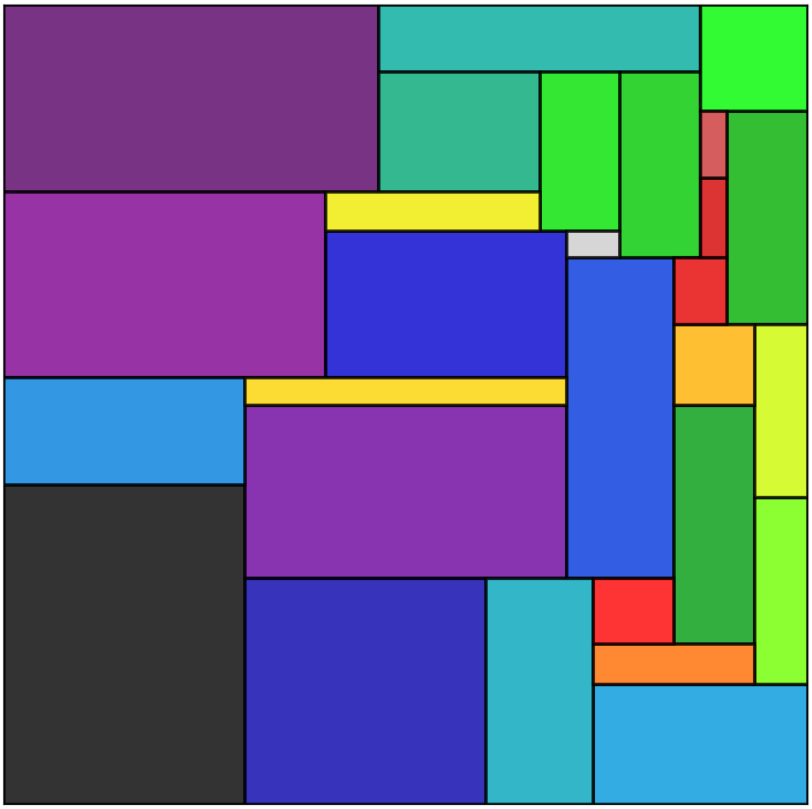
VLSI design: CP, SAT and SMT approaches

Diego Chinellato diego.chinellato@studio.unibo.it
Giorgia Campardo giorgia.campardo@studio.unibo.it

Alma Mater Studiorum - University of Bologna
Combinatorial Decision Making and Optimization course
Module 1 project report



Contents

1. Introduction

In the last couple of decades we have been witnessing a trend of shrinking and miniaturization of electronic components (transistors). Indeed, having smaller components allows to fit more of them in a given area; on the other hand, smaller transistors allows to obtain a smaller chip (in terms of area) with the same computational capabilities of a chip which uses bigger transistors. This process is called Very Large-Scale Integration (VLSI), that is, the process of designing an integrated circuit by embedding millions or even billions of transistors on a single silicon semiconductor microchip. Given the huge number of components involved, the development of smart techniques to address this process has become critical for developing modern-day devices.

This report describes a Combinatorial Optimization approach to the VLSI problem. In particular, three different technologies are employed to address the problem at hand, namely Constraint Programming (CP), propositional SATisfiability (SAT) and Satisfiability Modulo Theories (SMT).

2. Preliminaries

2.1 Instances

An instance of VLSI is defined by several integer variables. In particular, as input we are given w, which is the width of the plate; n, which is the number of circuits to fit in the plate; a set $\mathcal{C} = \{(x_i, y_i)\}_{i=1}^{n}$ where x_i and y_i are respectively the horizontal and vertical dimensions of the i-th circuit. A solution of a certain instance involves determining the minimum feasible length l of the plate, as well as the locations of each of the n circuits, determined by the $(\hat{x}_i, \hat{y}_i)$ coordinates of the bottom left corner. Table 1 shows an example of the input and output format (as it appears in a text file).

Input (*ins-n.txt*)	**Output** (*out-n.txt*)	**Description**
8	8 8	w, l
4	4	n
5 5	5 5 0 0	$x_1, y_1, \hat{x}_1, \hat{y}_1$
5 3	5 3 0 5	$x_2, y_2, \hat{x}_2, \hat{y}_2$
3 5	3 5 5 0	$x_3, y_3, \hat{x}_3, \hat{y}_3$
3 3	3 3 5 5	$x_4, y_4, \hat{x}_4, \hat{y}_4$

Table 1: Input/output format example.

2.2 Common preprocessing steps

There are two "preprocessing" steps that are common to every technology adopted. The first step concerns the definition of a proper domain for the l variable; this is done for all models through additional parameters `minl` and `maxl`, which represent respectively the lower and upper bounds on the variable's domain. The lower bound for the plate height is computed as

$$minl = \frac{\sum_{(x_i,y_i) \in \mathcal{C}} x_i \cdot y_i}{w}$$

i.e, by summing the areas of all circuits and then dividing by the given width. This minimum value is actually feasible if and only if there is at least one solution that can pack all the circuits together without leaving any blank space in between them. Concerning the upper bound, a trivial one can be computed by summing all the circuits' heights, i.e. $maxl = \sum_{(x_i,y_i)\in\mathcal{C}} y_i$ (as we can always find a solution by stacking all circuits one on top of the other). However, this is a huge overestimation of the actual minimum feasible height, so this upper bound was reduced through an approximation: we proceed in the same way as for the lower bound, but rather than using the horizontal dimension of each circuit, the maximum horizontal dimension was used

instead when computing the areas. The formula then becomes

$$maxl = \frac{\sum_{(x_i,y_i)\in\mathcal{C}} x^* \cdot y_i}{w}, \text{ with } x^* = \max_{(x_i,y_i)\in\mathcal{C}} x_i$$

This upper bound is in general much smaller than the previous, trivial one, thus reducing the complexity of the search problem while still allowing for a reasonable domain of the length variable.

The second preprocessing step, which is actually more of an heuristic, involved sorting the given set of circuits according to decreasing values of their areas, meaning that circuit 1 will be the one with the biggest area, circuit 2 the second-biggest one and so on. The intuition behind this is that it's more "difficult" to place bigger circuits as they occupy more space, thus we want our solvers to deal with them first and then move on the easier, smaller circuits. This approach has been already explored in the literature, e.g. in Huang and Korf (2013).

3. CP

Constraint Programming is a paradigm for solving combinatorial problems by stating, in a declarative fashion, a set of constraints that must hold on the feasible solutions for a given set of decision variables. In this section, a CP model for the VLSI problem is described.

3.1 Variables

In order to model the problem, the following parameters were defined:

- `w`: the width of the plate;
- `n`: the number of circuits to be placed in the plate;
- `minl, maxl`: the lower and the upper bound for the plate height;
- `x, y`: two arrays containing respectively the horizontal and the vertical dimensions of each circuit.

Then, we defined the model's decision variables:

- `l`: the plate height;
- `xhat, yhat`: two arrays that represent the coordinates of each circuit along the x and y axes. Note that the domain of these two variables differs from the domain of `x, y`, given that e.g. the highest feasible x-coordinate cannot be greater than `w - min(x)`.

Finally, we define the objective function simply as `solve minimize l`.

```
int: w;
int: n;
int: minl;
int: maxl;
set of int: circuits = 1..n;

array [circuits] of 1..w: x;
array [circuits] of 1..maxl: y;

var minl..maxl: l = max(i in circuits)(yhat[i] + y[i]);

array [circuits] of var 0..w-min(x): xhat;
array [circuits] of var 0..maxl-min(y): yhat;
```

Listing 1: Model parameters and variables

3.2 Constraints

Main problem constraints

The problem definition begins with the specification of a constraint which impose a no-overlap relationship between circuits, that is, a constraint imposing that each cell of the plate must be occupied by a single circuit. This can be simply done by specifying that, for each pair of circuits i, j, the origins of circuit i plus its dimension along the respective axis must be lower or equal to the origins of circuit j or vice-versa for circuit j with respect to circuit i.
Then we give bounds to the origins of each circuit: the origin point plus its dimension along that axis must be lower or equal than the width/heigth of the plate and the origin for the y axis must be lower than the plate's height.

```
constraint forall (i,j in circuits where i < j)
                  (xhat[i] + x[i] <= xhat[j]
                  \/ yhat[i] + y[i] <= yhat[j]
                  \/ xhat[j] + x[j] <= xhat[i]
                  \/ yhat[j] + y[j] <= yhat[i]);
constraint forall (i in circuits) (yhat[i] < l);
constraint forall (i in circuits) (xhat[i] + x[i] <= w);
constraint forall (i in circuits) (yhat[i] + y[i] <= l);
```

Listing 2: Main problem constraints

Implied constraints

Implied constraints are logical consequences of the initial specification of the problem. Though adding an implied constraint to the specification of a constraint satisfaction problem does not change the set of solutions, it can reduce the amount of search the solver has to do (Frisch et al. (2004)). Considering the problem at hand, we can make the following consideration: if we draw a horizontal line and sum the horizontal sides of the traversed circuits, the sum can be at most w. A similar property holds if we draw a vertical line.

```
constraint forall (k in 1..w) (sum([ y[i] | i in circuits
                                where xhat[i] < k
                                /\ xhat[i] + x[i] >= k ]) <= l);
constraint forall (k in 1..maxl) (sum([ x[i] | i in circuits
                                   where yhat[i] < k
                                   /\ yhat[i] + y[i] >= k ]) <= w);
```

Listing 3: Implied constraints

Global constraints

In Minizinc, when using the library of the global constraints the search process is faster as many solvers implement special, efficient inference (propagation) algorithms

for enforcing these constraints. Thus, the no-overlap constraint can be substituted with the `diffn(x, y, dx, dy)` global constraint that constrains rectangles specified by their origins `x, y` and sizes `dx, dy`, to be non-overlapping. A similar reasoning applies to the implied constraints, which can be substituted with the `cumulative(s, d, r, b)` global constraint, which requires that a set of tasks given by start times `s` (x and y coordinates), durations `d` (the respective sizes in the same coordinate), and resource requirements `r` (the other size), never require more than a global resource bound `b` (either width or length) at any one time.

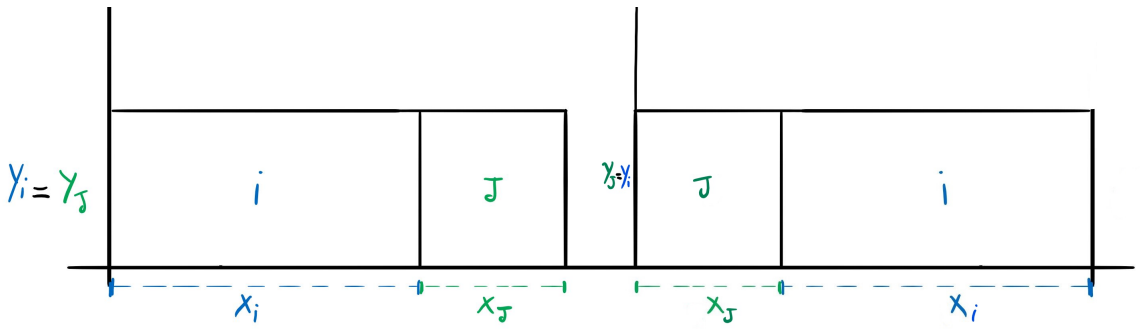
```
constraint diffn(xhat, yhat, x, y) :: domain;
constraint cumulative(xhat, x, y, l) :: domain;
constraint cumulative(yhat, y, x, w) :: domain;
```

Listing 4: Global constraints

Symmetry breaking constraints

This problem is full of symmetries. Given a perfect packed board of circuits, we can identify 6 main symmetries, given by rotating the plate by 90°, 180°and 270° and by reflecting the board over the x-axis, y-axis and both simultaneously. To break this symmetries, an ordering is imposed between the first (largest) circuit and the second (largest) circuit.

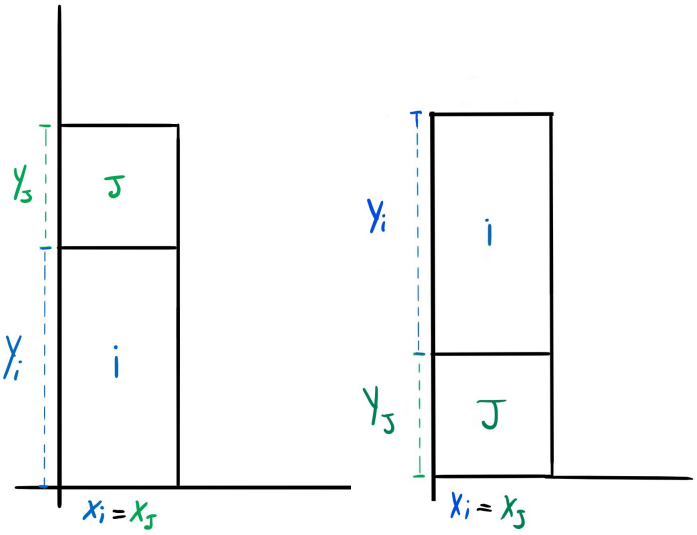
Another type of symmetry we found is the row-block and column-block symmetry, shown in Figures 1 and 2: whenever two adjacent blocks share the same x dimension (if they are one on top of the other) or the same y dimension (if they are one to the side of the other), then they can be freely swapped. To break this type of symmetries, we impose an ordering between the two circuits on the varying coordinate whenever we have this kind of situation.



(a) Retained solution (b) Pruned symmetric solution

Figure 1: Row-block symmetry

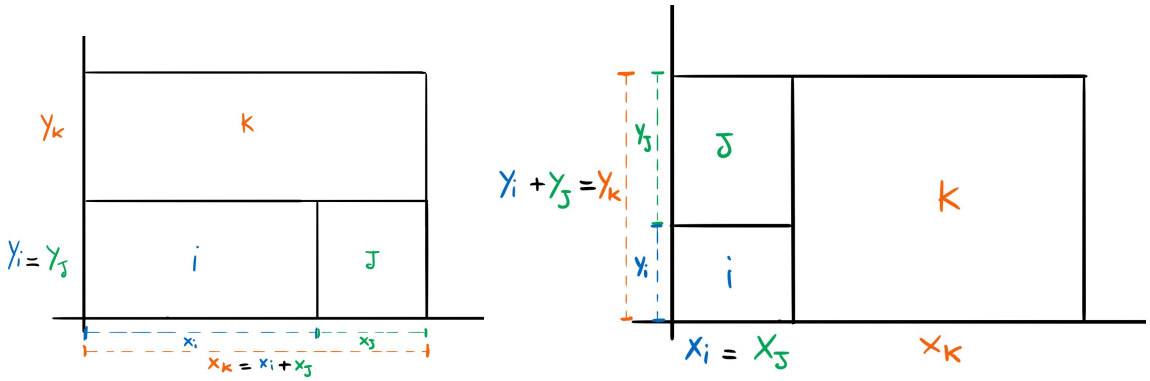
One more type of symmetry found is the one we called 3-blocks symmetry and that is shown in Figure 3. This symmetry involves 2 adjacent circuits i, j with the



(a) Retained solution (b) Pruned symmetric solution

Figure 2: Column-block symmetry

same y-dimension $y_i = y_j$, which are both adjacent to a bigger circuit k for which we have $x_i + x_j = x_k$. In this case, we can freely swap the "composite" circuit i, j with circuit k. A similar reasoning applies to the horizontal case. To break this symmetries, we impose an ordering on the varying coordinate of i and k.



(a) Vertical case (b) Horizontal case

Figure 3: Three-blocks symmetry

Finally, one last symmetry breaking constraint forces the first (biggest) block to be to the bottom left of the second (biggest) block, thus eliminating 3 symmetric solution (i.e. when the second block is to the top left, top right or bottom right of the first one).

```
% rows and columns symmetry
```

```
constraint symmetry_breaking_constraint(
    forall (i,j in circuits where i < j) ((xhat[i] == xhat[j] /\ x[i
       ] == x[j]) -> yhat[i] <= yhat[j] ));
constraint symmetry_breaking_constraint(
    forall (i,j in circuits where i < j) ((yhat[i] == yhat[j] /\ y[i
       ] == y[j]) -> xhat[i] <= xhat[j] ));

% three blocks symmetry
constraint symmetry_breaking_constraint(
    forall (i,j,k in circuits where i > j /\ j > k)
        ((xhat[i] == xhat[j] /\ x[i] == x[j] /\ yhat[i] == yhat[k]
           /\ y[i] + y[j] == y[k]) -> xhat[k] <= xhat[i] ));
constraint symmetry_breaking_constraint(
    forall (i,j,k in circuits where i > j /\ j > k)
        ((yhat[i] == yhat[j] /\ y[i] == y[j] /\ xhat[i] == xhat[k]
           /\ x[i] + x[j] == x[k]) -> yhat[k] <= yhat[i] ));

% force the biggest block to be always to the bottom left of the
   second biggest
constraint xhat[1] <= xhat[2] /\ yhat[1] <= yhat[2];
```

Listing 5: Symmetry breaking constraints

Dual model

During the process of trying to determine and break symmetries, we came up with a dual model that can be used in conjunction with the main one. In particular, we define two new arrays of variables `xsym`, `ysym` that represent the coordinates of each circuit in the respective axis symmetry and they are computed using the following formulas:

$$\forall i,\ (\tilde{x}_i = w - x_i - \hat{x}_i) \wedge (\tilde{y}_i = l - y_i - \hat{y}_i)$$

Then, we define the board of the circuits, a bi-dimensional array where each cell contains the index of the circuit that occupies that position in the board. Similarly, we also define the other three symmetric boards: `boardXsym` that represents the board reflected over the x-axis, `boardYsym` that represents the board reflected over the y-axis and `boardXYsym` that represents the board reflected over both axes. In order to break the symmetries, we impose an ordering among the symmetric solutions by putting the `lex_lesseq` constraint between the flattened board and the flattened symmetric board.

```
set of int: xRange = 0..w-1;
set of int: yRange = 0..maxl-1;
array [circuits] of var xRange: xsym;
array [circuits] of var yRange: ysym;
array [yRange, xRange] of var 1..n+1: board;
array [yRange, xRange] of var 1..n+1: boardXsym;
array [yRange, xRange] of var 1..n+1: boardYsym;
array [yRange, xRange] of var 1..n+1: boardXYsym;
```

```
% channeling constraints
constraint forall (i in circuits) (xsym[i] = w - x[i] - xhat[i] /\
   ysym[i] = l - y[i] - yhat[i]);

constraint forall (j in yRange, k in xRange, i in circuits)
        (k >= xhat[i] /\ k < (xhat[i]+x[i]) /\ j >= yhat[i]
        /\ j < (yhat[i]+y[i]) <-> board[j, k] = i);

constraint forall (j in yRange, k in xRange, i in circuits)
        (k >= xsym[i] /\ k < (xsym[i]+x[i]) /\ j >= yhat[i]
        /\ j < (yhat[i]+y[i]) <-> boardXsym[j, k] = i);

constraint forall (j in yRange, k in xRange, i in circuits)
        (k >= xhat[i] /\ k < (xhat[i]+x[i]) /\ j >= ysym[i]
        /\ j < (ysym[i]+y[i]) <-> boardYsym[j, k] = i);

constraint forall (j in yRange, k in xRange, i in circuits)
        (k >= xsym[i] /\ k < (xsym[i]+x[i]) /\ j >= ysym[i]
        /\ j < (ysym[i]+y[i]) <-> boardXYsym[j, k] = i);

% symmetry breaking constraints
constraint lex_lesseq(array1d(board), array1d(boardXsym));
constraint lex_lesseq(array1d(board), array1d(boardYsym));
constraint lex_lesseq(array1d(board), array1d(boardXYsym));
```

Listing 6: Dual model

3.3 Rotation

The general case of the problem at hand allows the rotations of the circuits, meaning that a circuit of size $x_k \times y_k$ can be placed on the plate as a $y_k \times x_k$ circuit. To allow our model to rotate circuits, we introduce additional variables and constraints.

In particular, we add a boolean array `rotation` of size n, to keep track of the rotations of the individual circuits; moreover, we add two arrays `inputx` and `inputy` which contain the original input, and we transform the original `x, y` arrays into decision variables (as they depend on whether each circuit has been rotated or not).

Finally, we introduce a simple symmetry breaking constraint, where we impose that square circuits are not to be rotated (as their size is not affected by rotations).

```
array [circuits] of 1..w: inputx;
array [circuits] of 1..maxl: inputy;
array [circuits] of var bool: rotation;
array [circuits] of var 1..w: x;
array [circuits] of var 1..maxl: y;

constraint forall (i in circuits) (if rotation[i]
    then y[i] == inputx[i] /\ x[i] == inputy[i]
    else x[i] == inputx[i] /\ y[i] == inputy[i] endif);
```

```
% simple symmetry breaking for square circuits
constraint forall (i in circuits) (inputx[i] == inputy[i] ->
    rotation[i] = false);
```

Listing 7: Handling rotations: new variables and constraints

3.4 Search

One of the key features of CP is the possibility of interacting with the search process by specifying different search strategies such as heuristics for variable and value ordering and restart strategies. In MiniZinc, by default there is no such specification, so the search process is handled entirely by the underlying solver (which can be chosen as well), but indeed we can specify how the search should be carried out. Note that the search strategy is not really part of the model, but for sure it's essential in tackling more difficult problems.

In MiniZinc, we can specify extra search information to the constraint solver using annotations. In particular, we consider a search annotation for specifying how to carry out the search on the `xhat, yhat` integer variables of our model, i.e. the `int_search ( <variables>, <varchoice>, <constrainchoice> )` annotation, where `<variables>` is a one dimensional array of `var int`, `<varchoice>` is a variable choice annotation (i.e., a variable ordering heuristic, VOH) and `<constrainchoice>` is a value ordering strategy. We will consider the following three VOHs:

- `input_order`: choose the variables in the given order, which for our model means (when ordering the circuits) to choose the biggest circuit not yet positioned.

- `dom_w_deg`: choose the variable x that minimizes $dom(x)/w(x)$, where $dom(x)$ is the (current) domain size of x and $w(x)$ is the weighted degree of x, computed as the sum of the weights of all constraints involving x (initially set to 1 and incremented each time the constraint fails).

- `impact`: choose the variable with the highest impact so far. The intuitive idea behind impact-based heuristics is to estimate, for each variable, how much each possible assignment reduces the search tree size (the bigger the reduction, the "harder" the assignment; an assignment which fails has an impact of 1); thus, the variable with the highest impact is the "most difficult" one to assign and it is chosen, based on the first-fail principle (Refalo (2004)).

As value ordering strategy, we use `indomain_min`, which simply assigns the variable its smallest domain value.

Moreover, any kind of depth first search used for solving optimization problems suffers from the problem that wrong decisions made at the top of the search tree can take an exponential amount of search to undo. One common way to alleviate this issue is to restart the search from the top, thus giving the solver the possibility of

making different decisions. In Minizinc, we can use restart annotations to control how frequently a restart occurs. We consider the following restart strategies:

- `restart_none`: no restart strategy is applied.
- `restart_luby(scale)`: the i-th restart occurs after $L[i] \cdot$ `scale` nodes in the search tree, where $L[i]$ is the ith number of the Luby sequence. In our experiments, we set `scale = 150`.
- `restart_geometric(base, scale)`: the i-th restart occurs after `scale` $\cdot$ `scale^i` nodes. In our experiments, we set `base = 2` and `scale = 50`.

Indeed, note that restart strategies only make sense when applied to non-deterministic search strategies - in our case, only `dom_w_deg` and `impact` - as it doesn't make any sense to restart the search if the solver always explores the search tree in the same way.

A summary comparison on the experimental results on the performances of the different search strategies can be found in Table 2, while full plots showing the runtimes on the individual instances can be found in Appendix I.

3.5 Experiments

Setup

The hardware and software specifications of the machine used to carry out the experiments are the following:

- CPU: Intel® Core™ i5-7600K @ 3.8 GHz (4 cores)
- RAM: 8 GB DDR4 2400 MHz
- OS: Ubuntu 20.04
- Environment: Python 3.9.7, MiniZinc version 2.5.5, Chuffed solver

Results

The first test carried out involved determining a baseline as well as testing the impact (if any) of the simple sorting heuristic described in Section 2.2. The results of this run are shown in Figure 4.

We can clearly see that this simple preprocessing step has a huge impact on the performances of the model: without circuit sorting, the model manages to solve 21 instances out of the provided 40 with an average solve time of 146.49 seconds, whereas enabling circuit sorting allows the same model to jump to 37 solved instances, with a much lower average solve time of 38.37 seconds. Indeed, this appear to confirm our intuition that injecting into the model some background, domain-specific knowledge on the problem (i.e., the fact that bigger circuits are more constraining and thus should be placed before the smaller ones) really helps a lot in driving up the performances

of the model. Based on this result, in the subsequent runs (for CP) we will always sort the circuits before feeding them to the model.

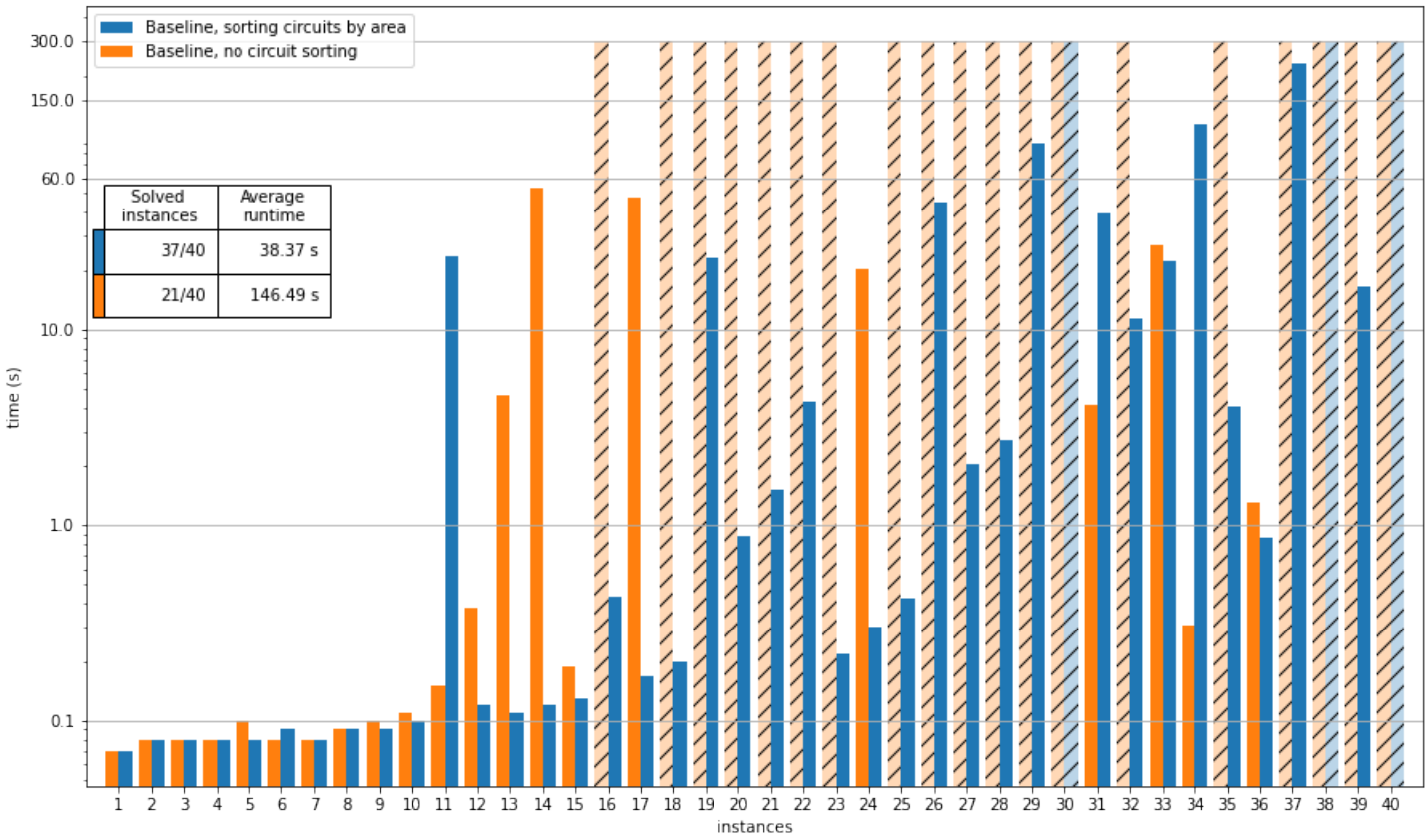


Figure 4: Baseline performances for the CP approach, with and without circuit sorting as preprocessing step.

Next, we moved on testing the dual model described in Section 3.2 and determining whether it could bring any improvement on the base model. The comparison of performances between the baseline model and the dual one is shown in Figure 5.

The results clearly show that this dual model effectively slows down the CP model, since the solve time is much higher even for simple instances (sometimes even one or two orders of magnitude higher, see e.g. instance 8 or instance 27). In general, the dual model manages to solve 30 instances out of 40 with an average solve time of 95.24 seconds, almost three times higher than the base model without the dual

one. This results are probably due to the fact that the dual model adds too many variables and constraints, making the overall model too heavy to be satisfied - indeed, compared to the base model the dual model introduces $O(2n+4(w \cdot maxl)) = O(n+w \cdot maxl)$ variables as well as $O(n^3)$ channeling costraints (and three, hopefully efficient, `leq_lesseq` global constraints).

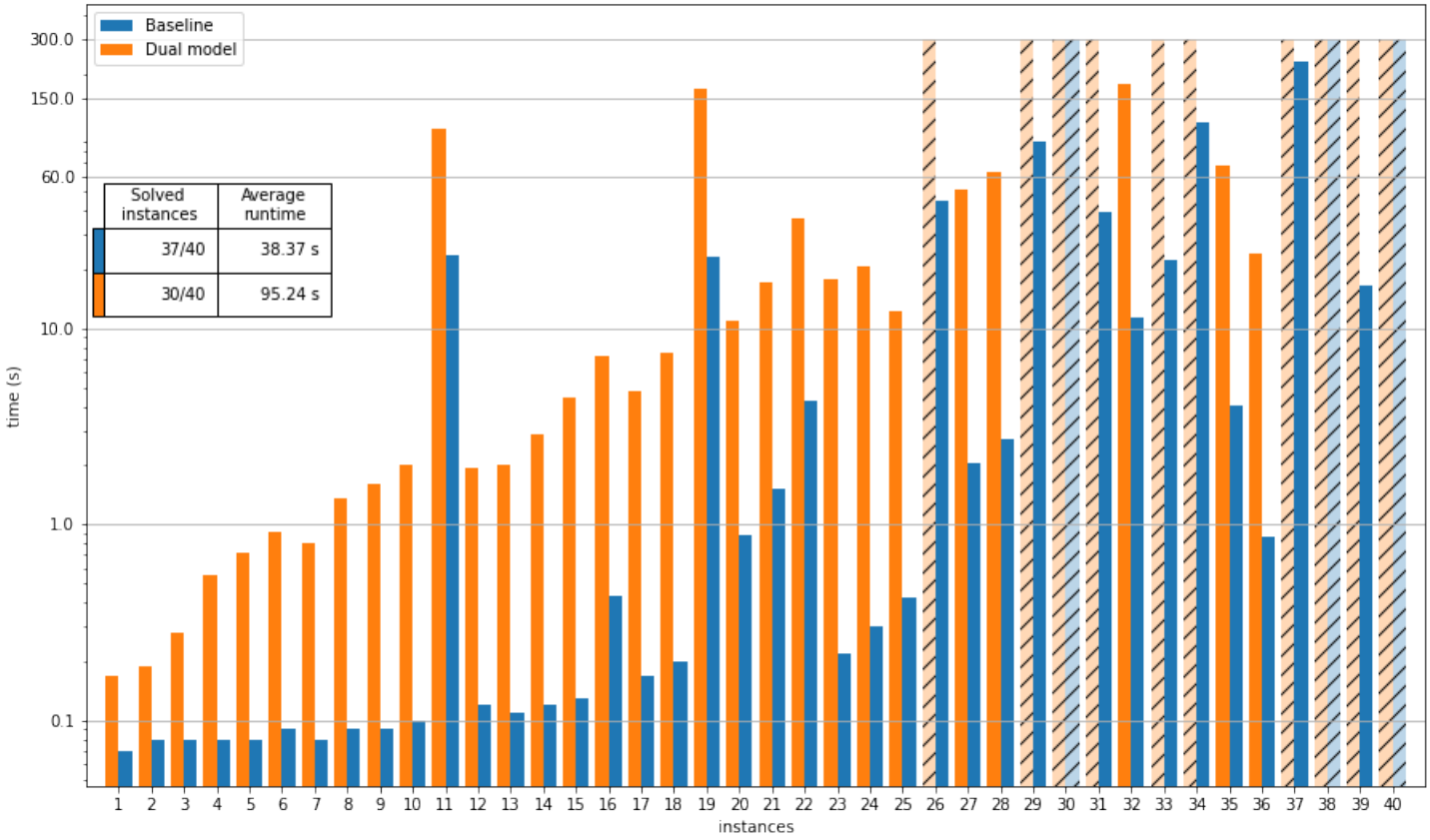


Figure 5: Performances of the CP dual model, compared to the baseline perfomances.

Finally, we moved on investigating the different search strategies discussed in Section 3.4, as well as the rotation-enabled model discussed in Section 3.3. A summary of the results is provided in Table 2, whereas full plots providing runtimes for the individual instances can be seen in Appendix I.

Indeed, the first thing we can notice is that the general case of the VLSI problem (i.e., with rotations) is much more difficult to handle than the simplified case - that's

Rotations	Search heuristics	Restart strategy	Solved instances	Average runtime
No	input order	None	31/40	78.62
		luby(150)	37/40	38.37
		geom(2, 50)	37/40	38.11
	domWdeg	None	31/40	78.38
		luby(150)	37/40	38.32
		geom(2, 50)	37/40	38.17
	impact	None	31/40	78.38
		luby(150)	31/40	84.57
		geom(2, 50)	31/40	84.43
Yes	input order	None	23/40	141.64
		luby(150)	13/40	214.94
		geom(2, 50)	13/40	214.81
	domWdeg	None	23/40	141.61
		luby(150)	13/40	215.00
		geom(2, 50)	13/40	214.87
	impact	None	23/40	141.93
		luby(150)	18/40	174.20
		geom(2, 50)	18/40	174.01

Table 2: Performance comparison of the different search and restart strategies. Full plots of the experiments can be seen in Appendix I.

not a surprise, given that the rotation model involves more variables and in general a much bigger search space (including also more symmetries).

We can also see that indeed different combinations of search and restart strategies interact very differently, also depending on the "version" of the problem (with/without rotations). For example, the `impact` heuristic do not appear to help a lot in the simpler case, while in the general cases it leads to slightly better results, compared to the other two strategies. Also, we can notice that enabling restarts helps when rotations are disabled, but conversely it negatively impacts the performances when rotations are allowed. Moreover, we can also see that there are different combinations that basically lead to the same result (same number of solved instances and very similar average runtimes), further highlighting the fact that there isn't one single "best" strategy to use for combinatorial optimization problems like the one we are dealing with, but rather it is very problem-dependant and it can be empirically determined only through a trial and error process.

Finally, we want to highlight the fact that the `input_order` heuristic, which is indeed a deterministic search strategy, works better when enabling restart. Although in principle this doesn't make any sense (restart should work only with search strategies that use randomization), we think this is due to the solver employed, which is Chuffed.

Chuffed is a SotA solver that combines features of finite domain propagation and Boolean satisfiability and which uses, among several other, lazy clause generation (LCG) techniques. LCG techniques allow (when restarting) to never repeat the same search, even when a static search strategy is used.

4. SAT

The Boolean Satisfiability problem (SAT) involves determining whether a given propositional formula has a satisfying assignment of its variables, i.e. whether there is an interpretation which makes the formula evaluate to true. In this section, a SAT model for the VLSI problem is described.

4.1 Encoding

Variables

Our SAT encoding of the VLSI problem differs, in term of formalization, from the CP model; in particular, while for CP the problem has been formalized as a Costraint Optimization Problem (COP), for SAT we use a Constraint Satisfaction Problem (CSP) formalization. The idea is to define an encoding that implicitly defines a length of the plate; then all possible lengths are iteratively tested, starting from the smallest one possible (`minl`), and then the first encoding that we can prove SAT is guaranteed to be an optimal solution.

In particular, we define a cube of boolean variables $B \in \mathcal{B}^{w \times l \times n}$, where each B_{ijk} is true if circuit k occupies cell i, j of the plate, and false otherwise.

Main problem formulas

Analogously to the CP model, we need first of all to define a no-overlap relationship between the circuits. This can be done by stating that, for each cell of the board, the slice along the third dimension (circuits) must contain exactly one variable true:

$$\bigwedge_{i=1}^{w}\bigwedge_{j=1}^{l}((\bigvee_{k=1}^{n} B_{ijk}) \wedge (\bigwedge_{k_1=1}^{n}\bigwedge_{k_2=k1+1}^{n} \neg(B_{ijk_1} \wedge B_{ijk_2}))) \tag{1}$$

Secondly, we need to define how to actually place circuits on the plate. To do so, we consider the set $\mathcal{L}_k = \{\ell_{k1}, .., \ell_{km}\}$ of possible locations for the circuit k, where each ℓ_{kr} with cardinality $|\ell_{kr}| = x_k \cdot y_k$ contains the set of boolean variables of the plate that represent a valid placement for the circuits k. For example, suppose $w = 3, l = 3$ and for a certain circuit k, $x_k = 2, y_k = 2$; then, we have $\mathcal{L}_k = \{\ell_{k1}, \ell_{k2}, \ell_{k3}, \ell_{k4}\}$, with $\ell_{k1} = \{B_{00k}, B_{01k}, B_{10k}, B_{11k}\}$ (placement in the bottom-left corner of the board, i.e. $\hat{x}_k = 0, \hat{y}_k = 0$), $\ell_{k1} = \{B_{10k}, B_{11k}, B_{20k}, B_{21k}\}$ (placement one cell above the previous location, i.e. $\hat{x}_k = 1, \hat{y}_k = 0$) and so on. Then, we can impose a constraint stating that only one among these possible placements must be true, that is:

$$\bigwedge_{k=1}^{n}((\bigvee_{\ell_{kr} \in \mathcal{L}_k}(\bigwedge_{B_{ijk} \in \ell_{kr}} B_{ijk})) \wedge (\bigwedge_{\ell_{kr} \in \mathcal{L}_k}\bigwedge_{\ell_{kp} \in \mathcal{L}_k, r \neq p} \neg((\bigwedge_{B_{ijk} \in \ell_{kr}} B_{ijk}) \wedge (\bigwedge_{B_{ijk} \in \ell_{kp}} B_{ijk})))) \tag{2}$$

Symmetry breaking formulas

Given this encoding of the problem, we can easily identify 3 symmetries, given by rotating the plate by 90°, 180° and 270°. To break this symmetries, a lexicographic order is imposed between the first (largest) circuit and the second (largest) circuit, following an approach similar to the one proposed in Roy (1996). First, we define the ordering between two boolean variables a, b, $a \leq b$ as the clause $a \implies b$. Then, given two sequences of boolean variables $A = \{a_1, ..., a_m\}$, $B = \{b_1, ..., b_m\}$, we define the predicate $L_i(A, B) \equiv (A_{i+1} = B_{i+1} \implies a_i = b_i)$. Finally, we define the ordering $A \leq B$ as the predicate

$$\bigwedge_{i=1}^{m} L_i(A, B) \tag{3}$$

that is, $a_1 \leq b_1 \wedge (a_1 = b_1 \implies a_2 \leq b_2) \wedge ((a_1 = b_1 \wedge a_2 = b_2) \implies a_3 \leq b_3) \wedge ...$

Given this ordering, we now define $S_1 = \{B_{001}, B_{011}, ..., B_{w(l-1)1}, B_{wl1})\}$ and $S_2 = \{B_{002}, B_{012}, ..., B_{w(l-1)2}, B_{wl2})\}$, i.e. the boolean plates of the first and second circuits, and we impose the ordering $S_1 \leq S_2$. Informally, this ordering predicate states that the first circuit must be placed to the top right of the second circuit, thus effectively removing the other 3 rotation symmetries (when the first circuit is to the top left, bottom left, and bottom right of the second circuit).

4.2 Rotation

Our encoding of the problem allows very easily to extend the definition to the more general case where circuits are allowed to be rotated, without having to introduce additional variables but just by making use of an already defined formula. In particular, we simply need to extend the sets of possible locations $\mathcal{L}_k$ so that they include also the locations that the circuits would occupy if they were rotated. Then, the formula described in Equation 2 can be used just like before, but it is capable of handling rotations as well. However, extra care must be taken when building the set L_k for square circuits: given that for such a circuit the two dimensions coincide, there's no need to add to L_k the locations after a rotation of the circuit, since they are exactly the same as those obtained without rotation. Adding the same locations twice would effectively make the encoding UNSAT (the second part of Equation 2 would always evaluate to false whenever there is a square circuit).

4.3 Search

Although in general the search process carried out by SAT solvers is managed entirely by the solver itself, the actual tool that we used, the Z3 solver, allows the user to slightly interfere with the search process. In particular, Z3 adopts a complex architecture that comprises a CDCL Sat Core solver, a set of in-processors that perform global inference automatically on a periodic basis and include several of the techniques developed for SAT solving, and a set of co-processors to support alternative

searching strategies and heuristics that can be tweaked by the user (Bjørner et al. (2018)). Moreover, Z3 is also capable of running in a parallel fashion to make full use of modern day multi-core CPUs.

During our experiments, we allowed Z3 to spawn (up to) 4 threads, comprising three concurrent main threads that share unit literals and learned clauses, and one local search thread that uses WalkSAT (Selman et al. (1993)) to find a satisfying assignment. Moreover, we also enabled the lookhadead solver as a simplifier during in-processing, which uses slighty more powerful techniques for learning clauses.

4.4 Experiments

Setup

The hardware and software specifications of the machine used to carry out the experiments are the following:

- CPU: Intel® Core™ i5-7600K @ 3.8 GHz (4 cores)
- RAM: 8 GB DDR4 2400 MHz
- OS: Ubuntu 20.04
- Environment: Python 3.9.7, Z3 Solver 4.8.12

Results

The first test carried out involved determining a baseline as well as testing the impact (if any) of the simple sorting heuristic described in Section 2.2, like we did for CP as well. The results of this run are shown in Figure 6. Differently from CP, the sorting heuristic seems to be not so helpful: both runs solve 14 out of 40 instances, but the run with the sorted circuits by decreasing area takes slightly higher runtimes.

In Figure 7, we show the results of the runs with and without the custom search described in Section 4.3. In the upper part of the plot, where the baseline models without rotation are shown, we can see that enabling the custom search allows to solve the eleventh instance that the baseline model without the custom search wasn't able to solve, but no other interesting improvement seems to be derived from the custom search. In the lower part, where the baseline models with rotations are shown, we can see that the baseline model with rotations is able to solve only 9 instances out 40 and the custom search activation again leads the baseline model being able to solve one instance more (10 out of 40) even though it has slightly higher runtimes.

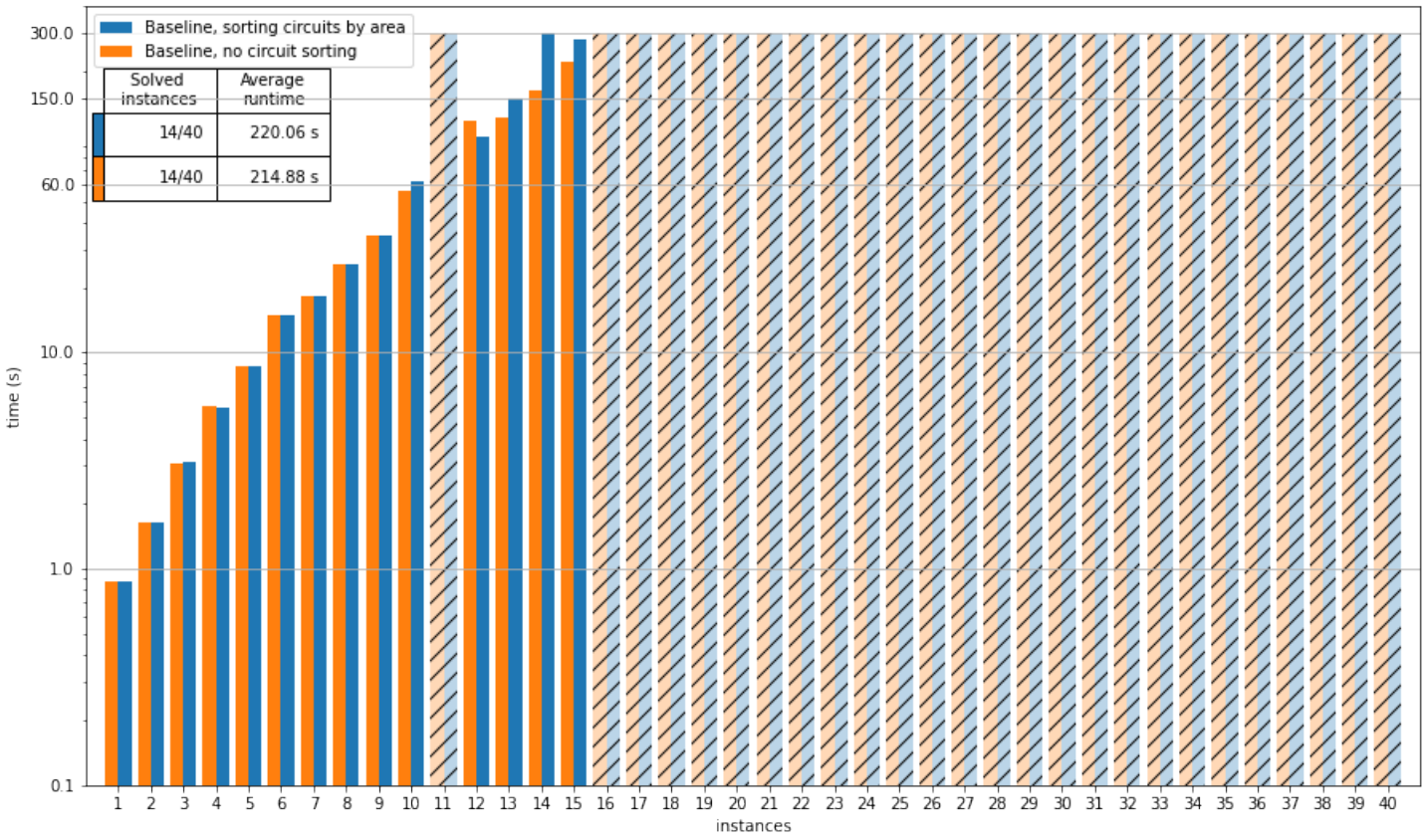


Figure 6: Baseline performances for the SAT approach, with and without circuit sorting as preprocessing step.

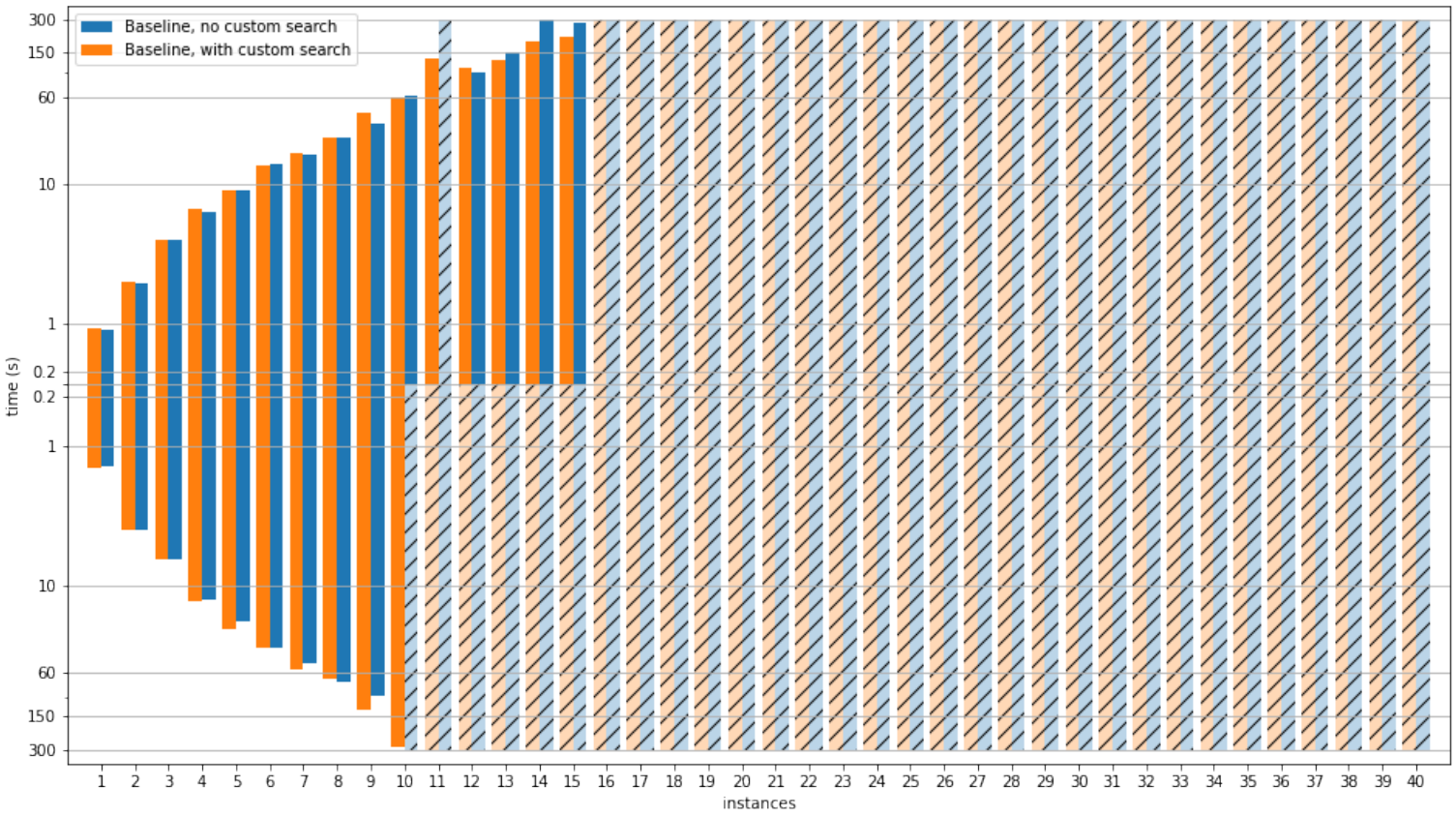


Figure 7: Performances for the SAT approach, with and without custom search enabled. In the upper section of the plot, models without rotations are shown, while in the lower section, models with rotations enabled are shown.

5. SMT

SAT solvers are a powerful tool, but they suffer one main drawback: they can only deal with propositional logic encodings, which for many classes of problems can be very complex and expensive. Indeed, many problem can be formalized in a more natural way using some other logic, rather than propositional logic.

Satisfiability Modulo Theories (SMT) is the problem of determining whether a certain first-order formula is satisfiable, based on some background theories. SMT solvers indeed allow for a greater expressivity and scalability compared to SAT solvers, although there is a slight loss of efficiency w.r.t. their propositional counterpart. In this section, a SMT model for the VLSI problem is described.

5.1 Encoding

Variables

The first encoding that we implemented was inspired by the CP model, and it made use primarily of the Arrays theory offered by SMT solvers. In particular, the main variables are four arrays `x, y, xhat, yhat` of size n, where `x[i]` and `y[i]` represent the dimensions of circuit i, whereas `xhat[i]` and `yhat[i]` represents the bottom-left corner of the circuit. However, preliminary tests of this encoding have shown that it performed very poorly, even on the simpler instances without circuits rotation (see Figure 10).

Hence, we dropped this encoding in favor of a different (and probably also better suited for a SMT solver) formalization. We first define an integer constant w for the plate's width and an integer variable[1] $l \in [minl, maxl]$ for the length; this latter variable will also act as the objective function that will be minimized. Then, for each circuit i, we define two integer constants x_i, y_i for its dimension and two integer variables $\hat{x}_i, \hat{y}_i$ for its coordinates.

Main problem formulas

The problem definition begins with the introduction of formulas to bound the circuits' coordinates:

$$\forall i,\ \hat{x}_i \geq 0 \wedge \hat{x}_i < w \wedge \hat{x}_i + x_i \leq w \tag{4}$$

$$\forall i,\ \hat{y}_i \geq 0 \wedge \hat{y}_i < l \wedge \hat{y}_i + y_i \leq l \tag{5}$$

Then, we need to impose the no-overlap relationship between circuits:

$$\bigwedge_{i=1}^{n} \Big(\bigwedge_{j=i+1}^{n} (\hat{x}_i + x_i \leq \hat{x}_j \vee \hat{y}_i + y_i \leq \hat{y}_j \vee \hat{x}_y + x_y \leq \hat{x}_i \vee \hat{y}_j + y_j \leq \hat{x}_i) \Big) \tag{6}$$

1. Note that what we call "constants" and "variables", in this context, formally are defined as constants in both cases (either interpreted or uninterpreted).

Implied constraints

Considering the problem at hand, we can make the following consideration: if we draw a horizontal line and sum the horizontal sides of the traversed circuits, the sum can be at most w. A similar property holds if we draw a vertical line. These implied constraints are added to the encoding with the following formulas:

$$\forall c \in [1, w], \sum_{i=1}^{n} f_{col}(c, i) \leq l \tag{7}$$

$$\forall r \in [1, maxl], \sum_{i=1}^{n} f_{row}(r, i) \leq w \tag{8}$$

where $f_{col}(c, i)$ and $f_{row}(r, i)$ are defined as

$$f_{col}(c, i) = \begin{cases} y_i & \hat{x}_i < c \wedge c \leq \hat{x}_i + x_i \\ 0 & otherwise \end{cases}$$

$$f_{row}(r, i) = \begin{cases} x_i & \hat{y}_i < r \wedge r \leq \hat{y}_i + y_i \\ 0 & otherwise \end{cases}$$

Symmetry breaking formulas

Ideas described in section 3.2 were replicated in the SMT model. Firstly, an ordering between the first block and the second one is imposed:

$$\hat{x}_0 \leq \hat{x}_1 \wedge \hat{y}_0 \leq \hat{y}_1$$

Then, rows and columns symmetry breaking constraints were formalized as follows:

$$\bigwedge_{i=1}^{n} \Big(\bigwedge_{j=i+1}^{n} ((\hat{x}_i = \hat{x}_j \wedge x_i = x_j) \implies \hat{y}_i \leq \hat{y}_j) \Big) \tag{9}$$

$$\bigwedge_{i=1}^{n} \Big(\bigwedge_{j=i+1}^{n} ((\hat{y}_i = \hat{y}_j \wedge y_i = y_j) \implies \hat{x}_i \leq \hat{x}_j) \Big) \tag{10}$$

Finally, the three-blocks constraints:

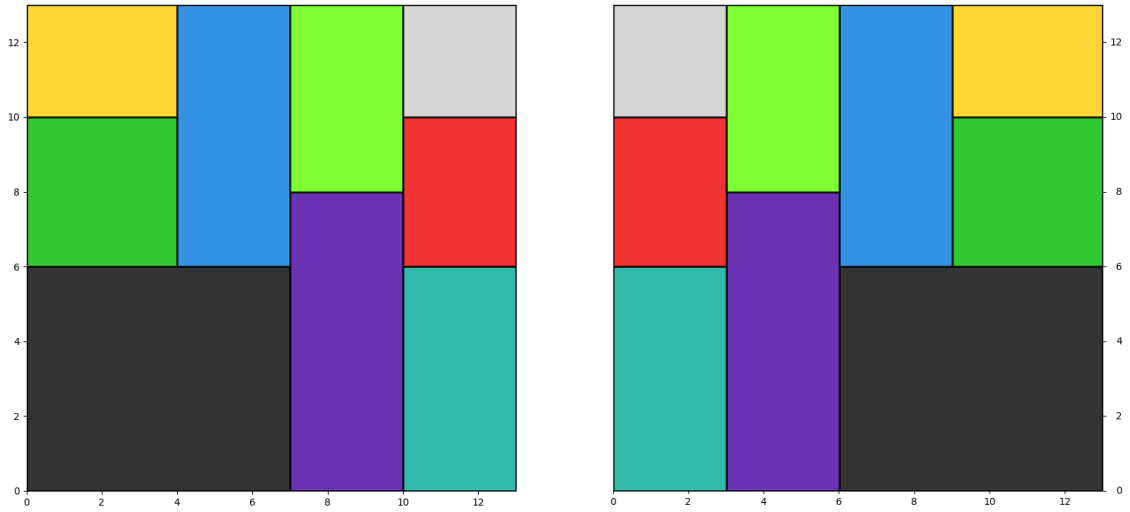
$$\bigwedge_{i=1}^{n} \Big(\bigwedge_{j=i+1}^{n} \Big(\bigwedge_{k=j+1}^{n} ((\hat{x}_i = \hat{x}_j \wedge x_i = x_j \wedge \hat{y}_i = \hat{y}_k \wedge y_i + y_j = y_k) \implies \hat{x}_k \leq \hat{x}_i) \Big) \Big) \tag{11}$$

$$\bigwedge_{i=1}^{n} \Big(\bigwedge_{j=i+1}^{n} \Big(\bigwedge_{k=j+1}^{n} ((\hat{y}_i = \hat{y}_j \wedge y_i = y_j \wedge \hat{x}_i = \hat{x}_k \wedge x_i + x_j = x_k) \implies \hat{y}_k \leq \hat{y}_i) \Big) \Big) \tag{12}$$

Dual model

In the process of trying to break more symmetries, we introduced a dual model. The idea is to define two sets of boolean variables $\tilde{X} = \{\tilde{x}_0, .., \tilde{x}_{w-1}\}$ and $\tilde{Y} = \{\tilde{y}_0, .., \tilde{y}_{maxl-1}\}$, such that $\tilde{x}_i, \tilde{y}_i$ is true only if the sum of the dimensions of the circuits that have their origin respectively in $\hat{x}_i, \hat{y}_i$, equals to the bound of the plate along that axis. Intuitively, $\tilde{x}_i$ being true means that there is a vertical line (at coordinate i) that is always "on the border" of different circuits, that is, we never end completely inside one block (the same applies to $\tilde{y}_i$). See Figure 8 for an example.

With these variables, we want to represent hyper-columns and hyper-rows of circuits that can be exchanged creating a symmetric solution. Then, by imposing a lexicographic ordering between the set of variables of an axis and its symmetric value over the same axis, we are effectively ordering hyper-blocks by their decreasing size (i.e., their aggregate dimension over that axis).



(a) Retained Solution (b) Pruned Solution

Figure 8: An example to explain how dual model works in the SMT encoding.

In solution 8a, $\tilde{X}$ has true values for $\hat{x}$ equal to 0, 7, 10, represented by the set of values $\{T, F, F, F, F, F, F, T, F, F, T, F, F\}$; while the symmetric solution 8b has true values for $\hat{x}$ equal to 0, 3, 6, represented by the set of values $\{T, F, F, T, F, F, T, F, F, F, F, F, F\}$. Actually, there are other symmetric solutions that can be derived from this representation of the board, like $\{T, F, F, T, F, F, F, F, F, F, T, F, F\}$, but we weren't able to address them into the SMT encoding.

Moreover, this representation has two general cases where there is no effective way to impose an ordering among the symmetric solutions:

- when the solution has only one hyper-block over an axis, like in Fig.8 where $\tilde{Y}$ is $\{T, F, F, F, F, F, F, F, F, F, F, F, F\}$

- when the hyper-blocks have the same dimensions, e.g.: $\{T, F, F, T, F, F, T, F, F\}$

The formalization is as follows:

$$\forall c \in [0, w-1],\ \tilde{x}_c \iff (l = \sum_{i=1}^{n} g_{col}(c, i)) \tag{13}$$

$$\forall r \in [0, maxl-1],\ \tilde{y}_r \iff (w = \sum_{i=1}^{n} g_{row}(r, i)) \tag{14}$$

where $g_{col}(c, i)$ and $g_{row}(r, i)$ are defined as

$$g_{col}(c, i) = \begin{cases} y_i & \hat{x}_i = c \\ 0 & otherwise \end{cases}$$

$$g_{row}(r, i) = \begin{cases} x_i & \hat{y}_i = r \\ 0 & otherwise \end{cases}$$

Then, we impose the LEX $\leq$ constraint between $\tilde{X}$ and its symmetric value with the same constraints explained in the SAT encoding (see Equation 3).

5.2 Rotation

In order to allow the circuits' rotation, x_i, y_i are encoded as integer variables, not constants anymore, and we introduce a boolean variable r_i for each circuit, to represent whether that circuit is rotated or not. Then, we add constraints to assign values to x_i, y_i:

$$x_i = \begin{cases} y_i^{input} & r_i \\ x_i^{input} & otherwise \end{cases}$$

$$y_i = \begin{cases} x_i^{input} & r_i \\ y_i^{input} & otherwise \end{cases}$$

where x_i^{input}, y_i^{input} are the dimensions of circuit i that are given in input.

5.3 Experiments

Setup

The hardware and software specifications of the machine used to carry out the experiments are the following:

- CPU: Intel® Core™ i5-7600K @ 3.8 GHz (4 cores)
- RAM: 8 GB DDR4 2400 MHz
- OS: Ubuntu 20.04
- Environment: Python 3.9.7, Z3 Solver 4.8.12

Results

The first test carried out involved determining a baseline as well as testing the impact (if any) of the simple sorting heuristic described in Section 2.2, like we did for the CP and SAT tests as well. The results of this run are shown in Figure 9.

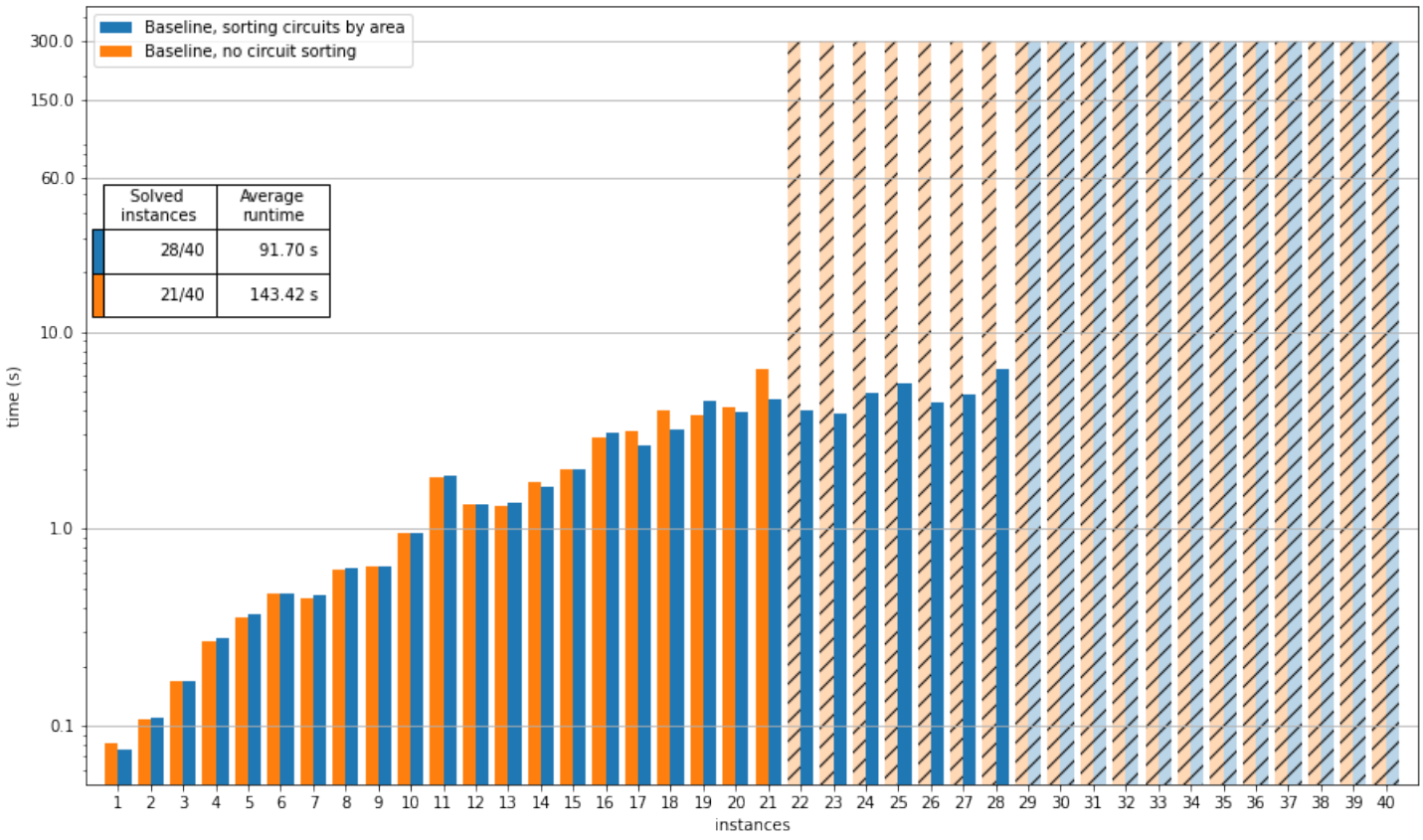


Figure 9: Baseline performances for the SMT approach, with and without circuit sorting as preprocessing step.

We can clearly see that this simple heuristic of sorting circuits by their decreasing area, helps also in solving SMT instances: without ordering the circuits the base model is able to solve only 21 instances out of 40, while having sorted circuits leads to 28 solved instances out of 40. This is interesting also because although this heuristic

did help with the CP model, it didn't for the SAT encoding (and the SMT encoding adopts the same solver as the SAT one).

In Figure 10, we show the results of the array encoding with respect to the baseline model with circuits sorted by area.

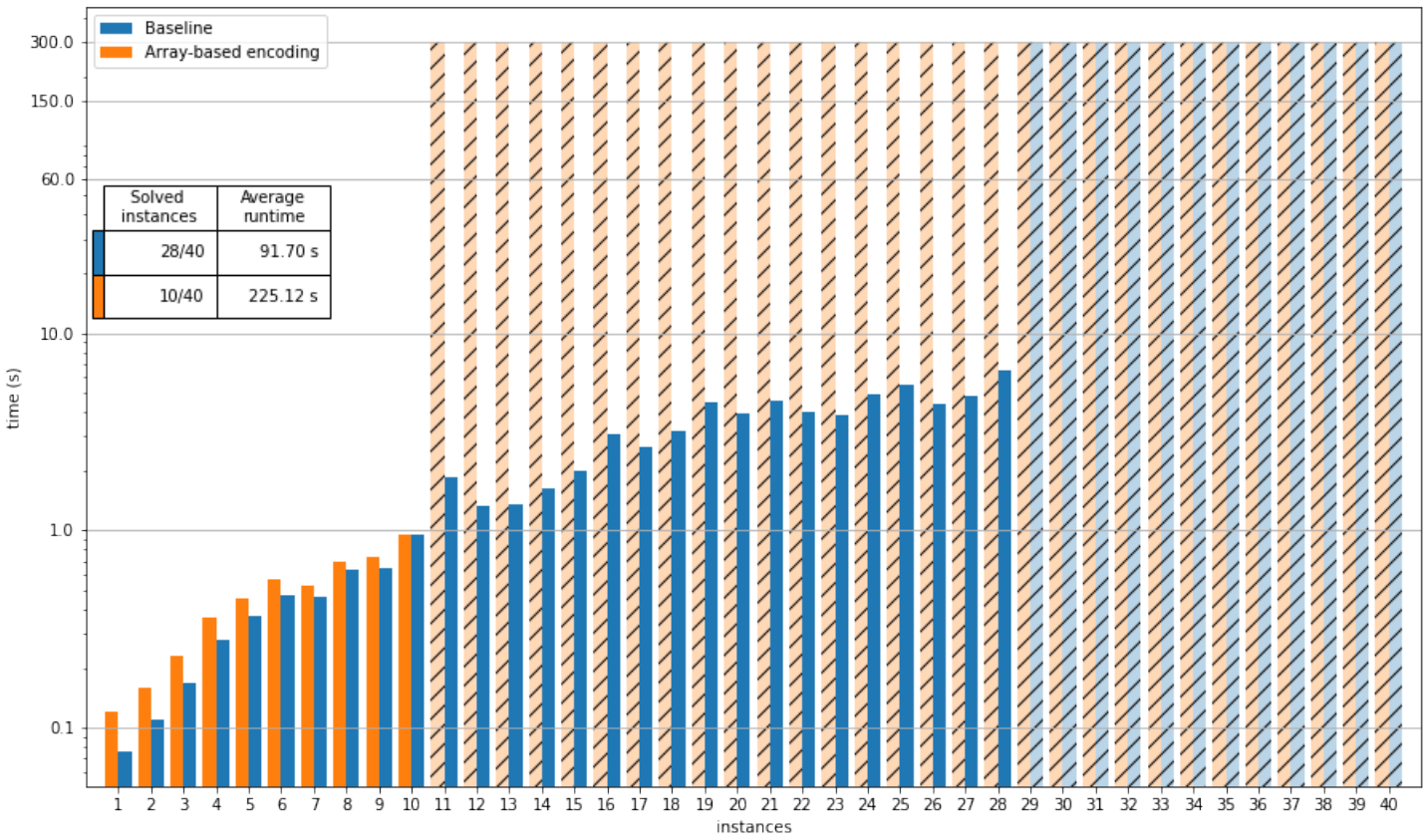


Figure 10: Performances of the array-based SMT encoding compared to baseline performances.

As described in section 5.1, the array-based model is performing way worse that the baseline model: it is able to solve only 10 instances out of 40 and with a higher average runtime. Due to this, we didn't do any more experiment with this model.

Finally, we run experiments for the baseline with the dual model and for the model with rotations; results are shown in Figure 11. Unfortunately, the baseline with the dual model is not improving the performances with respect to the simple

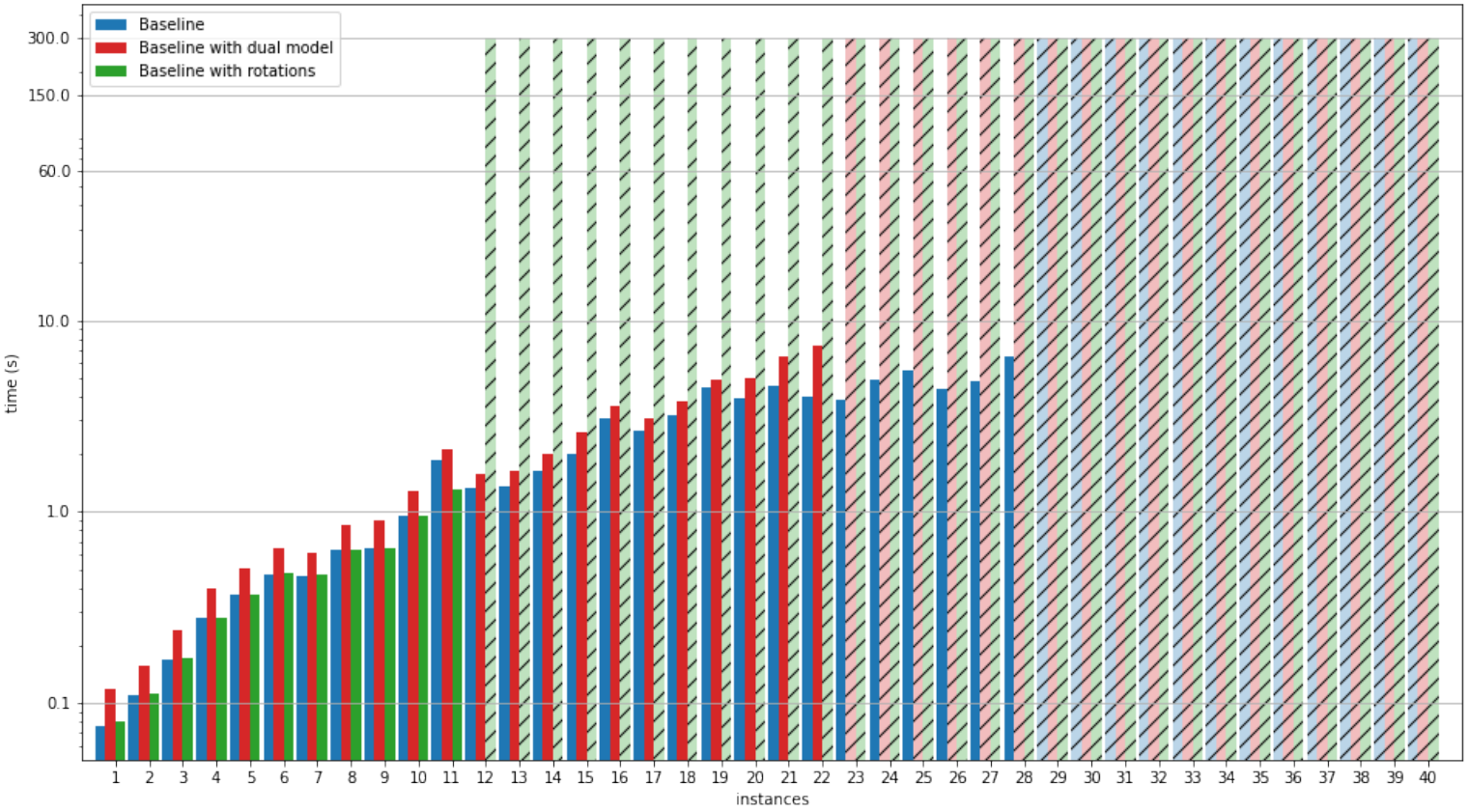


Figure 11: Performances of the SMT encoding with the dual model and with rotations enabled.

baseline model: it solves fewer instances (22 out of 40) and also takes a higher average runtime. We can conclude that the dual model only increases the complexity without really improving the performances. As seen in the previous technologies, the model with rotations is not able to solve as many instances as the baseline ones, to be precise it solves only 11 instances out of 40, even though the runtime of the solved instances isn't much higher than the baseline ones.

6. Conclusion

In this report, we have shown three different optimization approaches to a (simplified) version of the VLSI problem, which is formalized as a rectangle packing problem. In particular, a CP model as well as SAT and SMT encodings have been designed and developed, and experimental tests have been carried out to determine the performances of each proposed solution. Several techniques to build better performing models have also been investigated, implemented and tested, such as symmetry breaking constraints, dual models and tweaks to the underlying search process via search heuristics.

The CP model surely appears to be the best-performing approach, managing to solve 37 instances out of the 40 provided with an average solve time of 38 seconds, while the worst-performing approach appears to be the SAT encoding, which can solve only 15 instances; SMT lies somewhere in between, with 28 instances solved.

Moreover, this project has clearly highlighted the fact that, even if dealing with NP-Complete (i.e., theoretically unfeasible) problems, modern techniques such as CP and SAT/SMT solvers nowadays allow to reach very good performances even on difficult instances, if aided by a good model of the problem. Indeed, as we already said previously there is no general "best" approach that works for any combinatorial optimization problem, but rather, the best approach to such problems involves a (sometimes lengthy) trial and error process.

Bibliography

Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M Wintersteiger. Programming z3. In *International Summer School on Engineering Trustworthy Software Systems*, pages 148–201. Springer, 2018.

Alan M Frisch, Christopher Jefferson, and Ian Miguel. Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In *ECAI*, volume 16, page 171, 2004.

Eric Huang and Richard E Korf. Optimal rectangle packing: An absolute placement approach. *Journal of Artificial Intelligence Research*, 46:47–87, 2013.

Philippe Refalo. Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 557–571. Springer, 2004.

Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, volume 5, page 148. Morgan Kaufmann Pub, 1996.

Bart Selman, Henry A Kautz, Bram Cohen, et al. Local search strategies for satisfiability testing. *Cliques, coloring, and satisfiability*, 26:521–532, 1993.

Appendix I. Graphs

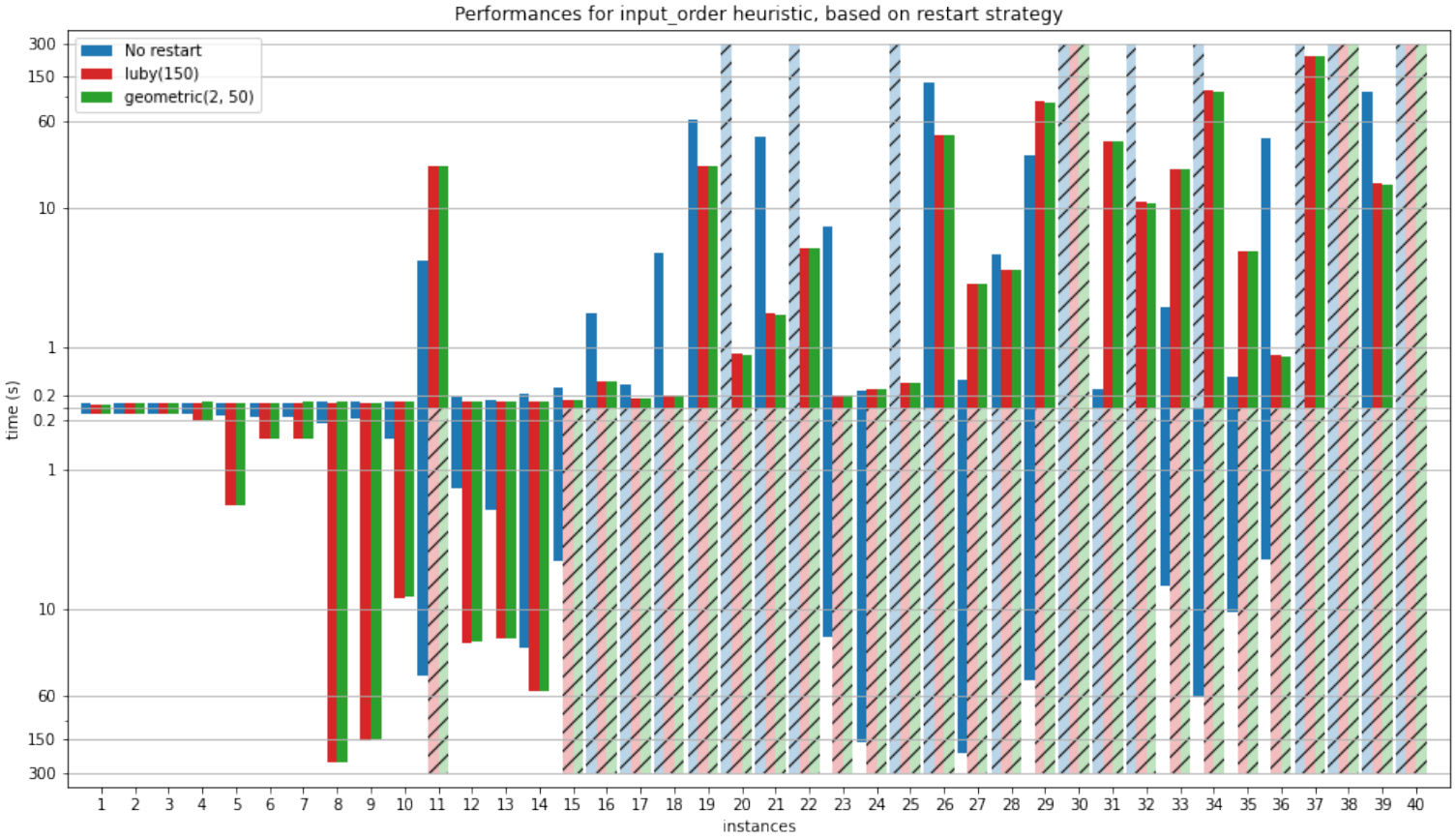


Figure 12: Performances for the input_order search heuristic, grouped by restart strategy. Upper part of the graph shows performances without rotations, the lower part with rotations.

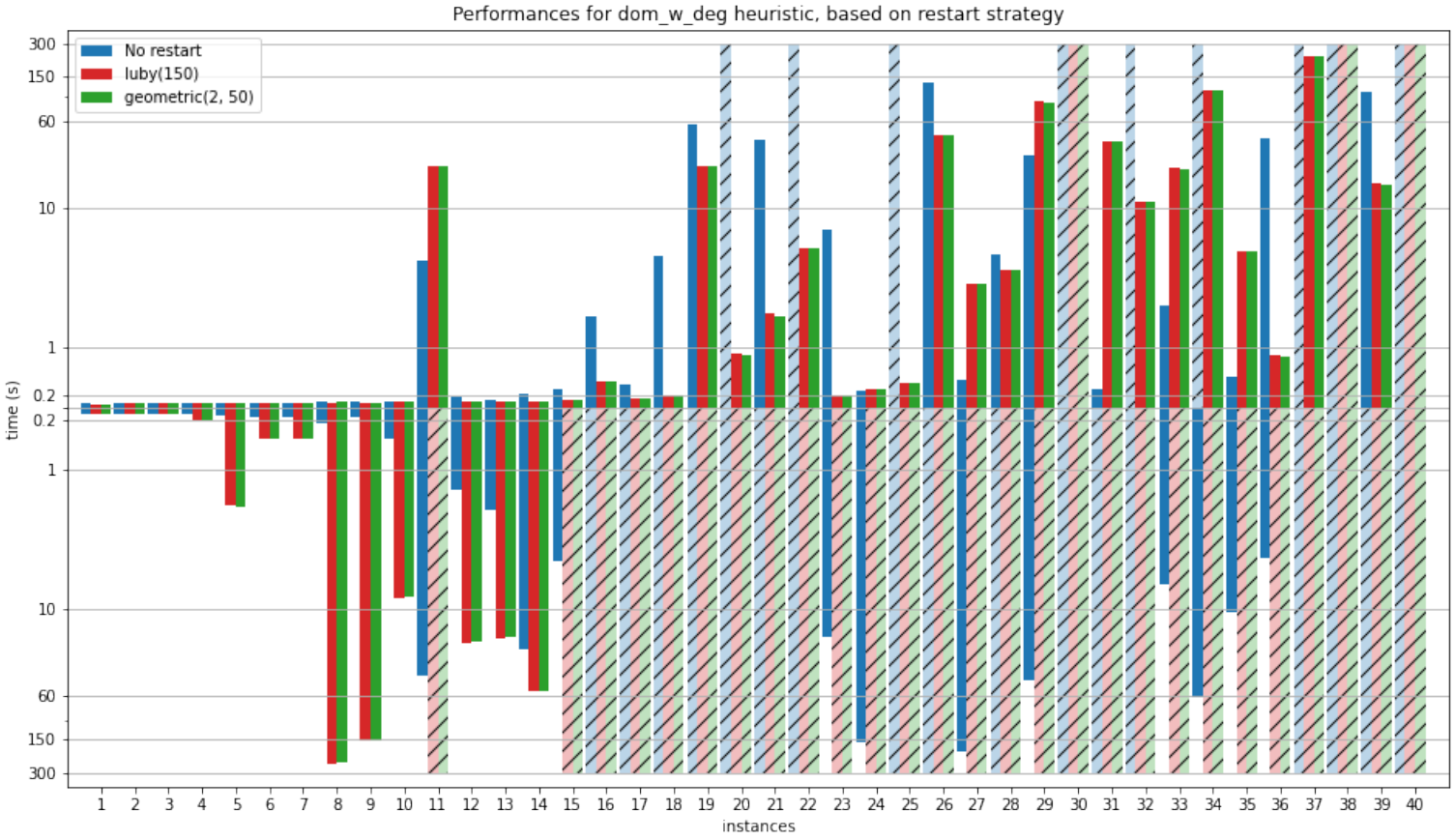


Figure 13: Performances for the domWdeg search heuristic, grouped by restart strategy. Upper part of the graph shows performances without rotations, the lower part with rotations.

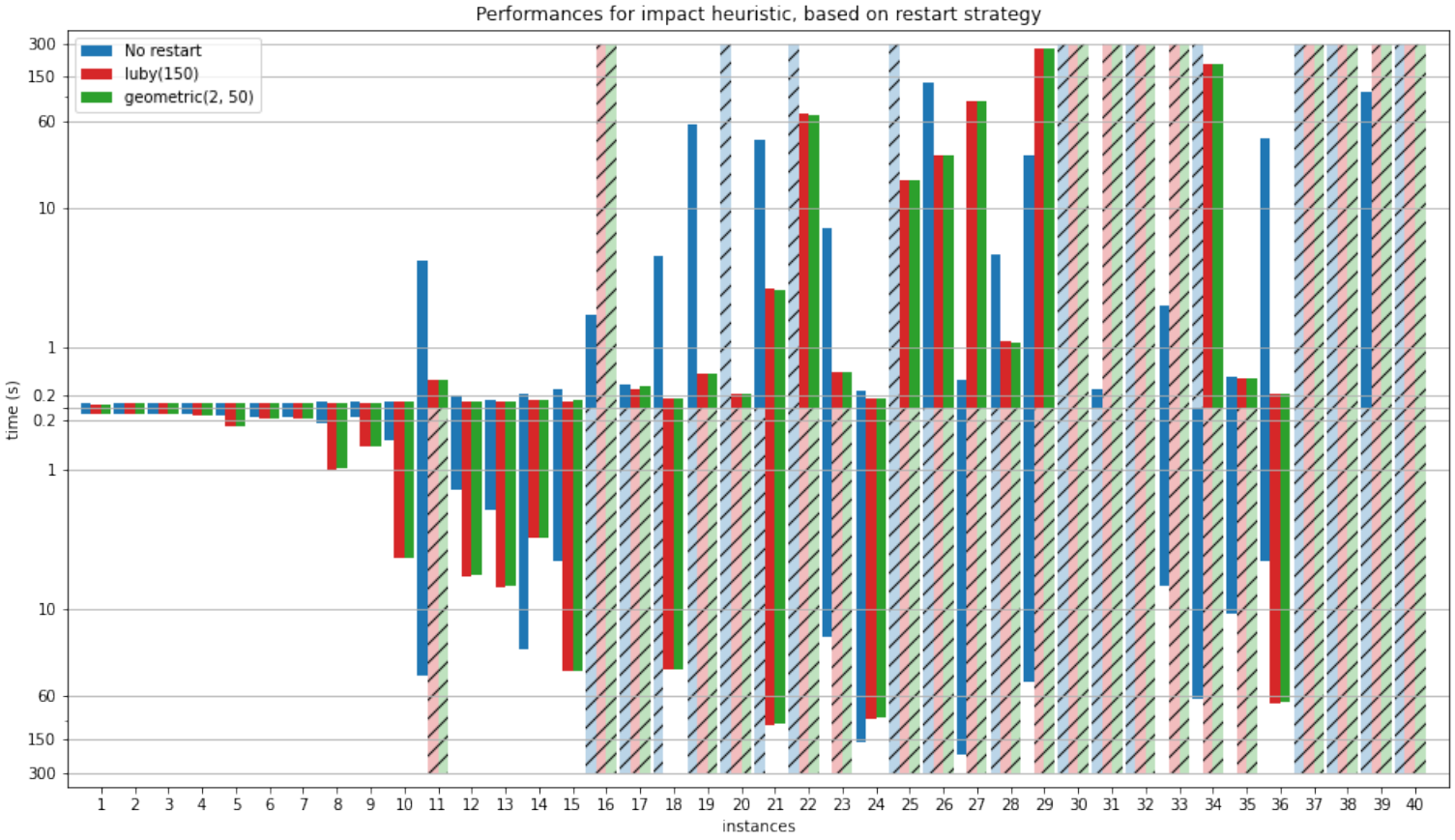


Figure 14: Performances for the impact search heuristic, grouped by restart strategy. Upper part of the graph shows performances without rotations, the lower part with rotations.