Combinatorial Decision Making and Optimization Project

Alessandro Lombardini - alessandr.lombardin3@studio.unibo.it
Giacomo Melacini - giacomo.melacini@studio.unibo.it
Matteo Rossi Reich - matteo.rossireich@studio.unibo.it
Lorenzo Tribuiani - lorenzo.tribuiani@studio.unibo.it

December 2022

Contents

1 Introduction

This section of the report is dedicated to an introduction over the *VLSI*[1] problem, particularly in the formulation proposed, its modelling and formalization. Moreover an overview over the main variable domains and constraints is proposed. The project has been completed in 5/6 weeks. The work has been divided among the students equally. Each one had the responsibility of developing a solution for one paradigm: Lorenzo Tribuiani dealt with CP, Alessandro Lombardini with SAT, Matteo Rossi Reich with SMT and Giacomo Melacini with MIP. The main difficulties have been to implement effective symmetry breaking constraints and to improve significantly the model after a first version was achieved.

1.1 Input parameters and variables

In order to formulate the series of mathematical constraints needed for the resolution of the problem through different *Paradigms*, a formalization of the problem into a more structured and solid mathematical model is needed.
To do so, some notation needs to be introduced, in particular, following the given formulation:

- w: The width of the plate.
- n: The number of circuits to place on the plate.
- $widths$: The list of circuits' widths. Since this element is defined as an array, $widths_i$ is referred as the width of the $i-th$ circuit.
- $heights$: The list of circuits' heights. As well as the $widths$ notation, $heights_i$ is referred as the height of the $i-th$ circuit.

Those elements represent the inputs of the problem, but are not sufficient to describe it completely. In order to have a complete description, three variables need to be introduced:

- h: Which represents the height of the plate once that all the circuits are placed. This variable represents the actual solution of the problem and it's the value to minimize.
- X: Which represents the list of all the *bottom-left* x coordinates of the circuits. Again, since this variable is defined as an array, X_i is referred as the bottom-left x coordinate of the $i-th$ circuit. This variable is used in all paradigms except SAT.
- Y: Which represents the list of all *bottom-left* y coordinates of the circuits. Again Y_i is referred as the bottom-left y coordinate of the $i-th$ circuit. As for X, also this variable is used in all paradigms except SAT.

[1] Very Large Scale Integration

All those elements are sufficient to formalize the problem, in particular what is wanted is to define a model to produce the positions X and Y of n circuits with widths $widths$ and heights $heights$ to place on a plate of width w such that the height of the plate h is minimum.

1.2 Variables domains

Since h, X and Y are the variables of the problem, a definition of the **domain** of each of them is needed and, moreover, some first *optimization* step could be taken by reducing this domain as much as possible to minimize the *Search Space*. Before starting a general assumption is needed:

Given the formulation of the problem is assumed that all the variables and values are natural numbers.

1.2.1 h

As said before the domain of this variable is simply $\mathbb{N}$. But, a *lower* and an *upper bound* must be introduced to reduce the domain:

- **Lower Bound:** Is known for sure that the height of the plate could never be smaller than the maximum circuits' height. So the lower bound could be set to:
$$min_h = max(heights) \tag{1}$$
Even though this statement is true, it is still possible to improve it by considering that:
$$min_h = max(heights) \leftrightarrow \sum_{i=1}^{n} widths_i \leq w$$
this states that $max(heights)$ is the actual minimum height reachable if and only if all the circuits could be stacked horizontally. More generally, given a set of rectangles R and a certain width w, the minimum height reachable is the one of the rectangle R' of base w and area:
$$\sum_{i=1}^{n} < S_i >$$
where $< S_i >$ is the area of the $i-th$ rectangle. So the following statement holds:
$$min_h = \frac{1}{w} \sum_{i=1}^{n} widths_i \cdot heights_i \tag{2}$$
Some modification are still needed, in particular:
  - The function 2 maps in $\mathbb{R}$, which is not acceptable. This limit could easily be exceeded by *rounding* or using an *integer division*.

– The function 2 has a minimum in 0, which, for the application of the VLSI problem, is not reasonable. Again, this limit can be exceeded by setting the minimum to be $max(heights)$

All those informations leads to the final definition of the domain as follows:

$$min_h = \begin{cases} max(heights) & if \quad \sum_n widths \leq w \\ (\sum_{i=1}^{n} whidths_i \cdot heights_i) \,\%w & else \end{cases} \tag{3}$$

where % is the integer division.

- **Upper Bound**: In this case the domain reduction is only possible by assuming that the maximum height reachable is when all the circuits are stacked together vertically, so:

$$max_h = \sum_{i=1}^{n} heights_i$$

any other restriction over the maximum height could not be applied, cause is known that min_h, in the way is defined, is bounded as follows:

$$max(heights) \leq min_h \leq \sum_{i=1}^{n} heights_i{}^2$$

so, having $max_h < \sum_{i=1}^{n} heights_i$ could imply $max_h < min_h$ which is unsatisfiable. The only step that could be taken is to apply the following reasoning:

$$max_h = \begin{cases} max(heights) & if \quad \sum_n widths \leq w \\ \sum_{i=1}^{n} heights_i & else \end{cases} \tag{4}$$

At this point we could bound the domain of h to be:

$$h \in [min_h, max_h]$$

With the following limit case:

$$min_h = max_h = max(heights) \qquad if \quad \sum_{i=1}^{n} widths_i \leq w$$

$$min_h = max_h = \sum_{i=1}^{n} heights_i \qquad if \quad widths_i = w \; \forall i \in [1, ..., n]$$

Which is coherent since, for those limit cases, the minimum and maximum coincide with the actual minimum height reachable.

[2] the maximum value is reached when all the values $widths_i$ are equal to w, so:

$$\left(\sum_{i=1}^{n} widths_i \cdot heights_i\right) \%w = \left(\sum_{i=1}^{n} w \cdot heights_i\right) \%w = \sum_{i=1}^{n} heights_i$$

1.2.2 X

Considering X as an array, it is known that each value X_i has $\mathbb{N}$ as domain. Even in this case a domain reduction is possible considering that X is the array of the *left-bottom* x coordinates of the circuits and, since all the circuits must be placed inside the plate, X_i could never be bigger than w. Moreover, a further reduction could be made considering that X_i could never be bigger the $w - min(widths)$, cause any circuits placed in the interval $(w - min(widths), w]$ would overlap the plate boundary. So:

$$0 \leq x_i \leq w - min(widths), \forall i \in [1, ..., n] \tag{5}$$

1.2.3 Y

The same domain reduction is applied to the Y array, obtaining:

$$0 \leq y_i \leq h - min(heights), \forall i \in [1, ..., n] \tag{6}$$

2 CP

Constraint Programming (CP) is a paradigm for solving *combinatorial problems* based on the declaration of *constraints* on the feasible solutions for a set of decision variables. This paradigm is implemented by various **constraint programming languages**, in particular, in the development of this particular project, **MiniZinc** constraint modelling language is used. MiniZinc is a *free and open source* constraint modelling language that allows an *high level* and *solver-indipendent* modelling for combinatorial problems.

2.1 Decision Variables

Decision variables are formalized exactly as described in the introduction section, in particular we have:

- h
- X
- Y

2.1.1 Domains

Even domains remains the same as previously described in Section 1.

2.2 Objective Function

Since the main objective is to find the smallest plate to fit all the circuits, the main *objective function* is to **minimize** the height of the plate.

2.3 Constraints

The following section is dedicated to the mathematical formalization of the fundamental constraints needed for the resolution of the VLSI problem. As mentioned before, *constraints* are the fundamental logical blocks which mainly serve two purposes:

- Create the structure for the problem resolution telling the model when a certain solution is coherent and acceptable
- Reduce the search space, as possible, in order to minimize the execution time for each instance

So, having good constraints could significantly improve the models behaviour. Given that, the following sections give a description of the constraints used.

Plate Boundary Constraints The main purpose of those constraints is to impose all the circuits to be placed inside the plate without overlapping the boundary given. Formally:

$$X_i + widths_i \leq w \quad \forall\, i \in [1, ..., n] \tag{7}$$

$$Y_i + heights_i \leq h \quad \forall\, i \in [1, ..., n] \tag{8}$$

Basically, those constraints impose the *right-bottom* corner of each circuits to be inside the plate. The *left-bottom* corner of each circuits needs no constraint since, by definition, is already constrained to be in $[(0,0), ..., (w - min(widths), h - min(heights))]$ (5) (6).
One more improvement could be done to this constraint considering that only the circuits placed on the boundary needs to be checked, Formally:

$$R_x = |X_1 + widths_1, X_2 + widths_2, ..., X_n + widths_n|$$

$$R_y = |Y_1 + heights_1, Y_2 + heights_2, ..., Y_n + heights_n|$$

$$max(R_x) \leq w \tag{9}$$

$$max(R_y) \leq h \tag{10}$$

Non Overlapping Constraint The *non overlapping constraint* imposes all the circuits to be placed inside the plate in such a way that none of them overlaps any other circuit. Formally we have:

$$\begin{aligned} &(X_i + widths_i \leq X_j \ \vee \ X_i \geq X_j + widths_j) \wedge \\ &\wedge (Y_i + heights_i \leq Y_j \ \vee \ Y_i \geq Y_j + heigths_j), \\ &\qquad\qquad \forall \ \ i, j \in [1, ..., n] \end{aligned} \tag{11}$$

Implied constraints Constraints (9), (10) and (11) are sufficient to implement a satisfiable model which produces the wanted solutions. Although, is still possible introduce some other constraints to optimize the model itself in order to reduce the execution time, in particular:

Cumulative constraints The *Cumulative Scheduling Constraint* or *Scheduling under resource constraint* is a constraint often used in scheduling problems. This constraint imposes that, given a set of tasks T, each described by a starting point s, a duration d and a resource consumption r, at each point in time the cumulated resources of the set of tasks that overlaps that point to be strictly smaller than a certain value R imposed. So, it's possible to use this constraint considering that:

- the starting point s is the X coordinate of each circuits (Y coordinate for width constraint).
- the duration d is the *width* of each circuits (*height* for width constraint).
- the resource usage r is the *height* of each circuit (*width* for width constraint).
- the resource usage limit R is the height of the plate h (w for width constraint).

so, Formally:

$$\sum_{i|X_i \leq u \leq X_i + widths_i} heights_i \leq h \quad \forall\ u\ in\ widths \tag{12}$$

$$\sum_{i|Y_i \leq u \leq Y_i + heights_i} widths_i \leq w \quad \forall\ u\ in\ heights \tag{13}$$

Simmetry breaking constraints *Symmetry Breaking Constraint* are generally introduced to avoid the model to explore symmetrical solutions. Those constraint are specific of the problem and their formalization depends directly on that.

Symmetric mapping function In order to introduce a *symmetric breaking constraint* a formalization of the symmetric solution is needed. In this particular formulation of the VLSI problem the two main symmetries are the *horizontal* and *vertical* ones. Given the X positions and a symmetry axis, which in this case is $w/2$ ($h/2$ for the *vertical* symmetry), is known from the basic algebra that:

$$X'_i = -X_i + 2\frac{w}{2} = -X_i + w$$

But, this symmetry equation is specular, this means that the origin is inverted and, in the new coordinates, the *right-bottom* corner is the new origin. In order to prevent that a transformation is applied:

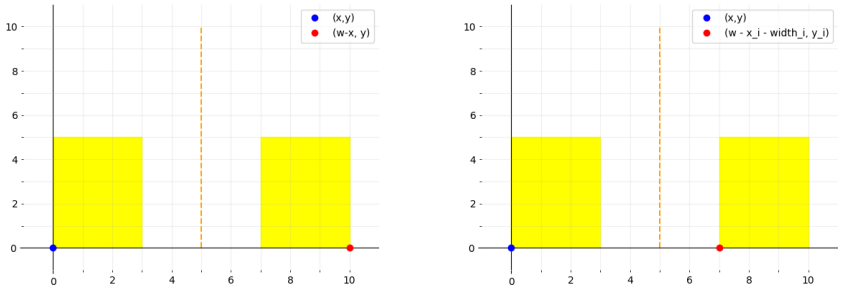
$$X'_i = w - X_i - widths_i$$

In this way the *left-bottom* indexing is restored. A this point is enough to impose the following *lexicographic symmetry constraint*:

$$X' = [w - X_1 - widths_1, ..., w - X_n - widths_n]$$

$$X \overset{\text{lex}}{\leq} X' \tag{14}$$

where $\overset{\text{lex}}{\leq}$ stands for *lexicographically minor or equal*.



(a) Specular symmetry (b) Non specular symmetry

Figure 1: Comparison of orgin point mapping for the two functions

2.4 Rotation

All the constraints introduced until now have no care about rotation. In order to preserve the usability of the models in case circuits' rotation is allowed, a new variable needs to be introduced: *rotation*. This variable is defined as an *array* of boolean values and $rotation_i$ describe whether or not the $i-th$ circuits has been rotated. In addiction to this variable a rule that put in relationship this variable with the height and the width of the circuits needs to be introducted, in particular:

$$widths^a = [widths_i \cdot (1 - rotations_i) + heights_i \cdot rotations_i, ...] \quad \forall\, i \ \in [1, ..., n]$$

$$heights^a = [heights_i \cdot (1 - rotations_i) + widths_i \cdot rotations_i, ...] \quad \forall\, i \ \in [1, ..., n]$$

So, $widths^a$ and $heights^a$ represents the new *widths* and *heights* array. Based on the value of $rotation_i$ the values of $widths_i$ and $heights_i$ could be swapped in $widths_i^a$ and $heights_i^a$.

2.5 Validation

The main models used are now presented.

Standard Model (STD) The standard model implements only the necessary constraints needed to produce feasible solutions. The constraint implemented are (9), (10) and (11). Clearly the simplicity of the model will affect its performance, but still is presented as a reference point for better understanding the impact of implicit constraint.

Cumulative Model (CML) the cumulative model implements, over the constraint already implemented in the **standard model**, the constraint (12) and (13)

Symmetry Breaking Model (SYB) The symmetry breaking model implements the same constraints implmented by the **cumulative model** and the constraint (14)

Experimental Design All the models have been implemented in **Minizinc** using *Minizinc to Flatzinc converter* version *2.6.4* and executed trough python script using the library *minizinc-python* version *0.7.0*. All the models have been executed with three different solvers:

- *Gecode 6.3.0*
- *Chuffed 0.10.4*
- *Or-tools 9.3*

on the same machine with the following hardware:

- Processor: *11th Gen Intel(R) Core(TM) i5 1135G7 @ 2.40 Ghz*
- RAM: *8 Gb*

Search Strategies Minizinc offers different possible search strategies according to the type of problem it has to deal with. After some test, since the problem stands on array search, a **sequential search** has been chosen together with an **integer search** (according to the assumption made before) for each element.

Experimental results

No rotation The following sub-paragraph is dedicated to the analysis of the result of the models over different solvers without rotation. The Table (1) reports the overall solution of the models, as is possible to see *Gecode* solver seems to be the worst performing with only 3 optimal solution for the **STD** model and different *non-optimal* solution for the **CML** and **SYB** model.

ID	STD Chuffed	CML Chuffed	SYB Chuffed	STD Gecode	CML Gecode	SYB Gecode	STD Or-T.	CML Or-T.	SYB Or-T.
1	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	N|A	**12**	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	N|A	**13**	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	N|A	**14**	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	N|A	**15**	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	N|A	**16**	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	N|A	**17**	**17**	**17**	**17**	**17**
11	**18**	**18**	**18**	N|A	19	19	**18**	**18**	**18**
12	**19**	**19**	**19**	N|A	**19**	**19**	**19**	**19**	**19**
13	**20**	**20**	**20**	N|A	**20**	**20**	**20**	**20**	**20**
14	**21**	**21**	**21**	N|A	**21**	**21**	**21**	**21**	**21**
15	**22**	**22**	**22**	N|A	**22**	**22**	**22**	**22**	**22**
16	**23**	**23**	**23**	N|A	24	24	**23**	**23**	**23**
17	**24**	**24**	**24**	N|A	**24**	32	**24**	**24**	**24**
18	**25**	**25**	**25**	N|A	**25**	**25**	**25**	**25**	**25**
19	**26**	**26**	**26**	N|A	27	27	**26**	**26**	**26**
20	**27**	**27**	**27**	N|A	**27**	N|A	**27**	**27**	**27**
21	**28**	**28**	**28**	N|A	**28**	**28**	**28**	**28**	**28**
22	30	**29**	**29**	N|A	30	30	**29**	**29**	**29**
23	31	**30**	**30**	N|A	31	31	**30**	**30**	**30**
24	**31**	**31**	**31**	N|A	**31**	**31**	**31**	**31**	**31**
25	34	**32**	**32**	N|A	35	35	**32**	33	**32**
26	**33**	**33**	**33**	N|A	**33**	34	**33**	**33**	**33**
27	**34**	**34**	**34**	N|A	**34**	42	**34**	**34**	**34**
28	**35**	**35**	**35**	N|A	36	36	**35**	**35**	**35**
29	**36**	**36**	**36**	N|A	N|A	N|A	**36**	**36**	**36**
30	38	**37**	**37**	N|A	38	38	**37**	38	38
31	**38**	**38**	**38**	N|A	40	52	**38**	**38**	**38**
32	40	**39**	40	N|A	42	40	41	40	40
33	**40**	**40**	**40**	N|A	42	42	**40**	**40**	**40**
34	**40**	**40**	**40**	N|A	41	41	41	**40**	**40**
35	**40**	**40**	**40**	N|A	41	41	**40**	**40**	**40**
36	**40**	**40**	**40**	N|A	41	41	**40**	**40**	**40**
37	61	**60**	**60**	N|A	66	62	61	61	**60**
38	61	**60**	**60**	N|A	61	61	61	61	61
39	61	**60**	**60**	N|A	61	61	**60**	**60**	**60**
40	92	92	92	N|A	N|A	N|A	92	92	92

Table 1: Best result obtained by each model with the three different solvers

Speaking about time comparisons the following graphs ((3), (2) and (4)) shows the different time execution of each model with respect to a specific solver.

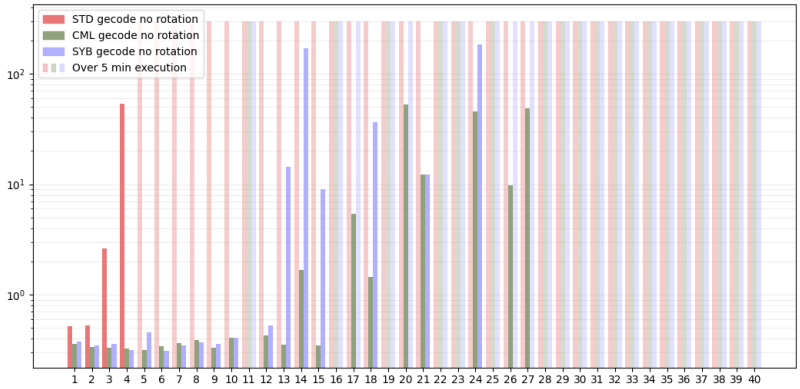


Figure 2: Time comparison over models with *Gecode* solver

As expected, *Gecode* solver has the highest time executions, over five minutes, leading the timeout to stop the execution returning a *non-optimal* solution or no solution at all. The **STD** model seems to be the most affected, which is reasonable given the simplicity of the model, but even **CML** and **SYB** are strongly affected with no optimal solution at all after the 28-th instance.
Even though this graph leads not much information over the constraints influence over the resolution of the problem is still visible the impact of the cumulative constraint leading the executions to be smaller over more instances

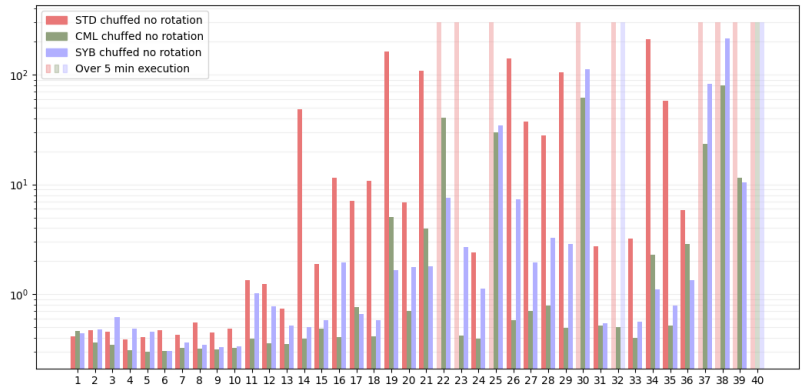


Figure 3: Time comparison over models with *Chuffed* solver

Chuffed solver severely decrease the execution time over the models bringing the **STD** to compute the great majority of instances optimally. **CML** with

chuffed solver founds the optimal solution for 39 instances within five minutes and same is for the **SYB** model, but with more time effort.
In this case is visible the impact of the cumulative constraint. **CML** and **SYB** (implementing it) have better execution time over almost all the instances with optimal solution found over the great majority of instances. *Symmetry breaking constraint* doesn't seem to be really affecting the time execution. Even tough the **SYB** model is performing better than the **CML** model on some instances it has a stronger degradation of times on others that leads the overall benefit to be negative.

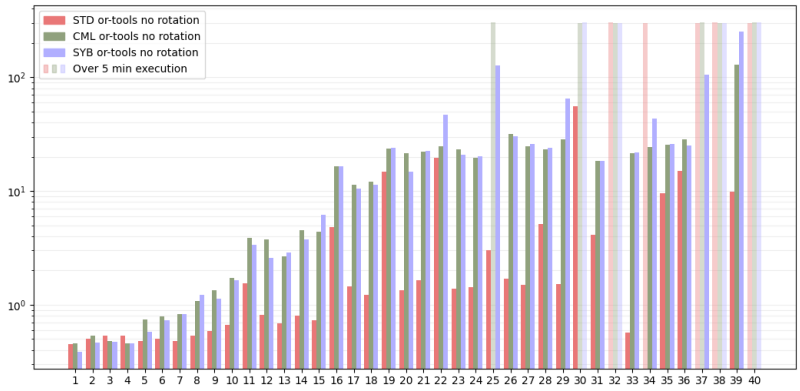


Figure 4: Time comparison over models with *Or-Tools* solver

Particular interest is given to the *Or-Tools* solver. This solver has the greatest impact on the **STD** model bringing down it's time execution and finding 35 solutions optimally. **CML** and **SYB** has not a real benefit with respect to *Chuffed* solver, but the behaviour seems to be more linear with respect to the increasing difficulty of the problem with less time peaks, but in general more time is required on each instance.
Or-Tools solver and *Chuffed* solver are the best performing one with, respectively, 36 optimal solution for the **STD** model and 31, 34 and 39 optimal solutions for the **CML** model, 36 and 38 optimal solution for the **SYB** model. In general, **CML** model with *Chuffed* solver seems to be the best performing combination even tough the impact of *Or-tools* solver over the *STD* model remains interesting.

Rotation This sub-paragraph is dedicated to the analysis of the models over different solvers with rotation enabled. Clearly, enabling rotation open many more solution and more search space to explore leading to a more complex and long execution. Again, table (2) report the results obtained by each model with each solver.

ID	STD Chuffed	CML Chuffed	SYB Chuffed	STD Gecode	CML Gecode	SYB Gecode	STD Or-T.	CML Or-T.	SYB Or-T.
1	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	N|A	**12**	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	N|A	**13**	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	N|A	**14**	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	N|A	**15**	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	N|A	**16**	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	N|A	**17**	**17**	**17**	**17**	**17**
11	**18**	**18**	**18**	N|A	N|A	N|A	**18**	**18**	**18**
12	**19**	**19**	**19**	N|A	N|A	N|A	**19**	**19**	**19**
13	**20**	**20**	**20**	N|A	21	21	**20**	**20**	**20**
14	**21**	**21**	**21**	N|A	22	22	**21**	**21**	**21**
15	**22**	**22**	**22**	N|A	N|A	N|A	**22**	**22**	**22**
16	**23**	**23**	**23**	N|A	25	25	24	**23**	**23**
17	**24**	**24**	**24**	N|A	25	25	**24**	**24**	**24**
18	**25**	**25**	**25**	N|A	N|A	N|A	**25**	26	26
19	**26**	**26**	**26**	N|A	27	27	27	**26**	27
20	**27**	**27**	**27**	N|A	30	29	28	**27**	28
21	29	**28**	**28**	N|A	N|A	N|A	29	29	29
22	30	**29**	30	N|A	N|A	N|A	30	30	30
23	**30**	**30**	**30**	N|A	32	32	**30**	**30**	31
24	**31**	**31**	**31**	N|A	34	34	**31**	**31**	**31**
25	33	**32**	33	N|A	44	44	33	33	33
26	**33**	**33**	**33**	N|A	39	39	**33**	**33**	**33**
27	**34**	**34**	**34**	N|A	37	37	**34**	**34**	**34**
28	**35**	**35**	**35**	N|A	38	38	**35**	**35**	36
29	**36**	**36**	**36**	N|A	39	39	**36**	**36**	**36**
30	38	38	38	N|A	N|A	39	38	38	38
31	**38**	**38**	**38**	N|A	N|A	N|A	**38**	**38**	**38**
32	40	40	40	N|A	48	48	40	40	40
33	**40**	**40**	**40**	N|A	43	43	**40**	**40**	**40**
34	**40**	**40**	**40**	N|A	N|A	45	**40**	**40**	**40**
35	**40**	**40**	**40**	N|A	45	44	**40**	**40**	**40**
36	**40**	**40**	**40**	N|A	41	41	**40**	**40**	**40**
37	61	61	61	N|A	N|A	N|A	61	61	61
38	61	**60**	61	N|A	N|A	N|A	61	61	61
39	61	**60**	**60**	N|A	66	62	61	61	61
40	93	92	92	N|A	N|A	N|A	92	230	292

Table 2: Best result obtained by each model with the three different solvers, rotation is enabled

Again, *Gecode* seems to be the worst performing over the three solvers, but even *Or-Tools* have had a reduction of the total optimal solution found.

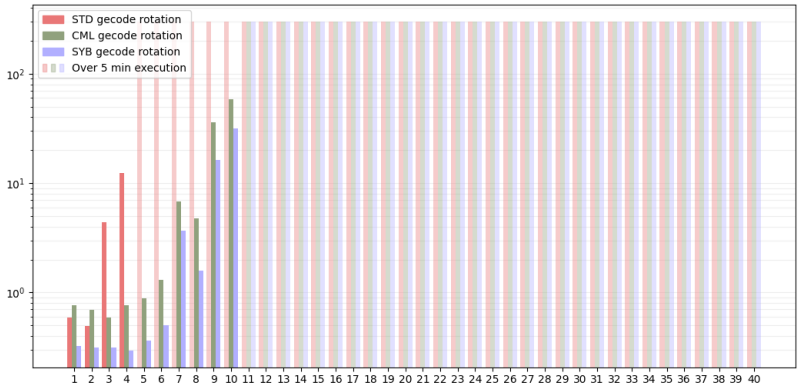


Figure 5: Time comparison over models with *Gecode* solver, rotation is enabled

As can be seen in figure (5) time execution have severely increased. *Gecode* solver has no model computing solution within five minutes just from the 11-th instance.

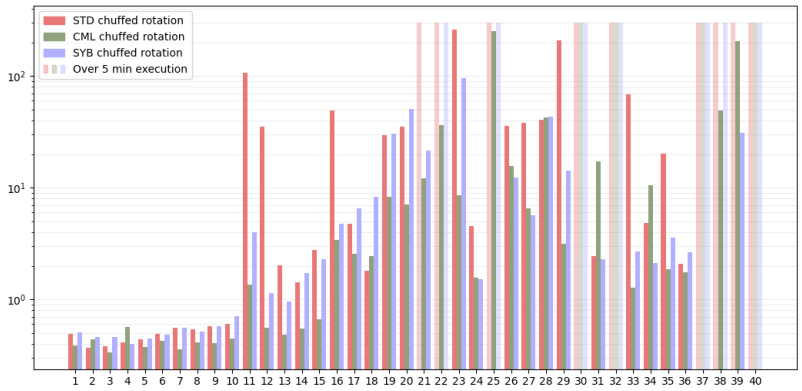


Figure 6: Time comparison over models with *Chuffed* solver, rotation is enabled

Chuffed solver's time execution has not increased in a several way, as can be seen in figure (6). Still, *Chuffed* is capable of finding optimal solutions over the great majority of instances within five minutes. Clearly, time has increased, but not enough to make the models's execution be stopped leading to non-optimal solution.

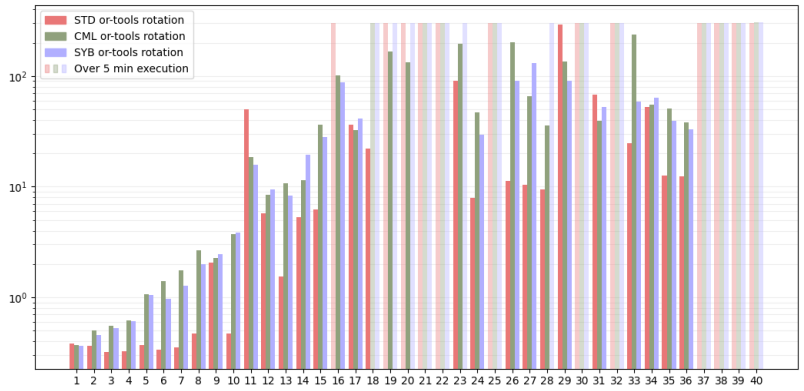


Figure 7: Time comparison over models with *Or-Tools* solver, rotation is enabled

Again *Or-Tools* seems to be working better with the **STD** model. Time has encreased, but the **STD** model is the best performing one over the three.

3 SAT Model

A **SAT solver** is a computer program whose purpose is to solve the **Boolean satisfiability problem** (abbreviated **SAT**), that consists in determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it checks whether the variables of a given formula can be consistently replaced by `true` or `false` in such a way that it is evaluated as `true`. If this is the case, the formula is called *satisfiable*, otherwise, if no such assignment exists, it is called *unsatisfiable*.

The aim of this section is to represent the VLSI problem as a propositional logic formula in order to find a solution to it using a SAT solver.

3.1 Decision variables

The problem has first of all to be modelled with literals. In particular, VLSI has been represented with the 3-Dimensional boolean array P of size $H * W * K$, where H is the height of the plate, W is the width and K is the number of circuits to place. A literal can be referred as $P_{h,w,k}$, and it defines if at the coordinates (h, w) of the plate is placed the k circuit. P is composed by the following set of literals:

$$\{P_{h,w,k} \ \in B\} \ \ \forall_{h \in 0...H-1} \ \ \forall_{w \in 0...W-1} \ \ \forall_{k \in 0...K-1}$$

3.2 Objective function

To do optimization, plates of different heights are tried iteratively. In particular, it is initially defined a model with a height equal to the lower bound, which is described in 1.2. If the model is unsatisfiable, the height is increased by one, and the model so expanded: this is done until a satisfiable one is reached. Since the algorithm tries all the possible heights, from the lower bound to the upper bound, and since the lowest value coincides with or is smaller than the optimal one, it is certain that the solution that first is evaluated as satisfiable will be optimal.

3.3 Constraints

The cardinality constraints are the ones in the form $p_1 + ... + p_n \ \begin{matrix} \leq \\ = \\ \geq \end{matrix} \ k$. In SAT different encodings of these constraints can be very different with big instances, so it is useful to test those available in order to decide which of them to use. For this solution, they have been considered four different encodings:

- Pairwise encoding
- Sequential encoding
- Bitwise encoding

- Heule encoding

The above constraints can be named as `at_most_k`, `at_least_k` and `exactly_k` respectively. Anyway, in this particular context, they have been considered and used only constraints with `k` equal to 1. The four encoding variants have been compared, and the `Bitwise` one has turned out to be the best (see paragraph 3.5). Here below are shown the Bitwise encodings of the three cardinality constraints cited before:

$$at\_least\_one([x_1, x_2, ..., x_n]) = \bigvee_{x \in [x_1, x_2, ..., x_n]} x$$

$$at\_most\_one([x_1, x_2, ..., x_n]) = \bigwedge_{1 \leq i \leq n} x_i \rightarrow (r_1 = b_{i,1} \wedge r_2 = b_{i,2} \wedge ... \wedge r_m = b_{i,m})$$

$$= \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} \bar{x}_i \vee r_j[\bar{r}_j]$$

* The Bitwise encoding of the AtmostOne constraint introduce m new variables r_i where $m = log_2(n)$. For $1 \leq i \leq n$ let $b_{i,1}, ..., b_{i,m}$ be the binary encoding of $i - 1$.

$$exactly\_one([x_1, x_2, ..., x_n]) = at\_most\_one([x_1, x_2, ..., x_n]) \wedge at\_least\_one([x_1, x_2, ..., x_n])$$

They can be also defined `all_true` and `all_false` constraints as follows:

$$all\_true([x_1, x_2, ..., x_n]) = \bigwedge_{1 \leq i \leq n} x_i$$

$$all\_false([x_1, x_2, ..., x_n]) = \bigwedge_{1 \leq i \leq n} \neg x_i$$

Now, through these constraints, more complex ones about the problem itself can finally be defined. First, let's define few terms which will be useful in the next paragraphs. A `block` represents the fundamental unit that makes up the plate and it is defined as the following set of literals:

$$block_{h,w} = \{P_{h,w,k}\} \quad k \in [0, K-1]$$

In particular, the plate is composed of $W * H$ blocks, and each block is composed of `k` literals. It can be now cited also the term `place`, which represents a potential area that a circuit `k` can occupy. It is defined as a set of contiguous blocks, whose size and shape of all of them together is equal to the ones of the circuit itself. To be precise, they are not taken the entire blocks, but just their literals associated with the `k` circuit. The set of literals that represent a place occupied by a circuit `k` with the bottom left corner in (h, w) are listed as follows:

$$place_{h,w,k} = \{P_{h+i,w+j,k}\} \quad i \in [0, H_k - 1], j \in [0, W_k - 1]$$

where H_k and W_k are respectively the height and the width of the `k` circuit. Now they can be finally defined the constraints to solve the VLSI problem.

Placing constraint This constraint allows to place the circuits on the plate: they have to be defined all the possible places that each circuit can occupy, and impose that only one position has to be chosen for each. The overlapping problem will be solved later with another constraint (see Paragraph 3.3). To define a possible position of a circuit is not enough to require that all the literals of a `place`, as it has been defined before, have to be true. It is also necessary to specify that all the other literals of the plate associated with the same circuit have to be false. It is called O_k the set of all the literals of the plate that the circuit k can potentially occupy, and it is defined as follows:

$$O_k = \{P_{i,j,k}\} \;\; i \in [0, H-1],\; j \in [0, W-1]$$

The literals which have to be false instead, as explained above, coincide with the following set:

$$not\_place_{h,w,k} = O_k - place_{h,w,k}$$

In the solution so, a circuit `k` occupies a specific place if the conjunction of the variables contained in $place_{h,w,k}$ is evaluated as `true` and if the conjunction of the negation of the variables contained in $not\_place_{h,w,k}$ is evaluated as `true` too. This concept is encoded into `place_check`.

$$place\_check_{h,w,k} = all\_true(place_{h,w,k}) \wedge all\_false(not\_place_{h,w,k})$$

Because a circuit can be placed on all the surface of the plate, they have to be encoded all the possible places that a circuit can occupy.

$$places\_check_k = \{place\_check_{i,j,k}\} \;\; i \in [0, H - H_k],\; j \in [0, W - W_k]$$

For each circuit, exactly one *place_check* contained in *places_check* should be evaluated as true in the solution. It is in this way defined the placing constraint.

$$\bigwedge_{\forall k} exactly\_one(places\_check_k)$$

The same constraint can be also defined requiring that only the literals that describe the border of the circuit have to be true, instead of those of its entire area. In particular, it can be defined as a new set of literals in substitution of `place`, which will be called `placeBorder`.

$$\begin{aligned} placeBorder_{h,w,k} = {} & \{P_{h,w+j,k}\} \;\; j \in [0, W_k - 1] + \\ & \{P_{h+H_k-1,w+j,k}\} \;\; j \in [0, W_k - 1] + \\ & \{P_{h+1,w,k}\} \;\; i \in [1, H_k - 2] + \\ & \{P_{h+1,w+W_k-1,k}\} \;\; i \in [1, H_k - 2] \end{aligned}$$

It contains the literals which represent the border of the circuit `k` when it has its left corner at the coordinates (h, w). Now let's define the corresponding *not_place* set of this variant, which will be called `not_placeBorder`. It is different from the one before because all the literals of the blocks inside the border have to be false (otherwise there could be cases in which some circuits are placed inside other ones).

$$\begin{aligned} not\_placeBorder_{h,w,k} = P - placeBorder_{h,w,k} + \\ \{block_{h+i,w+j,k}\} \quad i \in [1, H_k - 2], \\ w \in [1, W_k - 2], \\ k \in \{0...K-1\} - \{k\} \end{aligned}$$

It can be so defined another model just substituting, in the previous description of the placing constraint, the `place` set with the `placeBorder` one and `not_place` with `not_placeBorder`. This is enough to create another model, very similar to the first one, but which allows getting different performances as it will be shown in Chapter 3.5.

Overlapping constraint It is defined a constraint whose aim is to verify that the circuits do not overlap each other. It checks that at most one literal of each block is evaluated as true, in this way different circuits can not occupy the same area.

$$\bigwedge_{i=0,\ j=0}^{H-1,W-1} exactly\_one(block_{i,j})$$

This constraint, together with the previous one, allows to completely solve the VLSI problem.

Symmetry breaking constraints The two models that can be defined with the previous constraints and literals are subjects of three possible types of symmetries:

- Vertical symmetry
- Horizontal symmetry
- Vertical and horizontal symmetry (180° rotation)

In order to try to improve the models' performances it is added a symmetry-breaking constraint whose aim is to put the highest circuit always in the bottom-left corner of the plate. They are in this way avoided all those configurations in which that particular circuit is somewhere else, so all the symmetries that include it. This shouldn't stop the algorithm from reaching optimality.

3.4 Rotation

As explained in the introduction of the problem the height and length of each circuit can be swapped if the rotation is allowed. To manage this aspect the representation of the plate and the constraints have to be slightly expanded. This solution requires first of all a new array of literals called `rotated`, whose aim is to express if the circuits have been placed rotated or not. It corresponds to the following set:

$$\{rotated_k \in B\} \quad \forall_{k \in 1 \dots K}$$

Then, it has to be changed the `placing` constraint to make it able to manage also the rotated version of the circuits: they have to be defined the rotated places, and they have to be managed together with the previous ones in order to be able to place each circuit both rotated and not. A place for the k circuit rotated with the left corner in (h, w) is defined as follows:

$$rotated\_place_{h,w,k} = \{P_{h+j,w+i,k}\} \quad i \in [0, H_k - 1], j \in [0, W_k - 1]$$

Then it has to be computed the set `rotated_place_check` for that specific circuit, and this can be done in the exact same way it has been done before for *place_check*, just substituting *place* with *rotated_place*. Following the same pipeline of the previous case, it has also to be defined `rotated_places_check` with the already seen *places_check*. The new placing constraint can be defined as follows:

$$\bigwedge_{\forall k} exactly\_one((exactly\_one(places\_check_k) \wedge \neg rotated_k), \\ (exactly\_one(rotated\_places\_check_k) \wedge rotated_k))$$

It requires that exactly one place for each circuit is activated, and based on if it is rotated or not, the corresponding $rotated_k$ variable has to be true or false.

3.5 Validation

Experimental Design The modelling process has been done in this case using boolean variables and logical operators provided by the python library `Z3Py` (version 0.2.0), which is based on `Z3`, a high-performance theorem prover developed at Microsoft Research. The results have been computed with the following hardware:

- Processor: *11th Gen Intel(R) Core(TM) i7-1165G7 2.80GHz*
- RAM: *32,0 GB*

As explained in Paragraph 3.2, the height of the plate is increased iteratively until a satisfiable model is reached. In order to avoid creating all the models at each iteration, the project has been built in a way that instead of throwing it away every time, it is just expanded with the new literals and constraints. This has allowed increasing the performance of the optimization process.

Experimental results The first result which is useful to analyze is the one related to the use of the different encodings of the cardinality constraints. Table 3 shows the results of SATModel with each of them. It can be seen that they are not so much different from each other; anyway, in past executions, the `Bitswise` encoding has given the best results in most of the cases. In particular, it has been able to solve a few instances more than others; for this reason, it has been chosen as a predefined one. Figure 8 shows the times done on the instances.

ID	SATModel$_{seq}$	SATModel$_{np}$	SATModel$_{bw}$	SATModel$_{he}$
1	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	**17**
11	N|A	N|A	N|A	N|A
12	**19**	**19**	**19**	**19**
13	**20**	**20**	**20**	**20**
14	**21**	**21**	**21**	**21**
15	**22**	**22**	**22**	**22**
16	N|A	N|A	N|A	N|A
17	N|A	N|A	**24**	N|A
18	**25**	N|A	N|A	N|A
19	N|A	N|A	N|A	N|A
20	N|A	N|A	N|A	N|A
...	...	...	...	...
40	N|A	N|A	N|A	N|A

Table 3: Results of `SATModel` with different *encodings* of the cardinality constraints (without *rotation* and without *symmetry breaking constraint*).

They have been defined two models, `SATModel` and `SATModelBorder`. They differ from each other only for the placing constraint: the first one works with `place`, the second one with `placeBorder` (as explained in Paragraph 3.3). From all the other points of view, they are exactly the same. It is interesting comparing them, in particular their version with and without the symmetry-breaking constraint (the rotation is not taken into consideration yet). Table 4 and Figure 9 show the comparison. `SATModel` performs better respect `SATModelBorder`: it is able to solve more instances respect the second one. From these two graphs, it can be seen also that the symmetry-breaking constraint effectively reduces the time required to solve the instances.

The last result to be analyzed is one of the rotated versions of the same

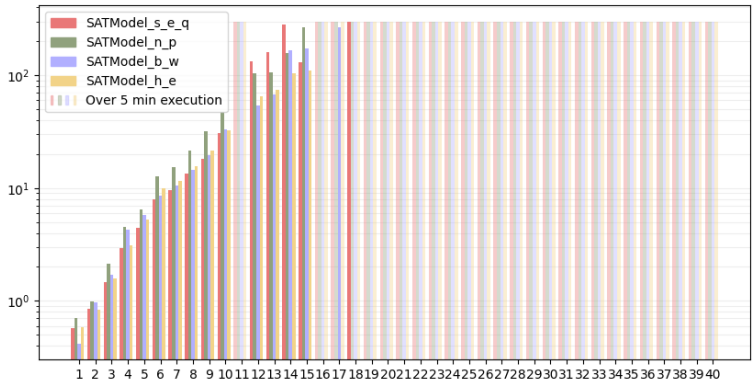


Figure 8: Performances of models show in Table 3

ID	SATModel$_{bw}$	SATModel$_{bw}$ + sb	SATModelBorders$_{bw}$	SATModelBorders$_{bw}$+ sb
1	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	**17**
11	N|A	**18**	N|A	**18**
12	**19**	**19**	**19**	**19**
13	**20**	**20**	**20**	**20**
14	**21**	**21**	**21**	**21**
15	**22**	**22**	**22**	**22**
16	N|A	**23**	N|A	N|A
17	**24**	**24**	N|A	N|A
18	N|A	**25**	N|A	N|A
19	N|A	N|A	N|A	N|A
20	N|A	N|A	N|A	N|A
...	...	...	...	...
40	N|A	N|A	N|A	N|A

Table 4: Results of **SATModel** and **SATModelBorder** with `Bitwise` encoding and with and without symmetry breaking constraint (`without rotation`).

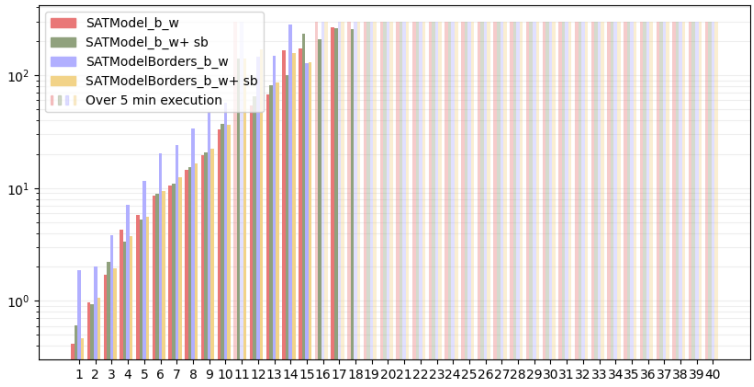


Figure 9: Performances of models shown in Table 4

models already described. Table 5 and Figure 10 describe it. All the models have solved the same number of instances, but with different times.`SATModel` performs still better respect `SATModelBorder` from a time point of view. Managing the rotation is expensive from a performance point of view, and this can be seen by the number of instances solved, which are less respect the previous situations.

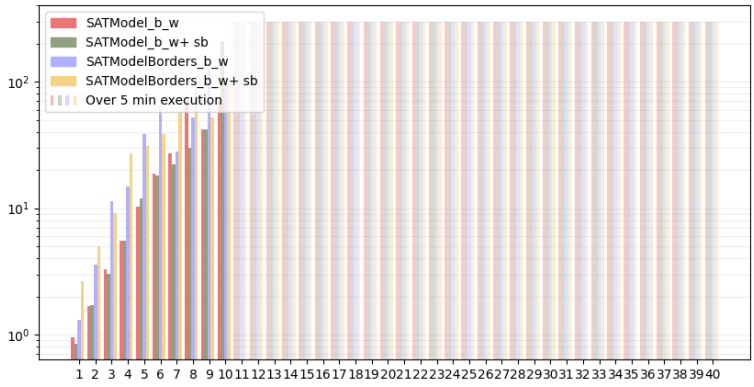


Figure 10: Performances of models shown in Table 5

ID	SATModel$_{bw}$	SATModel$_{bw}$ + *sb*	SATModelBorders$_{bw}$	SATModelBorders$_{bw}$+ *sb*
1	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	**17**
11	N|A	N|A	N|A	N|A
12	N|A	N|A	N|A	N|A
13	N|A	N|A	N|A	N|A
14	N|A	N|A	N|A	N|A
15	N|A	N|A	N|A	N|A
16	N|A	N|A	N|A	N|A
17	N|A	N|A	N|A	N|A
18	N|A	N|A	N|A	N|A
19	N|A	N|A	N|A	N|A
20	N|A	N|A	N|A	N|A
...	...	...	...	...
40	N|A	N|A	N|A	N|A

Table 5: Results of **SATModel** and **SATModelBorder** with `Bitwise` encoding and with and without symmetry breaking constraint (`with rotation`).

4 SMT

4.0.1 Decision Variables

The decision variables are five, two for both models and the remaining three for the rotation model:

- *x_positions* defines the position on the x-axis of the lower left corner of the circuit, defined as **X** in 1.1
- *y_positions* defines the position on the y-axis of the lower left corner of the circuit, defined as **Y** in 1.1
- *rotations* is a Boolean vector defining whether the circuit has been rotated, defined as **rotation** in 2.4
- *circuit_h_true* an Int vector defining the true height depending on whether the circuit is rotated or not, defined as $heigths^a$ in 2.4

- *circuit_w_true* an Int vector defining the true width depending on whether the circuit is rotated or not, defined as $widths^a$ in 2.4

4.1 Objective Function

In this case it was used the same one as for the SAT paradigm, refer to 3.2.

4.2 Constraints

The following constraints were used:

- Plate Boundary Constraint as described in 2.3
- Non Overlapping Constraint as described in 2.3
- Cumulative Scheduling Constraint on both columns and rows as described in 2.3
- Lexicographic Ordering Constraint was used to break symmetry as described in 2.3.

This last constraint was implemented using the Recursive OR Decomposition described in [1], which according to the paper seems to be the best implementation.

4.3 Rotation

This has been approached as in 2.4.

4.4 Validation

In this section the results as well as the other models will be presented.

4.4.1 Experimental Design

The solvers used in this section are z3Py and pySMT for which Z3 and MathSat were downloaded. The hardware used was:

- Processor: *AMD Ryzen 7 5800X3D 8-Core Processor 3.40 GHz (Boost 4.5 GHz)*
- RAM: *16.0 GB*

The problem has been firstly addressed using z3py, and later on using the solver-independent language pySMT, it was chosen because of its convenient Python API. Using pySMT two solvers where used, Z3 and MathSAT. It is worth noting that the solvers implemented in pySMT use only the plate boundary and the no-overlapping constraint.

4.4.2 Working in Parallel

To allow parallelism in pySMT, a portfolio solver was used which runs multiple solvers in parallel and once the first solver completes, all the solver are stopped. Unfortunately this did not produce any correct solution and hence it was discarded.
In z3Py it is necessary to explicitly specify the logic which is going to be used. To chose the right one the *get_logic* function from pySMT was used, this allows to perform an automatic selection which proposed *QF_LIA*. A parallel solver which allowed rotation was tried, but eventually discarded because it was producing bad solutions.

4.4.3 Experimental Results

As it is evident the best solver without rotation appears to be pySMT using Z3, it succeeds in solving all but two instances and it seems to be consistently faster than the others. The parallel solver, even though not necessarily faster than its non-parallel counterpart is able to solve more instances, it is worth noting that before settling down for *QF_LIA* as a logic various attempts were done, most of them performing terribly bad.
There is little to say about the rotation model, it is an harder problem as evident from the fewer solved instances.

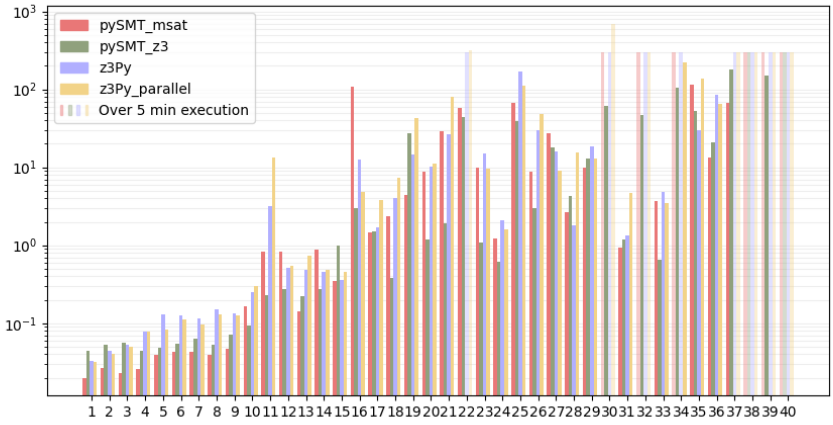


Figure 11: Time comparison between SMT executions with all the models. The times are shown in logarithmic scale. Rotation was not allowed.

ID	pySMT.msat	pySMT.z3	z3Py	z3Py.parallel
1	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	**17**
11	**18**	**18**	**18**	**18**
12	**19**	**19**	**19**	**19**
13	**20**	**20**	**20**	**20**
14	**21**	**21**	**21**	**21**
15	**22**	**22**	**22**	**22**
16	**23**	**23**	**23**	**23**
17	**24**	**24**	**24**	**24**
18	**25**	**25**	**25**	**25**
19	**26**	**26**	**26**	**26**
20	**27**	**27**	**27**	**27**
21	**28**	**28**	**28**	**28**
22	**29**	**29**	UNSAT	**29**
23	**30**	**30**	**30**	**30**
24	**31**	**31**	**31**	**31**
25	**32**	**32**	**32**	**32**
26	**33**	**33**	**33**	**33**
27	**34**	**34**	**34**	**34**
28	**35**	**35**	**35**	**35**
29	**36**	**36**	**36**	**36**
30	UNSAT	**37**	UNSAT	**37**
31	**38**	**38**	**38**	**38**
32	UNSAT	**39**	UNSAT	UNSAT
33	**40**	**40**	**40**	**40**
34	UNSAT	**40**	UNSAT	**40**
35	**40**	**40**	**40**	**40**
36	**40**	**40**	**40**	**40**
37	**60**	**60**	UNSAT	UNSAT
38	UNSAT	UNSAT	UNSAT	UNSAT
39	UNSAT	**60**	UNSAT	UNSAT
40	UNSAT	UNSAT	UNSAT	UNSAT

Table 6: Results using all the solvers. Rotation was not allowed.

ID	z3Py.rotation
1	**8**
2	**9**
3	**10**
4	**11**
5	**12**
6	**13**
7	**14**
8	**15**
9	**16**
10	**17**
11	**18**
12	**19**
13	**20**
14	**21**
15	**22**
16	UNSAT
17	**24**
18	**25**
19	UNSAT
20	UNSAT
21	UNSAT
22	UNSAT
23	UNSAT
24	**31**
25	UNSAT
26	UNSAT
27	UNSAT
28	UNSAT
29	UNSAT
30	UNSAT
31	**38**
32	UNSAT
33	**40**
34	**40**
35	**40**
36	**40**
37	UNSAT
38	UNSAT
39	UNSAT
40	UNSAT

Table 7: Results using the solvers for rotation.

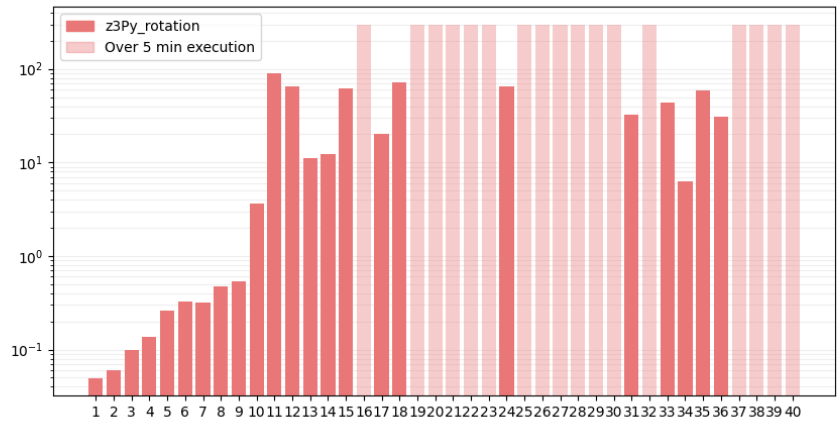


Figure 12: Time comparison between SMT executions with rotation. The times are shown in logarithmic scale.

5 MIP Model

Using the MIP paradigm, two models have been implemented. For each one of them follows a description of the decision variables, constraints, the way the rotation of the circuits was implemented and the validation.

5.1 Decision variables

5.1.1 Standard model

In the standard model there are the following decision variables:

- X, Y: The lists of the bottom-left x and y coordinates of all the circuits. Refer to section 1.1 for a complete description and to section 1.2 for their domains.
- $Circuit\_heights$: The list of the top y coordinates of all the circuits. For each circuit $i = 1, .., n$, $Circuits\_heights_i = Y_i + heights_i$. This variable has been defined as an helper to capture h.
- h: The variable representing the height of the plate. It must be the maximum among the $Circuit\_heights$. For its complete description and boundaries refer respectively to section 1.1 and 1.2.
- Max: A list of binary variables used to encode the MAX constraint needed to extract h from $Circuit\_heights$. For $i = 1, .., n$, $Max_i = 1$ if the i^{th} circuit is the one that has the maximum height, otherwise $Max_i = 0$.

- $Not\_overlaps$: a three dimensional array of binary variables to encode an OR constraint used to check that the circuits are not overlapping. The variable $Not\_overlaps_{d\_i\_j} = 1$ if the j^{th} circuit does not overlap the i^{th} circuit in direction d, for $d = 1, .., 4$, $i = 1, .., n$, $j = 1, .., n$. The four possible directions are right, left, up and down. For each dimension there will be an nxn matrix and in order to have complete information about the overlapping there is need to look only at the upper triangular portion of each one. This is true because we are never interested about the diagonal - each circuit is always overlapping itself - and we are not interested in the lower triangular portion because it is equal to the upper triangular portion of the matrix referring to the opposite direction. For example, if we consider $dl = 1$ (left) and $dr = 2$ (right) we have that $Not\_overlaps_{dl\_i\_j} = Not\_overlaps_{dr\_j\_i}$.

5.1.2 Strong bounds model

The strong bound model makes use of more constrained domains of the variables in order to reduce the total number of constraints. The only difference with respect to the standard model is that now the variable X_i has domain $[0..w - widths_i]$ for each $i = 1, .., n$ and that the variable Y_i has domain $[0..h_{upper\_bound} - heights_i]$ for each $i = 1, .., n$.

5.2 Objective function

The objective function of all the MIP models is to $minimize(h)$.

5.3 Constraints

5.3.1 Standard model

The standard model, given the variables formulation of the previous section, requires a constraint to enforce the circuits not to exceed the horizontal boundaries of the plate. Thus, it makes use of the following constraint:

- $Total\_width$: It is the constraint used to enforce the fact the each circuit must not exceed the boundaries of the plate on its right (the left side is already dealt by the X variable domain).

  $\forall\ i = 1, .., n,\ X_i + widths_i \leq w$.

There is also the need to impose the fact that h must be equal to the height of the plate. This is done using the following constraints:

- $Only\_one\_is\_max$: It enforces the fact that only one circuit can be the one having a maximum value for $Circuit\_heights$ by imposing

  $\sum_{i=1}^{n} Max_i = 1$.

- $Circuit\_i\_is\_max\_1$: It imposes the variable h to be greater or equal to all the top y coordinates of the circuits by imposing

  $\forall\ i = 1, .., n,\ h \geq Circuit\_heights_i$.

- $Circuit\_i\_is\_max\_2$: Enforces the variable h to be exactly the value of the maximum among all the top y coordinates of the circuits. It does it by imposing:

  $\forall\ i = 1, .., n,\ h - Circuit\_heights_i \leq M_{vertical} \cdot (1 - Max_i)$

  where $M_{vertical}$ is a big enough number (in this case we have $M_{vertical} = h_{upper\_bound}$). In this way if $Max_i = 1$, meaning that the i^{th} circuit is the one with maximum top y coordinate, we have $h \leq Circuit\_heights_i$. On the other hand, if $Max_i = 0$, we have $h - Circuit\_heights_i \leq M_{vertical}$ that, since $M_{vertical}$ is the upper bound for h and $Circuit\_heights_i$ is by definition non negative, is always true.

The last constraint required by the problem is the fact the all the circuits must be in distinct location, i.e. they are not overlapping one over the other. This requirement is fulfilled through the following constraints:

- $None\_overlapping$: For each pair of circuits, it checks that they are not overlapping by enforcing that they are not overlapped in at least one direction. This constraint is imposed linearly in the following way:

  $\forall\ i = 1, ..n,\ j = 1, .., n \mid i < j,\ \sum_{d=1}^{4} Not\_overlaps_{d\_i\_j} \geq 1$.

- J_is_right: Used to enforce that if the circuit j is to the right ($d = 1$) of the right boundary of the circuit i, then $Not\_overlaps_{1\_i\_j} = 1$. This is expressed linearly by imposing:

  $\forall\ i = 1..n,\ j = 1, .., n \mid i < j,\ X_i + widths_i - X_j \leq M_{horizontal} \cdot (1 - Not\_overlaps_{1\_i\_j})$

  In this case we make use of the same Big-M trick as in constraint $Circuit\_i\_is\_max\_2$ and the big enough $M_{horizontal} = w$.

- J_is_left, J_is_up, J_is_down: In a similar way as in constraint J_is_right, it is imposed a constraint for every other direction (left, up and down). The constraints are expressed as follows:

  Left ($d = 2$): $\forall\ i = 1..n,\ j = 1, .., n \mid i < j,\ X_j + widths_j - X_i \leq M_{horizontal} \cdot (1 - Not\_overlaps2\_i\_j)$

  Up ($d = 3$): $\forall\ i = 1..n,\ j = 1, .., n \mid i < j,\ Y_i + heights_i - Y_j \leq M_{vertical} \cdot (1 - Not\_overlaps3\_i\_j)$

  Down ($d = 4$): $\forall\ i = 1..n,\ j = 1, .., n \mid i < j,\ Y_j + heights_j - Y_i \leq M_vertical \cdot (1 - Not\_overlaps4\_i\_j)$

5.3.2 Strong bounds model

Thanks to the more constrained domains of the variables in X, in the strong bounds model the $Total\_width$ constraint described in section 5.3.1 is not useful anymore. The domain of each variable in X is strong enough to avoid the circuit to exceed the horizontal boundaries of the plate. All the other constraints are identical to the ones in the standard model.

5.4 Rotation

5.4.1 Standard model

In order to take into account the possible rotation of the circuits, some new variables must be introduced, the domains of the existing variables must be modified and some constraints must be updated.

A new list of binary variables *Rotated* is introduced as in section 2.4 where $Rotated_i = 1$ iff the i^{th} circuit is rotated, for $i = 1, .., n$.

The upper bound for the variables in X and Y must be modified. Let's call $min\_widths\_heights$ the minimum value among both widths and heights. The new domain for X_i becomes $[0, w - min\_widths\_heights]$ and the new domain for Y_i becomes $[0, h_{upper\_bound} - min\_widths\_heights]$.

Finally, in order to consider the possibility of a circuit to be rotated, every time the parameter $heights_i$ or $widths_i$ is used in a variable definition or a constraint, we must conditionally select the height or width of the circuit depending on its orientation. It is possible to express this linearly with the following substitutions:

$heights_i$ becomes $heights_i \cdot (1 - Rotated_i) + widths_i \cdot Rotated_i$

$widths_i$ becomes $widths_i \cdot (1 - Rotated_i) + heights_i \cdot Rotated_i$

In this way if the circuit i is not rotated ($Rotated_i = 0$) $heights_i$ stays the same. On the other hand, if the circuit is rotated ($Rotated_i = 1$) $heights_i$ is substituted with $widths_i$. E.g. the $Total\_width$ constraint of section 5.3.1 is transformed into:

For $i = 1, .., n$, $X_i + widths_i \cdot (1 - Rotated_i) + heights_i \cdot Rotated_i \leq w$.

5.4.2 Strong bounds model

In addition to the modification specified in the previous section, in the strong bounds model we need some other tricks to make the rotation of the circuits possible. First of all we want to make the bounds for X and Y stronger. Since we cannot introduce a variable in the bound of another variable, the domain for X_i becomes $[0, w - min(heights_i, widths_i)]$ and the domain for Y_i becomes $[0, h_{upper\_bound} - min(heights_i, widths_i)]$. This forces us to reintroduce the $Total\_width$ constraint, that we had removed in the strong bound model without rotation.

5.5 Validation

In this section the results of both the standard model and the strong bound model will be presented.

5.5.1 Experimental design

The experiments have been executed in a Windows machine with the following hardware:

- Processor: *Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz*
- RAM: *8.0 GB*

The models have been implemented in *AMPL* version *20221013 (MSVC 19.29.30146.0, 64-bit)* called from a python program with *Python 3.9.12* using the *AMPL* API *amplpy* version *0.8.5*. All the models, both with and without rotation, have been executed on all the instances - 1 to 40 - with 6 different solvers:

- *Gurobi version 10.0.0*
- *COpt version 5.0.1*
- *CPLEX version 20.1.0.0*
- *HiGHS version 1.2.2*
- *Xpress version 8.13.1(39.01.02)*

Also *CBC version 2.10.5* was tried, but, since it led to much worse results, it has been removed from the report. Each execution had a time limit of 300 seconds after which the best valid solution was returned, if any.

5.5.2 Experimental results

The results of the experiments on the MIP models show a considerable influence of the solver in the performance. On the other hand, the difference between the standard model and the strong bounds model is minor. As it can be seen in table 8, all the solvers were able to solve the problem to optimality up to instance 15.

The solver that generally performed best is **Gurobi**, while the other solvers had a similar performance. The strong bounds model outperformed the standard model in just a few instances: gurobi 40 and cplex 29; but it was defeated in others: cplex 33 and highs 37.

In order to have a better understanding of the difference between the standard model and the strong bound model let's have a look at figure 13. There it is possible to look at the execution times using Gurobi for all the instances and for both models. Also with respect to times, the difference between the two models seems instance dependant. Therefore, it is possible to conclude that the strong bound model did not bring an improvement in the performance of the solvers.

ID	std copt	stb copt	std cplex	stb cplex	std gurobi	stb gurobi	std highs	stb highs	std xpress	stb xpress
1	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**
11	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**
12	**19**	**19**	**19**	**19**	**19**	**19**	**19**	**19**	**19**	**19**
13	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
14	**21**	**21**	**21**	**21**	**21**	**21**	**21**	**21**	**21**	**21**
15	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**
16	24	24	**23**	**23**	24	24	24	24	**23**	**23**
17	**24**	**24**	**24**	**24**	**24**	**24**	**24**	**24**	**24**	**24**
18	**25**	**25**	**25**	**25**	**25**	**25**	26	26	**25**	**25**
19	**26**	**26**	27	27	**26**	**26**	28	28	27	N|A
20	**27**	**27**	28	28	**27**	**27**	28	28	28	28
21	29	29	29	29	29	29	29	29	29	29
22	30	30	30	30	30	30	31	31	30	30
23	**30**	**30**	31	31	**30**	**30**	**30**	**30**	32	32
24	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**
25	34	34	33	33	33	33	34	34	33	33
26	34	34	34	34	**33**	**33**	34	34	34	34
27	**34**	**34**	**34**	**34**	**34**	**34**	35	35	**34**	**34**
28	36	36	36	36	**35**	**35**	36	36	**35**	**35**
29	37	37	37	**36**	**36**	**36**	**36**	**36**	37	37
30	39	39	38	38	38	38	40	40	39	39
31	**38**	**38**	39	39	**38**	**38**	**38**	**38**	**38**	**38**
32	42	42	41	41	40	40	42	42	42	42
33	**40**	**40**	41	42	**40**	**40**	41	41	41	41
34	41	41	41	41	**40**	**40**	42	42	42	42
35	41	41	41	41	41	41	41	41	41	41
36	41	41	41	41	**40**	**40**	41	41	41	41
37	63	63	62	62	61	61	63	64	63	63
38	64	64	62	62	62	62	63	63	64	64
39	62	62	62	62	61	61	64	64	62	62
40	138	138	108	108	100	99	N|A	N|A	196	196

Table 8: Results using all the solvers both with standard model (std) and strong bounds model (stb). Rotation was not allowed.

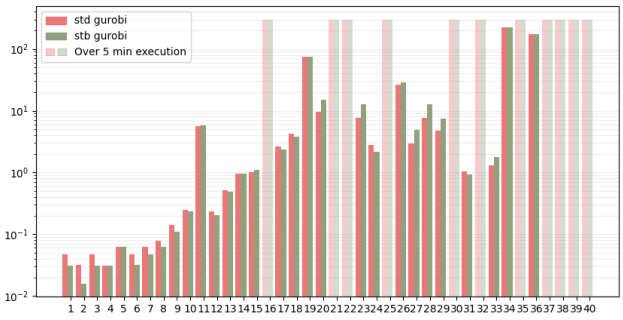


Figure 13: Time comparison between gurobi executions with standard model and strong bound model. The times are shown in logarithmic scale. Rotation was not allowed.

By introducing the possibility of rotating the circuits, as it can be seen in table 9, the results are generally worse. This happens because, in order to introduce the rotation, variables and constraints had to be added to the model, which made it slower. Still, there are some cases in which the model with rotation outperformed the model without such as Gurobi 35. Also in this case the performance is mostly dependent on the solver, while the model has no great impact on the result. Gurobi has again shown to be the best MIP solver for this task.

As for the models where rotation was not allowed, a comparison between the two models using Gurobi is shown in figure 14. Also in this case the models have a very similar time performance, confirming the fact that the strong bounds model is not really bringing any significant improvement.

ID	std copt	stb copt	std cplex	stb cplex	std gurobi	stb gurobi	std highs	stb highs	std xpress	stb xpress
1	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**
11	**18**	**18**	**18**	19	**18**	**18**	19	19	**18**	**18**
12	**19**	**19**	**19**	**19**	**19**	**19**	20	20	**19**	**19**
13	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
14	**21**	**21**	**21**	**21**	**21**	**21**	22	22	**21**	**21**
15	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**
16	24	24	24	24	**23**	**23**	24	24	24	24
17	25	25	**24**	**24**	25	25	**24**	**24**	25	25
18	26	26	**25**	**25**	26	26	26	26	**25**	**25**
19	27	27	27	27	27	27	28	28	27	27
20	28	28	28	28	28	28	28	28	28	28
21	29	29	29	29	29	29	30	30	29	29
22	30	30	30	30	**29**	**29**	31	31	30	30
23	31	31	31	31	31	31	31	31	31	31
24	**31**	**31**	32	32	32	32	32	32	32	32
25	34	34	33	33	33	33	34	34	34	34
26	34	34	34	35	34	34	34	34	34	34
27	**34**	**34**	35	35	**34**	**34**	36	36	**34**	**34**
28	36	36	36	36	36	36	36	36	36	36
29	37	37	37	37	37	**36**	38	38	37	37
30	39	39	40	40	38	38	40	40	38	38
31	**38**	**38**	39	39	**38**	**38**	39	39	**38**	**38**
32	41	41	41	41	40	40	42	42	41	41
33	**40**	**40**	41	41	**40**	**40**	41	41	**40**	**40**
34	41	41	41	41	**40**	**40**	42	42	41	41
35	41	41	41	41	**40**	**40**	42	42	41	41
36	41	41	41	41	**40**	**40**	42	42	41	41
37	62	62	62	62	61	61	63	63	62	62
38	62	62	61	61	61	61	63	63	62	62
39	62	62	61	61	61	61	64	64	61	61
40	137	137	115	116	104	103	N|A	N|A	191	194

Table 9: Results using all the solvers both with standard model (std) and strong bounds model (stb). Rotation was allowed.

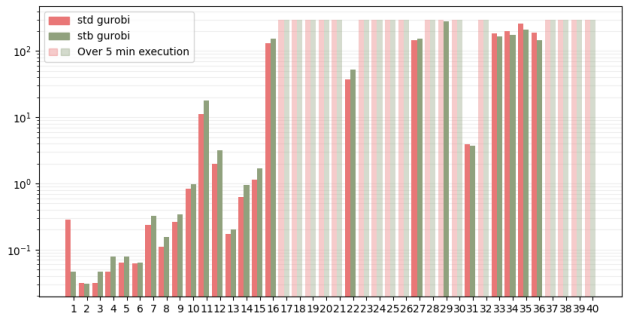


Figure 14: Time comparison between Gurobi executions with standard model and strong bound model. The times are shown in logarithmic scale. Rotation was allowed.

6 Conclusions

To conclude, the VLSI packing problem has been tackled via different paradigms that showed different performances. As it can be seen in table 10 and table 11, CP showed to be the best one both with and without rotation. It achieved very good results and it proved to be the most intuitive paradigm for modelling the problem. MIP performed somewhat worse than CP, but proved to be able to maintain good results also in presence of rotation. SMT showed great results in the problem that did not involve the possibility of rotating the circuits, while it performed much worse in the other case. On the other hand, the SAT models were not effective in modelling the problem and proved to be the most inefficient ones.

ID	CP	SAT	SMT	MIP
1	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	**17**
11	**18**	**18**	**18**	**18**
12	**19**	**19**	**19**	**19**
13	**20**	**20**	**20**	**20**
14	**21**	**21**	**21**	**21**
15	**22**	**22**	**22**	**22**
16	**23**	**23**	**23**	**23**
17	**24**	**24**	**24**	**24**
18	**25**	**25**	**25**	**25**
19	**26**	N|A	**26**	**26**
20	**27**	N|A	**27**	**27**
21	**28**	N|A	**28**	29
22	**29**	N|A	**29**	30
23	**30**	N|A	**30**	**30**
24	**31**	N|A	**31**	**31**
25	**32**	N|A	**32**	33
26	**33**	N|A	**33**	**33**
27	**34**	N|A	**34**	**34**
28	**35**	N|A	**35**	**35**
29	**36**	N|A	**36**	**36**
30	**37**	N|A	**37**	38
31	**38**	N|A	**38**	**38**
32	**39**	N|A	**39**	40
33	**40**	N|A	**40**	**40**
34	**40**	N|A	**40**	**40**
35	**40**	N|A	**40**	41
36	**40**	N|A	**40**	**40**
37	**60**	N|A	**60**	61
38	**60**	N|A	UNSAT	62
39	**60**	N|A	**60**	61
40	92	N|A	UNSAT	99

Table 10: Comparison between paradigms without the possibility of rotating the circuits.

ID	CP	SAT	SMT	MIP
1	**8**	**8**	**8**	**8**
2	**9**	**9**	**9**	**9**
3	**10**	**10**	**10**	**10**
4	**11**	**11**	**11**	**11**
5	**12**	**12**	**12**	**12**
6	**13**	**13**	**13**	**13**
7	**14**	**14**	**14**	**14**
8	**15**	**15**	**15**	**15**
9	**16**	**16**	**16**	**16**
10	**17**	**17**	**17**	**17**
11	**18**	N|A	**18**	**18**
12	**19**	N|A	**19**	**19**
13	**20**	N|A	**20**	**20**
14	**21**	N|A	**21**	**21**
15	**22**	N|A	**22**	**22**
16	**23**	N|A	N|A	**23**
17	**24**	N|A	**24**	**24**
18	**25**	N|A	**25**	**25**
19	**26**	N|A	N|A	27
20	**27**	N|A	N|A	28
21	**28**	N|A	N|A	29
22	**29**	N|A	N|A	**29**
23	**30**	N|A	N|A	31
24	**31**	N|A	**31**	**31**
25	**32**	N|A	N|A	33
26	**33**	N|A	N|A	34
27	**34**	N|A	N|A	**34**
28	**35**	N|A	N|A	36
29	**36**	N|A	N|A	**36**
30	38	N|A	N|A	38
31	**38**	N|A	**38**	**38**
32	40	N|A	N|A	40
33	**40**	N|A	**40**	**40**
34	**40**	N|A	**40**	**40**
35	**40**	N|A	**40**	**40**
36	**40**	N|A	**40**	**40**
37	61	N|A	N|A	61
38	**60**	N|A	N|A	61
39	**60**	N|A	N|A	61
40	92	N|A	N|A	103

Table 11: Comparison between paradigms with the possibility of rotating the circuits.

References

[1] Hani Elgabou and A.M. Frisch. "Encoding the lexicographic ordering constraint in sat modulo theories". In: *Thirteenth International Workshop on Constraint Modelling and Reformulation* (Jan. 2014), pp. 85–96.