \quad x_i \leq width - w_i$$

$$\forall i \in [1, n] \quad y_i \leq height - h_i$$

  This constraint is an implied constraint, i.e., it is semantically redundant, but computationally significant. Indeed, it is logically implied by the Cumulative constraints.

- **Symmetry Breaking Constraint**: Getting rid of equivalent solutions allows to decrease the search spaces and, for instance, to not lead to thrashing. We tried to break symmetries for three different cases:

  - **The biggest rectangle**: Thanks to the ordering (made in the preprocessing step), we can impose a lexicographic ordering between the positions of the first two rectangles (i.e., the largest rectangles).

- **Row symmetry**: When two blocks i and j have the same vertical position y_i and y_j and the same height h_i and h_j, but different horizontal position x_i and x_j, we can impose an ordering for x_i and x_j, i.e.,

$$x_i \leq x_j$$

- **Column symmetry**: When two blocks i and j have the same horizontal position x_i and x_j and the same width w_i and w_j, but different vertical positions y_i and y_j, we can impose an ordering for y_i and y_j, i.e.,

$$y_i \leq y_j$$

- **Three block symmetry**: In this case we consider three blocks with indices i, j, k. If we have two blocks i and j one on top of the other and next to them a bigger block k, i.e,.

$$\forall i, j, k \in [1, n] \mid k < j < i \quad (x_i = x_j) \wedge (w_i = w_j) \wedge (y_i = y_k) \wedge (h_i + h_j = h_k)$$

then we constrain the block k to be to the left of the blocks i and j

$$x_k \leq x_i.$$

If we have two blocks i and j next to each other and above a bigger block k, i.e,

$$\forall i, j, k \in [1, n] \mid k < j < i \quad (y_i = y_j) \wedge (h_i = h_j) \wedge (x_i = x_k) \wedge (w_i + w_j = w_k)$$

then we constrain the block k to be under the blocks i and j, i.e.,

$$y_k \leq y_i.$$

- **Global constraints**: We decided to use the global constraints `diffn` and `cumulative` to exploit their specialized and efficient propagation.

  - **Cumulative constraints**: The first cumulative constraint enforces that at each x coordinate in the plate, the cumulated height of the rectangles that occupies that coordinate, does not exceed the given capacity `height`. The constraint

$$cumulative([x_1, ..., x_n], [w_1, ..., w_n], [h_1, ..., h_n], height)$$

is satisfied if and only if

$$\forall x \in [1, width] \sum_{i | x_i \leq x < x_i + w_i} h_i \leq height.$$

The second cumulative constraint enforces that at each y coordinate in the plate, the cumulated width of the rectangles that occupies the given y coordinate, does not exceed the given capacity `width`. The constraint

$$cumulative([y_1, ..., y_n], [h_1, ..., h_n], [w_1, ..., w_n], width)$$

is satisfied if and only if

$$\forall y \in [1, height] \sum_{i | y_i \leq y < y_i + h_i} w_i \leq width.$$

  - **Non overlapping constraints**: To avoid overlapping, for each pair of rectangles i and j at least one of these two conditions must be true:

* the sum of the x coordinate and the horizontal dimension of one rectangle must be less than the x coordinate of the other;
* the sum of the y coordinate and the vertical dimension of one rectangle must be less than the y coordinate of the other.

In other words,

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

This constraints has been implemented using the `diffn` constraint. Indeed,

$$diffn([x_1, ..., x_n], [y_1, ..., y_n], [w_1, ..., w_n], [h_1, ..., h_n])$$

automatically constrains the rectangles to be non-overlapping.

2.3 Objective function

The goal is to minimize the final height of the device. We can constrain the objective function with a lower bound `min_height` and an upper bound `max_height`, i.e.,

$$min\_height \leq height \leq max\_height$$

2.4 Rotation

To handle rotation, we defined the array `rotated` of n boolean variables, such that the i-th variable is true if and only if the i-th rectangle has been rotated (with respect to its normal orientation in input). The rotation, can be implemented as the inversion of the dimensions of the rectangle: the width becomes the height, and viceversa. We stored the original dimensions in the arrays `w` and `h`, and the potentially inverted dimensions in the arrays `actual_w` and `actual_h`. To reduce the domain of these two arrays we defined

$$max\_dim = max\{h_1, ..., h_n, w_1, ..., w_n\}$$

$$min\_dim = min\{h_1, ..., h_n, w_1, ..., w_n\}$$

Accordingly, we can bound the "actual" dimensions of the rectangles through min_dim and max_dim:

$$\forall i \in [1, n] \quad min\_dim \leq actual\_w_i \leq max\_dim$$

$$\forall i \in [1, n] \quad min\_dim \leq actual\_h_i \leq max\_dim$$

If, according to the array `rotated`, the i-th rectangle is rotated, the actual dimensions `actual_w` and `actual_h` are inverted with the following constraints:

```
constraint forall(i in 1..n)(
  actual_w[i] = (if rotated[i] then h[i] else w[i] endif)
);
constraint forall(i in 1..n)(
  actual_h[i] = (if rotated[i] then w[i] else h[i] endif)
);
```

Moreover, if the dimensions of the rectangle are equal, we avoid the rotation forcing the element of the array `rotated` that corresponds to that rectangle to be 0.

2.5 Search

We investigated the best way to search for solutions in CP comparing different strategies. We considered different ordering of variables:

- `input_order`: choose in order from the array
- `first_fail`: choose the variable with the smallest domain size
- `dom_w_deg`: choose 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.

We discovered that thanks to `seq_search` the performance with `first_fail` improved. For ordering the domain of the variables we used `indomain_min`, which assigns to the variable its smallest domain value. To compare the different strategies we used the solver Gecode for `dom_w_deg`, that seems to be not compatible with Chuffed. We report some results in Table 1.

Instance	**Strategy**	**Results**			
		Failures	Restarts	Time	Propagations
5	input_order	5	2	0.001	2804
	first_fail	5	2	0.002	2581
	dom_w_deg	137	4	0.016	8429
21	input_order	22465	10	3.21	9762735
	first_fail	20932	11	1.32	13942433
	dom_w_deg	1290834	18	32.861	96539551
35	input_order	3741	7	0.427	2845959
	first_fail	1863	8	0.201	1294721
	dom_w_deg	51263	12	1.668	2557970

Table 1: Results for some instance using different heuristics.

Lastly, we compared the performance using different heuristics and using or not a restart strategy. In particular, we used `restart_geometric`, that restarts after L resources, with the new limit $\alpha \cdot L$. In Table 2, we report the the number of instances we were able to solve and the average time.

Strategy	**Restart**	**Results**	
		Solved instances	**Average time**
input_order	Yes	30	25.04
	No	30	24.59
first_fail	Yes	34	38.39
	No	33	33.29

Table 2: Global results using different heuristics, with restart or no restart.

2.6 Experimental Results

The histogram in Figure 1 reports the time to solve each instance with **no rotation**, **restart**, and two different search heuristics, `first_fail` and `input_order`.

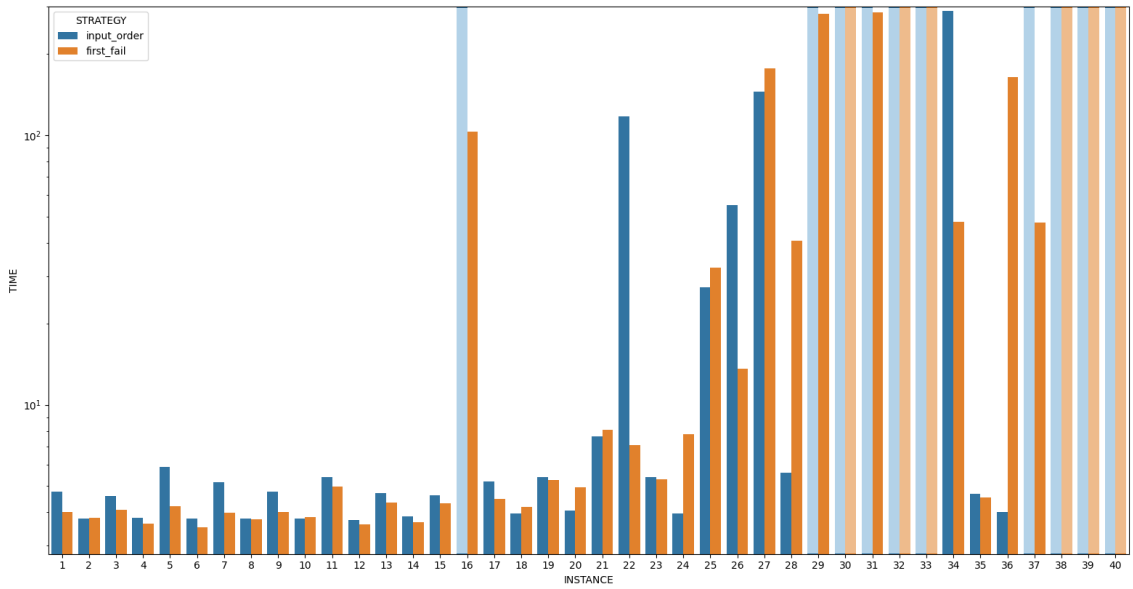


Figure 1: Results with **no rotation**, **restart**, and two different search heuristics

The histogram in Figure 2 that reports the time to solve each instance with **rotation**, **restart**, and two different search heuristics, `first_fail` and `input_order`.

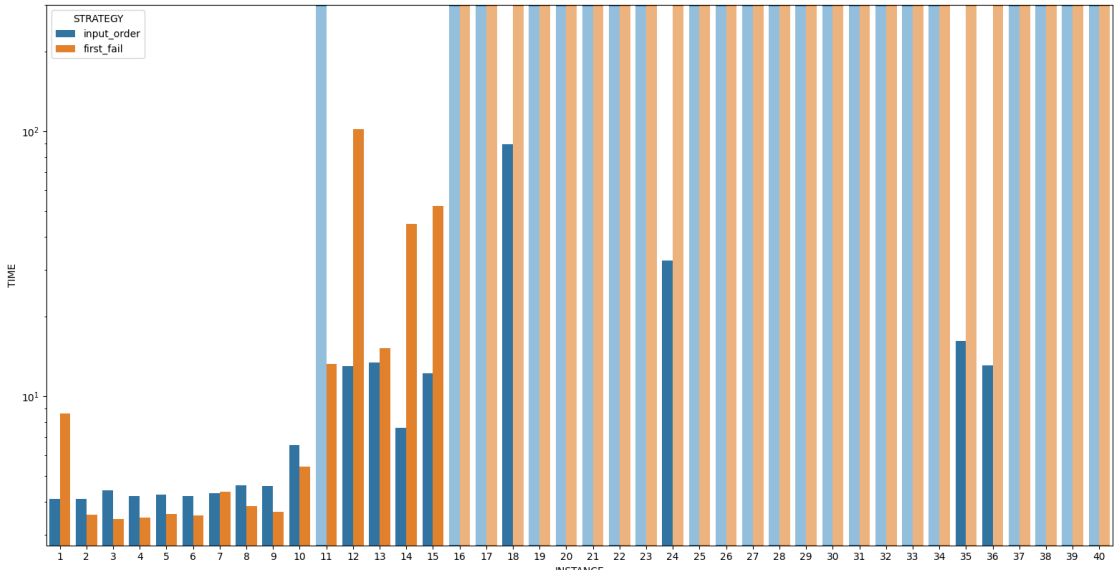


Figure 2: Results with **rotation**, **restart**, and two different search heuristics

The histogram in Figure 3 reports the time to solve each instance with **no rotation**, **no restart**, and two different search heuristics, `first_fail` and `input_order`.

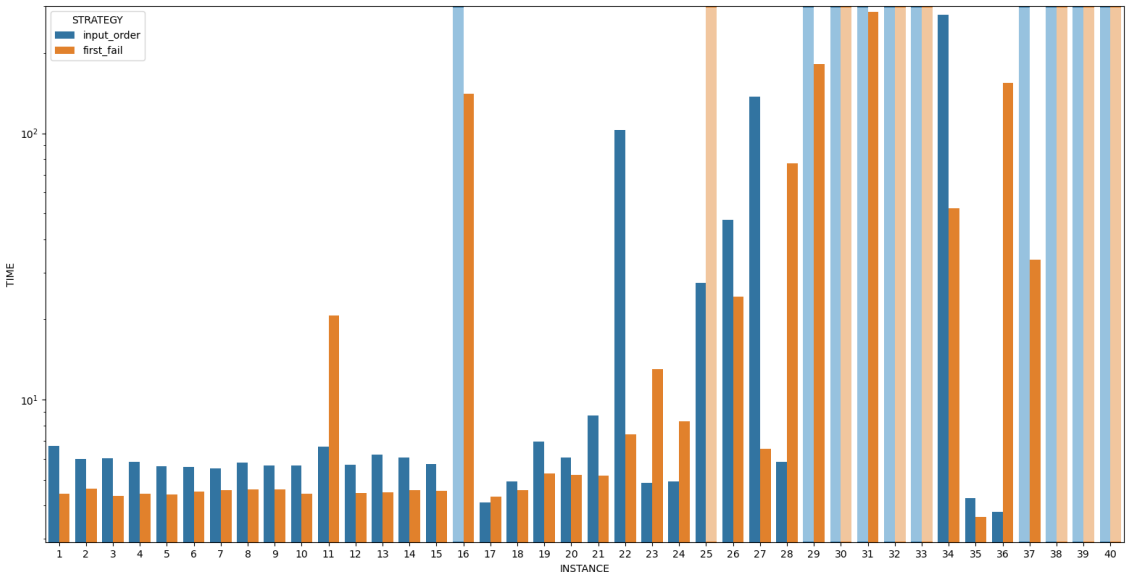


Figure 3: Results with **no rotation**, **no restart**, and two different search heuristics

The histogram in Figure 4 that reports the time to solve each instance with **rotation**, **no restart**, and two different search heuristics, `first_fail` and `input_order`.

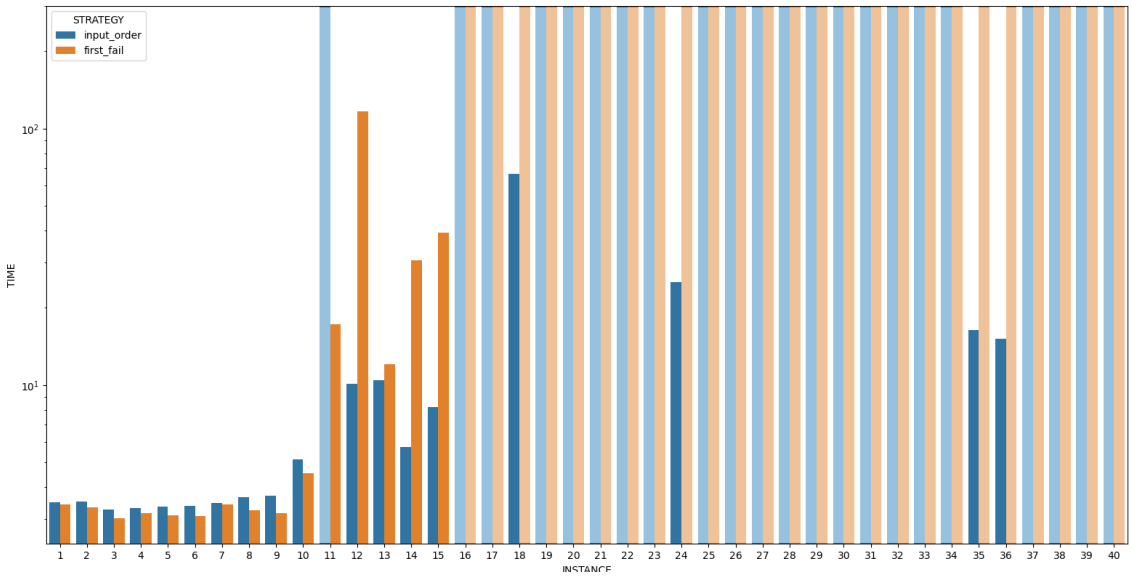


Figure 4: Results with **rotation**, **no restart**, and two different search heuristics

According to the discussed results we can conclude that the best possible configuration is the one that makes use of Chouffed as solver, `first_fail` as search strategy and with restart mode.

Some visual results

We plotted the results using the Python library `matplotlib`. Each rectangle is assigned to a different color. Moreover, we reported on each rectangle its respective number (e.g., the rectangle with number 0 is the first rectangle in the list of rectangles to place). We ordered the rectangles by decreasing area and we forced the biggest rectangle to be to the left or under the second biggest rectangle. Therefore, in the following figures, we can see that the biggest rectangle (with index 0) is in the bottom left of the plate.

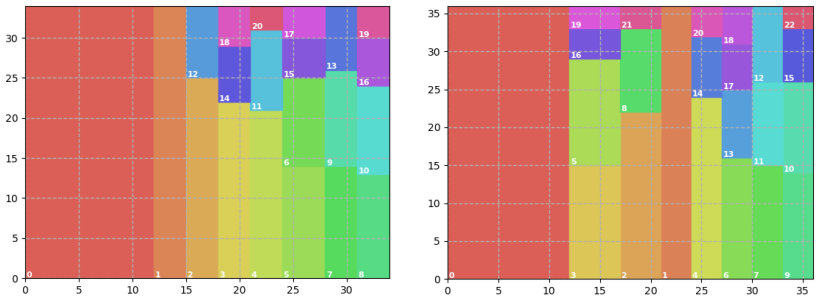


Figure 5: Instances 27 and 29

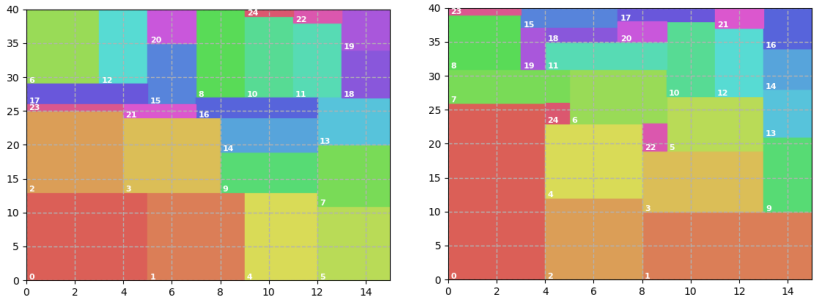


Figure 6: Instances 34 and 35

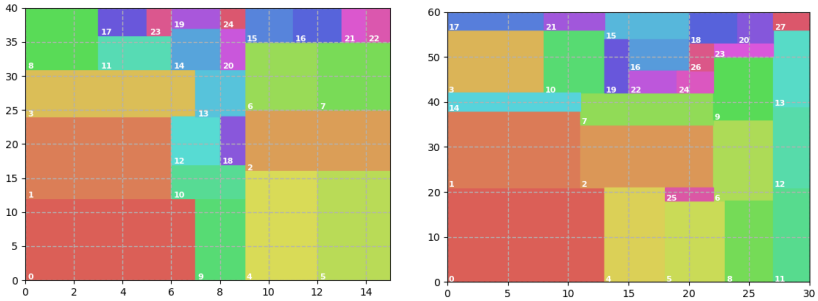


Figure 7: Instances 36 and 37

3 Propositional Satisfiability (SAT)

For the SAT approach we used Z3 API in Python.

3.1 Decision Variables

- `map`: As the SAT formulation requires Boolean Variables, we instantiated a three dimensional array of booleans (`map`), such that the element at position (i, j, k) is equal to 1 if and only if the (i, j)-th cell belongs to the k-th rectangle. This means that each layer of our data structure refers to a specific rectangle and it's made up by all zeros except for the area where the related rectangle is placed, that contains ones. Accordigly, given the `width` and the maximum length `max_height` of the plate, we defined $\texttt{n} \cdot \texttt{width} \cdot \texttt{max\_height}$ variables.
- `height`: As in the CP approach, the height of the plate can assume whatever value in the range between `min_height` and `max_height`. Accordingly, we defined an array `height` of dimension $max\_height - min\_height$ such that, the k-th element of the array is 1 if and only if the height of the plate is equal to the k-th height in the range $[min\_height, max\_height]$.

For simplicity, in the following:

- x_{ijk} refers to the k-th element of the array related to the position (i, j), where i indicates the vertical position and j the horizontal one.
- h_k refers to the k-th element of the array `height`.

3.2 Constraints

- **No overlapping**: For each position (i, j), at most one element of the corresponding array should be 1. This because a position (i, j) can be occupied by one and only one rectangle to avoid overlapping. In other words,
$$\forall (i,j) \bigwedge_{0<k_1<k_2\leq n} \neg(x_{ijk_1} \wedge x_{ijk_2})$$
- **At least one rectangle for each depth layer**: Each rectangle k should occupy at least one position, i.e.,
$$\forall k \in [1, n] \bigvee_{(i,j)} x_{ijk}$$
- **Exactly one height**: As the final height of the plate is unique, exactly one variable in the array `height` must be 1. Therefore, at least one variable and at most one variable in the array must be 1:
$$\left(\bigvee_{k\in[min\_height,max\_height]} h_k \right) \wedge \left(\bigwedge_{min\_height\leq k_1<k_2\leq max\_height} \neg(h_{k_1} \wedge h_{k_2}) \right)$$

- **Height channeling** The variable h_i is 1 if and only if at least one position (i, j) is occupied by a certain rectangle k, i.e., if, for all the possible j, k, there is at least one variable x_{ijk} equal to 1:

$$\bigvee_{j,k} x_{ijk}$$

Moreover, the variable h_i is 1 if and only if, for all the positions (l, j) with $l > i$, there is no variable x_{ijk} equal to 1, i.e.,

$$\bigwedge_{l=i+1}^{max\_height} \neg \bigvee_{j,k} x_{ljk}$$

Combining the two conditions, we get the final constraint:

$$\forall i \in [min\_height, max\_height] \quad h_i \iff \left(\bigvee_{j,k} x_{ijk} \right) \wedge \left(\bigwedge_{l=i+1}^{max\_height} \neg \bigvee_{j,k} x_{ljk} \right)$$

- **Position of rectangles**: To find the placement of rectangles that minimize the height we try different possible positions. For each rectangle k we set to 1 the variables of the rectangle (depending on its dimensions w_k and h_k) , i.e.,

$$\left(\bigwedge_{\substack{i \in [l,\, l + h_k] \\ j \in [m,\, m + w_k]}} x_{ijk} \right)$$

AND we set to 0 the variables outside the rectangle i.e.,

$$\left(\bigwedge_{\substack{i \notin [l,\, l + h_k] \\ j \notin [m,\, m + w_k]}} \neg x_{ijk} \right)$$

We repeat this operation for all the rectangles ($\forall k$) and for all the possible positions. To avoid the overflow of the rectangle k, we consider all the possible positions $(l,\, m)$ such that $0 \leq l \leq max\_height - h_k$ and $0 \leq m \leq width - w_k$, i.e.,

$$\forall k \bigvee_{\substack{l \in [0,\, max\_height\, -\, h_k] \\ m \in [0,\, width\, -\, w_k]}} \left[\left(\bigwedge_{\substack{i \in [l,\, l + h_k] \\ j \in [m,\, m + w_k]}} x_{ijk} \right) \wedge \left(\bigwedge_{\substack{i \notin [l,\, l + h_k] \\ j \notin [m,\, m + w_k]}} \neg x_{ijk} \right) \right].$$

- **Symmetry Breaking**: For the symmetry breaking constraint we implemented `lex_lesseq` for SAT variables. As for the CP formulation, we imposed an ordering between the first two rectangles (i.e., the biggest ones).

```
def lex_lesseq(x, y):
return And([Implies(x[0], y[0])] +
    [Implies(And([x[i] == y[i] for i in range(k)]), Implies(x[k], y[k]))
    for k in range(1, len(x))])
```

To apply this constraint, we considered the first two levels of the three dimensional array `map` and flattened them into two arrays:

```
s.add(lex_lesseq(flatten(map[:][:][0]), flatten(map[:][:][1])))
```

3.3 Search

To minimize the height of the plate, we try to solve the problem with the minimum height. If the problem is not satisfiable with that height, we increase the height, repeating this process until the maximum height.

3.4 Rotation

To handle rotation, we added new variables:

- an n dimensional array of Boolean variables `rotated` such that the i-th element of the array is 1 if and only if the i-th rectangle is rotated;
- a $n \times 2$ matrix `rotated_dimensions`, such that the i-th row indicates the (eventually) rotated dimensions of the i-th rectangle.

As we must consider all the possible positions for the non rotated and rotated rectangle, we modified the constraint previously named "Position of rectangle". Indeed, if the rotation is allowed, we add two implications: if the rectangle is not rotated, at least one among all the possible positions of the non rotated rectangle must hold, otherwise, at least one among all the possible positions of the rotated rectangle must hold.

4 Linear Programming (LP)

As regards the development of the Linear Programming model, we took advantage of the model we had already developed for Constraint Programming and adapted several concepts to the new context.

4.1 Decision Variables

- the $n \times 2$ matrix `positions` such that the i-th row of the matrix indicates the horizontal and vertical position of the i-th rectangle;
- the three dimensional array δ of Boolean variables (of dimension $n \times n \times 4$), which is used for the non overlapping constraint;
- the `height` of the plate.

4.2 Constraints

- **Height bounds**:

$$min\_height \leq height \leq max\_height$$

- **Positions bounds**:

$$\forall i \in [1, n] \quad 0 \leq x_i \leq width - w_i$$
$$\forall i \in [1, n] \quad 0 \leq y_i \leq height - h_i$$

- **Cumulative constraint**:

$$\forall x \in [1, width] \sum_{i|x_i \leq x < x_i + w_i} h_i \leq height.$$

$$\forall y \in [1, height] \sum_{i|y_i \leq y < y_i + h_i} w_i \leq width.$$

- **No overlapping constraint**: To avoid overlapping, there must be true that

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

According to the technique suggested in the following link, the "or" condition can be modeled with the help of binary variables δ and big-M constraints. In particular, in the array δ the first two dimensions i and j refer to the couple of rectangles i and j, and the third dimension k $\in [1, 4]$ refers to one among the four OR conditions. Accordingly, we get the following constraints:

$$\begin{aligned}
\forall i, j \in [1, n] \mid i \neq j \quad & x_i + w_i \leq x_j + M \cdot \delta_{ij1} \\
\forall i, j \in [1, n] \mid i \neq j \quad & x_j + w_j \leq x_i + M \cdot \delta_{ij2} \\
\forall i, j \in [1, n] \mid i \neq j \quad & y_i + h_i \leq y_j + M \cdot \delta_{ij3} \\
\forall i, j \in [1, n] \mid i \neq j \quad & y_j + h_j \leq y_i + M \cdot \delta_{ij4} \\
\forall i, j \in [1, n] \mid i \neq j \quad & \sum_{k=1}^{4} \delta_{ijk} \leq 3
\end{aligned}$$

We decide to set the constant M as $M = max\_height + width$.

4.3 Objective Function

The objective function is the total area of the plate, i.e.,

$$width \cdot height$$

that must be minimized. As the width of the plate is fixed, the area depends only on the height.

4.4 Rotation

In order to manage the potential rotation of the rectangles, we adopted a solution that reflects the one adopted in CP. Indeed, we exploited an array `rotation` whose elements belong to the domain $\{0, 1\}$, meaning that `rotation[i]` is equal to one if and only if the i-th rectangle has been rotated in the output solution. Furthermore, we made use additional variables, stored in the array `actual_dimensions`, that contains the dimensions of the rectangles in inverted order (wrt the array `dimensions`) in case they've been marked as rotated. This concept has been implemented through the following relations:

$$\forall i \quad actual\_w_i = rotated_i \cdot h_i + (1 - rotated_i) \cdot w_i$$

$$\forall i \quad actual\_h_i = (1 - rotated_i) \cdot h_i + rotated_i \cdot w_i$$

4.5 Experimental Results

The histogram in Figure 8 reports the time to solve each instance with and without rotation in LP. While without rotation we solved almost all the instances, with rotation we solved only 22/40 instances.

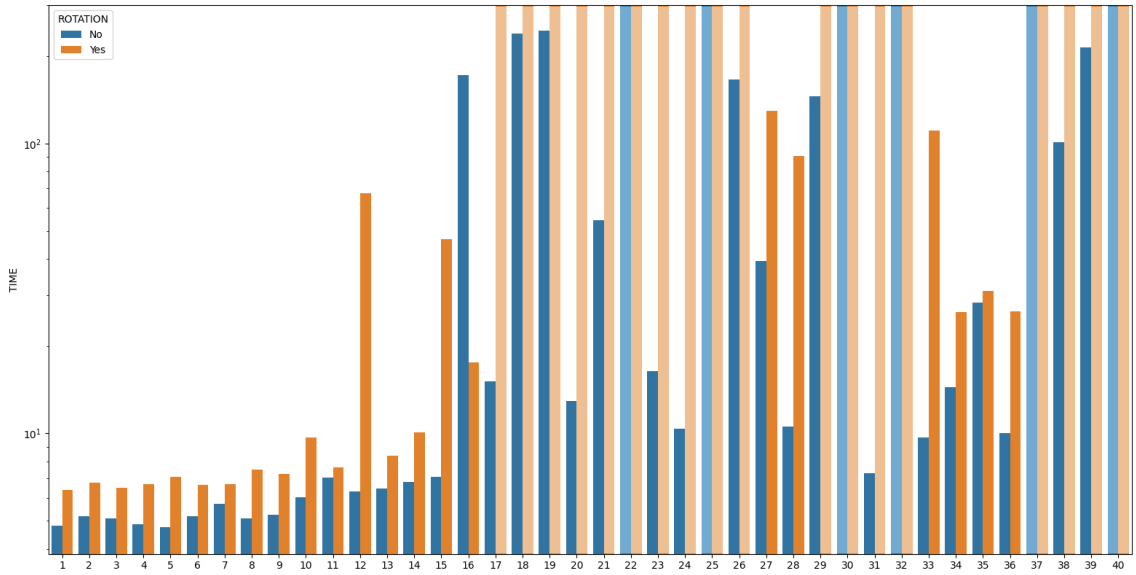


Figure 8: Results with **rotation** and **no rotation** in LP

5 Conclusion

While we obtained satisfying performances with CP and LP approaches, with SAT approach we did not solve many instances in five minutes. In general, we observed that allowing the rotation worsens the performance as a consequence of the increasing of variables and constraints.