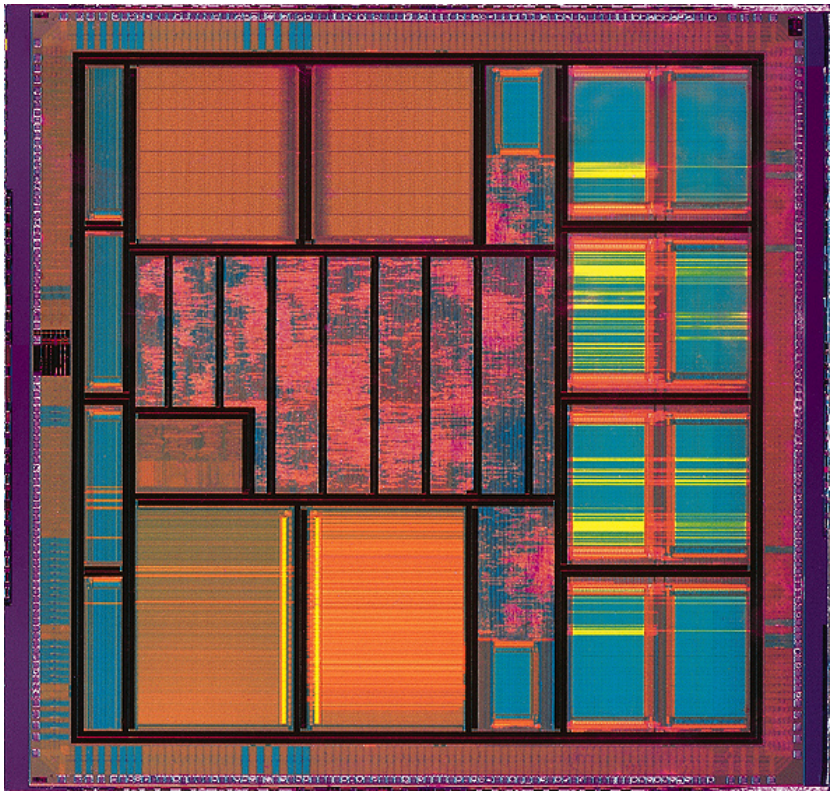
VLSI design

Combinatorial Decision making and Optimization

Project report

Alma Mater Studiorum

Ildebrando Simeoni - ildebrando.simeoni@studio.unibo.it
Diego Biagini - diego.biagini2@studio.unibo.it



Contents

1 Introduction

Very large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining millions of MOS (Metal Oxide Semiconductor) transistors onto a single chip. VLSI began in the 1970s when MOS integrated circuit chips were widely adopted, enabling complex semiconductor and telecommunication technologies to be developed. Structured VLSI design is a modular methodology originated by Carver Mead and Lynn Conway for saving microchip area by minimizing the interconnect fabrics area. This is obtained by repetitive arrangement of rectangular macro blocks which can be interconnected using wiring by abutment.
The aim of this project is to deal with this specific problem via a Combinatorial Optimization approach. In particular four different technologies have been adopted to address the problem, namely: Constraint Programming (CP), propositional SATisfiability (SAT), Satisifability Modulo Theory (SMT) and Mixed-Integer Linear Programming (MIP).

2 Preliminaries

Some preliminar information about the problem are shown here.

2.1 Problem description

The objective of the project is to design the VLSI of the circuits defining their electrical device, namely: given a fixed-width plate and a list of rectangular circuits, decide how to place them on the plate so that the length of the final device is minimized (improving its portability).
Two variants of the problem are considered. In the first, each circuit must be placed in a fixed orientation with respect to the others. This means that, an n×m circuit cannot be positioned as an m×n circuit in the silicon plate. In the second case, the rotation is allowed, which means that an n × m circuit can be positioned either as it is or as m × n.

2.2 Instance format

2.2.1 Input instance

An input instance of the problem is a text file consisting of lines of integer values. The first line gives w, which is the width of the silicon plate. The following line gives n, which is the number of necessary circuits to place inside the plate. Then n lines follow, each with x_i, y_i representing the horizontal and vertical dimensions of the i-th circuit.

8		w
4		n
3	3	x_i, y_i
3	5	x_i, y_i
5	3	x_i, y_i
5	5	x_i, y_i

From an instance we can then define the following parameters:

- **n-blocks**: the number of circuits given as input
- **max-width**: the maximum possible width of the plate, later referred to as W as well
- **height, width**: list of heights and widths, representing the height or width of each circuit, sometimes referred to as h, w

2.2.2 Output instance

An output instance of the problem is, as the input one, a text file consisting of lines of integer values. Where to place a circuit i can be described by the position of i in the silicon plate. The solution should indicate the length of the plate l, as well as the position of each i by its $\hat{x_i}$ and $\hat{y_i}$, which are the coordinates of the bottom-left corner i. This could be done by for instance adding l next to w, and adding $\hat{x_i}, \hat{y_i}$ next to x_i, y_i in the instance file.

8	8			w,l
4				n
3	3	0	0	$x_i, y_i, \hat{x_i}, \hat{y_i}$
3	5	0	3	$x_i, y_i, \hat{x_i}, \hat{y_i}$
5	3	3	0	$x_i, y_i, \hat{x_i}, \hat{y_i}$
5	5	3	3	$x_i, y_i, \hat{x_i}, \hat{y_i}$

2.3 Approximate Solutions

While the aim of this project is to solve the problem optimally, there have been many attempts to solve it approximately using less backtracking than exact methods or none at all [5].
While these methods do not provide us with what we are after they are a nice way to obtain an initial suboptimal solution which might be a good starting point. In particular such a solution can help us in determining the most important bound of the problem, that is the upper bound on the height of the plate.

For this reason in all but the CP implementation of the problem we initially obtained the solution given by the so called **BL-algorithm (Bottom-left)**.
This algorithm is extremely simple, it works as follows:

1. Order the circuits by non decreasing width
2. Scan each position of the plate, starting from the bottom left corner and moving in the right direction first, then in the up direction
3. If a position which can fit the circuit is found (there would be no overlap between the circuit, the already placed circuits and the plate bounds) place it there and go to the next one
4. Repeat steps 2 and 3 until all circuits have been placed

It turned out that the solutions found through this method gave very effective bounds for the given instances, in general they exceeded the optimal solution by around 5 squares for the easy-medium instances and around 10 squares for the harder ones.

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 this approach the following decision variables were defined:

- **cornerx, cornery**: x and y coordinates of the bottom left corner of each circuit

In particular a slight improvement in the overall performances has been registered when the domain of these two decision variables (and of the objective function) has been reduced from:

```
array[BLOCKS] of var int: cornery;
array[BLOCKS] of var int: cornerx;
```

to:

```
array[BLOCKS] of var 0..h-min(height): cornery;
array[BLOCKS] of var 0..w-min(width): cornerx;
```

3.2 Objective function

The aim of the program is to minimize the feasible height of the entire plate, so this has been selected as the objective function to minimize. Because of the similarities with a variety of operational research problems the decision variable encoding the height of the plate has been named **makespan** and it has been defined as the maximum y point reached by any of the circuits (blocks):

```
var int: makespan = max(b in BLOCKS)(cornery[b] + height[b]);
```

3.2.1 Bounds

In order to enhance model's performances and to solve more complex instances, tighter objective function's bounds were needed.
In particular the lower bound has been defined as:

$$lower\_bound = \frac{\sum_{i=1}^{n-blocks} width_i * height_i}{max - width}$$

defined by summing the areas of all circuits and then dividing by the given maximum width.
While, after a number of experiments, the upper bound has been defined as the naive formulation suggests, namely by the sum of all the heights of the blocks positioned one in top of the other:

$$upper\_bound = \sum_{i=1}^{n-blocks} height_i$$

Which, even if it's a high overestimation of the real upper-bound, works properly for almost all the instances provided, reaching even better results than more sophisticated tighter approximation of the upper bound.
Finally the objective function has been constrained into those bounds by the following constraints:

```
constraint makespan >= lower_bound;
constraint makespan <= upper_bound;
```

Which seemed to be a better encoding (evaluating CP model performances) rather than:

```
var lower_bound..upper_bound: makespan = max(b in BLOCKS)(cornery[b] +
height[b]);
```

3.3 Constraints

In order to let the solver find feasible solutions in a reasonable amount of time a variety of constraints need to be derived from the problem specification.
In this subsection the main problem constraints, implied constraints and symmetry breaking constraints are analyzed.

3.3.1 Main constraints

Starting from the problem specification we can define the two main constraints:

- considering each circuit in the plate, the sum of its height plus the y coordinate of its bottom left corner must be less or equal than the makespan
- the sum of the width of each circuit plus the x coordinate of its bottom left corner must be less or equal than the maximum width provided as input

```
constraint forall(b in BLOCKS) (cornery[b] + height[b] <= makespan);
constraint forall(b in BLOCKS) (cornerx[b] + width[b] <= max_width);
```

A sort of implied constraint which increases model's performances is the following one:

```
constraint forall(b in BLOCKS) (cornery[b]  < makespan);
```

namely that the starting position of each circuit must be strictly lower than the makespan.
Another important constraint is the non-overlapping one; the **diffn** MiniZinc global constraint has been chosen in order to express the aforementioned constraint. This one constrains rectangles i,j (given by their origins and sizes) to be non-overlapping.

```
constraint diffn(cornerx, cornery, width, height);
```

The use of global constraints as this one resulted in a drastic increase of model performance.

3.3.2 Implied constraints

As stated in the effective modelling practices in the MiniZinc handbook, another significant way to improve model's performances is by way of implied constraints.
Those constraints, which are semantically redundant (i.e. there is not a change in the set of solutions), are actually computationally significant because they can greatly reduce the search space by making more information available to the solver earlier.
In our solution a different view of the problem was proposed in order to add useful implied constraints that actually proved to be highly effective in the solving process. In particular our problem was seen as a resource allocation problem, in which the coordinates of the rectangles were the starting positions, height and width duration and resource requirements, max-width and makespan the limits, imposing that via global **cumulative** constraint as follows:

```
constraint cumulative(cornery, height, width, max_width);
constraint cumulative(cornerx, width,height, makespan);
```

3.3.3 Symmetry breaking constraints

Symmetry is very common in constraint satisfaction and optimisation problems, a state leading to a solution or a failure will have many symmetrically equivalent states, that's especially bad when proving optimality, infeasibility or looking for all solutions, that's why symmetry breaking constraints are so useful in order to reduce the set of solutions and search space.
In our problem we identified two main possible symmetries, namely the row-block and column-block symmetries (shown in Figure 1).
Both of them have been avoided by the way of the following two symmetry breaking constraints, which impose an order between the two rectangles.

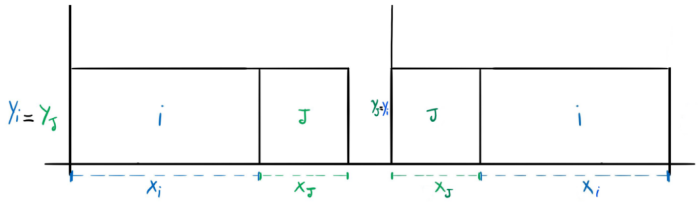


Figure 1: Row-block symmetry

```
constraint forall (i,j in BLOCKS where i < j) (
(cornerx[i] = cornerx[j] /\ width[i] = width[j]) ->
cornery[i] <= cornery[j] );

constraint forall (i,j in BLOCKS where i < j) (
(cornery[i] = cornery[j] /\ height[i] = height[j]) ->
cornerx[i] <= cornerx[j] );
```

Successively a more efficient solution was developed; rather then imposing the ordering via inequalities, the **lex-less** global constraint was preferred, and also an ordering between same sized rectangles was imposed. This resulted in a sizable performance improvement for our model.

```
constraint forall (i,j in BLOCKS where
                   i < j /\cornerx[i] = cornerx[j] /\
                   width[i] = width[j] /\ cornery[i] + height[i] = cornery[j])
                   (lex_less([cornery[i]], [cornery[j]]) );

constraint forall (i,j in BLOCKS where
                   i < j /\cornery[i] = cornery[j] /\
                   height[i] = height[j] /\ cornerx[i] + width[i] = cornerx[j])
                   (lex_less([cornerx[i]], [cornerx[j]]) );

constraint forall (i,j in BLOCKS where
                   i < j /\height[i] = height[j]/\
                   width[i] = width[j] )
                   (lex_less([cornerx[i], cornery[i]],
                             [cornerx[j], cornery[j]]) );
```

3.4 Rotation

The more general version of the VLSI problem we are considering allows circuits rotation, meaning that a n × m circuit can be placed as an m × n circuit on the plate.
In order to model this, a new array of boolean decision variables (i.e. **rotation**) has been defined; with this we can attach a variable to each circuit which defines whether the circuit has been rotated or not.
Some slight variations needed to be implemented on the previous model, like the new horizontal and vertical circuit dimensions:

```
array[BLOCKS] of var int: width_r = [if rotation[b]==true then height[b]
else width[b] endif | b in BLOCKS];

array[BLOCKS] of var int: height_r = [if rotation[b] then width[b]
else height[b] endif | b in BLOCKS];
```

Also some specific rotation related constraints have been implemented; it has been imposed that a circuit cannot be rotated if its height is greater than the maximum width (in order to obtain a feasible solution) and that square circuits should not be rotated (in order to avoid time wasting by considering "equivalent" solutions).

```
constraint forall(b in BLOCKS)(height[b] > max_width -> rotation[b]=false);
constraint forall (b in BLOCKS) (height[b] == width[b] -> rotation[b] = false);
```

3.5 Solvers and search strategy

By default in MiniZinc there is no declaration of how we want to search for solutions. This leaves the search completely up to the underlying solver, but most of the times we may want to specify how the search should be undertaken. This requires us to communicate to the solver a search strategy using annotations.
Not only the search strategy but also solver selection plays an important role in the tuning process of a CP model. For our problem two main solvers have been analyzed and tested, with some variations of the basic and most common search strategies in order to obtain the best performance.
In both cases a more complex search strategy using sequential search constructor annotations was preferred in order to obtain better results. The sequential search constructor first undertakes the search given by the first annotation in its list, when all variables in this annotation are fixed it undertakes the second search annotation and so on, until all search annotations are complete.

3.5.1 Gecode

The first analyzed solver is Gecode, an open-source constraint programming system which supports many of MiniZinc's global constraints natively.
Being this the most common choice it supports a variety of search annotations, in particular, for variable choice annotation:

- **input-order**: choose in order from the array
- **dom-w-deg**: choose the variable with the smallest value of domain size divided by weighted degree, which is the number of times it has been in a constraint that caused failure earlier in the search
- **smallest**: choose the variable with the smallest value in its domain

Any kind of depth first search 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 ameliorate this problem is to restart the search from the top thus having a chance to make different decisions. MiniZinc includes annotations to control restart behaviour, which can highly increase solver performance. Because restart search does not make much sense if the underlying search strategy does not do something different the next time it starts at the top, a randomization effect was needed in order to ensure that something is different in each restart.
After a number of experiments the following search strategy for Gecode solver was the one selected

```
solve :: seq_search([int_search([makespan], smallest, indomain_min),
                     int_search(cornerx, dom_w_deg, indomain_random)
                     ])::restart_luby(150) minimize makespan;
```

Thanks to the sequential search the solver tries to set the **makespan** first by choosing its smallest domain value, and then it tries to assign to the variable **cornerx** with dom-w-deg and indomain-random in order to ensure some randomization.
At the end luby as restarting strategy has been selected among the most popular ones for its effectiveness [3].

3.5.2 Chuffed

Although Gecode solver reached considerable results after an accurate parameters tuning (solving 33 out of the 40 instances provided), this was not the best performance obtained during our experiments. Another solver proved to be way more efficient for the problem in hand, Chuffed.
Chuffed is a constraint solver based on lazy clause generation, it adapts techniques from SAT solving, such as conflict clause learning, watched literal propagation and activity-based search heuristics, and can often be much faster than traditional CP solvers.

Because both indomain-random and dom-w-deg annotation were not supported by Chuffed solver a more basic approach was preferred, in particular the following sequential search strategy has been adopted:

```
solve :: seq_search([int_search([makespan], smallest, indomain_min),
                     int_search(cornerx, input_order, indomain_min)
                     ]) minimize makespan;
```

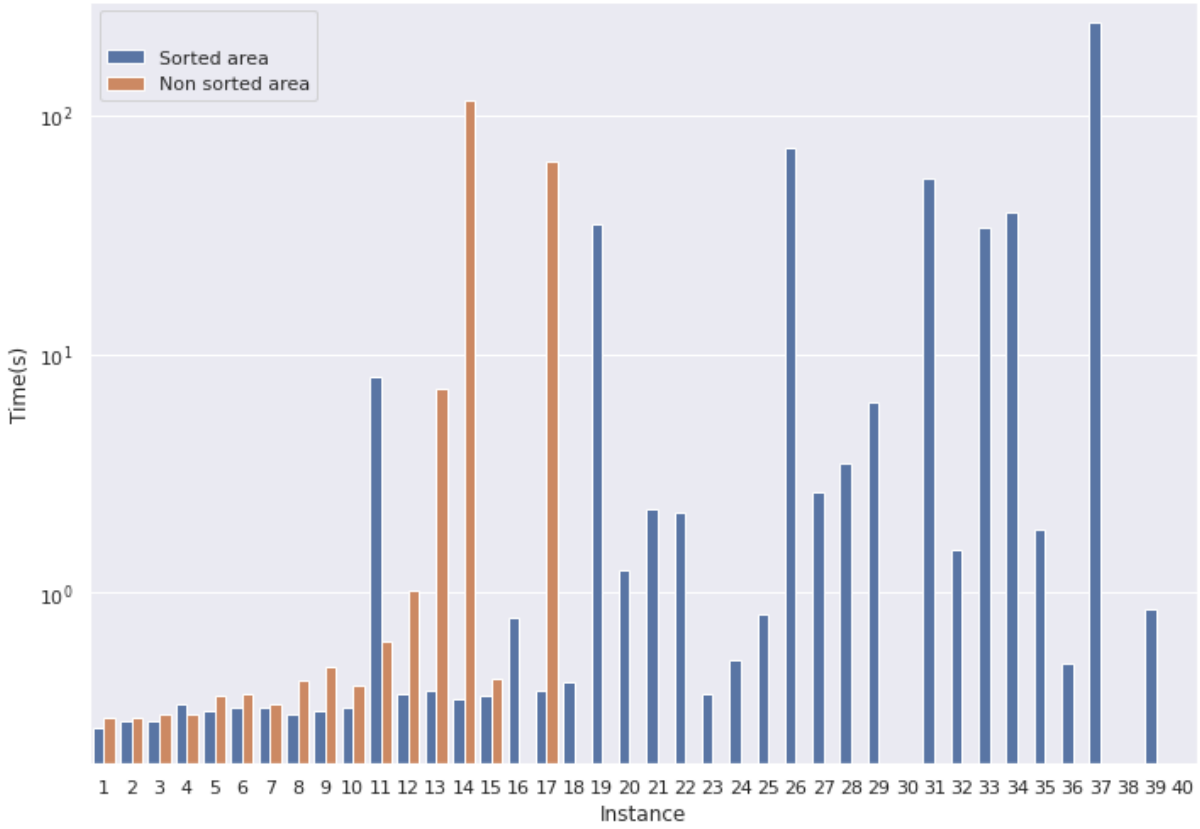
In this case restart has been avoided because the search strategy didn't implement any form of randomization.

3.6 Results

In the following subsection some of the highly interesting results are shown and discussed.
Results are shown in form of bar plots of elapsed time to solve each of the 40 instances provided with different strategies. Times are computed via **instance.solve().statistics** and plotted on a logarithmic scale to show clearly when an instance was solved or not.
The best combination of solver and search strategy (of which the outputs are present in the out folder) is Chuffed with circuits sorted by decreasing area and without rotation.

3.6.1 Area sorting

One of the key strategy specific to this problem that has been adopted was to sort circuits to place at each instance in decreasing order of area. This approach was already tested in the literature [2], observing that: "based on the observation that placing rectangles of larger area is more constraining than placing those of smaller area."
Because of that larger area rectangles will be placed first. This simple heuristic enabled our solver to highly increment its performances as shown:

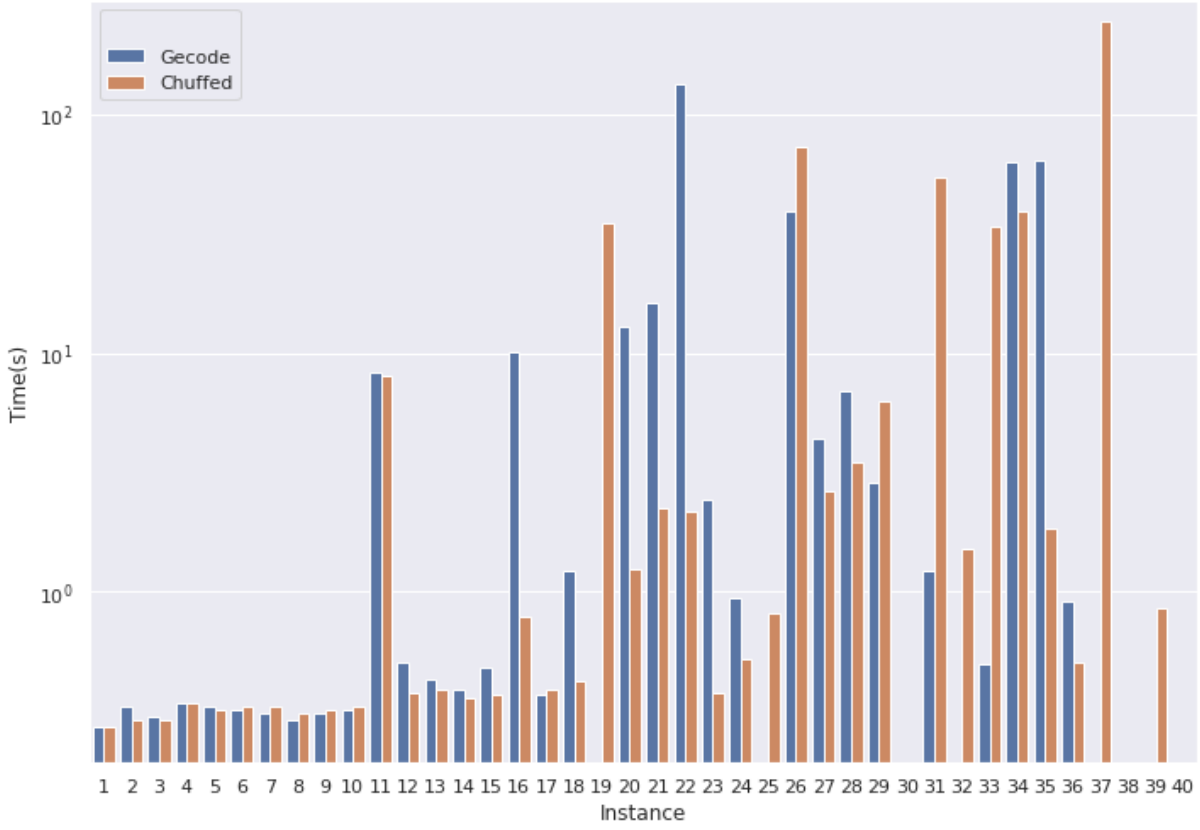


As can be seen without this preprocessing step a lot of the instances wouldn't actually be solved with our most performing solver.

Because of this performance enhancement, the aforementioned step is performed before evaluating all the following strategies.

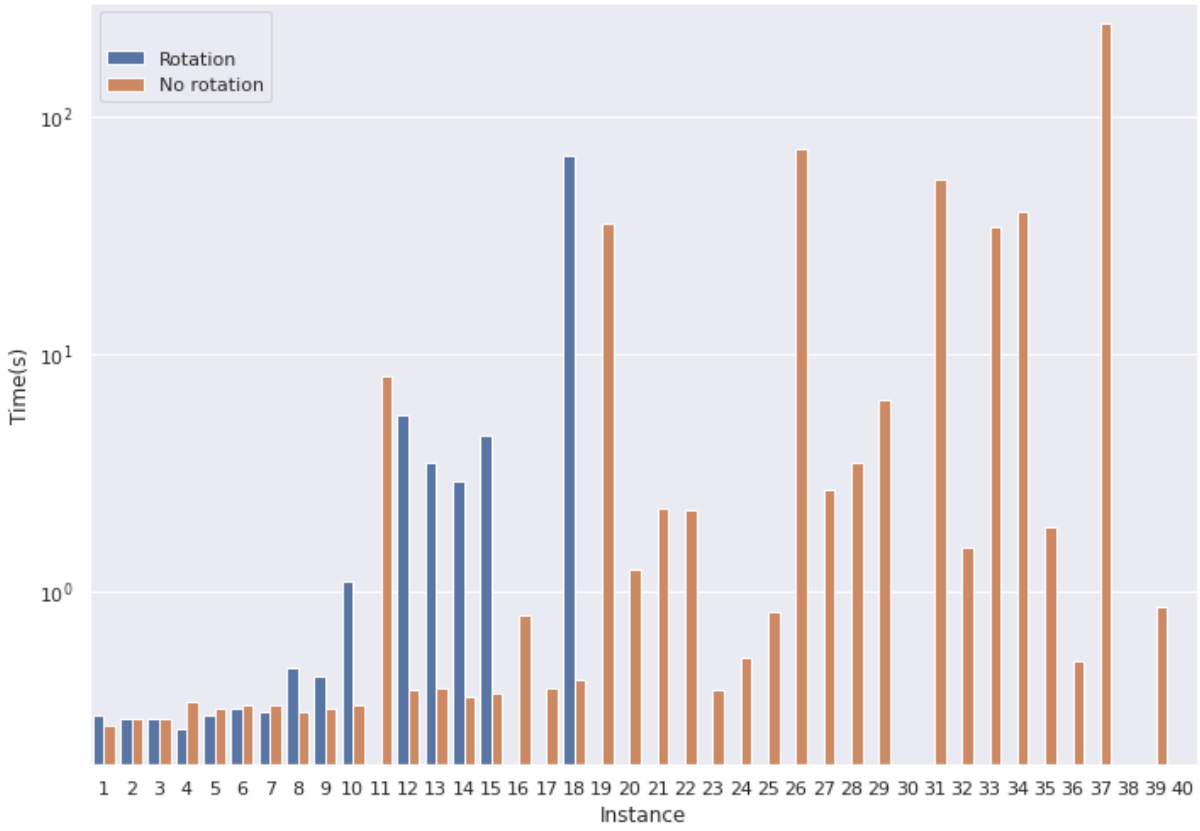
3.6.2 Chuffed vs Gecode

As previously stated in section 3.5, two main solver have been tested and analyzed, Chuffed reached an excellent result solving 37 out of the 40 instances provided, with a significantly lower average time for more complex instances compared to Gecode.



3.6.3 Rotations vs no rotations

In this subsection the two interpretations of the problem have been tested; the best solution found so far (i.e. Chuffed solver with sorted areas) has been tested both allowing and not allowing circuits rotation.



As can be easily observed, the performance of the solver degraded while increasing the complexity of the problem by letting it rotate the circuits.
So it can be deducted that allowing rotation of circuits results in an increasing in complexity way more influential than the higher freedom of the solver, worsening its performances.

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

As proposed in [4] even if we want to obtain the optimal height in our VLSI problem, SAT solvers can only determine the satisfiability of a given problem. Because of that, the problem is approached in a different way, namely by solving a sequence of two-dimensional orthogonal packing problems (2OPPs), which are decision problems just like the one we are interested in but with a fixed height of the strip. In our case both a fixed maximum width and a feasible maximum height needed to be specified.
There have been several studies on translation methods which encode a CSP into a SAT problem, namely: direct encoding, log encoding, support encoding, order encoding and log support encoding. Among them, order encoding aims to make a more natural explanation of the order relation of integers, and because of that it has been selected as the encoding for this problem.

4.1.1 Variables

In order to model the problem in a SAT encoding, the following boolean variables were defined:

- **px, py**: boolean variables set to true if rectangles are positioned before a certain width or height

- **lr, ud**: boolean variables that indicate if two rectangles are positioned one above the other (up down) or alongside (left right)
- **ph**: a useful set of boolean variables, ph_i set to true if all rectangles are placed under a specific height i

The pseudo-parameter **H** was also defined as the maximum possible height of the plate.

4.1.2 Bounds

For each rectangle r_i, and integers e and f such that $W - w_i \leq e < W$ and $H - h_i \leq f < H$, the two constraints that define the implicit domain of px and py are:

$$px_{i,e}$$

$$py_{i,f}$$

Those are true if the width and the height of the rectangle r_i are smaller than the upper bounds (i.e. W and H).

A suboptimal solution obtained using the BL-algorithm has been used as the upper bound (instead of a more generous upper bound) when deciding how many boolean variables px, py to instantiate. This was done in an effort to both generalize the algorithm for any kind of possible instance and to not instantiate too many variables due to a looser bound.

In order to represent the height of the plate as a fixed bound the ph variables were introduced. During the search for a satisfiable optimal solution, the lower bound for the height, computed as in CP, was first used as the maximum feasible height and satisfiability is checked.
After a first test if this was not attained the bound is increased by one by removing the previous constraint and adding another one corresponding to the next value to be checked.
In practice what is done is first we set ph_{lb}, where lb is the lower bound, if this results in UNSAT we remove it and add ph_{lb+1} and we continue until we get SAT. This removal/insertion is done using z3's push and pop methods for constraints.

This method is opposite to the classical SAT optimization method of only adding constraints, however it proved much more successful since in our experiments the lower bound seemed to be a feasible solution for most the 40 instances provided.
For the few ones which were not satisfied in the time limit of 300 seconds probably a feasible solution would have been found in the next few minutes.

4.2 Constraints

4.2.1 Under-height packing

The first constrain links py with ph, in particular ph_o is true if all rectangles are packed at the downward to the height o. Then, to solve our problem, for each rectangle r_i, and height o such that $lb \leq o < ub - 1$,we have the 2-literal clauses:

$$\neg ph_o \vee py_{i,o-h_i}$$

4.2.2 Order encoding

In order encoding, there are two encoding steps. Let x be an integer variable, and c be an integer value. In the first step, a constraint with comparison is translated into primitive comparisons which are in the form of $x \leq c$. In the next step, a primitive comparison is encoded into a Boolean variable px_c.
For each rectangle r_i, and integer e and f such that $0 \leq e < W - w_i$ and $0 \leq f < H - h_i$, we have the 2-literal axiom clauses due to order encoding:

$$\neg px_{i,e} \vee px_{i,e+1}$$

$$\neg py_{i,f} \vee py_{i,f+1}$$

Furthermore, for each o such that $lb \leq o < ub - 1$, we also have the 2-literal clauses related to the optimality checking:

$$\neg ph_o \vee ph_{o+1}$$

4.2.3 Non overlapping

Just like for the CP formulation, when dealing with SAT we need to define a no-overlap relationship between the circuits, in particular this has been done in the following way.
For each rectangles r_i, r_j (i<j), we have the following 4-literal clauses as the non-overlapping constraints:

$$lr_{i,j} \vee lr_{j,i} \vee ud_{i,j} \vee ud_{j,i}$$

For each pair of rectangles r_i, r_j (i<j), and integers e and f such that $0 \leq e < W - w_i$ and $0 \leq f < H - h_i$, we also have the following 3-literal clauses as the non-overlapping constraints:

$$\neg lr_{i,j} \vee px_{i,e} \vee \neg px_{j,e+w_i}$$
$$\neg lr_{j,i} \vee px_{j,e} \vee \neg px_{i,e+w_j}$$
$$\neg ud_{i,j} \vee py_{i,f} \vee \neg py_{j,f+h_i}$$
$$\neg ud_{j,i} \vee py_{j,f} \vee \neg py_{i,f+h_j}$$

For which helper python functions have been implemented in order to express it in a z3 model.

4.2.4 Symmetry breaking

Like in CP, symmetry breaking constraints are useful to improve model performance, in particular here we implemented a constraint in order to break the same size rectangle symmetry.
If we are given rectangles r_i, r_j which have the same dimension (wi, hi)=(wj, hj), we can fix the positional relation of rectangles. Thereby, we can add the constraints:

$$\neg lr_{i,j}$$

$$lr_{i,j} \vee \neg ud_{j,i}$$

4.2.5 Large rectangles

Another fundamental constraint in our problem is the one that ensure that no circuit exceed the maximum bounds imposed both on width and on height.
In particular if there exist any two rectangles r_i, r_j such that $w_i + w_j > W$ we can impose the following constraint:

$$\neg lr_{i,j} \wedge \neg lr_{j,i}$$

in such a way, the two rectangles cannot be placed one next to the other in the horizontal direction.
The same reduction technique has also been implemented in the vertical direction.

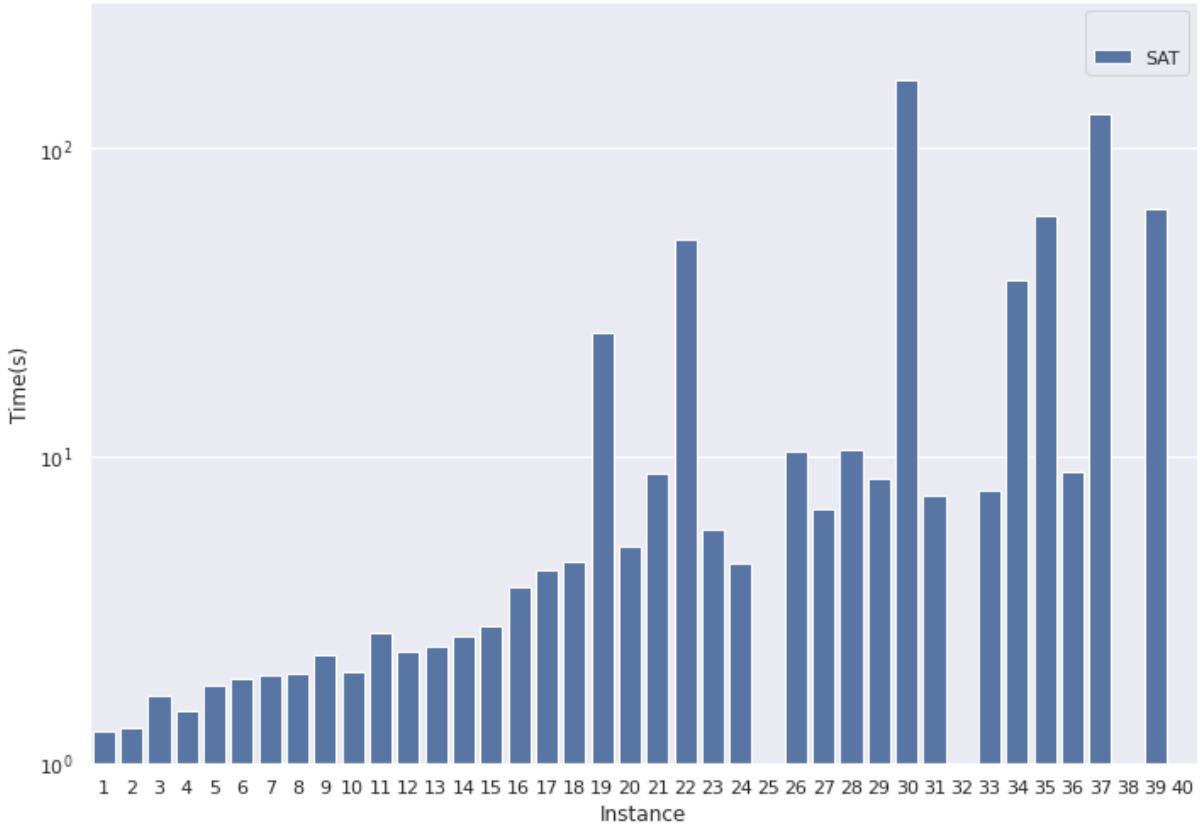
4.3 Rotation

If rotations are allowed the possible combinations of rectangles increase exponentially. In order to solve the problem, a possibility could be to consider an expanded list of rectangles, where all the rotated copies are included. The model requires a way to consider only one of each couple of rotated rectangles, so a list of used literal is implemented, with some modifications to the previous model, like a new array of booleans (i.e. **rotation**) which identifies each of the rotated circuits.
The major changes consist in applying the XOR operation between pairs of used variables, so that only one of each couple of rectangles will be placed. The rest of the code is kept virtually identical, with the exception of the non overlapping constraints: they will be valid only if the elements of the couple are both used.

A first draft of this possible implementation is present in the SAT folder.
The actual implementation of the suggestions above would require some other tweaks in order to work in any situation, since at the moment the model includes assumption on the variable bounds that are incompatible with the new approach.

4.4 Results

The results of SAT solver are here shown:



The solved instances can be found in the folder SAT /out.
The model managed to solve all but 4 of the instances (25,32,38,40), obtaining similar results as the CP model.

5 SMT

SMT(Satisfiability Modulo Theory) is a generalization of SAT to express first order logic formulas, it thus provides a much greater expressiveness in representing entities, namely it allows the representation of real numbers, arrays, bitvectors, etc. In this project we decided to use Z3 as an SMT solver, in particular the Z3 python API was used.

5.1 Encoding

A SMT formulation of the problem offers ample space for decisions on how to represent the problem, offering the best of the worlds of propositional logic and linear equation based constraints.
For this reason we opted for a simple encoding of the problem where the decision variables define the coordinates of the bottom-left corner of each circuit and a makespan variable which defines the maximum height of the solution.

The objective function is simply defined as the makespan variable, whose value should be minimal. In plain SMT it's not possible to perform optimization(only satisfiability checking is possible), however the Z3 solver allows us to perform optimization directly.
A valid solution of the problem has to respect the two following constraint:

- Each circuit must be completely contained in the board (**Containment**)
- Circuits cannot overlap (**No overlap**)

Once we have defined these basics constraints we need further refinement to let the solver find the optimal solution in reasonable time. At this point it's possible to implement such improvements in two ways, either following the CP approach, that is replicating the global constraints used in the CP formulation, or following the SAT approach, by introducing boolean decision variables and constraints on them that help reduce the search space.

5.1.1 Variables

The basic decision variables in our encoding are the following integer variables:

$$corner_i^x, corner_i^y \quad i \in [1, n\text{-}blocks]$$

$$makespan$$

Where $corner_i^x, corner_i^y$ define respectively the x and y coordinates of where the i-th circuit should be placed. The other variable *makespan* defines the maximum height of the entire board.

5.1.2 Bounds

Bounding the domain of the $corner^x, corner^y$ variables allows us to fulfill the containment constraint as well as shrinking the search space:

$$0 \leq corner_i^x \leq W - w_i \qquad \forall i \in [1, n\text{-}blocks]$$

$$0 \leq corner_i^y \leq makespan - h_i \qquad \forall i \in [1, n\text{-}blocks]$$

Constraining *makespan* however required a more sophisticated approach. As presented in the CP section we know that its lower bound can be defined as the sum of the circuit areas divided by the maximum width.
As for the upper bound we can either take the most naive upper bound (the sum of the heights of the circuits, simulating a solution which stacks all blocks on a single column) or we can use the solution obtained by using the bottom-left heuristic.
Unlike in CP, where shrinking the makespan domain did not yield any improvement, here a sizable speedup was observed by setting the upper bound as the one given by the BL-solution.

These bounds are expressed as constraints in the following way:

$$lower\_bound \leq makespan \leq upper\_bound$$

5.2 Constraints

The other constraint that it's necessary to represent is the no overlap one.
It's possible to express such a constraint by noticing that any two circuits i, j do not violate it if they are positioned in such a way that i is to the left of j or i is to the right of j or i is on top of j or is on the bottom of j.
A logical/mathematical formulation follows naturally from such a definition in natural language:

$$\begin{aligned} corner_i^x + w_i \leq corner_j^x \quad \vee \\ corner_j^x + w_j \leq corner_i^x \quad \vee \\ corner_i^y + h_i \leq corner_j^y \quad \vee \\ corner_j^y + h_j \leq corner_i^y \\ \forall i, j \in [1, n\text{-}blocks] \end{aligned}$$

Two symmetry breaking techniques were analyzed.
The first one is described in [4], it boils down to the idea that the widest block should be positioned in the left part of the plate.

$$corner_k^x \leq \frac{W - w_k}{2} \quad w_k \geq w_i \quad \forall i \in [1, n\text{-}blocks]$$

Furthermore we can assert that any other block whose width is bigger than $\frac{W}{2}$ should not be placed to the left of the biggest block.
These constraints can be replicated on the vertical direction as well.

The second one is described in [6], here we add some restrictions to the no overlap constraint in a way to separate the regions where the third or fourth elements are active from those where the first and second one are. In particular the third and fourth elements of the disjunction are replaced respectively with:

$$corner_i^y + h_i \leq corner_j^y \wedge corner_i^x + w_i \geq corner_j^x + 1 \wedge corner_j^x + width_j \geq corner_i^x + 1$$

And:

$$corner_j^y + h_j \leq corner_i^y \wedge corner_i^x + w_i \geq corner_j^x + 1 \wedge corner_j^x + width_j \geq corner_i^x + 1$$

However adding these last constraints worsened the overall performance.

As introduced before we can use additional constraints resembling either the CP or the SAT formulation to improve the solution.
One might think that putting both of them together should be the best solution, however it was actually found to be the worst one.

5.2.1 CP-like constraints

We can try to replicate the cumulative constraint implemented in the CP formulation. To do so we define the following set of binary variables:

$$v_{i,j} \quad i \in [1, upper\_bound] \quad j \in [1, n\text{-}blocks]$$

$$h_{i,j} \quad i \in [1, W] \quad j \in [1, n\text{-}blocks]$$

The variable $v_{i,j}$ is true if the j-th circuit has at least a block on the i-th row. Likewise the variable $h_{i,j}$ is true if the j-th circuit has at least a block on the i-th column.
We can force this behaviour using the following constraints:

$$v_{i,j} = (corner_j^y \leq i \leq corner_j^y + h_j) \quad \forall i \in [1, upper\_bound] \quad \forall j \in [1, n\text{-}blocks]$$

$$h_{i,j} = (corner_j^x \leq i \leq corner_j^x + w_j) \quad \forall i \in [1, W] \quad \forall j \in [1, n\text{-}blocks]$$

The horizontal cumulative constraint is enforced by saying that for each row the sum of the widths of the active circuits on that row (identified by $v_{i,j}$) should not exceed W:

$$\sum_j v_{i,j} \cdot w_j \leq W \qquad \forall i \in [1, upper\_bound]$$

The vertical cumulative constraint follows a similar logic for columns:

$$\sum_j h_{i,j} \cdot h_j \leq upper\_bound \qquad \forall i \in [1, W]$$

5.2.2 SAT-like constraints

Like in the SAT formulation we can define the following sets of binary variables:

$$lr_{i,j} \quad \forall i, j \in [1, n\text{-}blocks], i \neq j$$

$$ud_{i,j} \quad \forall i, j \in [1, n\text{-}blocks], i \neq j$$

We say that $lr_{i,j}$ is true if the i-th circuit is positioned to the left of the j-th one. Likewise $ud_{i,j}$ is true if the i-th circuit is positioned in a lower position than the j-th one.
This can be enforced in the same way the no-overlap constraint was enforced:

$$\begin{gathered}(corner_i^x + w_i \leq corner_j^x) = lr_{i,j} \\ (corner_j^x + w_j \leq corner_i^x) = lr_{j,i} \\ (corner_i^y + h_i \leq corner_j^y) = ud_{i,j} \\ (corner_j^y + h_j \leq corner_i^y) = ud_{j,i} \\ \forall i, j \in [1, n\text{-}blocks]\end{gathered}$$

Once we have these variables we can use them to enforce the large rectangles constraint (two rectangles can't be next to each other if the sum of their width is greater than W), like in SAT, in both the vertical and horizontal direction.
In particular with SMT we can improve the vertical direction reduction (with respect to SAT where we can only consider the upper bound), since we have the ability to manipulate the makespan:

$$\begin{gathered}(h_i + h_j \geq makespan) \rightarrow \neg ud_{i,j} \\ (h_i + h_j \geq makespan) \rightarrow \neg ud_{j,i} \\ \forall i, j \in [1, n\text{-}blocks], i \neq j\end{gathered}$$

5.3 Rotation

Injecting the ability to rotate the circuits is quite straightforward in SMT, we just need to add a set of binary variables:

$$r_i \quad i \in [1, n\text{-}blocks]$$

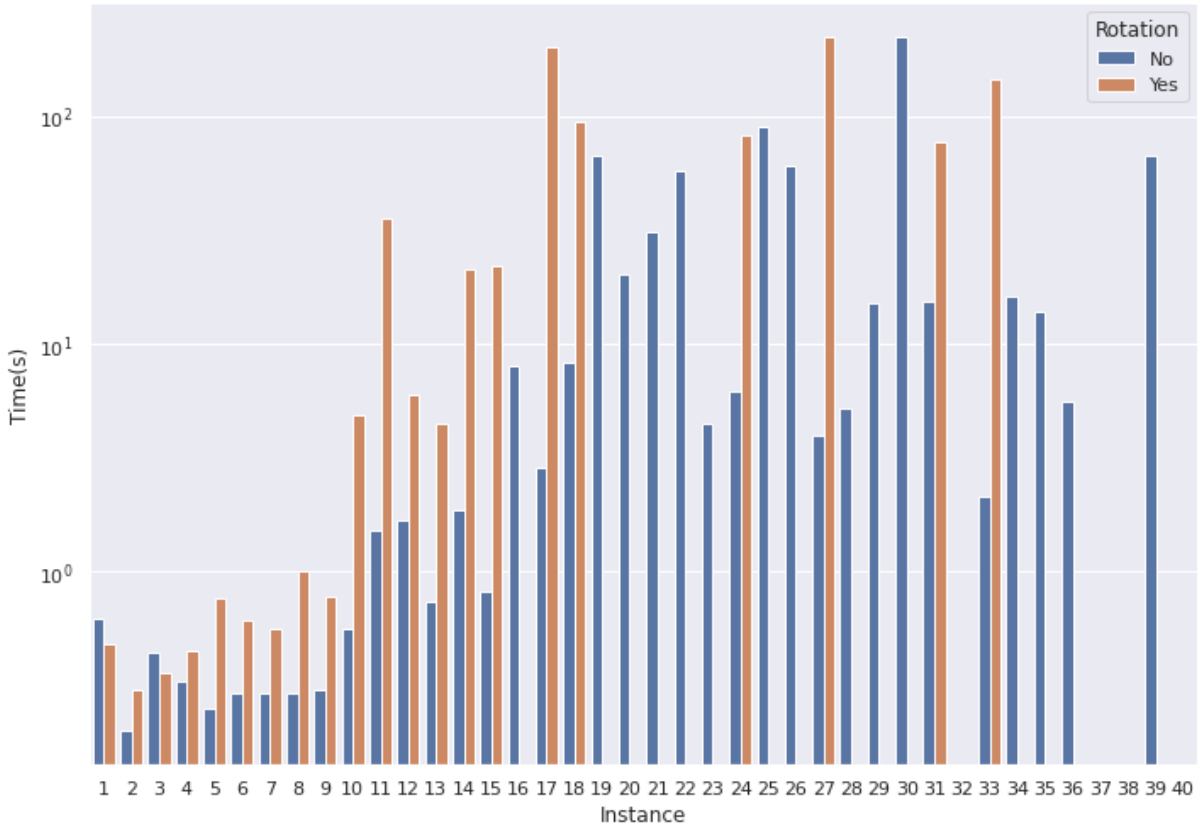
r_i is true if the i-th circuit should be rotated. By rotating circuits we have to transform the widths and heights into integer variables $\hat{w}_i, \hat{h}_i$, whose value is fixed by the following if-then-else constraints:

$$ite(r_i, \hat{w}_i = h_i, \hat{w}_i = w_i)$$

$$ite(r_i, \hat{h}_i = w_i, \hat{h}_i = h_i)$$

5.4 Results

Here we report the results on how many instances were solved and how much time it took, both with and without allowing rotations.

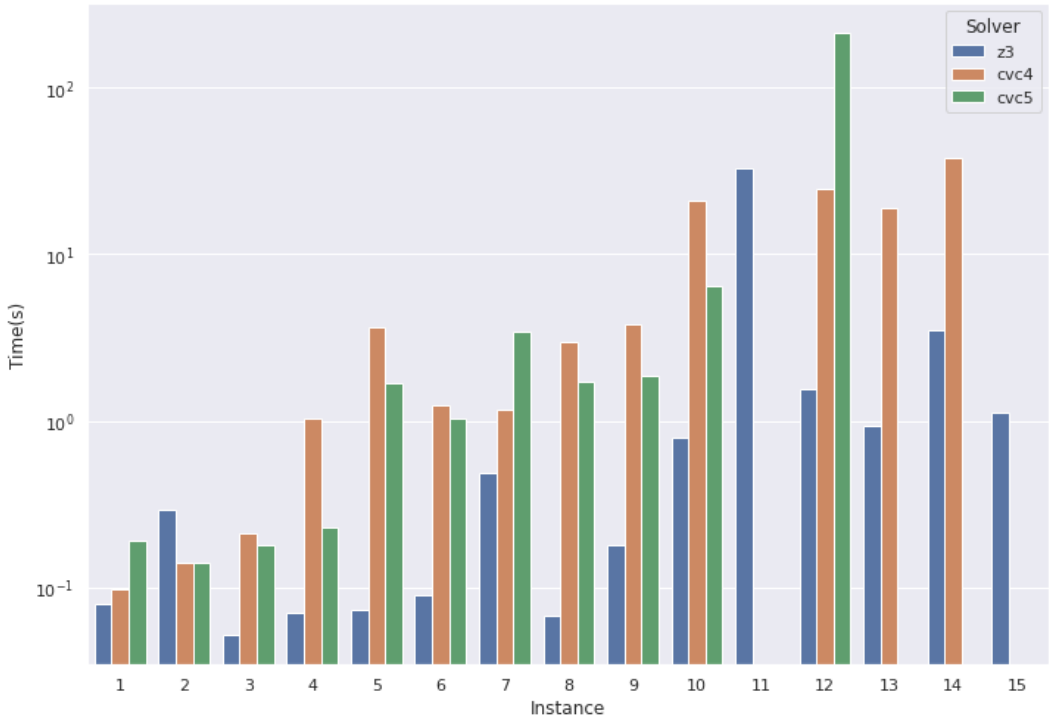


The solved instances can be found in the folder SMT/out.
It turned out that 36/40 instances were solved in 300 seconds without rotations, while 21/40 were solved with rotation.

5.5 Using SMTLIB

SMTLIB is a project aimed at standardizing SMT code in a way to make it solver independent. Usually SMTLIB code is written directly, however the Z3 python API allows us to actually export a problem definition which was written using Z3py into an equivalent SMT-LIB code.
This allows us to solve a problem using other solvers, like CVC4 or CVC5.
Furthermore SMT-LIB does not allow us to perform optimization as easily as with the Z3 python API, so we have to perform an incremental "search" over the objective function value, like in SAT. This has to be done by continuously adding constraints to the SMT-LIB code until we reach an UNSAT, such operations were performed using the offline_omt.sh file provided on Virtuale, which only performs a linear search over the objective value.

Using this method we can compare different solvers like Z3, CVC4 and CVC5.



It turned out that using this naive linear search makes it so only easy instances can be solved. The Z3 solver performed the best and surprisingly CVC4 performed better than its newer version, CVC5.

6 MIP

In MIP (Mixed Integer Programming) a problem is defined as a set of real valued or integer valued variables, constraints on these variables are expressed as linear (or sometimes quadratic) equalities or inequalities. The MIP solver that was used was Gurobi, using its python interface.

6.1 Encoding

The formulation of the problem used for CP and SMT, that is defining the decision variables as the x and y positions of the bottom left corner of the circuits on the plate, is sound in MIP as well.
Most of the constraints on these variables can be expressed through linear inequalities, however we also need to inject into the problem definition some logical notions, for example for the no-overlap constraint requirement.
It's easy to express boolean variables in MIP, we just have to say that a variable has to be an integer variable and it has to take value between 0 and 1. However MIP solvers cannot handle logical operations, for them we have to define some appropriate encodings.

The first of these is the encoding of the indicator function, that is a function that tells us whether a constraint is active or not.
Let's say we have an inequality/constraint $f(x) \leq b$, we want a binary variable y to be 1 when the constraint is active, to express this we can use the **big-M** method.
The big-M method makes use of a sufficiently large constant M to achieve our objective, we introduce a constraint like:

$$f(x) \leq b + M(1 - y)$$

This makes sense since if we have $y = 1$, the equation will be the normal constraint (which will be active), while if $y = 0$ we have that the constraint is inactive since the left hand side will always be lower than the right side. For this to work we have to be sure that M is large enough to nullify the constraint.

As for encoding logical operations between two binary variables using only inequalities the following is possible, as described in [1]:

- **Or**, given binary variables A,B we can define $C = A \vee B$ using the constraints:

$$C \geq A$$
$$C \geq B$$
$$C \leq A + B$$

- **And**, given binary variables A,B we can define $C = A \wedge B$ using the constraints:

$$C \leq A$$
$$C \leq B$$
$$C + 1 \geq A + B$$

- **Not**, given a binary variable A, $\bar{A}$ is simply obtained by the expression:

$$1 - A$$

Initially these encodings were used, however we later discovered that the Gurobi solver provided helper functions which abstracted them. No clear difference in performance between the two implementations was found, so for readability purposes we opted for the Gurobi implementation.

6.1.1 Variables and bounds

The decision variables and bounds on them are the same as the CP and SMT implementation, namely:

$$corner_i^x, corner_i^y \quad \forall i \in [1, n\text{-}blocks]$$
$$makespan$$

Where all of them are forced to be integer variables and to respect the following constraints:

$$0 \leq corner_i^x \leq W - w_i$$
$$0 \leq corner_i^y \leq makespan - h_i$$
$$lower\_bound \leq makespan \leq upper\_bound$$

Where $lower\_bound$ is defined as the sum of the areas of the circuits divided by W and $upper\_bound$ is obtained through the bottom-left heuristic.
Unlike with other technologies, the solution obtained using the bottom-left heuristic can also be used as some kind of warm start. On this regard Gurobi offers two options:

- Setting an initial **base** solution
- Adding **hints** to variables

The first option did not yield any benefit, however adding hints which corresponded to the value the variables took in the heuristic solution proved beneficial overall.

6.2 Constraints

The first set of constraint to be implemented is the one dealing with the no-overlap requirement. We first define the binary indicator variables $lr_{i,j}, ud_{i,j}$ for the constraints which encode the position between two circuits i, j:

$$corner_i^x + w_i \leq corner_j^x + M(1 - lr_{i,j})$$
$$corner_j^x + w_j \leq corner_i^x + M(1 - lr_{j,i})$$
$$corner_i^y + h_i \leq corner_j^y + M(1 - ud_{i,j})$$
$$corner_j^y + h_j \leq corner_i^y + M(1 - ud_{j,i})$$

And then we put them together in a 4-way OR by forcing their sum to be greater than 1:

$$lr_{i,j} + lr_{j,i} + ud_{i,j} + ud_{j,i} \geq 1$$

After this we implemented both horizontal and vertical cumulative constraints. Let's consider the horizontal cumulative constraint as an example. Like in SMT we defined the binary variables :

$$v_{i,j} \quad i \in [1, upper\_bound] \quad j \in [1, n\text{-}blocks]$$

These should take the result of an AND between two indicator dummy variables, let's call them $v^1_{i,j}, v^2_{i,j}$:

$$\begin{gathered} corner^x_j \leq i + M(1 - v^1_{i,j}) \\ i \leq corner^x_j + w_j + M(1 - v^2_{i,j}) \\ v_{i,j} \leq v^1_{i,j} \\ v_{i,j} \leq v^2_{i,j} \\ v_{i,j} + 1 \geq v^1_{i,j} + v^2_{i,j} \\ \forall i \in [1, upper\_bound] \quad j \in [1, n\text{-}blocks] \end{gathered}$$

The actual cumulative constraint is then implemented as:

$$\sum_j v_{i,j} \cdot w_j \leq W \qquad \forall i \in [i, upper\_bound]$$

This and the successive sums were implemented using the quicksum function provided in GurobiPy. A similar set of constraints was written for the vertical cumulative constraint.

Finally we implemented a generalization of the big-rectangles constraint over the vertical direction only, the same can of course be done on the horizontal direction, but it didn't prove beneficial.
Considering the vertical direction the idea is to re-use the binary variables $h_{i,j}$ which tells us whether a circuit j is active on column i.
We then take the superset of circuits up to a certain cardinality K (defined arbitrarily as a function of *n-blocks*).
Formally we say that if $C = \{c_1, \ldots, c_{n\text{-}blocks}\}$ is the set of circuits we obtain the restricted superset:

$$C_K = \{V| \quad V \in \mathcal{P}(C), |V| \leq K\}$$

We then compute the sum of the heights of the circuits in each of the subsets: $\sum_{c_i \in V} h_i \quad \forall V \in C_K$
In case this value is greater than *upper_bound* we add the following constraints for V:

- Define the binary variables: $a_i \quad i \in [1, W]$
- Set each of them to: $a_i = \bigwedge_{c \in V} h_{i,c}$
- Add the constraint: $\sum_i a_i = 0$

What each a_i symbolizes is whether all of the circuits in the subset V are active on column i at the same time. And what we want is that this is not true for any of the spaces (since we've said that the sum of their heights exceeds the bound).

Unlike in SMT it turned out that adding these last two types of constraints, cumulative and big rectangle, which need to perform a summation with a relatively high amount of terms, improves the performance of the solver even with this more complex general case, we could attribute this behaviour to the difference in nature between the SMT and MIP approach.
The same symmetry breaking constraints as SMT were tested however they proved to slow the solver down, probably due to their logic-based nature.

6.3 Solver settings

Despite the Gurobi solver being closed source and no explicit explanation of its inner workings being provided, it's possible to change some settings on how it solves an instance, most of these changes affect the variable selection and feasible space search/cutting processes.
Some settings which seemed promising and that we experiment on were:

- **MIPFocus**: allows the user to set the high level solution strategy for the solver. For example it's possible to focus on finding feasible solutions (MIPFocus=1), useful when the problem is hard to satisfy; on proving optimality (MIPFocus=2); or on lowering the bound faster (MIPFocus=3). In theory our problem has a high amount of feasible solutions and proving optimality/lowering the bound should be more important than finding feasible solution, however no improvement was found changing by changing the strategy.
- **Disconnected**: the higher it is the more aggressive the solver is in looking for independent sub-models of the problem. At first glance we thought that lowering this would be good, since our problem can't really be cut into independent parts.
- **BranchDir**: forces the solver to always choose a direction for branching in the branch-and-bound procedure. This kind of looks like a search strategy hint in CP, with much less information however.
- **Symmetry**: the higher this parameter is set to the more aggressive the solver is in looking for symmetries, both during the MIP and LP parts of the solver. Since no explicit symmetry breaking was enforced, a methodology which detects symmetries implicitly should work well.

In the end none of these changes gave an overall performance benefit.

6.4 Rotation

Like in the other technologies adding the possibility to rotate the circuits amounts to adding a set of binary variables:

$$r_i \quad i \in [1, n\text{-}blocks]$$

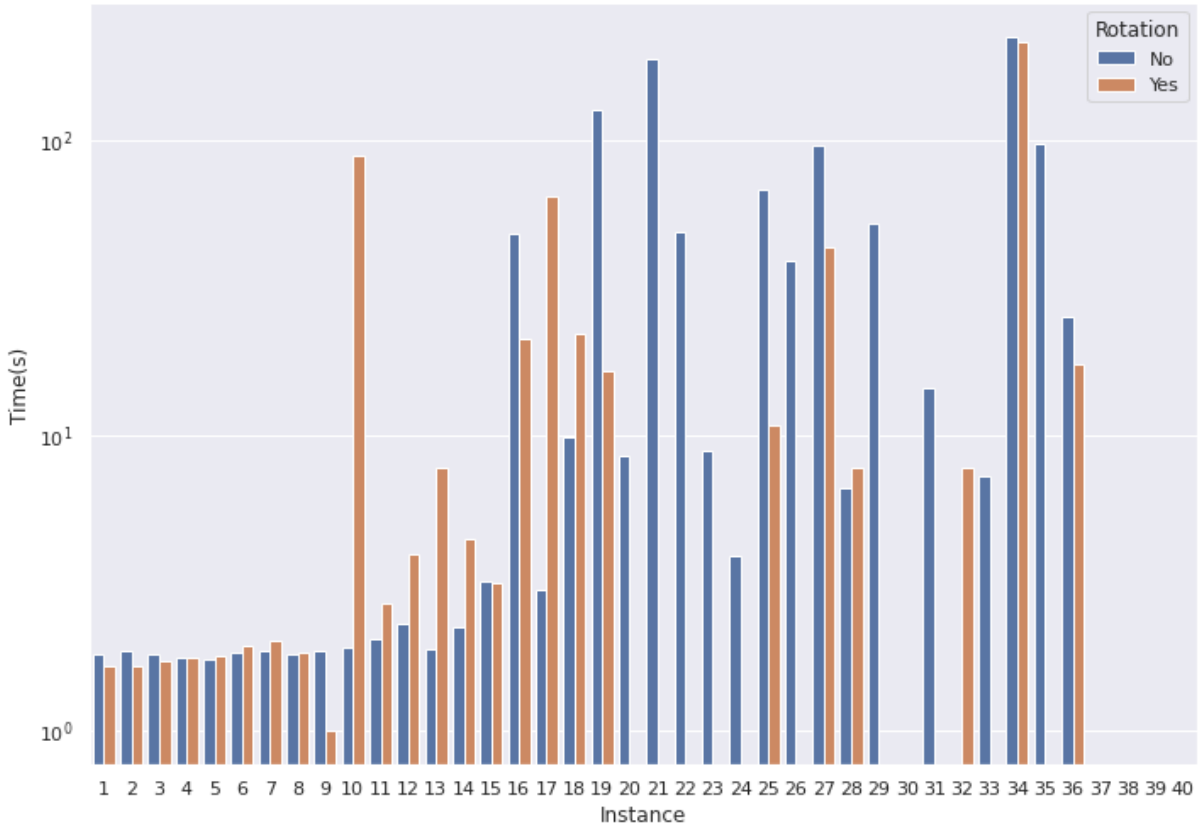
And subsequently turning the parameters h_i, w_i describing the size of the circuits into variables $\hat{h}_i, \hat{w}_i$. These variables were set in the following way:

$$\hat{h}_i = (r_i \cdot w_i + (1 - r_i) \cdot h_i)$$
$$\hat{w}_i = (r_i \cdot h_i + (1 - r_i) \cdot w_i)$$
$$\forall i \in [1, n\text{-}blocks]$$

The additional constraints are then the same as the version without rotation, except for the big-rectangle one, which could not be implemented efficiently with a variable width/height.

6.5 Results

Here we report the results on how many instances were solved and how much time it took, both with and without allowing rotations.



The solved instances can be found in the folder MIP /out.
It turned out that 34/40 instances were solved in 300 seconds without rotations, while 25/40 were solved with rotations.

7 Conclusion

In the end we can infer some insights about the problem in hand and the proposed solutions.
Our approach appeared to produce excellent results solving 38 out of the 40 provided instances considering all the technologies together, and a minimum of 34 with the best result for each technology; being the CP model the best performing one with 37 out of 40 solved instances.
Some instances have proven to be intrinsically more difficult to approach than other independently from the technology in hand, in particular instances 38 and 40 couldn't be solved within the 300 seconds time limit with any of the technologies tried.
Observing the results we can also conclude that considering the more general version of the VLSI problem, namely allowing circuits rotation, largely decreased model performances independently from the technology.
This is probably due to the higher amount of variables that need to be instantiated in the latter case, drastically increasing the effort required to the model. Furthermore if we allow rotations we can't make use of convenient heuristics that work due to the assumptions of a fixed height and width for each block, for example the big rectangle constraints would require much more computationally expensive constraints to be enforced and thus bring the overall performance down.

Finally, a number of approaches have been tested but only some of the them proved valid among all the different technologies. Namely:

- sorting circuits by area gave us a great performance increase in the CP model, while no substantial improvement has been registered in the other technologies.
- placing the widest block first in the left part of the plate, as exploited in 5.2 for SMT didn't seem to provide any substantial improvement to the other solutions

- cumulative constraints and similar constraints which work on linear combinations of variables worked well in CP and MIP but they proved too computationally expensive for SMT
- finding an upper bound for the height via the BL algorithm proved to be much more effective than a basic one (like the sum of the heights of the circuits) in all technologies but the CP one

References

[1] Gerald G Brown and Robert F Dell. "Formulating integer linear programs: A rogues' gallery". In: *INFORMS Transactions on Education* 7.2 (2007), pp. 153–159.

[2] Eric Huang and Richard E Korf. "New improvements in optimal rectangle packing". In: *Twenty-First International Joint Conference on Artificial Intelligence*. Citeseer. 2009.

[3] M. Luby, A. Sinclair, and D. Zuckerman. "Optimal speedup of Las Vegas algorithms". In: *[1993] The 2nd Israel Symposium on Theory and Computing Systems*. 1993, pp. 128–133. DOI: 10.1109/ISTCS.1993.253477.

[4] Takehide Soh et al. "A SAT-based Method for Solving the Two-dimensional Strip Packing Problem". In: *Fundam. Inform.* 102 (Jan. 2010), pp. 467–487. DOI: 10.3233/FI-2010-314.

[5] *Survey on two-dimensional packing*. https://www.csc.liv.ac.uk/~epa/surveyhtml.html.

[6] Francisco Trespalacios and Ignacio E Grossmann. "Symmetry breaking generalized disjunctive formulation for the strip packing problem." In: ().