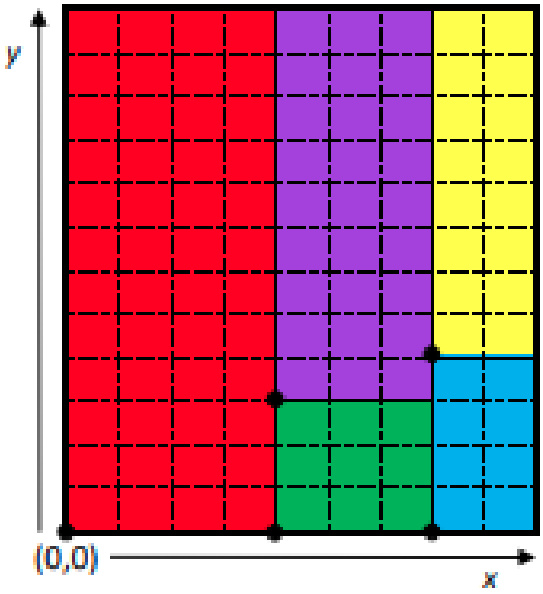
VLSI design: CP and SMT approaches

Riccardo Fava - riccardo.fava6@studio.unibo.it

Combinatorial and Decision Making Course - Module 1



Contents

1 Introduction

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. The trend of shrinking transistor sizes, allowing engineers to fit more and more transistors into the same area of silicon, enabled the modern cellphone to mature into a powerful tool with incredible features, that we can comfortably carry in a pocket. In this way the VLSI problem led us to the development of search strategies in order to determine the optimal placement of the circuits on the silicon chips. This report describes a Combinatorial and Decision making approach to deal with the VLSI problem; two different technologies have been employed, namely Constraint Programming (CP) and Satisfiability Modulo Theories (SMT).

2 Input and Output

An instance of VLSI is a text file consisting of lines of integer values. The first line of the input file gives w, which is the width of the silicon plate. The following line gives n, which is the number of necessary circuits to place inside the plate. Then n lines follow, each with x_i and y_i , representing the horizontal and vertical dimensions of the i-th circuit. A solution of an particular instance involves the computation of the minimum height h as well as determining the coordinates $(\tilde{x_i}, \tilde{y_i})$ of the bottom left corner of each circuit. All these informations are stored into the output files.
In the following table the is an example of the input and output files of the first instance:

Input	Output	Description
8	8 8	w h
4	4	n
5 5	5 5 0 0	x_1 y_1 $\tilde{x_1}$ $\tilde{y_1}$
5 3	5 3 0 5	x_2 y_2 $\tilde{x_2}$ $\tilde{y_2}$
3 5	3 5 5 0	x_3 y_3 $\tilde{x_3}$ $\tilde{y_3}$
3 3	3 3 5 5	x_4 y_4 $\tilde{x_4}$ $\tilde{y_4}$

3 CP

Constraint programming is a paradigm for solving combinatorial problems. The basic idea in CP is that the user states the constraints in a declarative fashion and a general purpose constraint solver is used to solve them.
In this section we present a CP model for encoding the VLSI problem through the use of the Minizinc constraint modeling language.

3.1 Variables

3.1.1 Decision variables

First we define the model's decision variables:

- x and y: array containing respectively the horizontal and vertical coordinates of the bottom left corner of each circuit
- *plate_height*: represent the actual height of the plate

```
array[circuits] of var 0..w - min(widths): x;
array[circuits] of var 0..max_height - min(heights): y;
var min_height..max_height: plate_height;
```

3.1.2 Input variables

Then the following input variables were defined:

- n: number of circuits to place inside the plate
- w: width of the plate
- *widths* and *heights*: are two arrays containing the width and the height of each circuit

```
int: w;
int: n;
set of int: circuits = 1..n;
array[circuits] of int: heights;
array[circuits] of int: widths;
```

3.1.3 Other variables

- *min_height* and *max_height*: values corresponding to the lower and upper bound for the plate height
- *big_circuit_idx*: represent the index of the circuit with the largest area

```
int: max_height = sum(heights);
int: min_height = max(heights);
int: big_circuit_idx = arg_max([heights[c]*widths[c] | c in circuits]);
```

3.1.4 Objective function

Lastly we define the objective function simply as the minimization of the decision variable *plate_height*:

```
solve minimize plate_height
```

3.2 Constraints

The CP solver tries to find a solution by scanning the search space; therefore including some constraints can improve significantly the performances of the model by simply reducing the domain of the decision variables.

3.2.1 Reducing variables domain

For each decision variable we defined a specific domain:

- x: for what concerns the horizontal coordinate of the bottom left corner of each circuit, it cannot assume a value greater than the width of the plate minus the width of the less wide rectangle; otherwise it will fall outside the board.
- y: the same holds for the vertical coordinate that cannot be greater than the upper bound of the *plate_height* minus the height of the shortest rectangle.
- *plate_height*: its domain must be included into the range composed by the height of the highest rectangle as lower bound, and the sum of the heights of all the circuits as upper bound.

3.2.2 Main constraints

We know that each circuit, must be inserted into an allowed position on the plate, without falling outside the perimeter. This can be encoded through the following constraints, where we specify that the horizontal coordinate x of the bottom left corner must be lower than the difference between the width of the plate and the width of the rectangle itself. The same holds for the vertical coordinate y that cannot exceed the difference between the height of the plate and the height of the circuit.

```
constraint forall(i in circuits)(x[i] <= w - widths[i])::domain;
constraint forall(i in circuits)(y[i] <=plate_height - heights[i])::domain;
```

3.2.3 Implied constraints

Implied constraints are logical consequences of the initial specification of the problem. Considering the problem we can make the following statement: if we draw a horizontal line and sum the horizontal sides of the traversed circuits, the sum can be at most equal to the plate width w. A similar property holds if we draw a vertical line.

3.2.4 Global constraints

One of the most important features of CP solvers is constraint propagation.
Constraint propagation consists in the examination of the constraints in order to remove inconsistent values from the domains of the decision variables. In our problem we have several constraints that have to be propagated; therefore for reaching the best performances in terms of efficiency there is necessity to have an effective constraint propagation. For this reason in our model we used Minizinc global constraints. These kind of constraints represent high-level modelling abstractions, for which many solvers implement specialized inference algorithms. In this way, using global constraints, we can obtain a potentially much more efficient constraint propagation with respect to the generalized one. We deployed the following global constraints:

- $diffn(x, y, dx, dy)$ [4]: this constraint force the rectangles specified by their origins x, y and dimensions dx, dy to be non-overlapping.
- $cumulative(s, d, r, b)$ [3]: using this global constraint we implemented what written before in the implied constraint section (3.2.3). This constraint is usually

employed during the modeling of scheduling problems, but can be useful also in our case. Indeed we can rephrase our problem in this way: a set of tasks given by start time s (x or y coordinates), duration d (the respective width or height), and resource requirements r (the other dimension), never require more than a global resource bound b (either the width or the height of the plate) anytime.

```
constraint cumulative(x, widths, heights, plate_height)::domain;
constraint cumulative(y, heights, widths, w)::domain;
constraint diffn(x, y, widths, heights)::domain;
```

3.2.5 Symmetry breaking constraints

Symmetries increase the size of the search space and therefore, time is wasted in visiting new solutions which are symmetric to the already visited ones. Using symmetry breaking constraints we can eliminate symmetries variants of a solution, reducing the search space size. The VLSI problem is full of symmetries; having a board of circuits and applying a rotation or a reflection over the axis (x, y or both) we can end up into a completely symmetric solution to the starting one.
Another symmetry that we have found consists in the interchangeability of equivalent circuits; when two or more rectangles have equal width and height, they can be easily swapped creating a symmetric solution.
In order to rule out these symmetries we inserted into the model the following constraints:

- For dealing with the **rotation** and **reflection** symmetries we placed the bottom left corner of the circuit with the largest area into the origin of the plate.
- For solving the **circuit equality** symmetry instead, we applied an ordering between the rectangles of same size through the use of a lexicographic global constraint [5].

```
constraint symmetry_breaking_constraint(
    x[big_circuit_idx] = 0 /\ y[big_circuit_idx] = 0);
```

```
constraint symmetry_breaking_constraint(
    forall(i in 1..n-1, j in 2..n where i<j)
        (widths[i]==widths[j] /\ heights[i]==heights[j]) ->
        lex_less([x[i],y[i]],[x[j],y[j]])
);
```

3.3 Rotation

The original formulation of the problem doesn't allow the rotation of the circuits. If instead we have to take into account also the possibility that an $n \times m$ circuit can be positioned as an $n \times m$ circuit in the silicon plate, we have to apply some changes into our model.

3.3.1 Variables

In order to handle the rotation property we decided to add the following variables to our model:

- *rotations*: An array of boolean variables of size equal to the number of circuits, that is useful to keep track of the rotation of each circuit present in the plate.
- *actual_widths*: An array of integers where we stored the actual value of the width of each circuit. It is useful because in this array when the rotation of an individual rectangle is applied, we store the value of the width after the swap with the value of the height.
- *actual_heights*: An array of integers where we stored the actual value of the height of each circuit. The same discussion made before holds; after the eventual rotation and the consequential swap, the new value of the height is stored here.

```
array[circuits] of var bool: rotations;
array[circuits] of var int: actual_widths =
    [(if rotations[i] then heights[i] else widths[i] endif) | i in circuits];

array[circuits] of var int: actual_heights =
    [(if rotations[i] then widths[i] else heights[i] endif) | i in circuits];
```

3.3.2 Constraints

Next to the constraints already explained before, we added a new symmetry breaking constraint in order to avoid the rotation of the squared circuits that would not change the solution obtained.

```
constraint symmetry_breaking_constraint(
    forall (i in circuits) (if actual_heights[i] == actual_widths[i]
    then rotations[i] = 0 endif)
);
```

3.4 Search

In Minizinc there is the possibility to interact with the solver specifying how to carry out the search process. Through the use of search annotations [6] it is possible to guide the search determining the value ordering and the variables ordering. By default in Minizinc there is no specification on how to apply the search, but like in our case this feature is very important for tackling difficult problems and obtaining good results in terms of efficiency.
We used the `int_search(<variables>, <varchoice>, <constrainchoice>)` annotation in order to perform the search process into the domain of the integer variable *plate_height*; the second and the third argument are respectively the variable ordering and domain search heuristics.
We tried the following variable ordering heuristics:

- `input_order`: choose the variables in the given order
- `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
- `first_fail`: choose the variable with the smallest domain size

And the following domain search heuristics:

- `indomain_min`: assign the variable its smallest domain value
- `indomain_random`: assign the variable a random value from its domain

3.4.1 Restarts

Given the fact that CP solvers are based on a depth first search approach, they suffers from the problem that wrong decisions made at the top of the search tree can take an exponential amount of search to undo. A way to partially solve this problem is to restart the search from the top thus having a chance to make different decisions. MiniZinc includes annotations to control restart behaviour. We tried the following restart strategies [7]:

- `restart_luby(<scale>)`: where `<scale>` is an integer. The kth restart gets `<scale>` $\times L[k]$ where $L[k]$ is the kth number in the Luby sequence.
- `restart_geometric(<base>,<scale>)`: where `<base>` is a float and `<scale>` is an integer. The kth restart has a node limit of `<scale>` $\times$ `<base>`^{k}

3.5 Experiments and Results

Hardware setup used during this project:

- **Environment**: Minizinc 2.6.1, Python 3.7.10
- **CPU**: Intel core i7-8550U
- **RAM**: 8 GB DDR3
- **OS**: Ubuntu 20.04

The first phase of experimentation that we performed is to build a baseline model in order to become familiar with the VLSI problem, then we improved this model by adding more specific and suitable constraints.
First of all we substituted the following non-overlapping constraint:

```
constraint forall (i,j in circuits where i<j)
    (x[i] + widths[i] <= x[j] \/ x[j] + widths[j] <= x[i]
    \/ y[i] + heights[i] <= y[j] \/ y[j] + heights[j] <= y[i]);
```

with the `diffn(x,y,dx,dy)` global constraint described in section 3.2.4.
Then in order to encode in the model the property (implied constraint) described in the section 3.2.3, we inserted also the `cumulative(s,d,r,b)` global constraint

reported in section 3.2.4. With this model we solved 26 instances out of 40. In order to further improve the performances of model we decided to add the symmetry breaking constraints described in section 3.2.5. Once applied this change we manage to solve 29/40 instances. These results are not very satisfying so we decided to sort the circuits by decreasing value of their area, following the intuition that once placed the largest rectangles in the plate, for the CP solver it's easier to find a position for the smaller ones. This simple step enabled us to improve again the performances of our model, so based on this fact, in the consequently tests we always ordered the circuits.
Next we moved our focus on the several search strategies, domain search heuristics and restart strategies described in paragraph 3.4. We experimented all the possible combinations establishing a comparison in the following tables.
The table below describes the results that we got using `indomain_random` as search domain heuristic:

Search strategy	Restart Strategy	Solved Instances
input_order	None	25/40
	luby(100)	28/40
	geometric(1.5,100)	31/40
first_fail	None	25/40
	luby(100)	29/40
	geometric(1.5,100)	31/40
dom_W_deg	None	25/40
	luby(100)	29/40
	geometric(1.5,100)	31/40

Here instead we can see the performances of our model using `indomain_min` as search domain heuristic:

Search strategy	Restart Strategy	Solved Instances
input_order	None	32/40
	luby(100)	30/40
	geometric(1.5,100)	32/40
first_fail	None	32/40
	luby(100)	30/40
	geometric(1.5,100)	32/40
dom_W_deg	None	32/40
	luby(100)	29/40
	geometric(1.5,100)	32/40

The results clearly show that the best configurations of the search annotation is with the use of `indomain_min` as search domain heuristic. While switching the search strategy doesn't seem to influence too much the performances of the model, in fact we manage to solve 32/40 instances in all the three tested cases (`input_order, first_fail` and `dom_w_deg`). Moreover we can say that the use of the restart strategies doesn't allow us to improve our results, on the contrary in some cases using them led us to solve less instances.
We tried also to switch from the `Gecode` solver to the `Chuffed` solver in order to test if this change can influence the performances of our model but we didn't get any improvement.
In the end we can say that finding the best configuration of all these parameters is very problem dependent and in most cases can be done only using a trial and error approach.

3.5.1 Rotation

We followed the same approach mentioned before while dealing with the formulation of the problem that includes the possible rotation of the circuits. In the following tables there is a comparison of the results of our model using `indomain_min` as domain search strategy:

Search strategy	Restart Strategy	Solved Instances
input_order	None	17/40
	luby(100)	15/40
	geometric(1.5,100)	15/40
first_fail	None	17/40
	luby(100)	15/40
	geometric(1.5,100)	15/40
dom_W_deg	None	17/40
	luby(100)	15/40
	geometric(1.5,100)	15/40

As we can see from the comparison, we manage to solve the same number of instances in all the three configurations of the search strategy; furthermore we can observe that in all these cases the restart strategies did not allow us to obtain better outcomes.
We performed several tests also using `indomain_random` as domain search strategy but we noticed a marked worsening of the results, with only 9 instances solved out of 40. Moreover we tried inserting into the model also the `bool_search(<variables>, <varchoice>, <constrainchoice>)` annotation in order to perform the search process into the domain of the boolean array *rotations* but this change didn't lead us to further improvements.

3.5.2 Comparison

In general, we can notice that our best model for the general case manage to solve many more instances with respect to the model implemented for dealing with rotation (32/40 vs 17/40). This is not a surprise, indeed the rotation formulation involves more variables and in general we have a bigger search space. In the following graph we can see a comparison of the performances of the two best models with and without rotation.

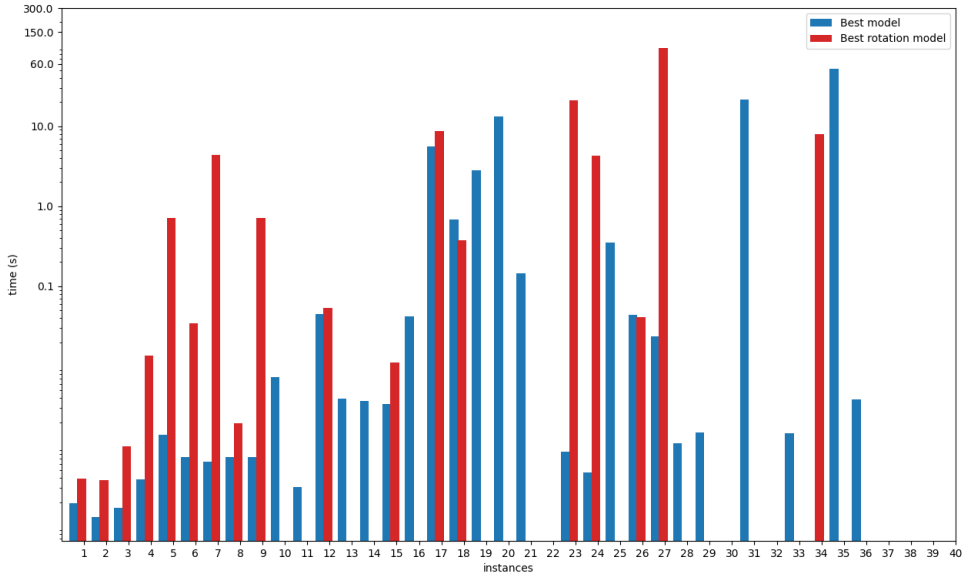


Figure 1: Comparison between CP standard and CP with rotation

4 SMT

Satisfiability modulo theories (SMT) is the problem of determining whether a mathematical formula is satisfiable. It generalizes the Boolean satisfiability problem (SAT) to more complex formulas in order to improve expressivity and scalability.
SMT solvers are tools which aim to solve the SMT problems; for this project we used the Python implementation of the Z3 solver [8].
In the following section we describe an SMT encoding to solve the VLSI problem.

4.1 Variables

We decided to encode the VLSI problem using more or less the same variables that we used in our CP model.

4.1.1 Decision variables

First we define the model's decision variables:

- x and y: are two `IntVector` containing respectively the horizontal and vertical coordinates of the bottom left corner of each circuit
- $plate\_height$: is an `IntValue` representing the actual height of the plate

4.1.2 Input variables

Then the following input variables were defined:

- n: number of circuits to place inside the plate
- w: width of the plate
- $widths$ and $heights$: are two arrays containing the width and the height of each circuit

4.1.3 Other variables

- *min_height* and *max_height*: values corresponding to the lower and upper bound for the plate height
- *big_circuit_idx*: represent the index of the circuit with the largest area

4.1.4 Objective function

Lastly we define the objective function simply as the minimization of the decision variable *plate_height* through the method `minimize()` of the Z3py `Optimize()` class.

4.2 Constraints

4.2.1 Reducing variables domain

For each decision variable we defined a specific domain:

- *plate_height*: its domain must be included into the range composed by the height of the highest rectangle as lower bound, and the sum of the heights of all the circuits as upper bound.

$$plate\_height \geq max(heights) \ \wedge \ plate\_height \leq \sum_{i \in \{1..n\}} heights_i$$

- x: for what concerns the horizontal coordinate of the bottom left corner of each circuit, it cannot assume a value greater than the width of the plate minus the width of the less wide rectangle; otherwise it will fall outside the board.

$$\bigwedge_{i \in \{1..n\}} x_i \geq 0 \ \wedge \ x_i \leq w - min(widths)$$

- y: the same holds for the vertical coordinate that cannot be greater than the upper bound of the *plate_height* minus the height of the shortest rectangle.

$$\bigwedge_{i \in \{1..n\}} y_i \geq 0 \ \wedge \ y_i \leq \sum_{j \in \{1..n\}} heights_j - min(heights)$$

4.2.2 Main constraints

- We deployed these constraints in order to guarantee that each circuit can't fall outside the perimeter; this can be encoded in the following way:

$$\bigwedge_{i \in \{1..n\}} x_i \leq w - widths_i$$

$$\bigwedge_{i \in \{1..n\}} y_i \leq plate\_height - heights_i$$

- Instead, with the constraint below we impose a no-overlapping restriction between each pair of circuits:

$$\bigwedge_{i,j \in \{1..n\}, i \neq j} x_i + widths_i \leq x_j \vee x_j + widths_j \leq x_i \vee y_i + heights_i \leq y_j \vee y_j + heights_j \leq y_i$$

4.2.3 Implied constraints

We inserted into the model a cumulative constraint in order to encode the following property: if we draw a horizontal line and sum the horizontal sides of the traversed circuits, the sum can be at most equal to the plate width w (a similar statements holds if we draw a vertical line).
A set of tasks given by start time S_i (x or y coordinates), duration D_i (the respective width or height) and resource requirements R_i (the other dimension), never require more than a global resource bound B (either the width or the height of the plate) anytime.

$$\forall u \in widths \sum_{i \in \{1..n\} | x_i \leq u < x_i + widths_i} heights_i \leq plate\_height$$

$$\forall u \in heights \sum_{i \in \{1..n\} | y_i \leq u < y_i + heights_i} widths_i \leq w$$

4.2.4 Symmetry breaking constraints

Aiming to speed up and improving the efficiency of our model through a reduction of the search space of the decision variables, we defined the following symmetry breaking constraints:

- We placed the bottom left corner of the circuit with the largest area into the origin of the plate:

$$x_{\ big\_circuit\_idx} = 0 \ \wedge \ y_{\ big\_circuit\_idx} = 0$$

- We applied an ordering constraint between the rectangles of same size:

$$\bigwedge_{i,j \in \{1..n\}, i \neq j} (heights_i = heights_j \wedge widths_i = widths_j) \rightarrow (x_i \leq x_j \wedge (x_i = x_j \rightarrow y_i \leq y_j))$$

4.3 Rotation

For handling the rotation property we applied the some changes to our model:

- As already done in the CP formulation, we defined a new array of boolean variables (`BoolVector`) called *rotations*, that is useful to keep track of the rotation of each circuit present in the plate. Concerning the circuit i , we swap the dimensions $widths_i$ and $heights_i$ depending on the value of $rotations_i$.

$$\bigwedge_{i \in \{1..n\}} (rotations_i = true) \rightarrow (widths_i = heights_i) \ \wedge \ (heights_i = widths_i)$$

- We added also the following symmetry breaking constraint in order to avoid the rotation of the squared circuits that would not change the solution obtained.

$$\bigwedge_{i \in \{1..n\}} (widths_i = heights_i) \rightarrow (rotations_i = false)$$

4.4 Experiments and Results

Fist of all we implemented a good baseline, then we improved this model executing several tests aimed to improve the performances and the efficiency. At first we inserted into the model the constraints described at section 4.2 in order to reduce the domain of the variables and for avoiding the circuits to fall outside the perimeter. With this simple model we manage to solve 24 instances out of 40. Then we added the symmetry breaking constraints defined in section 4.2.4 and the sorting of the circuits by decreasing value of their area; inserting this changes we succeeded in solving 25/40 instances.
In the end we obtained our best results by adding the cumulative constraint described in section 4.2.3; this final model was able to solve 26 instances out of 40.

4.4.1 Rotation

Instead for what concerns the formulation of the problem that allows the rotation of the circuits, after several tests we manage to solve 13/40 instances with our best encoding.

4.4.2 Comparison

Like happened for the CP encoding, we obtained better results with the general case, succeeding in solving 26/40 instances against the 13/40 solved with rotation model.
In the following graph we can visualize the performances of the two best models with and without rotation.

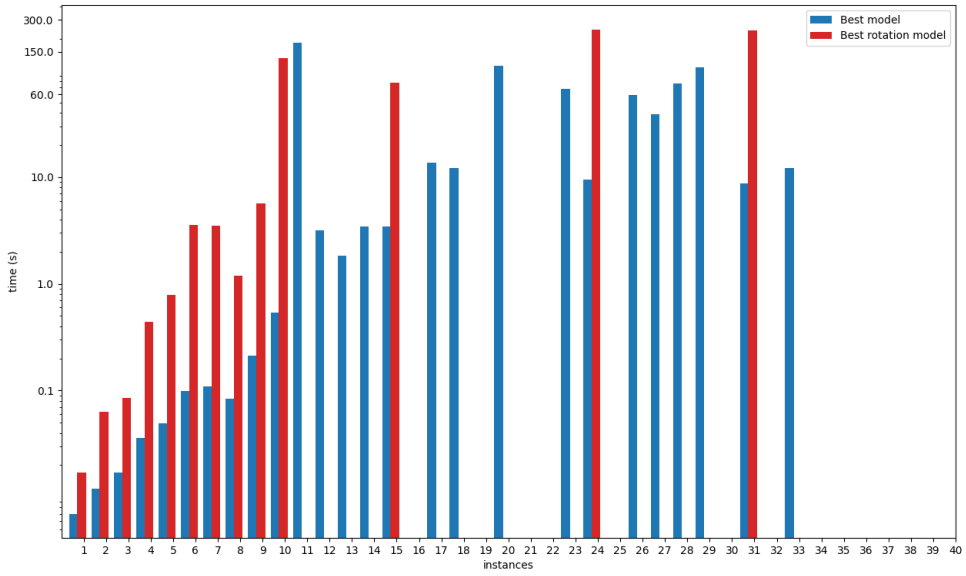


Figure 2: Comparison between SMT standard and SMT with rotation

5 Conclusions

In this project we explored two different techniques for solving a simplified version of the VLSI problem. We designed, developed and tested a CP model and a SMT encoding both for the general and the rotation formulation of this problem, reaching quite good results. In particular we investigated several methods for reducing the domain of the variables and the search space of the solver; in this way we were able to improve the results and the efficiency of our models.
As we can see carrying out a comparison of the approaches used, the CP model is the best in terms of performances, managing to solve 32/40 instances against the 26/40 of the SMT encoding. The same holds for the rotation formulation of the problem, where the CP model is able to solve 17 instances versus the 13 solved by the SMT encoding. Through the use of the following graphs we can make a comparison of the CP and SMT best models, in terms of number of instances solved and time spent for solving each instance (both for the general and the rotation formulation of the problem).

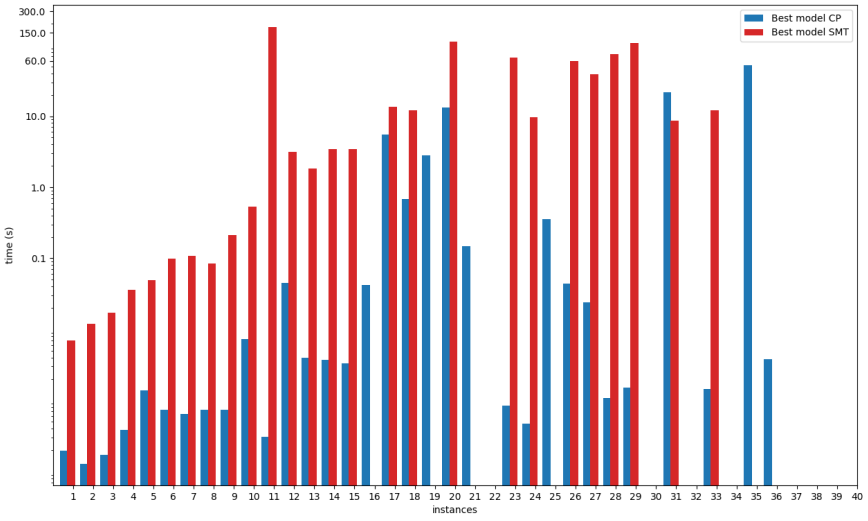


Figure 3: Comparison between CP and SMT standard

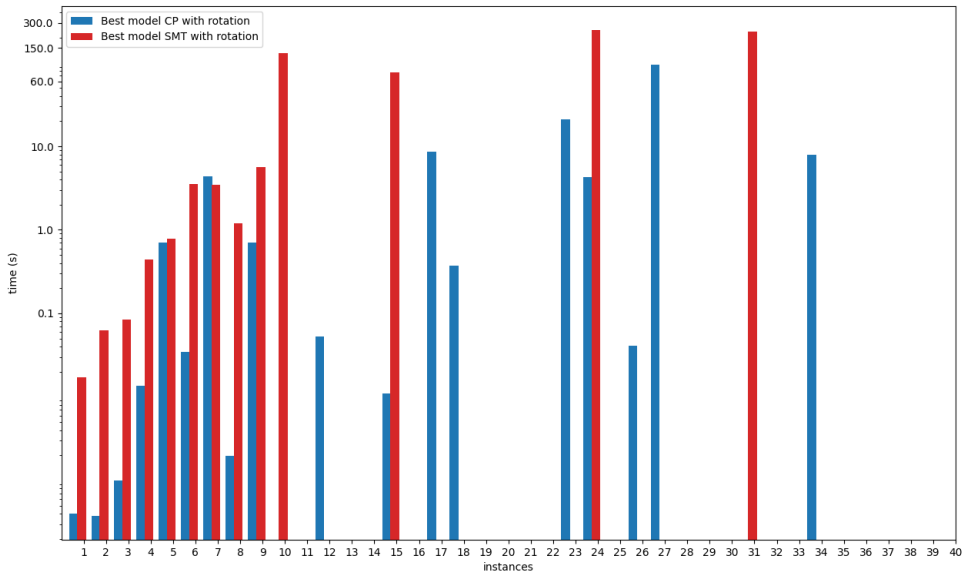


Figure 4: Comparison between CP and SMT with rotation

As we can deduce from the two graphs above, the SMT encoding employs also much more time for solving almost all instances with respect to the CP model, that turns out to be more efficient and effective.

References

[1] Optimal Rectangle Packing: An Absolute Placement Approach
Eric Huang and Richard E. Korf. 2012

[2] Symmetries in Rectangular Block-Packing
Hay-Wai Chan and Igor L. Markov.

[3] Cumulative global constraint:
https://www.minizinc.org/doc-2.5.5/en/predicates.html#cumulative

[4] Diffn global constraint:
https://www.minizinc.org/doc-2.5.3/en/lib-globals.html#packing-constraints

[5] Lexicographic global constraint:
https://www.minizinc.org/doc-2.5.3/en/lib-globals.html#lexicographic-constraints

[6] Search strategies:
https://www.minizinc.org/doc-2.5.5/en/mzn_search.html

[7] Restart strategies:
https://www.minizinc.org/doc-2.5.5/en/mzn_search.html#restart

[8] Z3Py tutorial guide:
https://ericpony.github.io/z3py-tutorial/guide-examples.htm