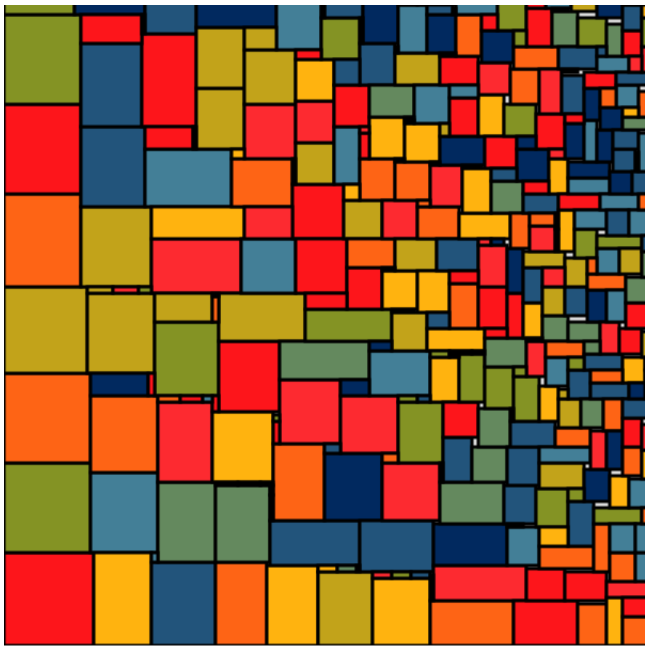
Alma Mater Studiorum
Università di Bologna

VLSI Design

SMT Solution



Yuri Noviello yuri.noviello@studio.unibo.it
Francesco Olivo francesco.olivo2@studio.unibo.it

Contents

1 Introduction

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. A typical example is the smartphone. The modern trend of shrinking transistor sizes, allowing engineers to fit more and more transistors into the same area of silicon, has pushed the integration of more and more functions of cellphone circuitry into a single silicon die (i.e. plate). This enabled the modern cellphone to mature into a powerful tool that shrank from the size of a large brick-sized unit to a device small enough to comfortably carry in a pocket or purse, with a video camera, touchscreen, and other advanced features. As the combinatorial decision and optimization expert, the student is assigned to design the VLSI of the circuits defining their electrical device: 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). Consider two variants of the problem. In the first, each circuit must be placed in a fixed orientation with respect to the others. This means that, an $n \times m$ circuit cannot be positioned as an $m \times n$ circuit in the silicon plate. In the second case, the rotation is allowed, which means that an $n \times m$ circuit can be positioned either as it is or as $m \times n$.

2 Format of the instances

This section describes the format in which the VLSI instances are written, as well as the expected format of the corresponding solutions.

2.1 Input format

An instance of VLSI 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 and y_i, representing the horizontal and vertical dimensions of the i-th circuit. For example, a file with the following lines:

```
9
5
3 3
2 4
2 8
3 9
4 12
```

describes an instance in which the silicon plate has the width 9, and we need to place 5 circuits, with the dimensions 3×3, 2×4, 2×8, 3×9, and 4×12.

2.2 Solution format

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 left-bottom corner i. This could be done by for instance adding l next to w, and adding $\hat{x}_i$ and $\hat{y}_i$ next to x_i and y_i in the instance file. To exemplify, the solution of the previous example looks like:

```
9  12
5
3  3  4  0
2  4  7  0
2  8  7  4
3  9  4  3
4  12  0  0
```

Which says for instance that the left-bottom corner of the 3×3 circuit is at (4, 0). For the model with rotation the solution contains the (eventually) inverted dimensions of the rotated circuits.

3 Model

3.1 Variables

The starting point requires finding all the necessary variables to model the problem. In particular, since the output format requires the left bottom corner coordinates of a given rectangle, we need two arrays to store the x and the y coordinates, respectively p_x and p_y. These arrays must be sorted coherently with the original arrays x and y that state the dimensions of the rectangles, respectively the width and the height. All said arrays have length n, i.e. the number of circuits to be placed on the board.
We want to model all of our variables to have the smallest possible domain, in order to reduce the search space. Thus, the coordinates p_x and p_y will be bounded within $[0, w - min(x)]$ and $[0, l_{max} - min(y)]$ respectively.

3.2 Bounds for the height

Since the goal is to minimize the height, it is crucial to find a good pair of bounds for the height variable l. For the lower bound of l we considered that if all the plates can be inserted in the plate without leaving empty space, the height is given by the division between the summation of the areas and the fixed width. At the same time we need to consider the case in which there is a plate with a very large height, in this case the lower bound is given by the larger height.

$$l_{min} = max(max(y), \lfloor \frac{\sum_{i=1}^{n} y_i * x_i}{w} \rfloor)$$

For the upper bound we approximated an estimator that (largely) estimate the value of l. We also noticed that the upper bound does not influence particularly the performances since the solver searches firstly for values near to the lower bound, since we need to minimize it.

$$l_{max} = \lfloor l_{min} + \frac{\sum_{i=1}^{n} y_i}{2} \rfloor$$

3.3 Symmetries

There are many possible symmetric solutions to this particular problem. We shall avoid them in order to reduce the search space, thus improving the overall performances.

- Symmetry across the x-axis
- Symmetry across the y-axis
- Symmetry around the center of the board (180° rotation)
- Two rectangles with the same dimensions in the same position

After many tests, we noticed that in many cases the optimal solution leads to a square board, so

$$l_{optimal} = w$$

Which means that also 90° and 270° rotations lead to symmetric solutions. This may happen only when we allow the rotation of the circuits.

4 Constraints

There are many constraints to impose in order to correctly solve the problem: we want to respect the width and the height constraint, while trying to minimize the height. On the other hand we do not want circuits to overlap, and we want to avoid symmetries in order to reduce the search space.

4.1 Limits constraint

All the rectangles must be places within the board limits. This can be expressed as:

$$x_i + p_{x_i} \leq w, i \in [1, n]$$

$$y_i + p_{y_i} \leq l, i \in [1, n]$$

Moreover, the coordinates p_x and p_x are bounded between 0 and the board width (resp: height) minus the width (resp: height) of the narrowest (resp: shortest) rectangle, thus:

$$0 \leq p_x \leq w - min(x)$$

$$0 \leq p_y \leq l - min(y)$$

4.2 No overlapping

We want to avoid rectangles to overlap with one another: this means that if we divide our board in $w * l$ squares, each one with an area of 1, only one circuit can be placed in each square.

$$\begin{gathered}\forall i, j \in [1, n] \text{ s.t. } i < j \\ p_{x_i} + x_i \leq p_{x_j} \vee \\ p_{x_j} + x_j \leq p_{x_i} \vee \\ p_{y_i} + y_i \leq p_{y_j} \vee \\ p_{y_j} + y_j \leq p_{y_i}\end{gathered}$$

4.3 Rectangle packing as a schedule problem

The rectangle packing problem can be seen as a bi-dimensional scheduling problem. In particular, for the x-axis, the board's height represents the resource bound, the x-coordinate p_x represents the starting time, the width of a circuit x represents the duration, while the height of a circuit represents the required resource.

$$cumulative(p_x, x, y, l) := \begin{array}{l} \forall j \in [1, n] \\ \displaystyle\sum_i^n \begin{cases} y_i, & p_{x_i} \leq y_j \wedge y_j < p_{x_i} + x_i \\ 0, & \text{otherwise} \end{cases} \end{array} \leq h$$

These considerations can be applied in a symmetrical way to the y-axis.

$$cumulative(p_y, y, x, w) := \begin{array}{l} \forall j \in [1, n] \\ \displaystyle\sum_i^n \begin{cases} x_i, & p_{y_i} \leq x_j \wedge x_j < p_{y_i} + y_i \\ 0, & \text{otherwise} \end{cases} \end{array} \leq w$$

4.4 Symmetry breaking

Since we want to remove the symmetries which were previously discussed, we need some more constraints. In particular, we decided to manually place the rectangle with the biggest area in (0, 0). This leads to breaking the symmetries

in most cases. We chose the area in area to keep the model consistent with the one described in CP.
Given the complexity of writing SMT constraints, we did not implement any lexicographic constraint.

4.5 No empty spaces at the bottom of the plate

With the previous constraints we reached good performances but we tried to achieve better results. We observed that in some cases the solver tried to produce solutions where not all bottom positions were occupied, so we introduced the following constraint:

$$full\_bottom := \sum_{i}^{n} \begin{cases} x_i, & \text{if } y_i = 0 \\ 0, & \text{otherwise} \end{cases} = w$$

This constraint says that the summation of the widths of the rectangles that are at the bottom of the plate is exactly equal to w.

5 Rotation

We are required to create another model, where it is possible to rotate the circuits on the board with respect to the given dimensions.
To represent rotation, we simply need a boolean array r, where r_i is $true$ if the i-th rectangle in the input is rotated, $false$ otherwise.
Then, we need a couple of arrays, x_r and y_r, to represent the effective dimensions of a rectangle. Note that x_r and y_r do not always contain the swapped dimensions of the input rectangles, they may according to the analyzed node. Therefore:

$$x_{r_i} = \begin{cases} y_i, & \text{if } r_i \text{ is true} \\ x_i, & \text{otherwise} \end{cases}$$

$$y_{r_i} = \begin{cases} x_i, & \text{if } r_i \text{ is true} \\ y_i, & \text{otherwise} \end{cases}$$

We also noticed that the model with rotation performed better without cumulative constraints, so we removed them.

5.1 Symmetry breaking

To reduce the search space, we set the rotation of squared circuits as false, we also set the rotation false if a rectangle has height greater than the width of the plate:

$$x_i = y_i \lor y_i > w \quad \Rightarrow \quad r_i = \text{false} \qquad \forall i \in i, n$$

6 Implementation choices

We decided to implement our solution in `pySMT`[1], a Python library that provides an intermediate step between the `SMT-LIB` and solvers API. The reason that led us to make this choice is that using `pySMT`, our solution is completely independent of the specific solver, indeed we have also been able to test more than one solver, such that `Z3` and `CVC4`.
Since `pySMT` does not support natively the minimization of a variable (h in our case) we decided to implement the following algorithm:

Algorithm 1 Minimization via pySMT

```
function MINIMIZE($l, solver, constraints, l_{min}$)
    $k = l_{min}$
    $solver.set(constraints, Equals(l, l_{min}))$
    while !$solver.is\_sat()$ do
        $k = k + 1$
        $solver.set(constraints, Equals(l, l_{min}))$
    end while
    $l = solver.get("l")$
end function
```

In this way if we find a solution, we are sure that it is the optimal one since we started from the lower bound of l.
As we mentioned in section 5, we did not implement cumulative constraints in the rotated model since we empirically noticed better performances.

6.1 Solvers

`pySMT` supports 7 different solvers, but only 4 of them can manage integers variables (`Z3`[2], `CVC4`[3], `MSAT`[4], `YICES`[5]). We focused mainly on `Z3` and `CVC4`, since they both support the "timeout" option via `pySMT`.

6.1.1 CVC4

`CVC4` is one of the a prover for SMT problems that can be implemented in `pySMT`. With the final model and this solver we obtained decent results only without rotation. It solved only 31 instances with the optimal solution without rotation, following there is the table that contains the time of the solved instances for both the model. We reported only the first 33 instances since from 34th on we do not found any optimal solution.

Name	**Time**	**Time-Rot**
ins-01	0.03115	0.06568
ins-02	0.01803	0.01833
ins-03	0.02182	0.04014
ins-04	0.07218	0.03614
ins-05	0.09601	0.75146
ins-06	0.12771	0.27191
ins-07	0.17746	1.17983
ins-08	0.31016	7.83662
ins-09	0.2638	1.17023
ins-10	0.3901	42.05413
ins-11	6.74009	300
ins-12	0.66777	38.76906
ins-13	0.90091	24.97785
ins-14	1.06072	128.75152
ins-15	8.07331	300
ins-16	48.36219	300
ins-17	4.83715	300
ins-18	2.48491	300
ins-19	109.13373	300
ins-20	6.16167	300
ins-21	16.58847	300
ins-22	127.26796	300
ins-23	18.9531	300
ins-24	16.34698	33.2955
ins-25	142.32429	300
ins-26	136.93896	300
ins-27	16.7328	300
ins-28	105.35609	300
ins-29	17.78156	300
ins-30	300	300
ins-31	1.55492	300
ins-32	300	300
ins-33	4.11694	300

6.1.2 Z3

The configuration with `Z3` as solver returned the best results. We setted the solver with the option "*auto_config*" that enable to use heuristics to automatically configure the solver [6], noticing an improvement in performance.
In our first model (without considering the *full_bottom* constraint 6.5) we solved 35 instances with an average time of 10 seconds. With the final model we are able to find 38 optimal solutions with an average time of 17 seconds, so even if the computational time is increased we think that a slight increase in execution

time is justified by a model capable of finding more solutions.

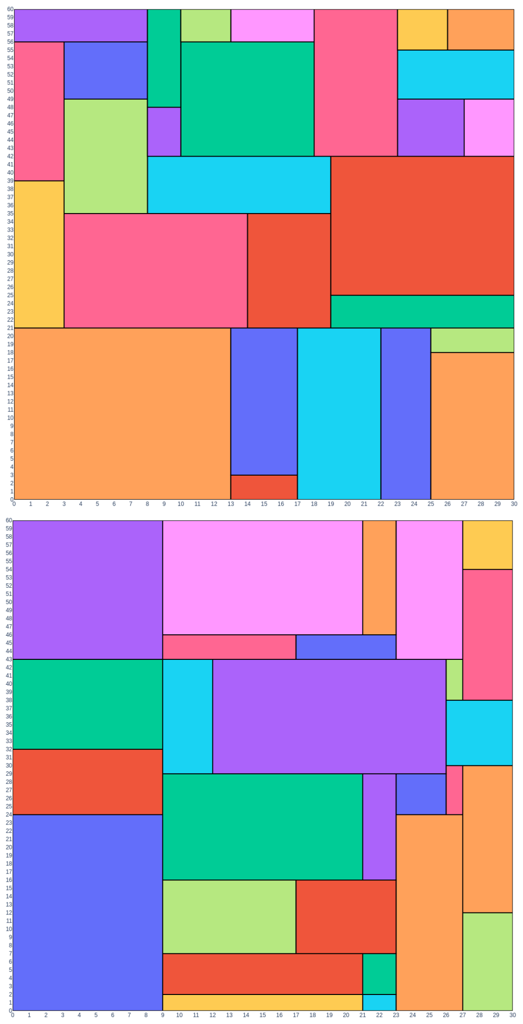


Figure 1: Solutions found for the instances 37 and 39 with no rotation

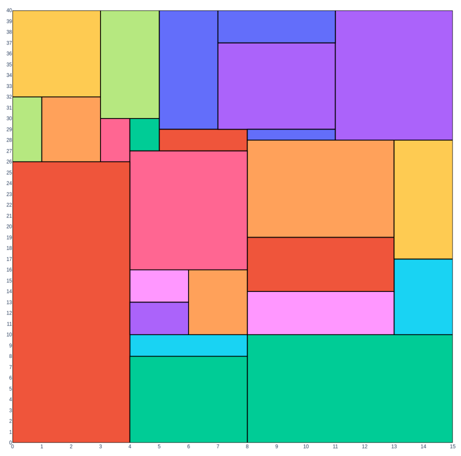


Figure 2: ins-35 with no rotation

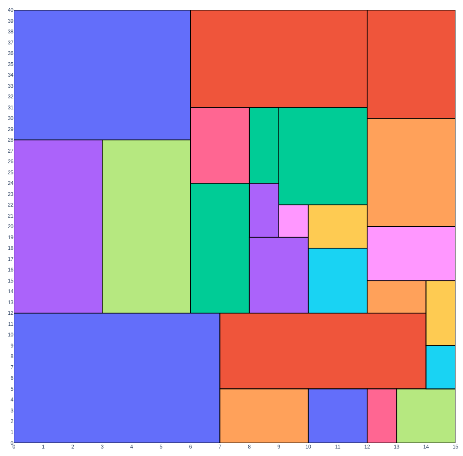


Figure 3: ins-36 with no rotation

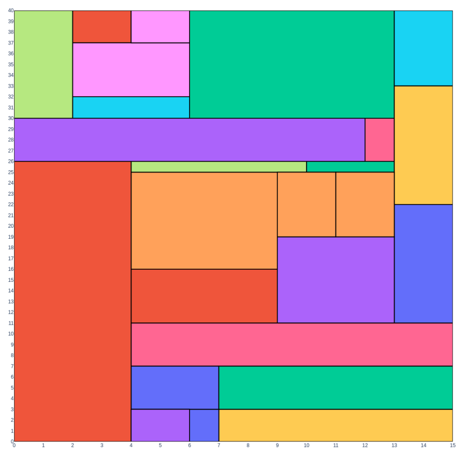


Figure 4: ins-35 with rotation

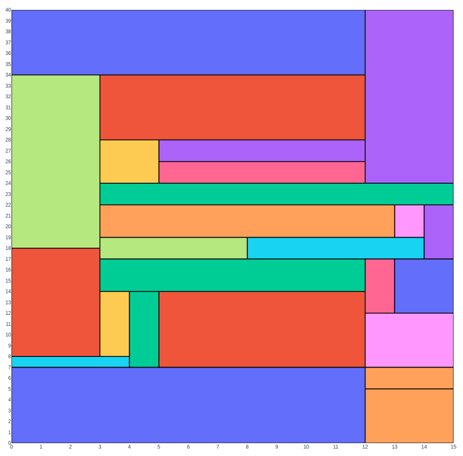


Figure 5: ins-36 with rotation

7 Final results and comments

The model with rotation performs way worse than the basic one: it solves less instances and has a higher average time; nonetheless, this is a common behavior with all the developed techniques.
Following there is a comparison between Z3 and CVC4, we can observe that Z3 is definitively faster and more effective, for this reason our final model uses it as solver.

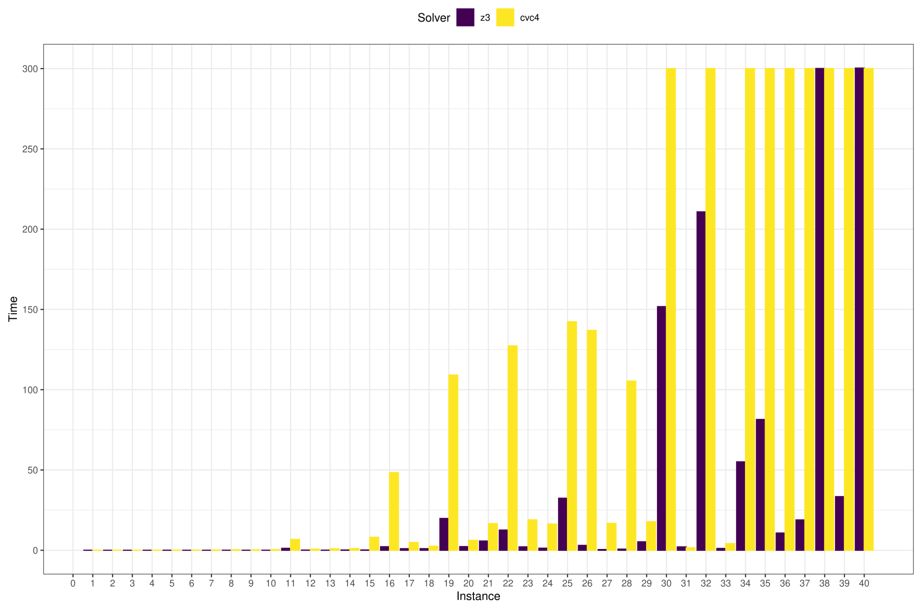


Figure 6: comparison between Z3 and CVC4 without rotation

Following there are the results for the whole set of instances with the final model, both with rotation and not:

Results of the final model with `Z3`:

Instance	**Time**	**Time-Rot**
ins-01	0.02927	0.02743
ins-02	0.01126	0.01411
ins-03	0.01289	0.01351
ins-04	0.01501	0.0162
ins-05	0.03851	0.06528
ins-06	0.01815	0.04824
ins-07	0.02100	0.08508
ins-08	0.02924	0.10978
ins-09	0.02966	0.17121
ins-10	0.03264	0.55984
ins-11	1.21335	13.55553
ins-12	0.10971	11.83042
ins-13	0.06177	2.55147
ins-14	0.16833	4.29923
ins-15	0.18448	5.38161
ins-16	2.22151	194.08484
ins-17	1.00431	52.20132
ins-18	1.01046	154.09229
ins-19	19.82373	300
ins-20	2.28858	229.92705
ins-21	5.75623	300
ins-22	12.59271	300
ins-23	2.15358	166.77124
ins-24	1.30359	21.25979
ins-25	32.41716	300
ins-26	3.04722	211.00699
ins-27	0.46319	225.87103
ins-28	0.72015	300
ins-29	5.28485	157.02511
ins-30	151.76241	300
ins-31	2.11907	32.19336
ins-32	210.76945	300
ins-33	1.14893	19.44357
ins-34	55.08106	216.47268
ins-35	81.47718	63.49064
ins-36	10.74554	55.66537
ins-37	18.91117	300
ins-38	300	300
ins-39	33.42447	300
ins-40	300	300

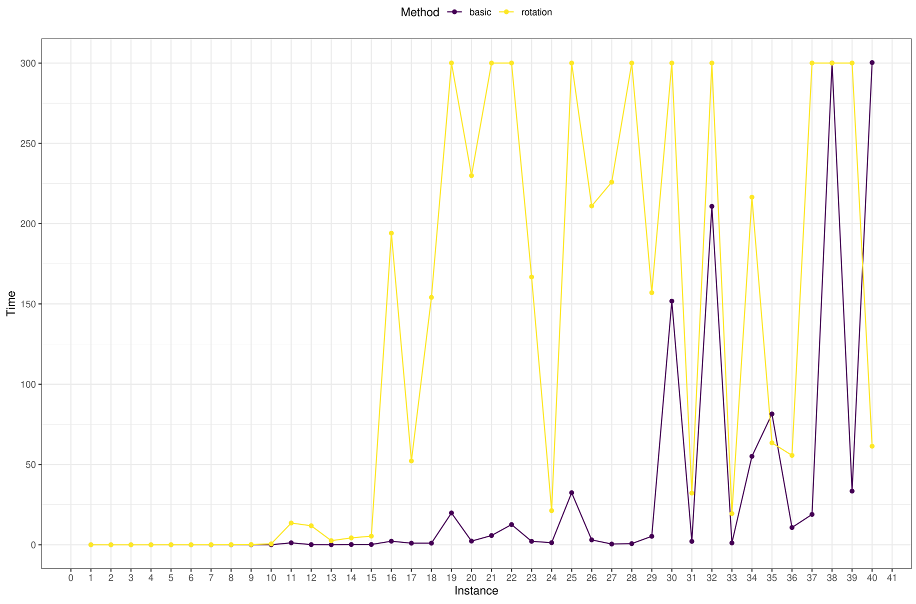


Figure 7: Comparison between basic and rotation models

References

[1] Andrea Micheli and Marco Gario. "pySMT's documentation". In: (). URL: `https://pysmt.readthedocs.io/en/latest/`.

[2] de Moura, Leonardo, and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: (2008), pp. 337–340.

[3] Clark W. Barrett et al. "CVC4". In: Lecture Notes in Computer Science 6806 (2011). Ed. by Ganesh Gopalakrishnan and Shaz Qadeer, pp. 171–177. DOI: `10.1007/978-3-642-22110-1\_14`. URL: `https://doi.org/10.1007/978-3-642-22110-1%5C_14`.

[4] Guillaume Bury. *MSAT*. URL: `https://github.com/Gbury/mSAT`.

[5] Bruno Dutertre et al. *YICES*. URL: `https://github.com/SRI-CSL/yices2`.

[6] CM Wintersteiger. "Module Z3". In: (). URL: `https://z3prover.github.io/api/html/ml/Z3.html`.