Very Large Scale Integration Project

Davide Sangiorgi
University of Bologna
davide.sangiorgi3@studio.unibo.it

Leonardo Monti
University of Bologna
leonardo.monti3@studio.unibo.it

Riccardo Falco
University of Bologna
riccardo.falco2@studio.unibo.it

Matteo Conti
University of Bologna
matteo.conti16@studio.unibo.it

Abstract

Very Large Scale Integration problem refers to the trend of integrating circuits into silicon chips. In particular, in this report we present the results of four different optimization technique for a specific instance of VLSI set of problems (based on a plate with a fixed width inside which a set of given circuits must be placed). The goal of the objective function is to find the minimum value for the height of the plate. The techniques used during the experiments are: Constraint Programming (CP), Proposition Satisfability (SAT), Satisfiability Modulo Theory (SMT) and Mixed Integer Linear Programming (MILP). Despite the use of the similar constraints formulation (with respect to the proper optimization technique formalization) we obtained quite differents results over the four techniques. In particular, CP and SAT models out perform the other techinques (SMT, MILP) in terms of number of solved instances.

Contents

1 Introduction

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. A typical example are systems on a chip for smartphones. The modern trend of shrinking transistor sizes, allowing engineers to fit more and more transistors into the same area of silicon, has pushed the integration of more and more functions of cellphone circuitry into a single silicon die (i.e. plate). This enabled the modern cellphone to mature into a powerful tool that shrank from the size of a large brick-sized unit to a device small enough to comfortably carry in a pocket or purse, with a video camera, touchscreen, and other advanced features.

This report describes a Combinatorial Optimization approach to the VLSI problem. In particular, four different optimization techniques have been implemented to address the problem at hand, namely Constraint Programming (CP), propositional Satisfiability (SAT), Satisfiability Modulo Theories (SMT) and Mixed Integer Linear Programming (MILP).

All the material (source code) used for running the experiments can be found at the following link.

2 Background

2.1 Problem Description

The purpose of this project is to model and solve the *VLSI* problem: given a fixed-width plate and a list of rectangular circuits, place them on the plate so that the length of the final device is minimized. We faced first the easier case in which the circuits cannot be rotated and then the other case on a distinct model.
The solution of the problem gives the position of each circuit (the coordinates of the bottom left corner) and, in case a circuit has been rotated, its width and height are coherently modified.

2.2 Format of Files

Instance Format An instance of *VLSI* is a text file consisting of lines of integer values. The first line gives *width*, which is the width of the silicon plate. The following line gives *nc*, which is the number of necessary circuits to place inside the plate. Then *nc* lines follow, each with w_i and h_i, representing the horizontal and vertical dimension of the *i-th* circuit. The file is terminated by an empty line.

Solution Format A solution of *VLSI* is a text file built starting from the input file. To the first line is added the *makespan*, which is the length of the final device. The second line is kept the same while to the following *nc* lines are added x_i and y_i, which are the coordinates of the bottom left corner of the *i-th* circuit. In case a solution is obtained through the rotation of a circuit i, then, in the output file, the horizontal and vertical dimensions (w_i and h_i) will be swapped w.r.t. the input text file.

2.3 Shared Notation

In this section we list the parameters and the variables that are shared in the mathematical formalization of models and constraints from now on.

Parameters:

$$
\begin{aligned}
nc &= \text{total number of circuits} \\
width &= \text{width of the plate} \\
c_i &= \text{index of circuit } i \\
w_i &= \text{width of circuit } i \\
h_i &= \text{height of circuit } i \\
min\_makesapan &= \text{lower bound of } makespan \text{ variable} \\
max\_makesapan &= \text{upper bound of } makespan \text{ variable} \\
C &= \{c_i \mid i \in [1, nc]\} \\
CC &= \{(i, j) \in C \times C \mid i < j\}
\end{aligned}
$$

Variables:

$$
\begin{aligned}
x_i &= \text{x coordinate of the bottom-left corner of circuit } i \\
y_i &= \text{y coordinate of the bottom-left corner of circuit } i \\
r_i &= \text{boolean variable indicating if circuit } i \text{ is rotated} \\
X &= \{x_i \mid i \in [1, nc]\} \\
Y &= \{y_i \mid i \in [1, nc]\} \\
Xcross(x) &= \{c \in C \mid (x_c \leq x) \wedge (x_c + w_c > x)\} \\
Ycross(y) &= \{c \in C \mid (y_c \leq y) \wedge (y_c + h_c > y)\} \\
makespan &= \max_{c \in C} y_c + h_c \\
w\_index_{min} &= i \in C \mid w_i = \min_{c \in C} w_c \\
h\_index_{min} &= i \in C \mid h_i = \min_{c \in C} h_c
\end{aligned}
$$

2.3.1 Makespan lower bound

In order to limit the search domain of the *makespan* variable, we defined an algorithm to find its lower (*min_makespan*) and upper (*max_makespan*) bound.
The lower bound is determined starting from the area that the circuits occupy:

$$a = \sum_{c \in C} w_c \cdot h_c$$

The best situation is the one in which all circuits fit in a rectangle whose width=*width* and height=*makespan*, without leaving any empty space, so that the *makespan* would be $makespan = \lceil a/width \rceil$. We could set accordingly $min\_makespan = \lceil a/width \rceil$, but there is simple a case in which the *makespan* can be easily improved (increased), even if there are no empty spaces left. Indeed, we may have a very "high" circuit (let its name be c_{high}) such that:

$$h_{c_{high}} >= \lceil \frac{\sum_{c \in C - \{c_{high}\}} w_c \cdot h_c}{width - w_{c_{high}}} \rceil \tag{1}$$

where the numerator is the sum of the areas of all circuits $c_i \neq c_{high}$. On the right of the inequality we obtain the $min\_makespan'$ computed without c_{high} and with a reduced width of the plate $width' = width - w_{c_{high}}$.
Let m be the right side of the inequality. It's almost intuitive that the *min_makespan* computed as before is lower than $h_{c_{high}}$:

$$m = \frac{\sum_{c \in C - \{c_{high}\}} w_c \cdot h_c}{width - w_{c_{high}}} \tag{2}$$

then

$$
\begin{aligned}
min\_makespan &= m + \frac{(h_{c_{high}} - m) \cdot w_{c_{high}}}{width} \\
&= h_{c_{high}} \cdot \frac{w_{c_{high}}}{width} + m \cdot \left(1 - \frac{w_{c_{high}}}{width}\right) \\
h_{c_{high}} &= \frac{width}{w_{c_{high}}} \cdot min\_makespan - \left(\frac{width}{w_{c_{high}}} - 1\right) \cdot m
\end{aligned}
$$

From the first one we have $min\_makespan \geq m$ since $h_{c_{high}} \geq m$ for definition. From the second we can easily prove $h_{c_{high}} \geq min\_makespan$:

$$\begin{aligned} h_{c_{high}} &= \frac{width}{w_{c_{high}}} \cdot min\_makespan - \left(\frac{width}{w_{c_{high}}} - 1\right) \cdot m \geq min\_makespan \\ \left(\frac{width}{w_{c_{high}}} - 1\right) \cdot min\_makespan &- \left(\frac{width}{w_{c_{high}}} - 1\right) \cdot m \geq 0 \\ \left(\frac{width}{w_{c_{high}}} - 1\right) &\cdot (min\_makespan - m) \geq 0 \end{aligned}$$

that is true since $width \geq w_{c_{high}} \rightarrow \frac{width}{w_{c_{high}}} \geq 1$ for model constraints and $min\_makespan \geq m$ as shown before; the situation we are describing is also shown in Fig.1.

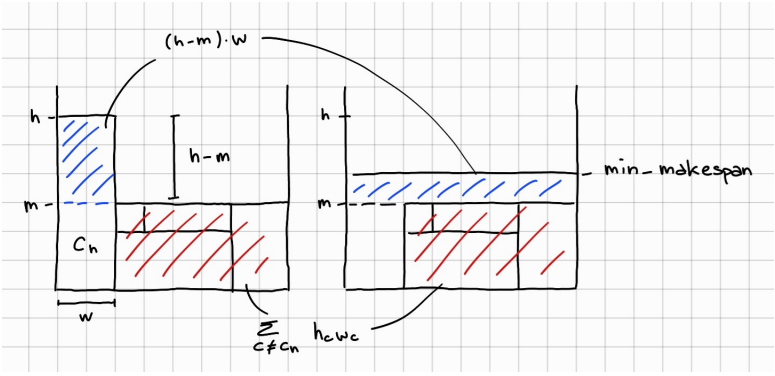


Figure 1: On the left a possible solution with circuit $c_n = c_{high}$ satisfying condition 1, with m defined in 2, $h = h_{c_{high}}$ and $w = w_{c_{high}}$. On the right, how $min\_makespan$ was computed before improving it. In this situations we always have $m \leq min\_makespan \leq h_{c_{high}}$.

The improvement for the lower bound of *makespan* is easy and immediate:

$$min\_makespan = max\left(\max_{c \in C} h_c, \lceil a/width \rceil\right)$$

2.3.2 Makespan upper bound

The upper bound of the *makespan* variable (*max_makespan*) is determined as the makespan of a possible solution that must be built using an effective and efficient algorithm. The solution with the greatest possible *max_makespan* is obtained stacking one on top of the other all the circuits of the instance. This strategy is very simple and fast but the result easily over estimates by a lot the final *makespan* value. We adopted a more effective strategy.

A much better algorithm requires to consider each circuit as a node so that we can build a solution by building a tree of the nodes. Each circuit is represented as a node that contains the dimensions of the circuit w_c, h_c, its coordinates x_c, y_c, the index of the circuit, the index of the parent node and a list of the children nodes $children\_list_c$. The $altitude_c$ of the node c can be computed as:

$$altitude_c = y_c + h_c$$

The remaining horizontal free space $remaining\_space_c$ on the top of each circuit can be calculated as:

$$remaining\_space_c = w_c - \sum_{i \in children\_list_c} w_i$$

The relationship of a circuit being stacked on top of another is represented with the node of the circuit at the bottom being the father of the node of the circuit stacked on top. All the nodes are added to a list of nodes that is sorted on the width of the circuit in descending order:

$$\forall (i,j) \in CC.\ \ w_i \geq w_j$$

In this way the nodes are added to the tree from the widest to the least wide.

A conceptual root node is computed. The width of the root node is the *width* of the plate and its height is 0. The root node is inserted in the head of the list of nodes.

All the nodes are then inserted into the tree using the following strategy: a list of possible parent nodes $fringe$ is kept. A node must have enough free space on top to be a parent of another. Let z be the index of the circuit that is being inserted; $fringe_z$ is the list of all the nodes c already in the tree that satisfy:

$$fringe_z = \left[c \in C \mid w_z \leq w_c - \sum_{i \in children\_list_c} w_i \right]$$

Each new node is appended to the node in the $fringe_z$ whose altitude is the lowest.

$$altitude_{father_z} = \min_{c \in fringe_z} altitude_c$$

Once all the nodes are in the tree we can calculate the maximum altitude $altitude_{max}$ reached by the nodes.

$$altitude_{max} = \max_{c \in C} altitude_c$$

This value is the makespan of a fully legit solution so it is intuitive that the makespan of the best solution cannot be greater than it.
The time required for this calculation is negligible w.r.t the time required to obtain the best solutions with any of the technologies used.

2.4 Shared Predicates

We have defined the following support predicates:

$$\text{diff\_n} \leftarrow \bigwedge_{i,j \in CC} (x_i + w_i \leq x_j) \vee (y_i + h_i \leq y_j) \vee\ (x_j + w_j \leq x_i) \vee (y_j + h_j \leq y_i) \tag{3}$$

$$\text{cumulative\_x} \leftarrow \bigwedge_{x \in X}\ \sum_{c \in X cross(x)} h_c \leq makespan \tag{4}$$

$$\text{cumulative\_y} \leftarrow \bigwedge_{y \in Y}\ \sum_{c \in Y cross(y)} w_c \leq width \tag{5}$$

$$\text{disj\_cond\_x} \leftarrow \bigwedge_{c \in C} (h_c + h_{min} > makespan) \vee (c = h\_index_{min})$$

$$\text{disj\_cond\_y} \leftarrow \bigwedge_{c \in C} (w_c + w_{min} > width) \vee (c = w\_index_{min})$$

$$\text{disj\_x} \leftarrow \bigwedge_{(c1,c2) \in CC} (x_{c1} + h_{c1} \leq x_{c2}) \vee (x_{c2} + h_{c2} \leq x_{c1})$$

$$\text{disj\_y} \leftarrow \bigwedge_{(c1,c2) \in CC} (y_{c1} + h_{c1} \leq y_{c2}) \vee (y_{c2} + h_{c2} \leq y_{c1})$$

2.5 Shared Constraints

The following formalization will be the one followed for the CP solution and then adapted for SAT and SMT. The ILP model will be described after.

Starting from the predicates defined in Section 2.4, we have defined the following constraints valid over all models (CP, SAT, SMT, ILP):

$$\bigwedge_{c \in C} w_c > 0 \ \wedge \ w_c \leq width$$
$$\bigwedge_{c \in C} h_c > 0 \ \wedge \ h_c \leq makespan$$
$$\bigwedge_{c \in C} x_c \geq 0 \ \wedge \ x_c \leq width - w_c$$
$$\bigwedge_{c \in C} y_c \geq 0 \ \wedge \ y_c \leq makespan - h_c$$
$$\text{diff\_n} \ \leftrightarrow \ \top$$

Being common to CP, SAT and SMT, here we list the cumulative constraints adopted:

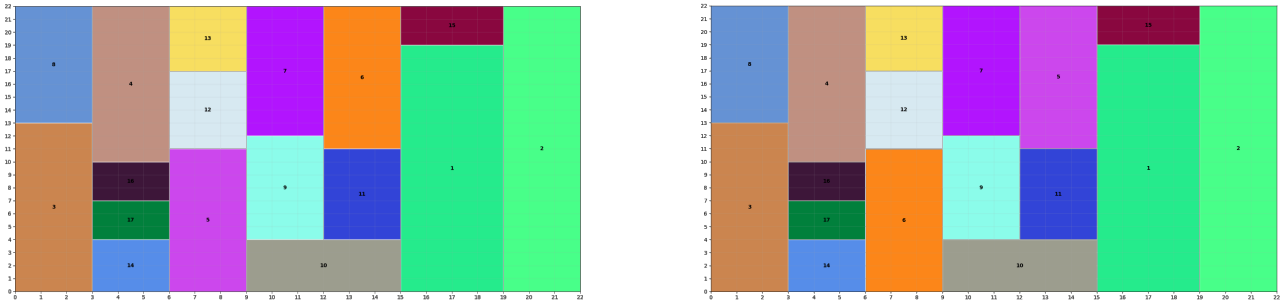
$$(\text{disj\_cond\_x} \rightarrow \text{disj\_x}) \ \wedge \ (\neg\ (\text{disj\_cond\_x}) \rightarrow \text{cumulative\_x})$$
$$(\text{disj\_cond\_y} \rightarrow \text{disj\_y}) \ \wedge \ (\neg\ (\text{disj\_cond\_y}) \rightarrow \text{cumulative\_y})$$

Like in the "cumulative" of MiniZinc [1], the constraint first checks if disjunctive ($disj_x$, $disj_y$) can be used instead. Actually in the documentation[1] is checked also if *all_different* can be used, which is even faster, but also improbable in our use case.

2.5.1 Symmetries

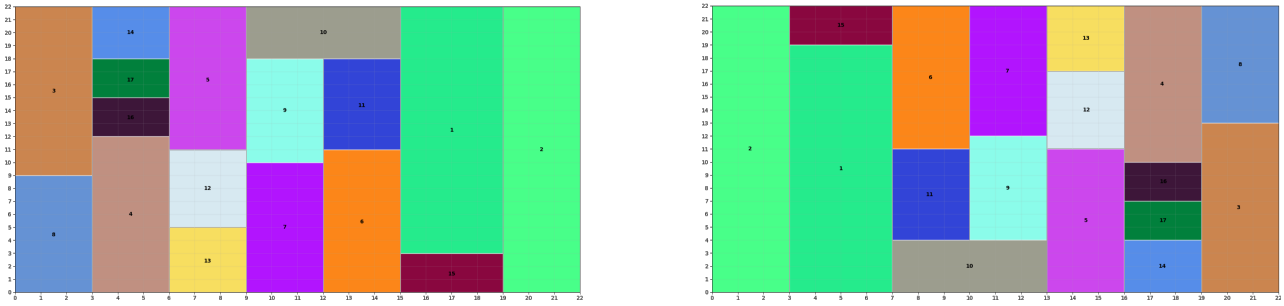
Given a solution there are two main ways to get a different one with the same *makespan*: swap circuits with same dimensions [Fig.2] or get the solution specular w.r.t. the horizontal or the vertical axis [Fig.3]. If we consider also sub-rectangles which do not cross their edges with any circuit and call them *virtual* circuits [Fig.4], we can generalize what mentioned before and find all solutions with the same *makespan* [Fig.5]. In particular we can find a symmetric solution swapping any couple of circuits or *virtual* circuits with same dimensions, or we can substitute any *virtual* circuit with the set of *real* circuits within it, but with specular positions. Obviously, combinations of the previous cases will lead to other symmetric solutions [Fig.5c].

[1] https://github.com/MiniZinc/libminizinc/blob/master/share/minizinc/std/cumulative.mzn,



(a) (b)

Figure 2: Plotted solution of an instance created for explanatory puropose. From the solution on the left [Fig.2a] we can create a different one with the same *makespan* just swapping circuits with same dimensions. In the plot on the right [Fig.2b] we swapped circuit 5 (at originial position of (6,0)) and circuit 6 (at originial position of (12,11)).



(a) (b)

Figure 3: Other possible solutions with same makespan as the one plotted in [Fig.2a]. The left one [Fig.3b] is the specular w.r.t. the vertical axis, while the left one [Fig.3a] is the specular w.r.t. the horizontal axis

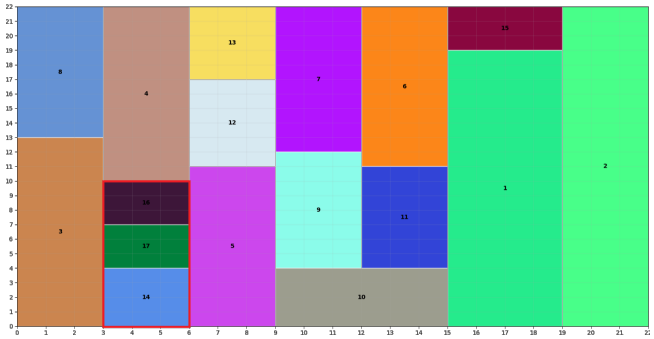
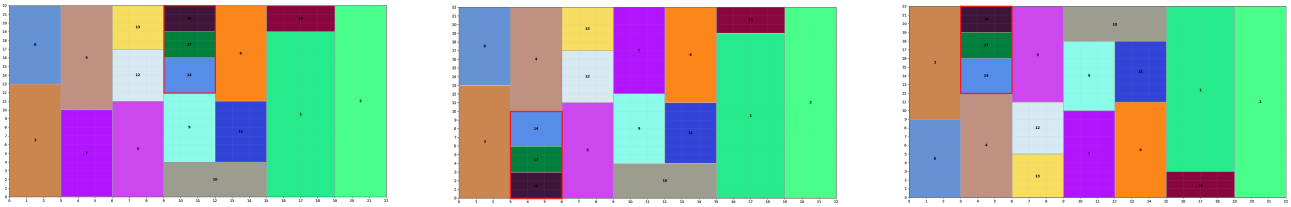


Figure 4: At position $(3, 0)$ an example of *virtual* circuit highlighted in red, with $w = 3$ and $h = 10$, which includes inside the circuits 14, 16 and 17.



(a) (b) (c)

Figure 5: Other possible solutions with same makespan as the one plotted in [Fig.2a], now including also *virtual* circuits in the reasoning. In the left one [Fig.5a], the *virtual* circuit introduced in [Fig.4] at original position (3,0) is swapped with the *real* circuit 7 at original position (9,11). In the middle one [Fig.5b] the same *virtual* circuit is substituted with its specular w.r.t. the horizontal axis. The right one [Fig.5c] is an example of combination of what mentioned before: starting from the solution [Fig.3a], the subset of circuits 14, 16, 17, belonging to the highlighted *virtual* circuit, is *"mirrored"* again.

Catching all the symmetric solutions described before is quite demanding and we did not find an efficient way of implementing all the symmetry breaking constraints, so we tried to keep them as simple as possible.
We define first some support variables:

$$\begin{aligned} x\_v &= [x_c \mid c \in C] \\ y\_v &= [y_c \mid c \in C] \\ x\_v' &= [width - (x_c + w_c) \mid c \in C] \\ y\_v' &= [makesapan - (y_c + h_c) \mid c \in C] \end{aligned} \tag{6}$$

where x_v and y_v are respectively the vector of all horizontal and vertical coordinates, while x_v' and y_v' are the horizontal and the vertical coordinates of the specular circuits.

The symmetry breaking constraints proposed for CP, SAT and SMT are the lexicographic orderings between y_v and y_v' and between x_v and x_v'; in this way we break the symmetries shown respectively in [Fig.3a] and [Fig.3b].

$$lex\_lesseq(x_v, x'_v)$$
$$lex\_lesseq(y_v, y'_v)$$

Later we will specify with more detail the adopted constraints and compare the performances of the CP, SAT and SMT models with and without the symmetry breaking constraints.

3 CP

3.1 Background

The first paradigm used to solve the VLSI optimization problem is Constraint Programming (CP), developed using MiniZinc as modeling language. Starting from a base model following the constraints discussed in Section 2.5, we added: rotation variables r_i, symmetry breaking constraints and search strategies. We have a model for each combination of the previous elements. To test different behaviour of solvers we chose to run each model on both Chuffed and Gecode solvers.

3.2 Notation

In CP we tried to break also the symmetries related to *virtual* circuits, but we decided to keep only the simplest ones for efficiency reasons, as anticipated in Section 2.5.1.
This section is mainly needed to introduce constants related to *virtual* circuits:

$$\begin{aligned}
vc_{i,j} &= \textit{virtual} \text{ circuit obtained from } (i,j) \in CC \\
n_vc &= \text{maximum number of } \textit{virtual} \text{ circuits} \\
&= |CC| = \frac{nc!}{2! \cdot (nc-2)!} = \frac{nc \cdot (nc-1)}{2} \\
n_rvc &= n_vc + nc \\
VC &= \{c_i \mid i \in [1, n_vc]\} \\
VV &= \{(i,j) \ \in VC \times VC \mid i < j\} \\
RVC &= \{c_i \mid i \in [1, n_rvc]\}
\end{aligned}$$

The first definition needs deeper explanation: a rectangle can be defined giving the position of its bottom left and its top right corner. For a *virtual* circuit the corners are the bottom left corner of circuit i and the top right corner of circuit j. The maximum number of virtual circuits is then given by the number of possible combinations (without repetition) of the circuit indexes grouped in pairs, which is also the cardinality of CC.

3.3 Functions & Predicates

The additional functions and predicates we define in this section will later be used in the models that adopt symmetry breaking constraints or that allow the rotation of circuits. The other models work with the constraints already defined in Section 2.5.

Functions

$$\begin{aligned}
vc_x(vc_{i,j}) &= min(x_i, x_j) \\
vc_y(vc_{i,j}) &= min(y_i, y_j) \\
vc_width(vc_{i,j}) &= max(x_i + w_i, x_j + w_j) - vc_x(vc_{i,j}) \\
vc_height(vc_{i,j}) &= max(y_i + h_i, y_j + h_j) - vc_y(vc_{i,j}) \\
r_w(c) &= bool2int(\neg\, r_c) \cdot w_c + bool2int(r_c) \cdot h_c \qquad (7) \\
r_h(c) &= bool2int(\neg\, r_c) \cdot h_c + bool2int(r_c) \cdot w_c \qquad (8)
\end{aligned}$$

where $vc\_x(vc_{i,j})$, $vc\_y(vc_{i,j})$, $vc\_width(vc_{i,j})$ and $vc\_height(vc_{i,j})$ return respectively the $x_{vc_{i,j}}$, $y_{vc_{i,j}}$, $w_{vc_{i,j}}$ and $h_{vc_{i,j}}$ of the *virtual* circuit $vc_{i,j}$, while $r\_w(c)$ and $r\_h(c)$ check if circuit c is rotated and update coherently w_c and h_c.

Predicates

$$\text{vc\_valid}(vc_{i,j}) \leftarrow \bigwedge_{c \in C} (\neg(x_c < vc_x \wedge x_c + w_c > vc_x) \wedge \neg(x_c + w_c > vc_x + vc_w)) \vee \vee (\neg(y_c < vc_y \wedge y_c + h_c > vc_y) \wedge \neg(y_c + h_c > vc_y + vc_h)) \quad (9)$$

$$\text{c\_eq\_dim\_sym} \leftarrow \bigwedge_{(c_1,c_2) \in CC} (w_{c_1} == w_{c_2} \wedge h_{c_1} == h_{c_2}) \rightarrow lex\_lesseq([x_{c_1}, y_{c_1}], [x_{c_2}, y_{c_2}]) \quad (10)$$

$$\text{vc\_eq\_dim\_sym} \leftarrow \bigwedge_{(c,vc) \in C \times VC} (\text{vc\_valid}(vc) \wedge w_c == w_{vc} \wedge h_c == h_{vc}) \rightarrow \rightarrow lex\_lesseq([x_c, y_c], [x_{vc}, y_{vc}]) \quad (11)$$

$$\text{vv\_eq\_dim\_sym} \leftarrow \bigwedge_{(vc_1,vc_2) \in VV} (\text{vc\_valid}(vc_1) \wedge \text{vc\_valid}(vc_2) \wedge \neg(vc_{1x} == vc_{2x}) \wedge \wedge vc_{1w} == vc_{2w} \wedge vc_{1h} == vc_{2h} \wedge \wedge (|vc_{1x} - vc_{2x}| \geq vc_{1w} \vee |vc_{1y} - vc_{2y}| \geq vc_{1h})) \rightarrow \rightarrow lex\_lesseq([vc_{1x}, vc_{1y}], [vc_{2x}, vc_{2y}]) \quad (12)$$

$$\text{c\_x\_consec}(c_1, c_2) \leftarrow x_{c_1} + w_{c_1} == x_{c_2} \vee x_{c_2} + w_{c_2} == x_{c_1} \quad (13)$$

$$\text{c\_y\_consec}(c_1, c_2) \leftarrow y_{c_1} + h_{c_1} == y_{c_2} \vee y_{c_2} + h_{c_2} == y_{c_1} \quad (14)$$

$$\text{c\_on\_x\_swap}(c_1, c_2) \leftarrow \text{c\_x\_consec}(c_1, c_2) \wedge y_{c_1} == y_{c_2} \wedge h_{c_1} == h_{c_2} \quad (15)$$

$$\text{c\_on\_y\_swap}(c_1, c_2) \leftarrow \text{c\_y\_consec}(c_1, c_2) \wedge x_{c_1} == x_{c_2} \wedge w_{c_1} == w_{c_2} \quad (16)$$

$$\text{c\_consec\_sym} \leftarrow \bigwedge_{(c_1,c_2) \in CC} (\text{c\_on\_x\_swap}(c_1, c_2) \rightarrow lex\_less([x_{c_1}], [x_{c_2}])) \wedge \wedge (\text{c\_on\_y\_swap}(c_1, c_2) \rightarrow lex\_less([y_{c_1}], [y_{c_2}])) \quad (17)$$

As mentioned in Section 3.2, $vc_{i,j}$ defines a rectangle in the plate, but, in order to be defined as *virtual* circuit, its edges must not cross any *real* circuit. This condition is checked by the predicate $vc\_valid(vc_{i,j})$, where $vc_x = vc\_x(vc_{i,j})$, $vc_y = vc\_y(vc_{i,j})$, $vc_w = vc\_width(vc_{i,j})$, $vc_h = vc\_height(vc_{i,j})$. The predicates *c_eq_dim_sym*, *vc_eq_dim_sym*, *vv_eq_dim_sym* apply lexicographic order to circuits with same dimensionality; in particular the first predicate keeps in consideration only *real* circuits, the second a *real* circuit and a *virtual* circuit, the third only couples of *virtual* circuits. The last one must also check that the circuits vc_1 and vc_2 do not overlap, otherwise it would be in contrast with any lexicographic order constraint, making the solution unfeasible. Another possible case in which a couple of circuit can be swapped is when they have a shared side with same length; the last predicates are needed to catch those situations and apply to those couple of circuits the lexicographic order.

3.4 Constraints

3.4.1 Base

The reference model is the one with the constraints described in Section 2.5 with the following simple symmetry breaking constraint:

$$x_1 <= x_2 \wedge y_1 <= y_2$$

which is a simpler version of:

$$lex\_lesseq(x\_v, x\_v') \wedge lex\_lesseq(y\_v, y\_v')$$

where x_v, x_v', y_v, y_v' are defined in 6.

3.4.2 Rotation

In order to introduce rotations to the model described in Section 3.4.1 or any of the following CP models, we need to add rotation variables r_i, functions 7, 8. We also need to modify already existing constraints substituting w_c with $r\_w(c)$ and h_c with $r\_h(c)$.

3.4.3 Symmetry

As already discussed in Section 2.5.1, we decided to break only a few symmetries among all the ones we detected. The following are the constraints added to the models in order to deal with symmetries.

First of all we removed the solutions specular w.r.t. the horizontal and vertical axis [Fig.3]:

$$lex\_lesseq(x\_v, x\_v') \wedge lex\_lesseq(y\_v, y\_v')$$

with x_v, x_v', y_v, y_v' already defined in 6.
Then we avoid swapping of circuits with same dimensions [Fig.2, 5a], which obviously lead to a different solution, but with same *makespan*:

$$\text{c\_eq\_dim\_sym} \wedge \text{vc\_eq\_dim\_sym} \wedge \text{vv\_eq\_dim\_sym}$$

As demonstration of the inefficiency of *virtual* circuit constraint, we will show separately in the results [Section 3.6] a model with only *c_eq_dim_sym* and a model with also *vc_eq_dim_sym* $\wedge$*vv_eq_dim_sym*.

The case we simplified is the one figured in [Fig.5b]: instead of applying lexicographic order to all *real* circuits within a generic *virtual* circuit, we decided to limit ourself to *virtual* circuits containing only two *real* circuits. We can see this like avoiding adjacent circuits, having the shared edge with same length, to swap their position:

$$\text{c\_consec\_sym} \leftrightarrow \top$$

in order to have better understanding of the predicates above, recover Section 3.3.

3.5 Search

The performances of all the models are compared with and without the search strategy we selected. Actually in the input of all models the circuits are sorted in decreasing order according to their area, but then we also implemented a sequential search [2] with the following search annotations, in this specific order:

[2] https://www.minizinc.org/doc-2.5.5/en/mzn_search.html#search-annotations

1. ann_search_makespan = int_search([*makespan*], input_order, indomain_split)
2. ann_search_x = int_search(x, input_order, indomain_min)
3. ann_search_y = int_search(y, input_order, indomain_min)

Models with both rotation and search strategies at the end of the sequential search list also have:

4. ann_search_rot = bool_search($[r_c \mid c \in C]$, input_order, indomain_min)

To avoid getting stuck during the search of the solution we added also luby restart strategy [3], choosing empirically the value of parameter *scale*, where *scale* is an integer defining after how many nodes to restart. For Luby restart the *k-th* restart gets $scale \cdot L[k]$ where $L[k]$ is the *k-th* number in the Luby sequence.

3.6 Results

Hardware specifcations All the experiments for the CP technology have been executed on a laptop computer running Linux Mint 20.3, Linux kernel 5.15 equipped with the following hardware: `Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz, 16Gb Ram 2.4GHz`.

The results obtained for the *base* models are plotted in the Figures [6,7]. The performances of the base model (Section 3.4.1) are compared with:

- *base search*: base model with search strategy described in Section 3.5
- *base symmetry*: base model with symmetry breaking constraints described in 3.4.3 (no *virtual* circuits)
- *base search symmetry*: *base symmetry* model with search strategy

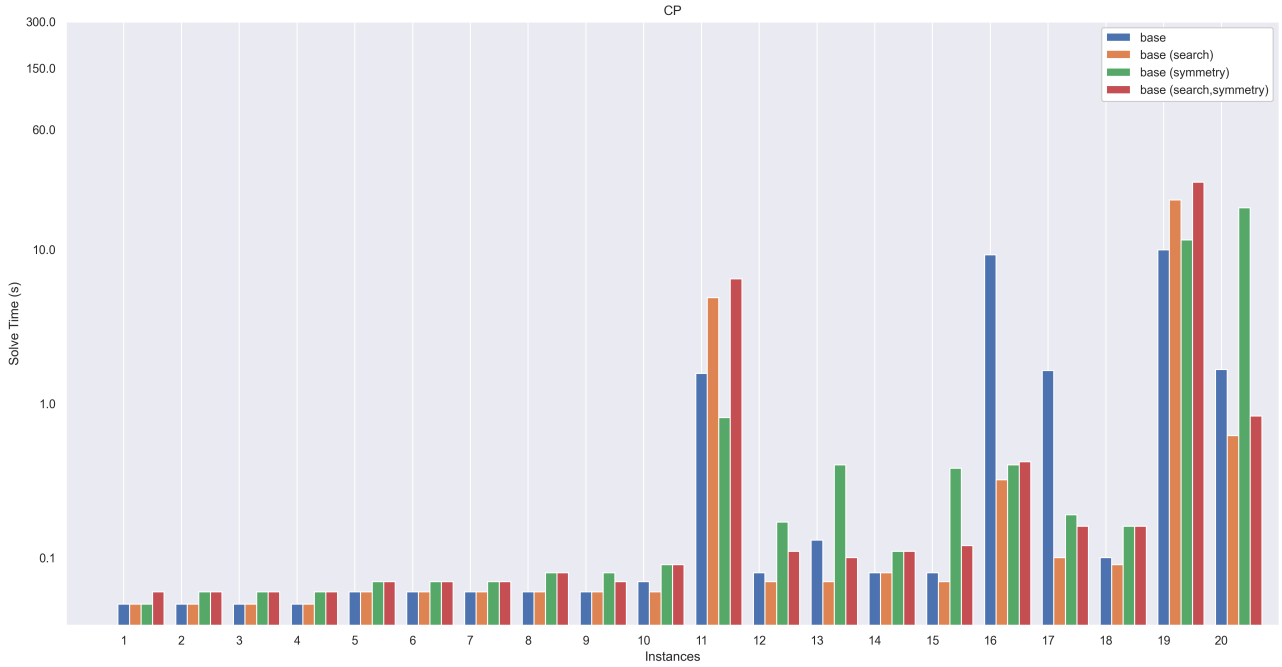


Figure 6: CP Results obtained with base model - base search - base symmetry - base search symmetry - instances [1,20]

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

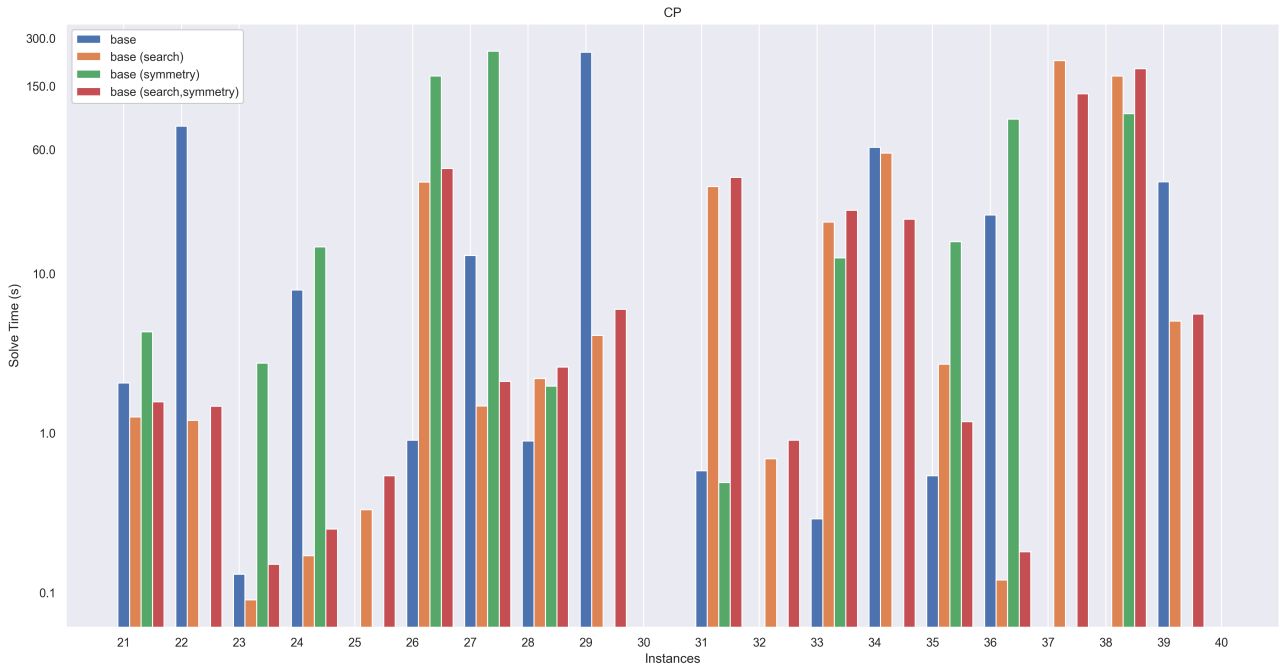


Figure 7: CP Results obtained with base model - base search - base symmetry - base search symmetry - instances [21,40]

The results are taken fixing the random seed, but should be taken in consideration that changing seed the solve time of the same model on the same instance may vary a lot. As consequence we will describe only the most evident trends.

The first 20 instances do not show any particular difference in performances between the models. We can already notice an increasing behaviour of solve time, but with some peaks for "random" instances showing how the solve time does not depend only on the number of circuits to place.

The second 20 instances are not solved by all models, showing the first difference: the model solving most of the instances is *base search* and if we compare *base symmetry* with *base symmetry search* we have the proof that the search and restart strategies are quite effective. We can also notice that most of the time adding the symmetry breaking constraints that we defined in Section 3.4.3 make the performances worse, confirming what already said.

The results obtained for the *rotation* models are plotted in the Figures [8,9]. The performances of the rotation model (Section 3.4.2) are compared with:

- *rotation search*: rotation model with search strategy described in Section 3.5
- *rotation symmetry*: rotation model with symmetry breaking constraints described in 3.4.3 (no *virtual* circuits)
- *rotation search symmetry*: *rotation symmetry* model with search strategy

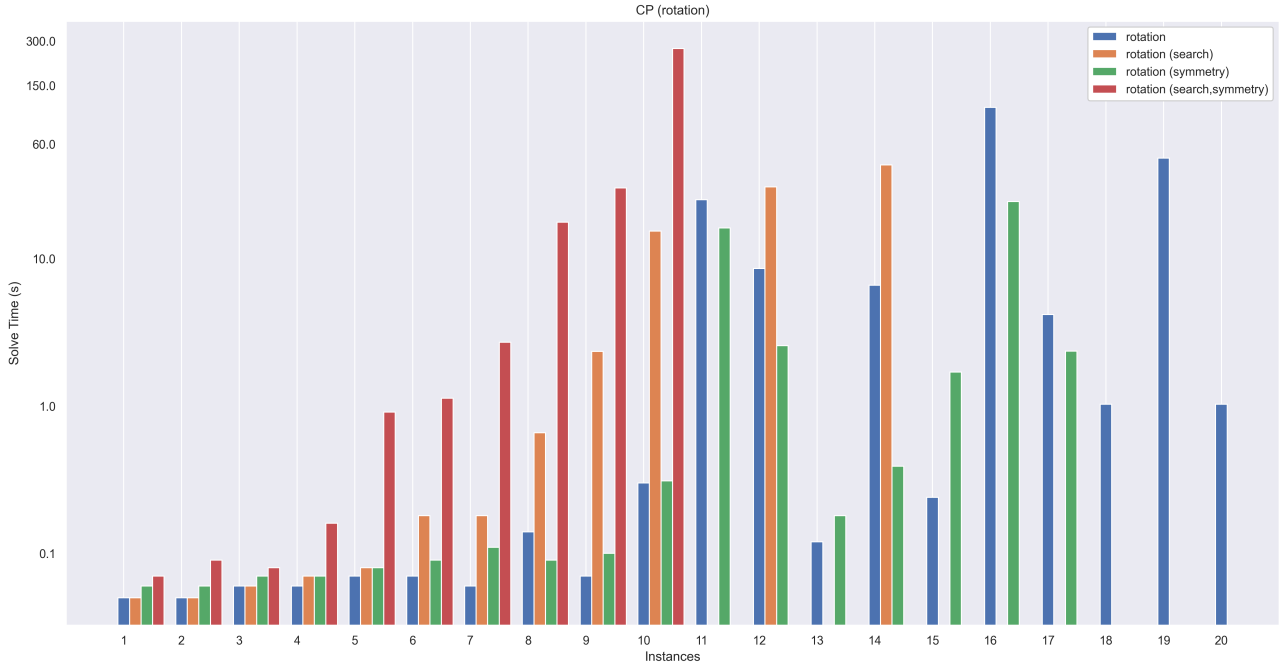


Figure 8: CP Results obtained with Rotation model - rotation search - rotation symmetry - rotation search symmetry - instances [1,20]

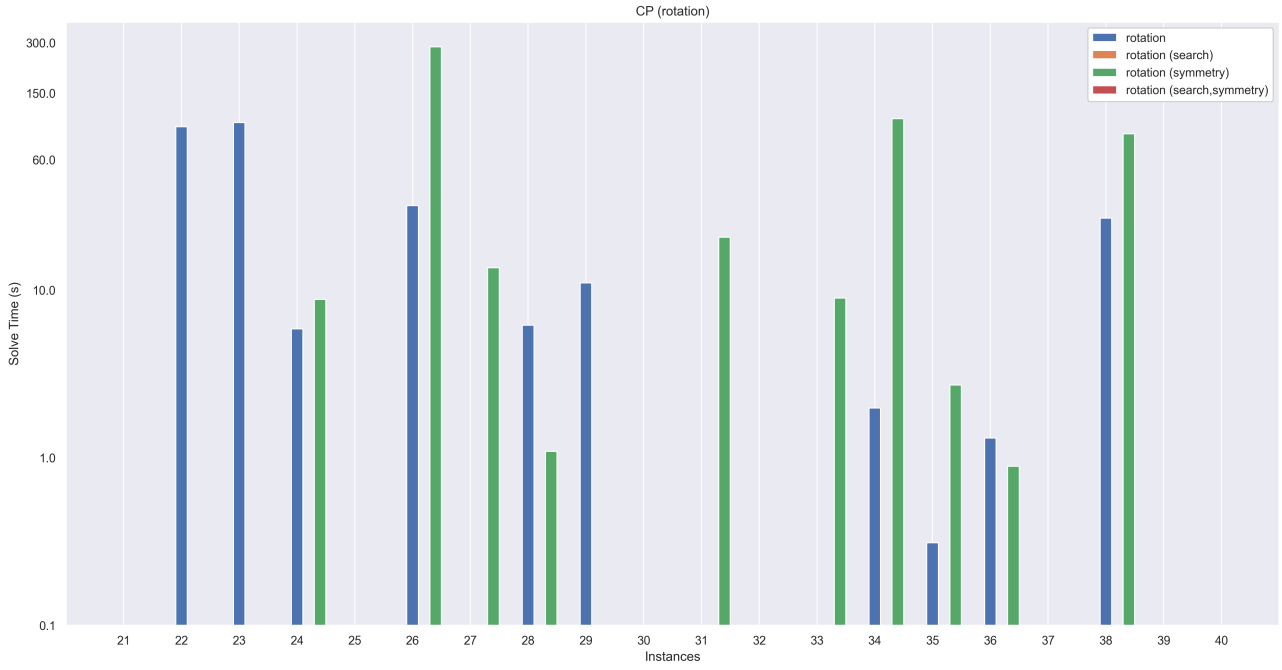


Figure 9: CP Results obtained with Rotation model - rotation search - rotation symmetry - rotation search symmetry - instances [21,40]

The results obtained for *rotation* models are quite different from the ones presented before before: we still have an increasing solve time, but the first main difference is the impact of search strategies. While, with *base* models, search strategies improved the efficiency, with *rotation* model is the opposite. We were not expecting any significant impact from the search strategy for the boolean variables r_i

since their domain is highly limited and random search was not even implemented. After checking the solutions and seeing that most of the *makespan* values (in *base* models) match with our definition of *min_makespan*, we may hazard that the dimensions of the circuits have been designed to e perfectly placed over the plate without neither empty spaces nor rotation.

The second main difference from *base* models is that in this case symmetry breaking constraint are more competitive. Maybe *base* model was too simple and adding more constraints (referring to the symmetry breaking constraints) to compute was not balanced by the number of truncated solutions during the search.

The comparisons between different solvers (Chuffed and Gecode) are plotted in the Figures [10,11]. The models we selected for the comparison are:

- *base*: base model described in Section 3.4.1
- *base search symmetry*: base model with search strategy (Section 3.5) and symmetry breaking constraints (Section 3.4.3, no *virtual* circuits)

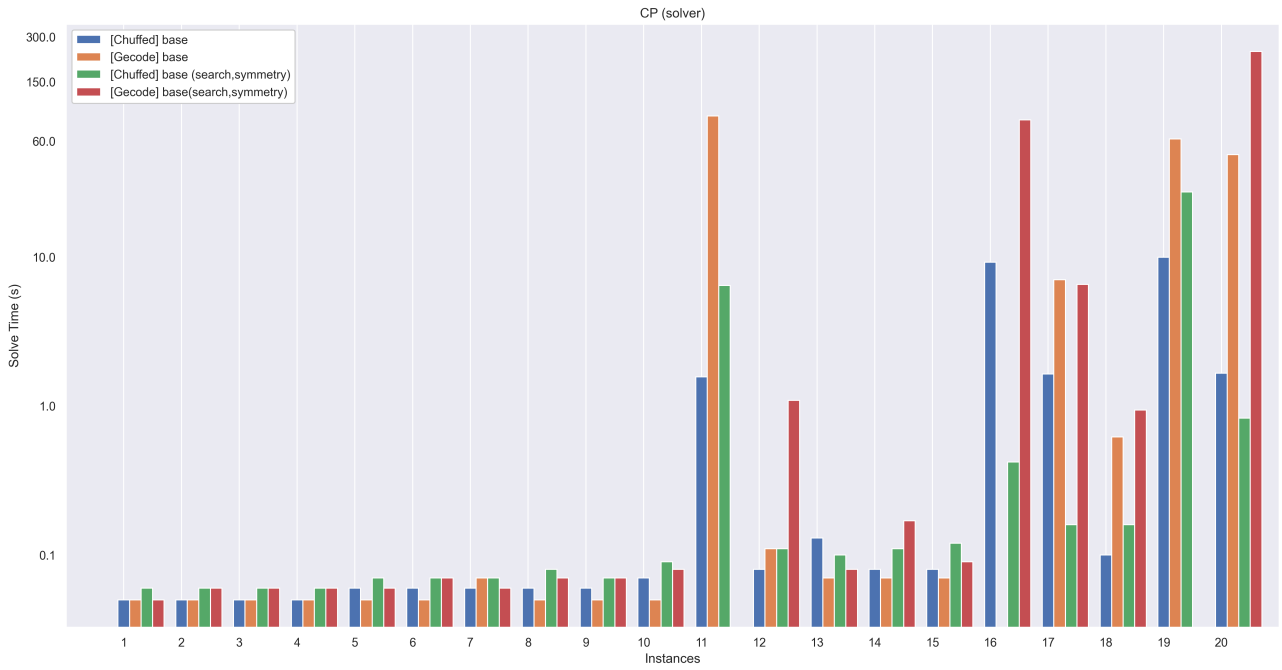


Figure 10: CP Results obtained with base model - base search symmetry - instances [1,20]. Both with Chuffed and Gecode solver.

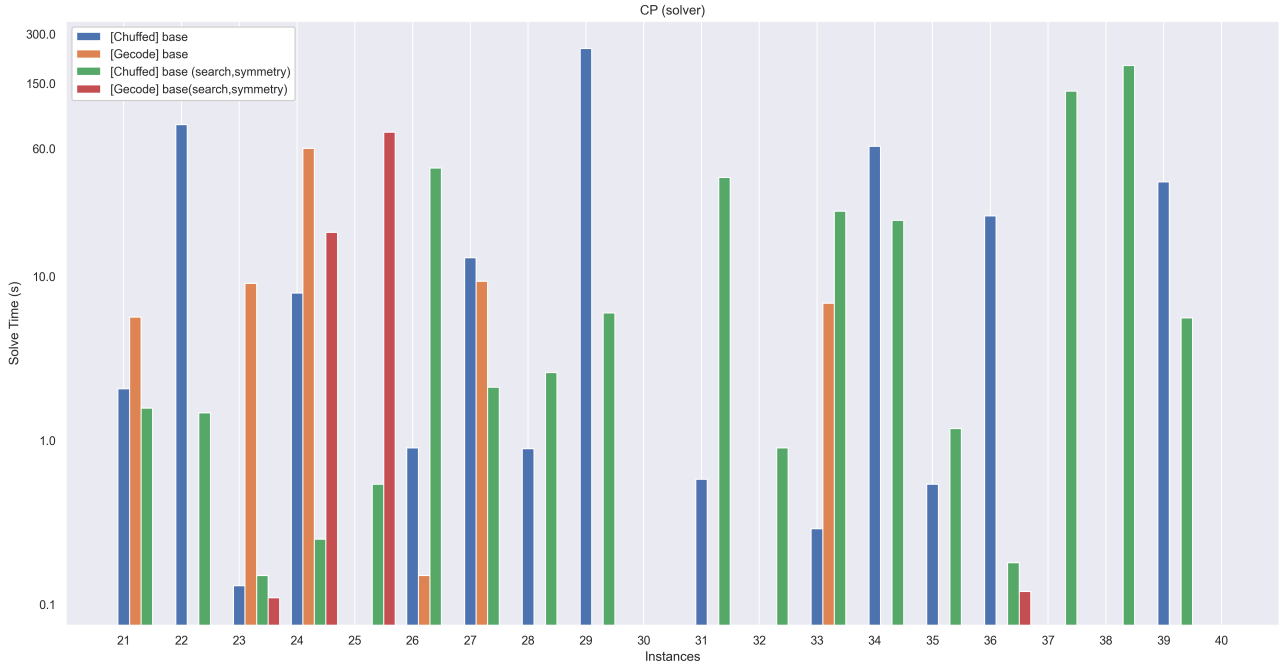


Figure 11: CP Results obtained with base model - base search symmetry - instances [21,40]. Both with Chuffed and Gecode solver.

For both the models, Chuffed is the one solving the higher number of instancess, most of the times with lower solver time.

The following plot show how much slower becomes a model when also *virtual* circuit are included in the symmetry breaking constraints, stopping us in implementing constraints even more complex than the ones shown in Section 3.4.3.

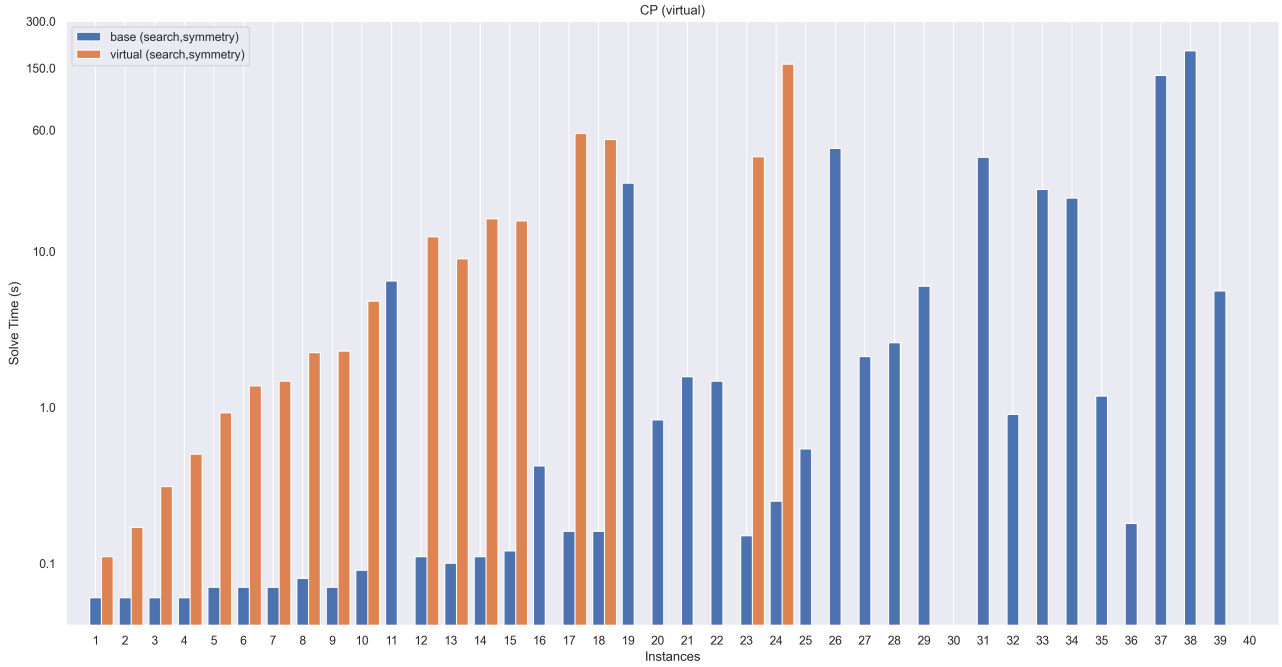


Figure 12: CP Results obtained with base search symmetry with and without virtual circuits - instances [1,40].

4 SAT

4.1 Background

Representing integers with list of booleans SAT problems are based only on boolean variables. Hence, one of the issues is that we have to represent positive integer numbers. One-hot encoding is a possible solution for representing integers using only boolean variables.
How does one-hot encoding work?
For every integer variable we declare one boolean variable for every item of the domain of the numeric variable. Only one boolean per integer variable is set to `True`. The only `True` variable corresponds to the value of the domain that is being represented.

This solution leads to a simple implementation of the operations but is not very efficient because it requires the SAT solver to manage a very big quantity of variables and clauses. We decided to use decimal-to-binary encoding.

Let *domain_max* be the maximum value of the domain of a positive integer variable. The number of boolean variables allocated to represent it is:

$$var\_size = \lceil \log_2(domain\_max + 1) \rceil$$

We have a list of boolean variables that take the value of the corresponding bit of the binary encoding of the positive integer. We implemented specific functions to perform mathematical operations. They are discussed in Section 4.3.

Implementation of the Models We decided to create a default abstract model that has the common parts needed by every concrete model. The abstract model is extended by the Base model and the Rotation model. The details of the Rotation model are discussed in Section 4.5

In order to implement the SAT formulation we have used the solver called `z3`.

4.2 Variables

The variables for the SAT solver are lists of boolean variables but their meaning does not differ from the definitions in 2.3. The size of the lists is calculated to be minimal but safe against overflows of the possible operations.

$$x\_domain\_size = \lceil \log_2(width - \min_{0 \le i < nc}(w_i) + \max_{0 \le i < nc}(w_i) + 1) \rceil$$

x_domain_size is the length of each x_i list of booleans.

$$y\_domain\_size = \lceil \log_2(max\_makespan - \min_{0 \le i < nc}(h_i) + \max_{0 \le i < nc}(h_i) + 1) \rceil$$

y_domain_size is the length of each y_i list of booleans.

4.3 Predicates and Functions

We created several support functions for dealing with the conversion from integers to lists of boolean variables and back.

If an integer value is provided than it is converted to a list of python `bool` variables that are consistent with `z3 Bool` variables and can be used to perform calculations.

In this way we achieve the transparency of the mathematical functions for the type of operand used. The length of the lists of booleans is calculated upon the maximum value of the domain of the variable. Padding of the shortest list is performed if necessary in case of operations between two values of different domain sizes.

All the bits of a list set to False

$$all\_F(list) = \bigwedge_{b \in list} \neg\, b$$

At least one bit of a list must be set to True

$$at\_least\_one(list) = \bigvee_{b \in list} b$$

At most one
We implemented the Heule encoding making this a recursive function.
The base case of the recursion is if the list has less than four elements. In this case the pair-wise encoding is used to calculate the result.

Let n be the length of *list*

$$(n < 4) \Longrightarrow at\_most\_one(list) = \bigwedge_{0<i<n} \bigwedge_{i+1 \leq j \leq n} \neg\, (b_i \wedge b_j)$$

In the recursive case we split *list* into two parts: $list_A$ has length two, $list_B$ is the remainder. We introduce a new variable y. We append y to $list_A$ and $\neg y$ to $list_B$. We than calculate the result through recursion:

$$(n \geq 4) \Longrightarrow at\_most\_one(list) = at\_most\_one(list_A) \wedge at\_most\_one(list_B)$$

We chose this encoding because it achieves a number of clauses of $3n-6$ using $(n-3)/2$ new variables.

Exactly one

$$exactly\_one(list) = at\_most\_one(list) \wedge at\_least\_one(list)$$

Not equal
We must consider the cases where the two lists have different lengths. Let *min_len* be the minimum between the length of the lists, exc_1 be the part of $l1$ exceeding *min_len* and exc_2 be the part of $l2$ exceeding *min_len*. At least one among exc_1 and exc_2 will be an empty list and will provide no clauses.

$$ne(l1, l2) = at\_least\_one(exc_1) \vee at\_least\_one(exc_2) \ \vee \bigoplus_{i \leq min\_len} (l1_i, l2_i)$$

Equal
One of the lists is eventually padded with bits set to `False` than we can calculate the result:

$$eq(l1, l2) \ = \bigwedge_{0 \leq i < n} \neg\, (l1[i] \oplus l2[i])$$

Greater than or equal
We must consider the cases where the two lists have different lengths. Let min_len be the minimum between the length of the lists, exc_1 be the part of $l1$ exceeding min_len and exc_2 be the part of $l2$ exceeding min_len. At least one among exc_1 and exc_2 will be an empty list and will provide no clauses.

$$gte(l1, l2) = at\_least\_one(exc_1) \vee (all\_F(exc_2) \wedge gte\_same\_len(l1\_cut, l2\_cut))$$

Where $l1\_cut$ and $l2\_cut$ are the result of pruning $l1$ and $l2$ at length min_len.

Greater than or equal of numbers represented with the same quantity of bits
The encoding used is the AND encoding with Common Subexpression Elimination as described in [1] and [2].

Given n as the length of the lists $l1$, $l2$, we have for the case $n = 1$:

$$gte\_same\_len(l1, l2) = l1_0 \vee \neg\, l2_0$$

otherwise, let k be a list of boolean variables of length $n - 1$

$$first = l1_0 \vee \neg\, l2_0$$

$$second = k_0 \iff \neg\, (l1_0 \oplus l2_0)$$

$$third = k_{i+1} \iff k_i \wedge \neg\, (L1_{i+1} \oplus l2_{i+1}) \quad \forall i \in [0, n-3]$$

$$fourth = k_i \iff (l1_{i+1} \vee \neg\, l2_{i+1}) \quad \forall i \in [0, n-2]$$

We chose this encoding because of the considerations of [2] among other linear encodings.

Greater than
We must consider the cases where the two lists have different lengths. Let min_len be the minimum between the length of the lists, exc_1 be the part of $l1$ exceeding min_len and exc_2 be the part of $l2$ exceeding min_len. At least one among exc_1 and exc_2 will be an empty list and will provide no clauses.

$$gt(l1, l2) = at\_least\_one(exc_1) \vee (all\_F(exc_2) \wedge gt\_same\_len(l1\_cut, l2\_cut))$$

Greather than of numbers represented with the same quantity of bits
The encoding used is the AND encoding with Common Subexpression Elimination. We slightly modified the implementation of [2] to exclude the equality.
Len n be the length of the lists $l1$, $l2$.
In the case $n = 1$

$$gt\_same\_len(l1, l2) = l1_0 \wedge \neg\, l2_0$$

otherwise, let k be a list of boolean variables of length $n - 1$

$$first = l1_0 \wedge \neg\, l2_0$$

$$second = k_0 \iff \neg\, (l1_0 \oplus l2_0)$$

$$third = \bigwedge_{0<n-2} k_{i+1} \iff k_i \wedge (\neg\, (l1_{i+1} \oplus l2_{i+1})) \quad \forall i \in [0, n-3]$$

$$fourth = \bigvee_{0 \le i < n-1} k_i \wedge (l1_{i+1} \wedge \neg\, l2_{i+1}) \quad \forall i \in [0, n-2]$$

$$gt\_same\_len(l1, l2) = first \vee (second \wedge third \wedge fourth)$$

Less than

$$lt(l1, l2) \;=\; gt(l2, l1) \qquad \text{;notice the reverse order of the parameters}$$

Less than or equal

$$lte(l1, l2) \;=\; gte(l2, l1) \qquad \text{;notice the reverse order of the parameters}$$

Sum of two numbers
Padding is performed on the shortest list if need be. Let n be the length of the lists, eventually padded. Let *carry* be a list of boolean variables of length n. $carry_0$ is set to `false`.

$$carry_{i+1} = ((l1_i \oplus l2_i) \wedge carry_i) \vee (l1_i \wedge l2_i) \quad \forall i \in [0, n]$$

$$sum\_b(l1, l2) = (l1_i \oplus l2_i) \oplus carry_i \quad \forall i \in [0, n-1]$$

The result is a list that represents the sum of the numbers. $carry_{n-1}$ could be used to notify an overflow but we did not use it because the domains size are designed to avoid this.

Subtraction of two numbers
Padding is performed on the shortest list if need be. Let n be the length of the lists, eventually padded. Let *borr* be a list a list of boolean variables of length n. $borr_0$ is set to `false`

$$sub\_b(l1, l2) = (l1_i \oplus l2_i) \oplus borr_i \quad \forall i \in [0, n-1]$$

$$borr_{i+1} = ((\neg\ (l1_i \oplus l2_i) \wedge borr_i) \vee (\neg\ l1_i \wedge \quad l2_i)) \quad \forall i \in [0, n-1]$$

4.4 Constraints

The constraints of the base model are the ones defined in Section 2.5. We use functions defined in the previous Section 4.3 to implement the constraints.

The following Table contains the mapping between the operators used in Section 2.5 to define the constraints and the functions of Section 4.3 to implement them in the SAT model.

Math operator	←	SAT function equivalent
$a \geq 1$	←	at_least_one(a)
$a \leq 1$	←	at_most_one(a)
$a = 1$	←	exactly_one(a)
$a = b$	←	equal(a, b)
$a < b$	←	$lt(a, b)$
$a \leq b$	←	$lte(a, b)$
$a > b$	←	$gt(a, b)$
$a \geq b$	←	$gte(a, b)$
$a + b$	←	$sum\_b(a, b)$
$a - b$	←	$sub\_b(a, b)$

4.4.1 Static constraints

The process of finding the best solution is iterative. Some constraints do not need to be modified along the iteration process because the values on which they are based do not change during the iteration. They are constraints about the horizontal direction of the plate.

diff_n
The diff_n constraint, defined in Section 2.5, is implemented as follows.

$$\bigwedge_{(i,j)\in CC} = lte(sum\_b(x_i, widths_i), x_j) \vee lte(sum\_b(y_i, heights_i), y_j) \vee lte(sum\_b(x_j, widths_j), x_i) \vee lte(sum\_b(y_j, heights_j), y_i)$$

Circuits must not be placed over the side of the board The constraint is defined in Section 2.5. For greater clarity the implementation of the constraint is provided:

$$\bigwedge_{c\in C} lte(x_c, sub_b(width, widths_c))$$

cumulative_y Optionally we can impose a cumulative constraint along the horizontal axis, as defined in Section 2.5. This is an implied constraint.

$$cumulative\_y$$

Symmetry breaking constraint Optionally we can impose a constraint to break symmetries between circuits along the horizontal axis.

$$axial\_symmetry\_x = lte(x_i, sub\_b(sub\_b(width, x_i), w_i) \quad \forall i \ in[0, nc]$$

4.4.2 Dynamic constraints

Dynamic constraints are the constraint whose values change during the iteration. We need to update their limits when we perform the optimization. They are the constraints about the height of the board. We use $z3.push()$ at the beginning of every loop and z3.pop() at the end to avoid jamming the solver with out of date constraints.

Circuits must not be placed over the current limit of height The constraint is defined in Section 2.5. For greater clarity the implementation of the constraint is provided:

$$\bigwedge_{c\in C} lte(y_c, sub\_b(target\_makespan, heights_c))$$

With *target_makespan* is intended the makespan that we are currently searching a solution for.

cumulative_x Optionally we can impose a cumulative constraint along the vertical axis, as defined in Section 2.5. This is an implied constraint.

$$cumulative\_x$$

Symmetry breaking constraint Optionally we can impose a constraint to break symmetries between circuits along the vertical axis. The constraint to be applied is:

$$axial\_symmetry\_y = lte(y_i, sub\_b(sub\_b(target\_makespan, y_i)), h_i) \quad \forall i \ in[0, nc]$$

With *target_makespan* is intended the makespan that we are currently searching a solution for.

4.5 Rotation

We tackled the problem of rotating the circuit on the board by extending the base model. Every extention provided by the model for the rotations is presented in this Section.

$$domain\_size = \lceil \log_2(\max_{0 \leq i < nc}(\max(x_i, y_i)) + 1) \rceil$$

When rotation is available all the items of h and w must have the same length. *domain_size* is the number of bits that we use to represent every w_i and h_i, $is\_rotated_i$ = is c_i rotated or not, w_b_i list of length *domain_size* of booleans to represent the effective horizontal size of circuit i

$$w\_b_i = (is\_rotated_i \wedge w_i) \vee (\neg\ is\_rotated_i \wedge h_i) \quad \forall i \in [0, domain\_size - 1]$$

h_b_i list of length *domain_size* of booleans to represent the effective vertical size of circuit i

$$h\_b_i = (is\_rotated_i \wedge h_i) \vee (\neg\ is\_rotated_i \wedge w_i) \quad \forall i \in [0, domain\_size - 1]$$

4.6 Search

SAT itself does not perform an optimization of the solution that provides. The optimization is made outside from the SAT solver. Initially we have a lower and an upper bound for *makespan*, that is the value that we want to minimize.

We have to perform a search on the space $[min\_makespan, max\_makespan]$. This is done by performing the solution of the SAT problem in a loop, updating the dynamic constraints at every iteration.

The choice of what *target_makespan* to use every time is done according to a search strategy. We implemented two possible search strategies: linear search starting from *min_makespan* and binary search but starting from *min_makespan* in this case too. The latter starts from *min_makespan* and then performs a binary search over the remaining search space.

A comparison between the performances of the two is made in Section 4.7

4.7 Results

Hardware specifcations All the experiments for the SAT technology have been executed on a laptop computer running Linux Mint 20.3, Linux kernel 5.15 equipped with the following hardware: `Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz, 16Gb Ram 2.4GHz`.

Observations We can see how all of the models are able to solve the large part of the instances withint the 5 minutes time limit, despite the modest computation power provided. The linear complexity of the used encodings is the key to the success of the model.

The symmetry breaking is not of any help in this model because it adds more clauses and more to be elaborated and satisfied. The same holds for the implied cumulative constraint. Despite not introducing any new literal it nearly always has a slowing effect. It is interesting how the combination of the symmetry breaking constraints and of the implied cumulative constraint obtains an overall performance very similar to the only cumulative constraint one.

In nearly all the solutions of the proposed instances the circuits can be fitted in order to achieve a flat upper overll margin and no empty space is left between circuits. This is the reason why the *min_makespan* nearly always prove to be the final *makespan* of the solution.In this circumstances a sequential search from the bottom to the top performs better than any other search strategy. In a more general situation, where empty spaces may have to be left in between circuits, the binary search strategy could be more effective than the linear one.

With no surprise the complexity of the problem arises much more faster when we allow the circuits to rotate but the SAT model dedicate to the problem supported well the effort.

Base model The results obtained with the Base model are in Figures [13, 14] for linear search and in Figures [15, 16] for binary search.

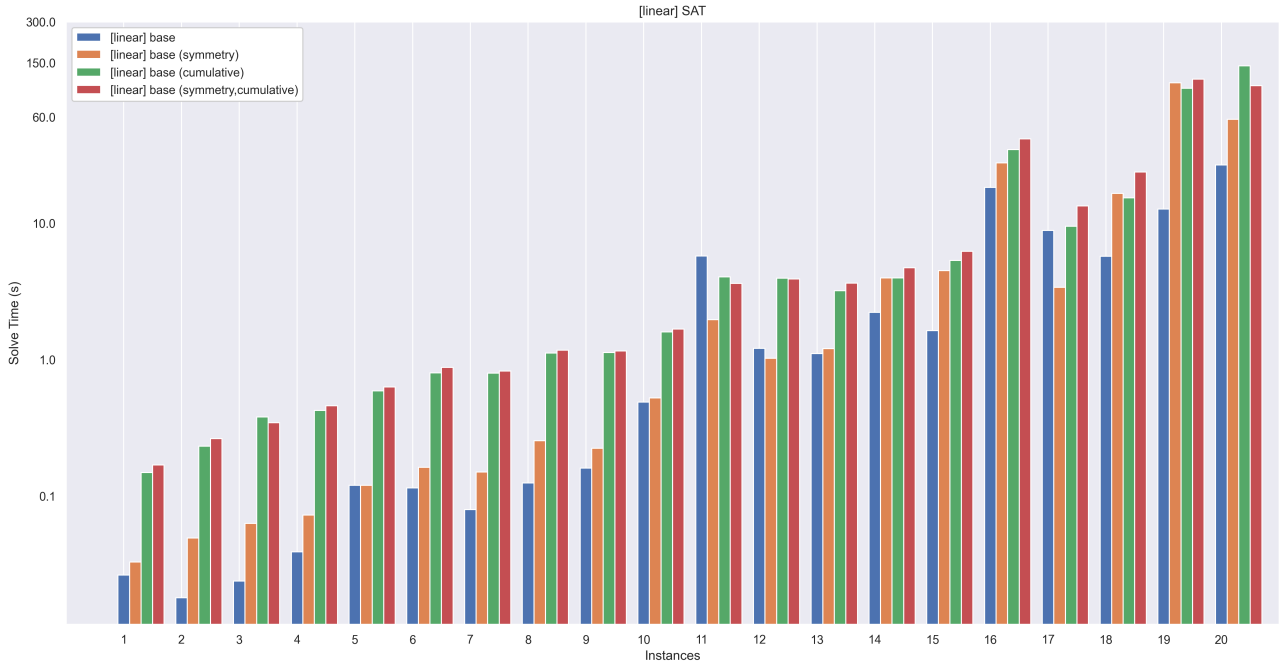


Figure 13: SAT Results obtained with Base model - linear search - instances [1,20]

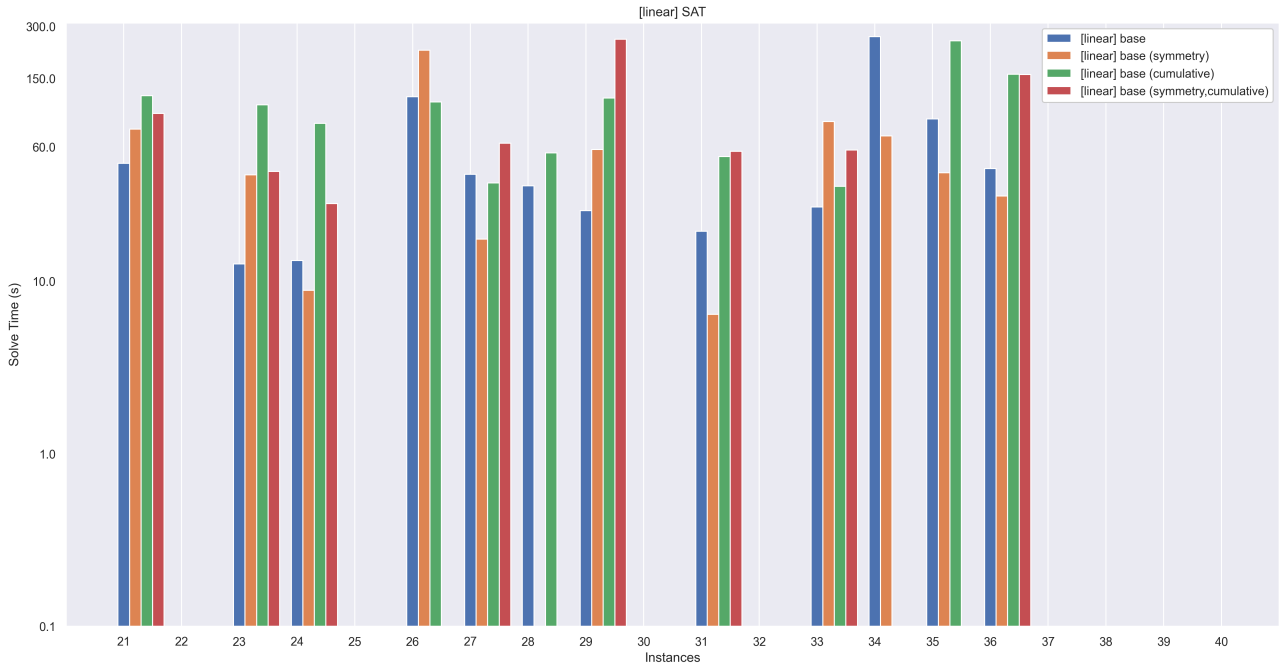


Figure 14: SAT Results obtained with Base model - linear search - instances [21,40]

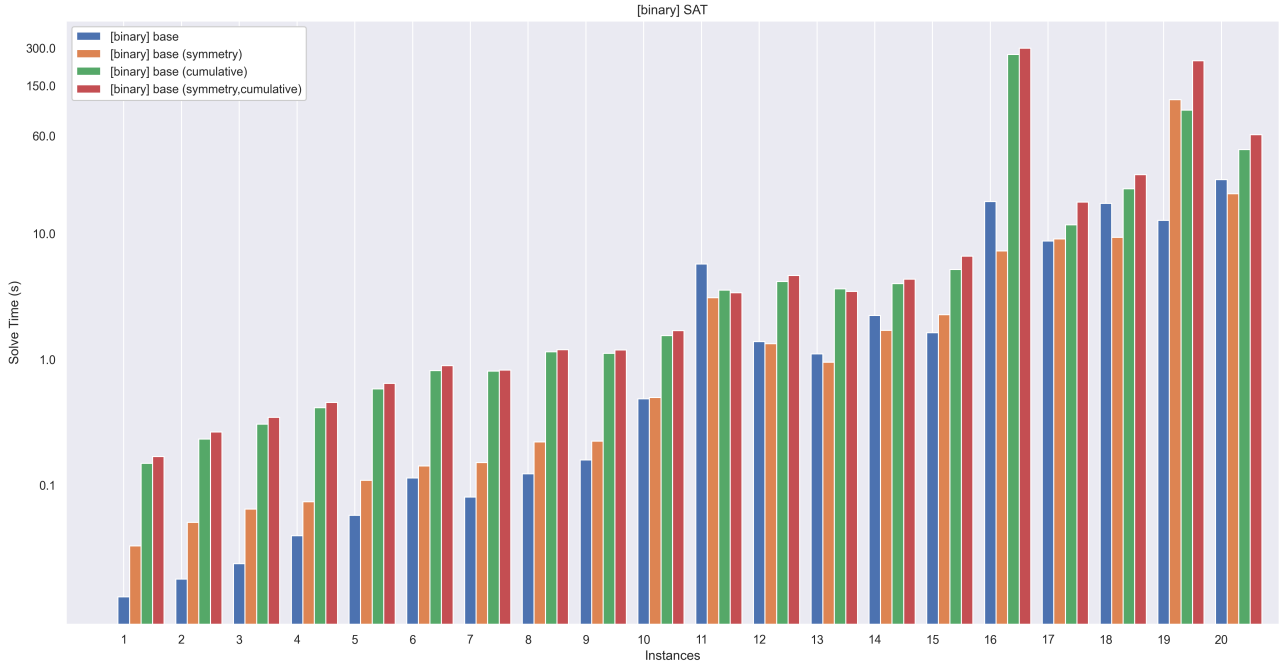


Figure 15: SAT Results obtained with Base model - Binary search - instances [1,20]

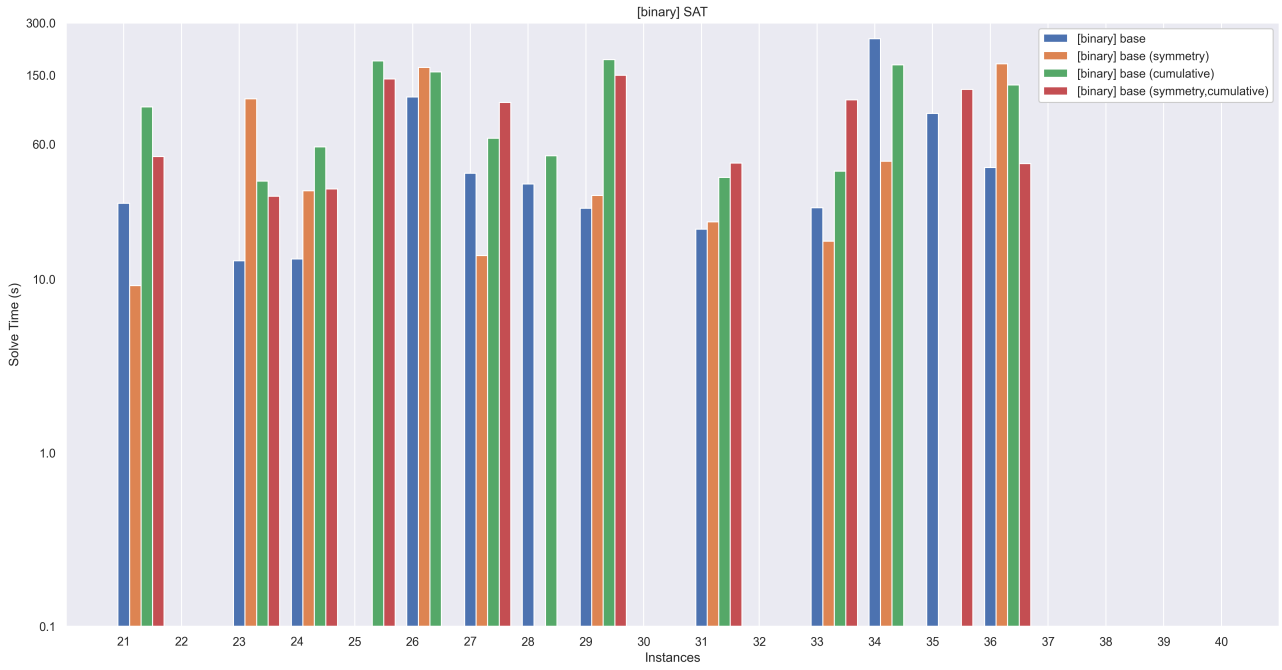


Figure 16: SAT Results obtained with Base model - binary search - instances [21,40]

Rotation model The results obtained with the Rotation model are in Figures [17, 18] for linear search and in Figures [19, 20] for binary search.

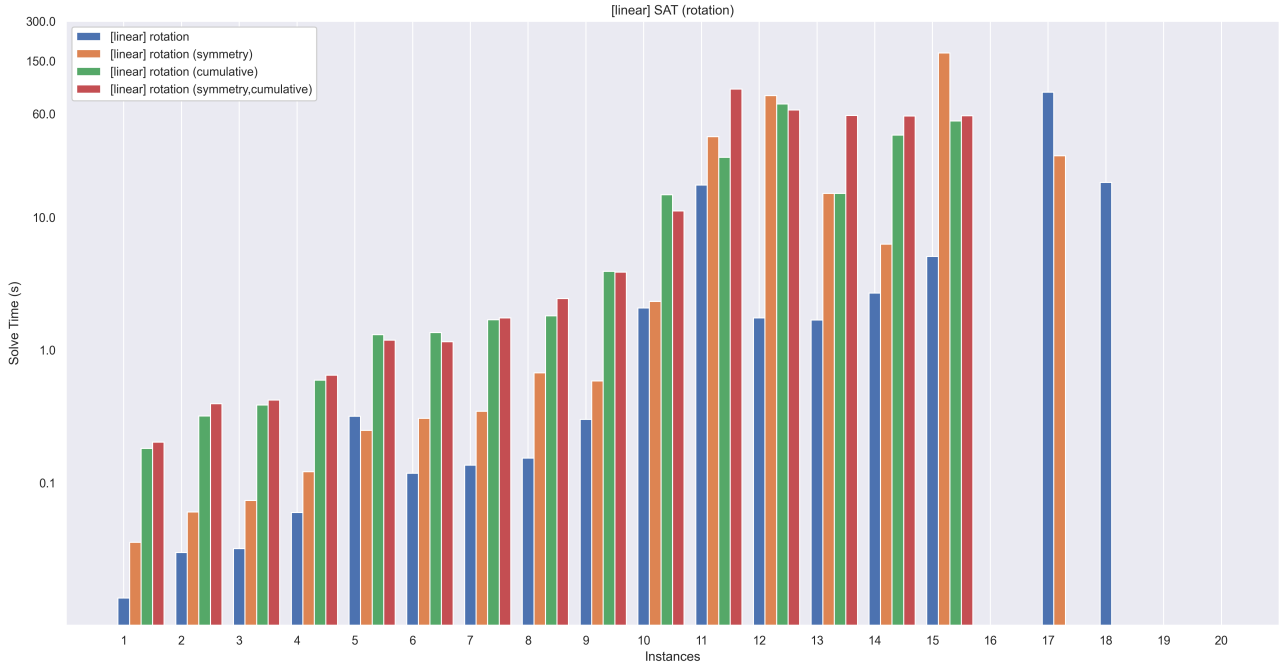


Figure 17: SAT Results obtained with Rotation model - linear search - instances [1,20]

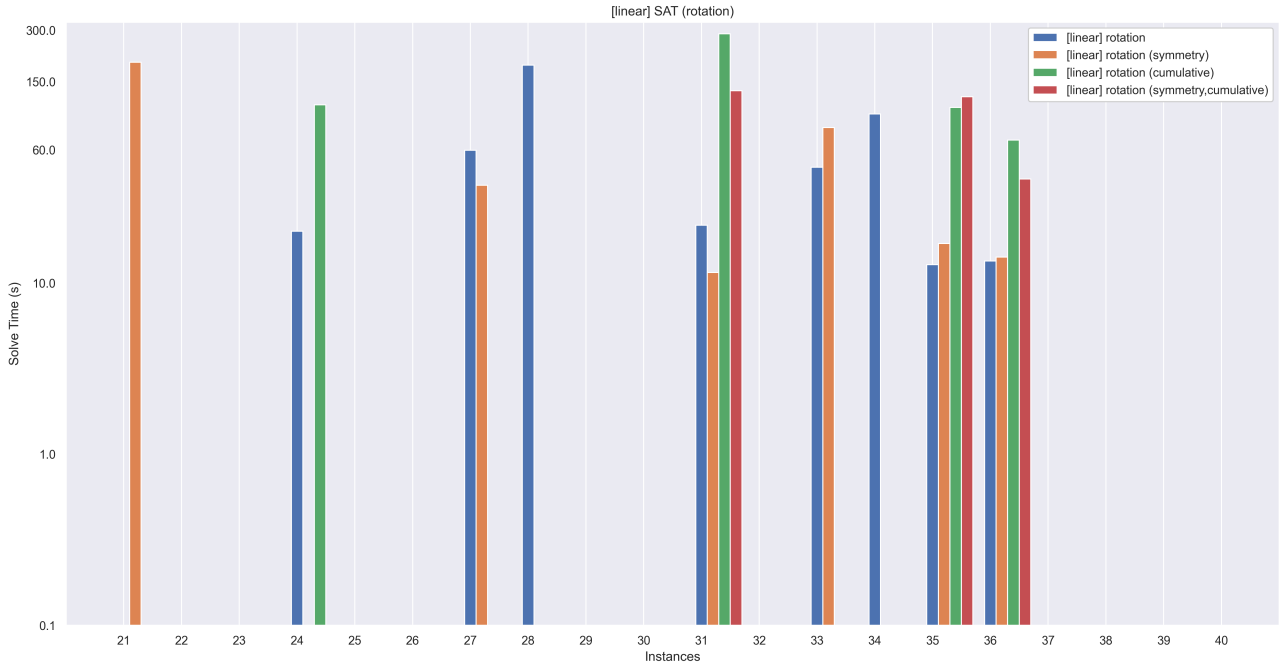


Figure 18: SAT Results obtained with Rotation model - linear search - instances [21,40]

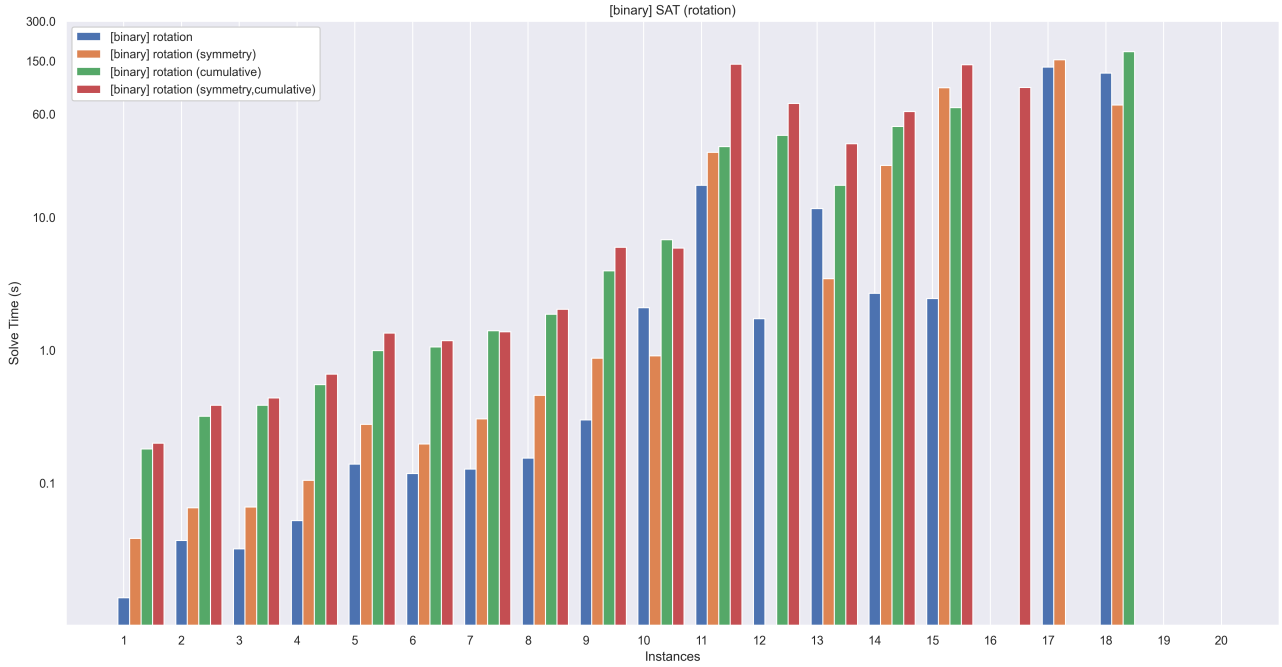


Figure 19: SAT Results obtained with Rotation model - binary search - instances [1,20]

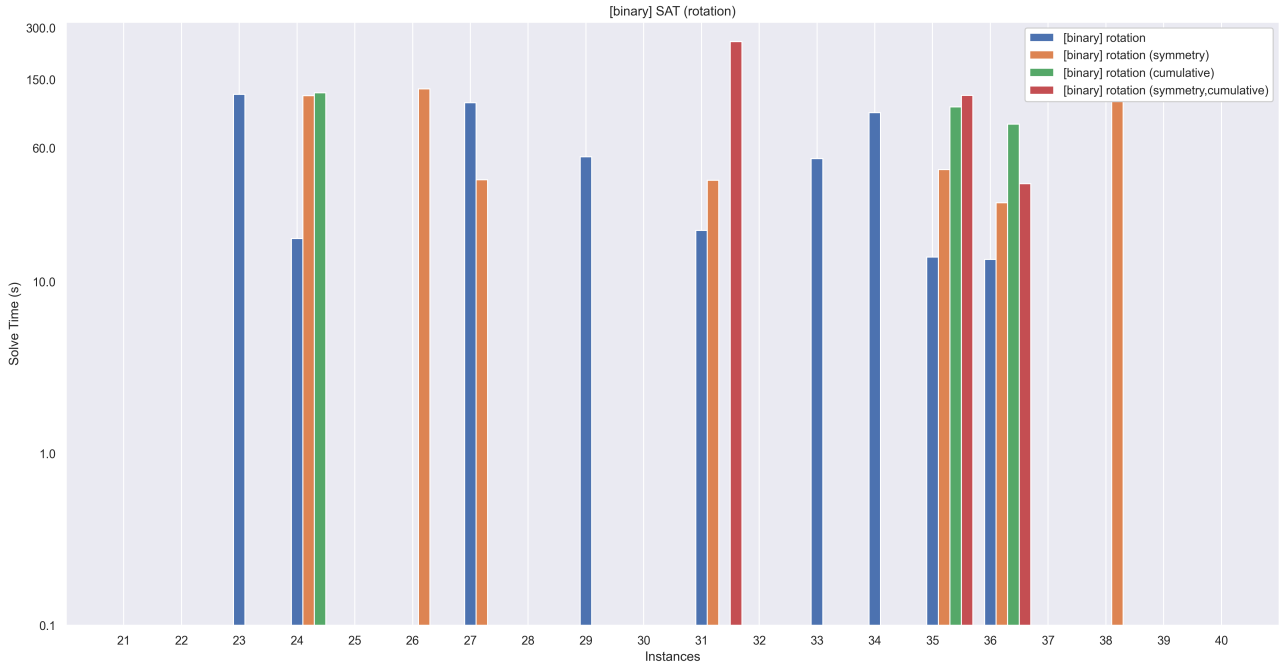


Figure 20: SAT Results obtained with Rotation model - binary search - instances [21,40]

5 SMT

5.1 Background

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 involving real numbers, integers, and/or various data structures such as lists, arrays, bit vectors, and strings.

5.2 Formulation

In order to write a proper SMT formulation, we started from the SAT formulation (available in the Section 4) and we have replaced each boolean representation of integer operators with the operators themself.

In particular, the following replacements have been done:

SMT	←	SAT
$a \geq 1$	←	at_least_one(a)
$a \leq 1$	←	at_most_one(a)
$a = 1$	←	exactly_one(a)
$a = b$	←	equal(a, b)
$a < b$	←	$lt(a, b)$
$a \leq b$	←	$lte(a, b)$
$a > b$	←	$gt(a, b)$
$a \geq b$	←	$gte(a, b)$
$a + b$	←	$sum\_b(a, b)$
$a - b$	←	$sub\_b(a, b)$

The final formulation obtained is very "near" to the one defined for CP in the Section 3.

5.3 Results

All the experiments for the SMT technology have been executed on a laptop computer running `macOS Monterey 12.3` equipped with the following hardware: `Apple M1 Pro 10-core, 16Gb Ram LPDDR5`.

The results obtained for the *Base* Model, presented in the Figures [21,22], are in line with those obtained previously in SAT (Section 4.7).

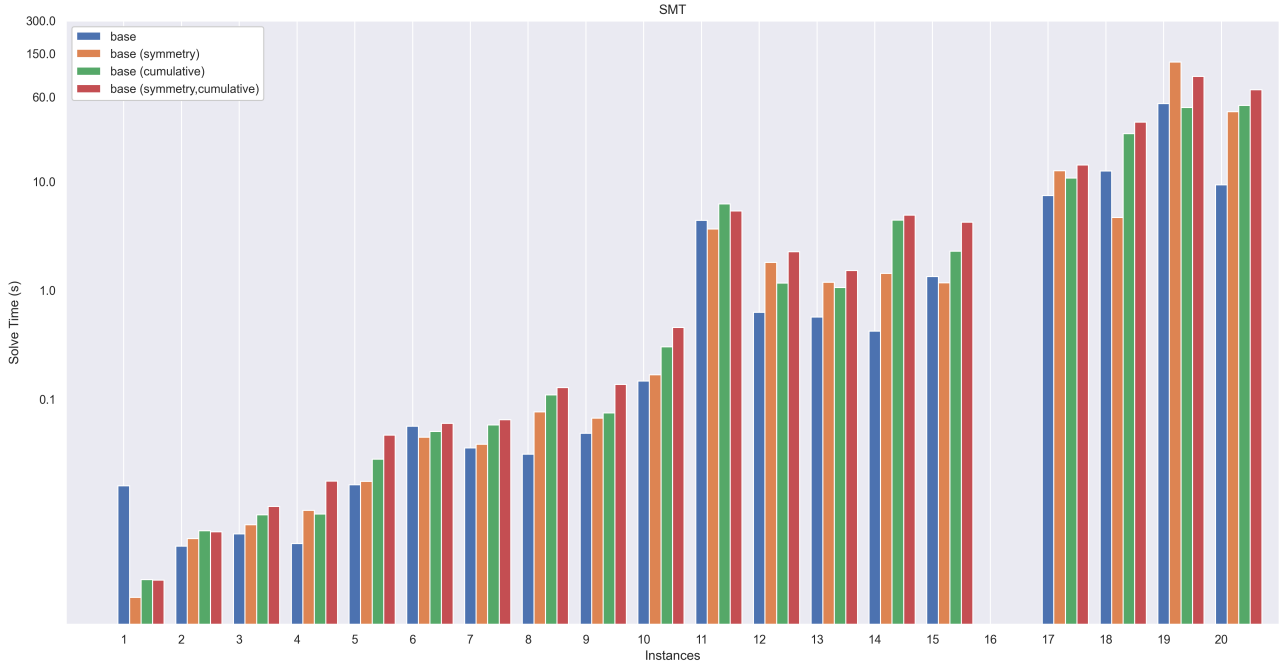


Figure 21: SMT *Base* Model: instances [1,20]

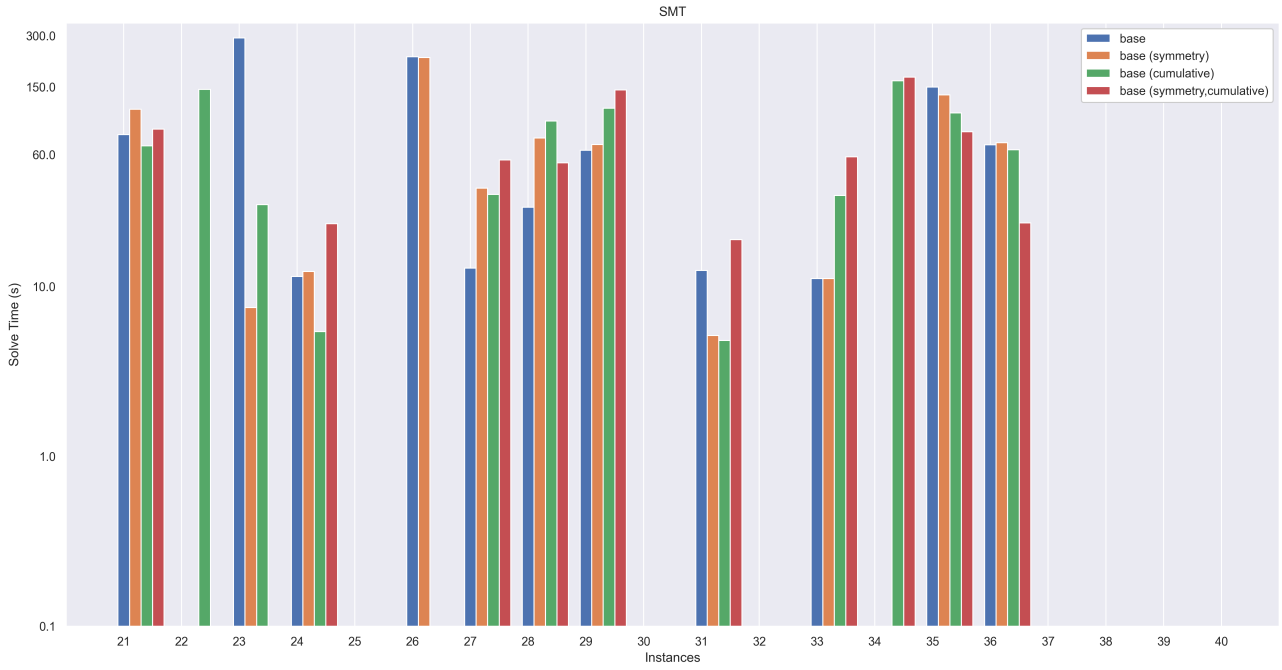


Figure 22: SMT *Base* Model: instances [21,40]

The results obtained for the *Rotation* Model, presented in the Figures [23,24], are "slightly" better with those obtained previously in SAT (Section 4.7). The latter statement refers, in particular, to the average solve time needed for solve the same instances. Despite this, as for the *base* Model, the total number of instances correctly solved haven't changed too much.

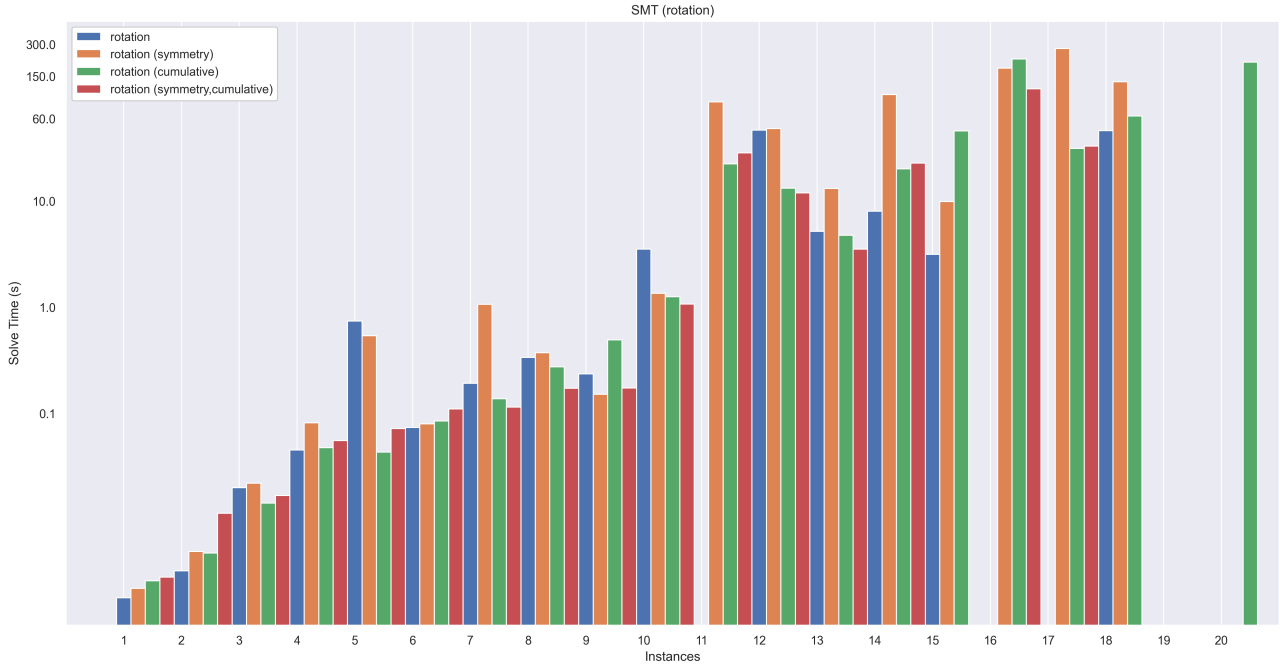


Figure 23: SMT *Rotation* Model: instances [1,20]

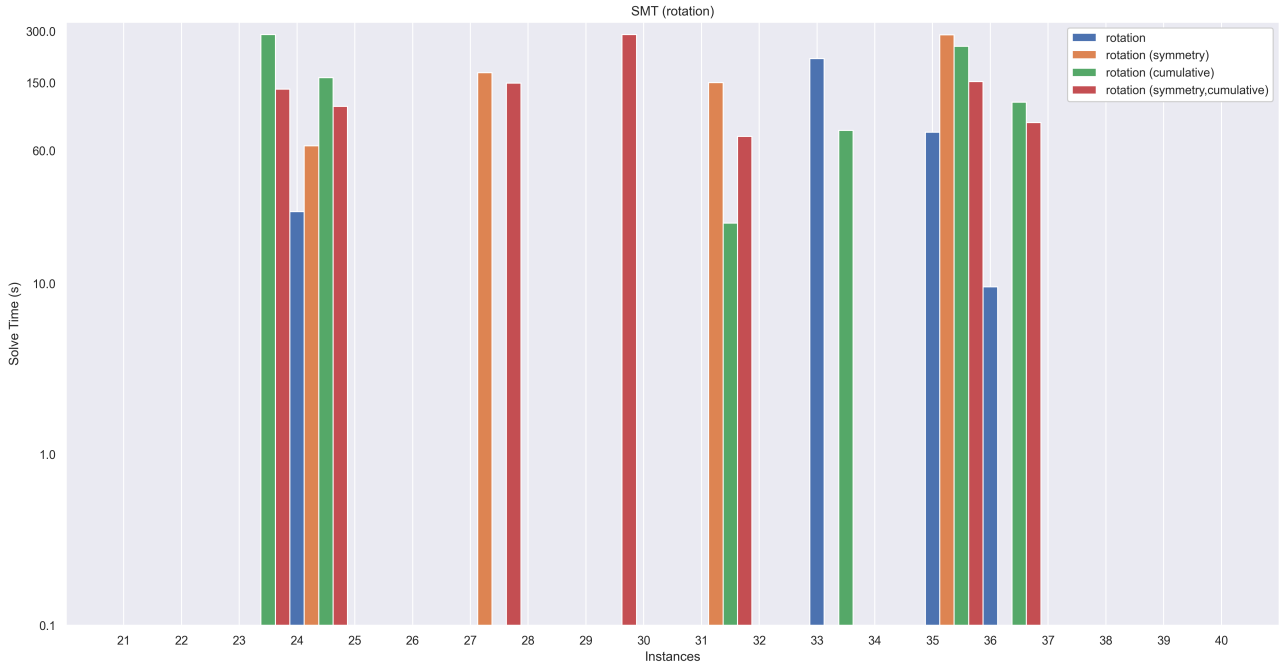


Figure 24: SMT *Rotation* Model: instances [21,40]

Considerig the fact that SMT is an extension of SAT, we can see how similar are the results for these two technologies (SAT results are available in Section 4.7) when we inspect the total number of correctly solved instances.

Finally, through the similarity between the results of SAT and SMT, we have also showed the validity of the conversion between the two formalizations (SAT: Section 4.3, SMT: Section 5.2).

6 ILP

6.1 Background

This section focuses on the floorplan design problem of VLSI circuits by using a Mixed Linear Programming approach (MILP). To implement our model we used the IBM library of *ILOG cplex*. More precisely we choosed to implement the model by using the extension of *DOcplex* python modeling API, *Decision and Optization cplex*, which is a library composed of two modules:

- Mathematical Programming Modeling for Python using *docplex.mp* (*DOcplex.MP*), which is the module we used.
- Constraint Programming Modeling for Python using *docplex.cp* (*DOcplex.CP*)

6.2 Notation

A new parameter will be considered in the following solution in order to correct define our model:

$$M \quad = \quad \text{A very large positive number} > \max(width, max\_makespan)$$

Next, we need to add to the variables defined in Section 2.3 the following:

$$u_i \quad = \begin{cases} 1 & \text{if circuit } i \text{ is rotated} \\ 0 & \text{otherwise} \end{cases}$$

$$l_{i,j} \quad = \begin{cases} 1 & \text{if circuit } i \text{ is on the left side of the circuit} j \\ 0 & \text{otherwise} \end{cases}$$

$$r_{i,j} \quad = \begin{cases} 1 & \text{if circuit } i \text{ is on the right side of the circuit} j \\ 0 & \text{otherwise} \end{cases}$$

$$a_{i,j} \quad = \begin{cases} 1 & \text{if circuit } i \text{ is above the circuit} j \\ 0 & \text{otherwise} \end{cases}$$

$$b_{i,j} \quad = \begin{cases} 1 & \text{if circuit } i \text{ is below the circuit} j \\ 0 & \text{otherwise} \end{cases}$$

Moreover, for what concerns the rotation model, we decide to implement a similar procedure to the other previously seen approaches. We implemented a couple of functions which have the scope of managing the rotation and the true dimension of each circuit i according to the value of its specific variable rotation u_i:

$$f_h(u_i) \quad = w_i \; u_i + h_i \; (1 - u_i) \tag{18a}$$

$$f_w(u_i) \quad = h_i \; u_i + w_i \; (1 - u_i) \tag{18b}$$

Please notice that the use of such a function will not remove the linearity of each constraint, but it is just a name we gave to increase the readability of the report.

6.3 Model Formalisation

In the following we are going to present the model formalisation for the VLSI problem as a MILP, by considering the notation previously cited. Let's start with the base model without the rotations.

$$\text{minimize } makespan \tag{19a}$$

$$y_c + h_c \leq makespan \qquad \forall c \in C \tag{19b}$$

$$x_{c_1} - x_{c_2} \geq w_{c_2} - M\ (1 - l_{c_1c_2}) \qquad \forall c_1, c_2 \in C \tag{19c}$$

$$x_{c_1} - x_{c_2} \geq w_{c_2} - M\ (1 - r_{c_1c_2}) \qquad \forall c_1, c_2 \in C \tag{19d}$$

$$y_{c_1} - y_{c_2} \geq h_{c_2} - M\ (1 - a_{c_1c_2}) \qquad \forall c_1, c_2 \in C \tag{19e}$$

$$y_{c_1} - y_{c_2} \geq h_{c_2} - M\ (1 - b_{c_1c_2}) \qquad \forall c_1, c_2 \in C \tag{19f}$$

$$l_{c_1c_2} + r_{c_1c_2} + a_{c_1c_2} + b_{c_1c_2} \geq 1 \qquad \forall c_1, c_2 \in C \tag{19g}$$

$$l_{c_1c_2} + r_{c_1c_2} + a_{c_1c_2} + b_{c_1c_2} \leq 2 \qquad \forall c_1, c_2 \in C \tag{19h}$$

$$0 \leq x_c \leq width - w_c \qquad \forall c \in C \tag{19i}$$

$$0 \leq y_c \leq max\_makespan - h_c \qquad \forall c \in C \tag{19j}$$

$$min\_makespan \leq makespan \leq max\_makespan \qquad \forall c \in C \tag{19k}$$

$$0 \leq a_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{19l}$$

$$0 \leq b_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{19m}$$

$$0 \leq r_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{19n}$$

$$0 \leq l_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{19o}$$

Also for what concerns ILP, the VLSI problem had been defined with the objective of minimizing the chip height. This can be seen in the model in the equation 19a where it is shown the problem direction, namely the minimization of the integer variable *makespan*. In order to complete the project task, it is necessary to minimize the variable *makespan* which is an integer variable with lower bound *min_makespan* and with upper bound *max_makespan*. Please notice also that the boundaries *min_makespan* and *max_makespan* had been computed in the same way as for the previous approaches (i.e. SAT, CP and SMT).

For what concerns the constraints, the first one to analyse is the one that constraints (equation 19b) the variable y to place blocks for which vertical dimension doesn't exceed the value associated with the variable *makespan*.

Now the foucus should be posed on the constraints 19c-19h. These constraints achieve the same task as the *diffn* constraint in CP. More in detail, we implemented this constraint in ILP by imposing a set of equations. As we know it is defined in CP as 3. So, in order to have the same result of the disjoint clause conditions, we need to control the variable's results. The constraints 19c-19f states the position of a circuit c_1 respect to another circuit c_2. The auxiliary parameter M has the scope to falsify the equations 19c-19f when the correspondent variable is set to 0. Please notice that M is a parameter that can be set by the user, we just want to highlight the fact that for its nature M is constrained to be higher than the difference between any couple of x variables and any couple of y variables, so we decided to set it:

$$M = \max max\_makespan, width + 1$$

We used for our implementation the smaller value Then the subsequent constraints 19g-19h are needed to allow the sum of the binary variables (namely $r_{c_1c_2}$, $l_{c_1c_2}$, $a_{c_1c_2}$ and $b_{c_1c_2}$) belonging in in the range between 1 and 2.

Finally, the last ones are just bounding constraints, which describe the upper and lower bounds of the variables. More in details, while the integer variables x_c can be in the range between 0 and $width - w_c$ (see constraint 19i), the variables y_c can have values in the range between 0 and $makespan - h_c$ (see constraint 19j). This is due to the fact that we cannot place boxes in a position such that the chip's dimension exceeds the circuit plate both in high and in width. Then, since all the variables of the constraints *diffn* are binary, their value should fall in the range 0 and 1.

6.4 Rotation

The next step consists to extend the model in order to take into account also the box rotation. For this scope, we introduced, as we did also in the previous sections, a new set of binary variables u_c.

$$\text{minimize } makespan \tag{20a}$$

$$y_c + f_h(u_c) \leq makespan \qquad \forall c \in C \tag{20b}$$

$$x_{c_1} - x_{c_2} \geq f_w(u_{c_2}) - M\ (1 - l_{c_1c_2}) \qquad \forall c_1, c_2 \in C \tag{20c}$$

$$x_{c_1} - x_{c_2} \geq f_w(u_{c_2}) - M\ (1 - r_{c_1c_2}) \qquad \forall c_1, c_2 \in C \tag{20d}$$

$$y_{c_1} - y_{c_2} \geq f_h(u_{c_2}) - M\ (1 - a_{c_1c_2}) \qquad \forall c_1, c_2 \in C \tag{20e}$$

$$y_{c_1} - y_{c_2} \geq f_h(u_{c_2}) - M\ (1 - b_{c_1c_2}) \qquad \forall c_1, c_2 \in C \tag{20f}$$

$$l_{c_1c_2} + r_{c_1c_2} + a_{c_1c_2} + b_{c_1c_2} \geq 1 \qquad \forall c_1, c_2 \in C \tag{20g}$$

$$l_{c_1c_2} + r_{c_1c_2} + a_{c_1c_2} + b_{c_1c_2} \leq 2 \qquad \forall c_1, c_2 \in C \tag{20h}$$

$$0 \leq x_c \leq width - w_c \qquad \forall c \in C \tag{20i}$$

$$0 \leq y_c \leq max\_makespan - f_h(u_c) \qquad \forall c \in C \tag{20j}$$

$$min\_makespan \leq makespan \leq max\_makespan \qquad \forall c \in C \tag{20k}$$

$$0 \leq a_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{20l}$$

$$0 \leq b_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{20m}$$

$$0 \leq r_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{20n}$$

$$0 \leq l_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{20o}$$

$$0 \leq u_{c_1c_2} \leq 1 \qquad \forall c_1, c_2 \in C \tag{20p}$$

As we previously said, the use of the functions defined in the equation 18, is needed to express the correct dimension to the model to each box in each condition. In this way, it is possible to extend the model in a way that it can be possible to preserve the formalisation of the no rotations model, i.e. the base one (see equations 19). Indeed, looking at the model, we didn't any changes to the objective function. We just modified the constraints 20c-20f, so that they take into account the correct dimensions and similarly the constraints 20b and 20j.

6.5 Results

All the experiments for the MILP technology have been executed on a laptop computer running Windows 10 Pro, 64-bit, equipped with the following hardware: `Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, 4 cores, 16Gb Ram 2.30GHz`.

As we can see in Figure 25, we obtained quite good results for the first 20 instances in the model *base*. In particular, we can notice that the time spent for proving the optimal solution is for the first 11 instances always less than 1 second. Things start to get worse after the 17th instance when the time spent reached suddenly the threshold of 60 seconds. In later instances (Figure 25), things get even

more difficult. The tests we did shows that many instances had been jumped due to the complexity of the problem, showing moreover an instability in for what concerns the time used for solve the problem.

In the case of *rotations* the Figure 25 shows a similarity across the two models, it seems like the complexity of the problem didn't exceed the one of *base* until the 15th instance. Indeed, the subsequent ones appeared to be more difficult for the solver which failed to prove the optimality many times (look at Figure 25 for more details).

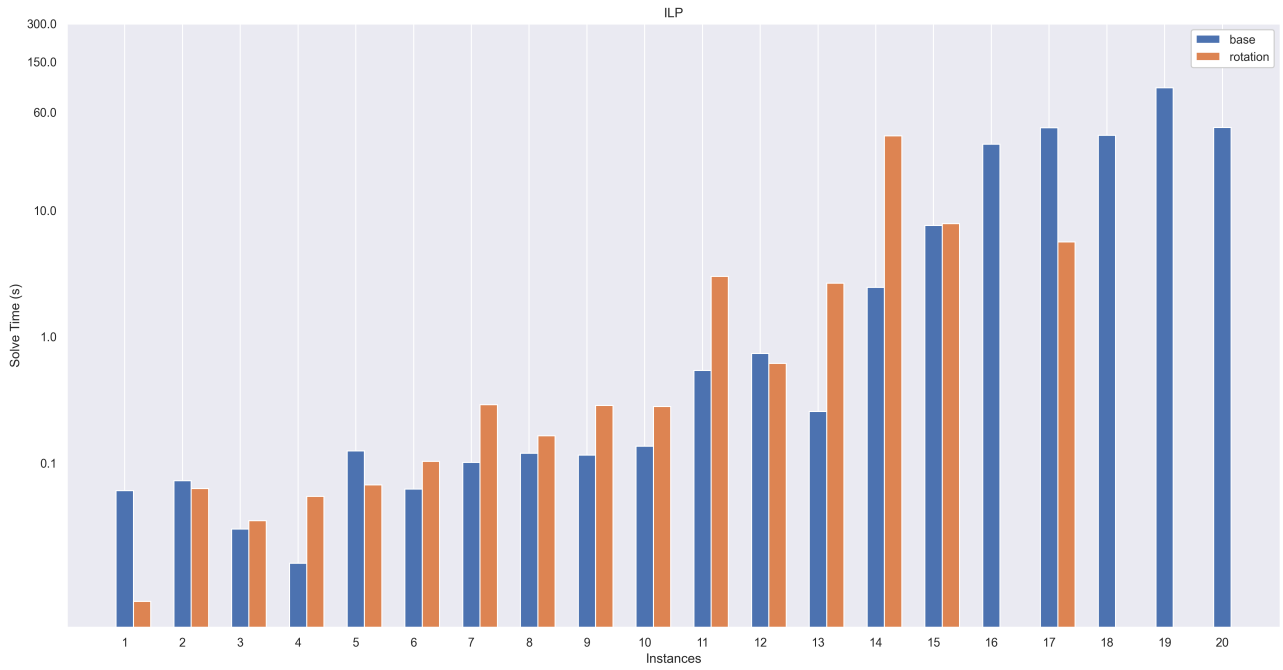


Figure 25: ILP: First 20 instances total time spent for proving optimality ILP cplex solver. Quite good results obtained in the fist attempt, with a progressive decrease in the quality of the results as the difficulty of the problem increases.

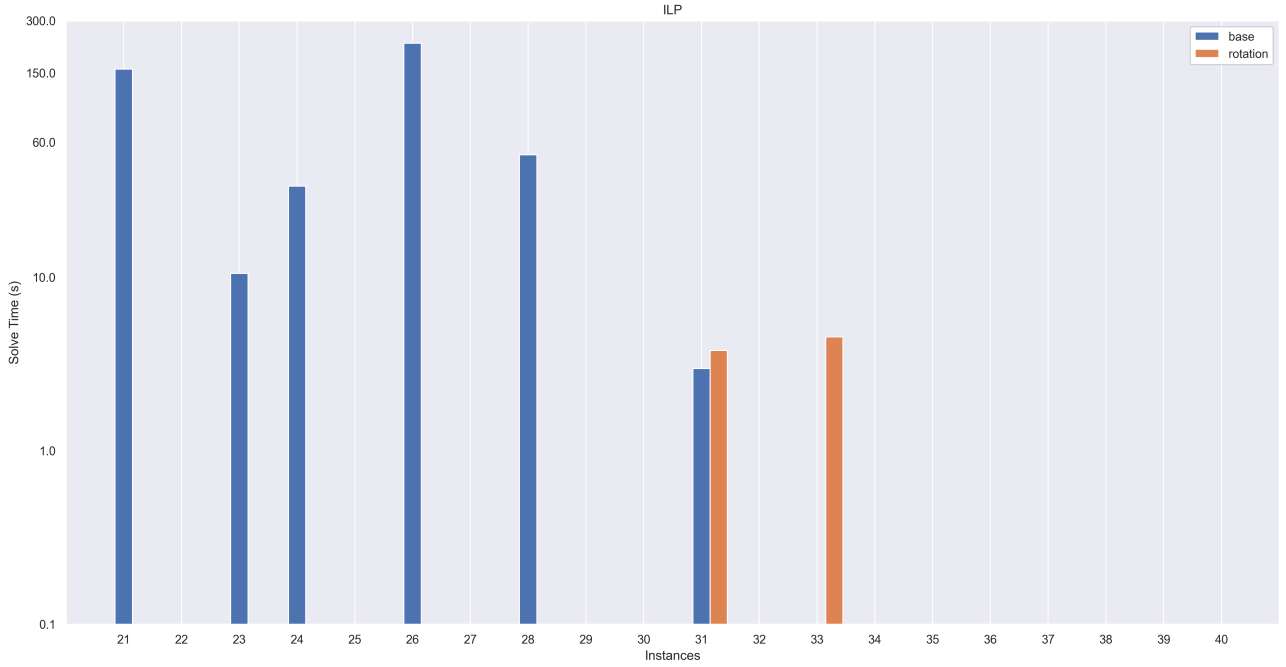


Figure 26: ILP: Last 20 instances total time spent for proving optimality ILP cplex solver. Many instances failed to reach the optimality with a with greater difficulty in the case of *rotation*, as it was easy to expect.

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.

[2] Wenting Zhao. "Encoding Lexicographical Ordering Constraints in SAT". In: (2017). URL: https://digitalcommons.iwu.edu/cgi/viewcontent.cgi?article=1022&context=cs_honproj.