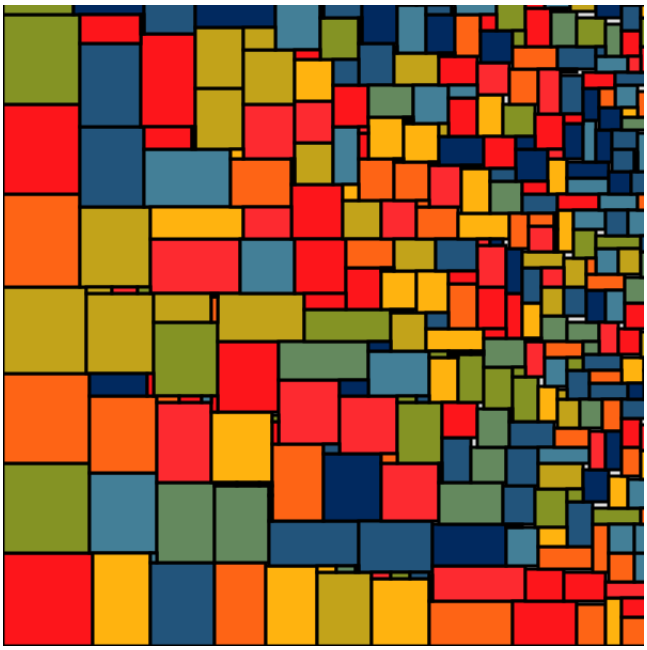
Alma Mater Studiorum
Università di Bologna

VLSI Design

CP 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) estimates 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. We can formalize this with the *diffn* packing constraint [1]:

$$diffn(p_x, p_y, x, y)$$

4.3 Rectangle packing as a schedule problem

The rectangle packing problem can be seen as a bi-dimensional scheduling problem. 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.

$$cumulative(p_x, x, y, l)$$

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

$$cumulative(p_y, y, x, w)$$

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 a significant circuit in (0, 0), where significant means the first one according to a particular feature. The feature we evaluated are:

- Area
- Height
- Perimeter
- Length / width ratio

In section 7 we will discuss which of these strategies leads to the best results.
No matter the strategy we adopt, placing a relevant circuit in (0, 0) leads to breaking the symmetries in most cases.
Moreover, we add a lexicographic constraint in order to ignore solutions where two rectangles with the same size are placed at the same coordinates.

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:

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

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

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:

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

6 Search heuristics

In order to find the set of heuristics that allows the best performances, we tested many combinations of search heuristics, both for variables and for domains.
As far as variables are concerned, we tested these heuristics:

- *input_order*
- *smallest*

- $first\_fail$
- dom_w_deg

As suggested in MiniZinc documentation.
As far as domain is concerned, we tested the following heuristics:

- $indomain\_min$
- $indomain\_median$
- $indomain\_random$

7 Testing the models

As we discussed in the previous sections, there are many combinations of heuristics and symmetry breaking constraints. Thus, we tested all the different models on the first 30 instances of the problem, without rotation, using the Chuffed[2] solver with the free search option. We did not use the $indomain\_random$ heuristic, and therefore restarting, since Chuffed does not support it yet.

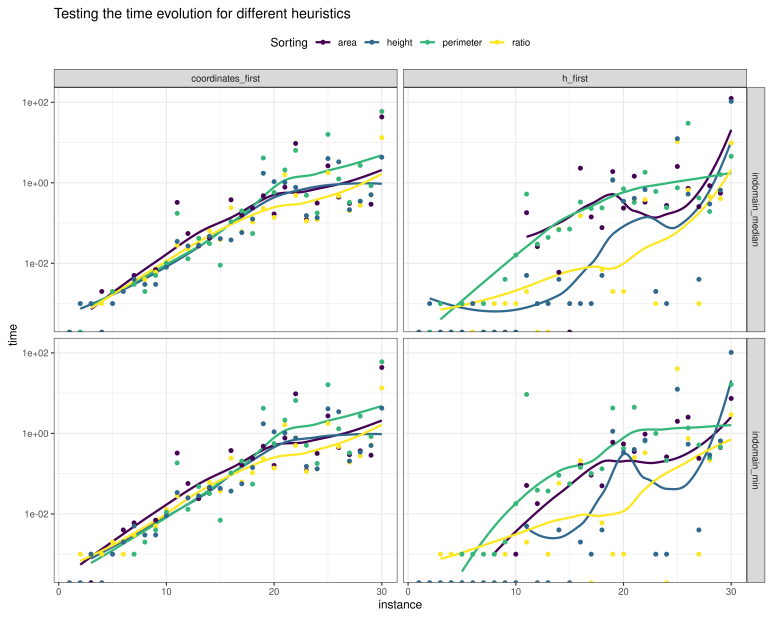


Figure 1: Performances using different search heuristics and symmetry breaking strategies

As expected, the time increase with the difficulty of the instance. Nonetheless, it is quite positive that all found solutions are optimal. In particular:

search_domain	search_order	sorting	avg_time	median_time	sd_time	optimal
indomain_min	l_first	area	0.532	0.034	1.423	30
indomain_median	coordinates_first	height	0.600	0.044	1.189	30
indomain_min	coordinates_first	height	0.606	0.045	1.203	30
indomain_median	coordinates_first	ratio	0.658	0.063	2.409	30
indomain_min	coordinates_first	ratio	0.668	0.062	2.453	30
indomain_median	l_first	ratio	0.760	0.001	2.556	30
indomain_median	l_first	perimeter	1.472	0.212	5.495	30
indomain_min	l_first	ratio	1.524	0.001	7.352	30
indomain_min	l_first	perimeter	1.751	0.117	4.003	30
indomain_median	coordinates_first	area	1.999	0.140	7.992	30
indomain_min	coordinates_first	area	2.011	0.140	8.024	30
indomain_median	coordinates_first	perimeter	3.162	0.082	11.053	30
indomain_min	coordinates_first	perimeter	3.215	0.080	11.232	30
indomain_min	l_first	height	3.976	0.001	18.788	30
indomain_median	l_first	height	4.063	0.001	19.252	30
indomain_median	l_first	area	4.545	0.052	22.665	30

The overall best model uses the *indomain_min* heuristic for the domain, and it looks for the l variable before looking for the coordinates.
Nonetheless, it is important to note that, since we allowed the solver to adopt a free search, it may have ignored the heuristic annotations. On the other hand, this proved to be the best model also with other solvers such as Gecode.

8 Implementation details

Once we found the model that offers the best performances, we tested it using different solvers in order to find the best one, and maximize the number of found solutions.
The solvers we tested are:

- Gecode
- Or-tools[3]
- Or-tools with free search
- Chuffed
- Chuffed with free search

Once again, we tested only instances from 1 to 30 for time reasons. These are the overall results:

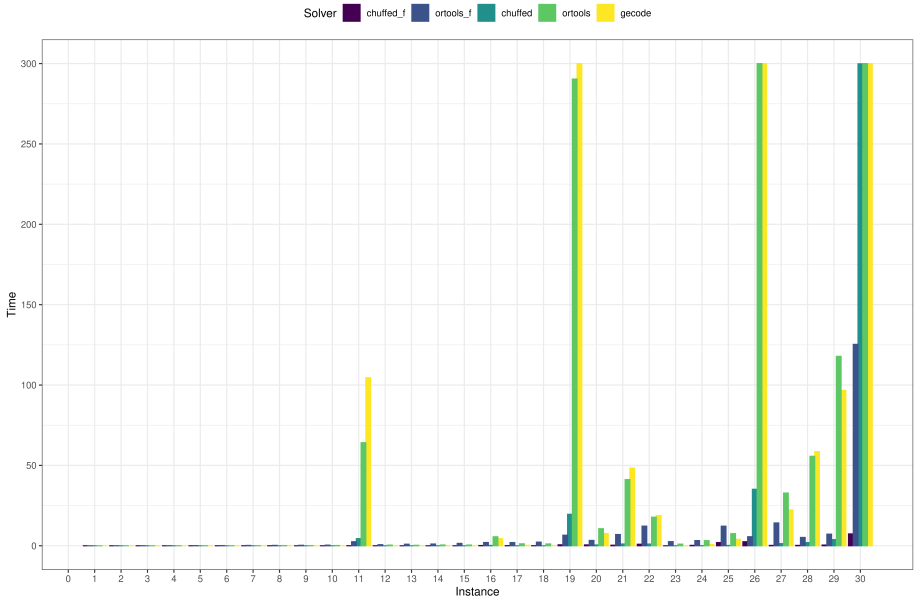


Figure 2: Execution time for different solvers

solver	avg_time	median_time	sd_time	found	optimal_found
Chuffed_f	0.532	0.034	1.423	30	30
Ortools_f	7.315	2.02	22.634	30	30
Chuffed	12.317	0.011	54.783	29	29
Ortools	41.756	1.083	90.292	28	28
Gecode	42.216	0.061	91.636	27	27

As the table suggests, using Chuffed with free search gives the best performances, and it is more than 10 times faster than the second best option, which is Ortools with free search. Moreover, it is able to solve all of the instances.
It is important to note that all the found solutions are optimal, and that there are no suboptimal solutions: this is due to the fact that solvers without free search use the *indomain_min* heuristic, which leads to the optimal solution, or, in case there is a timeout, to no solution at all.

9 Final results

Following there are the tables with the final results with **Chuffed free search**. We solved 39 instances withou rotation and 37 with rotation.

No rotation model:

Name	**Time**	l
ins-01	0.00000	8
ins-02	0.00000	9
ins-03	0.00000	10
ins-04	0.00000	11
ins-05	0.00000	12
ins-06	0.00000	13
ins-07	0.00000	14
ins-08	0.00000	15
ins-09	0.00000	16
ins-10	0.00100	17
ins-11	0.07500	18
ins-12	0.01900	19
ins-13	0.00000	20
ins-14	0.00500	21
ins-15	0.00100	22
ins-16	0.95700	23
ins-17	0.11700	24
ins-18	0.04500	25
ins-19	1.63400	26
ins-20	0.30000	27
ins-21	0.51300	28
ins-22	3.20500	29
ins-23	0.00100	30
ins-24	0.22500	31
ins-25	1.56200	32
ins-26	1.24700	33
ins-27	0.17000	34
ins-28	0.36700	35
ins-29	0.73400	36
ins-30	6.53000	37
ins-31	0.23900	38
ins-32	4.00800	39
ins-33	0.19300	40
ins-34	1.00100	40
ins-35	0.55300	40
ins-36	0.00400	40
ins-37	13.04500	60
ins-38	21.84800	60
ins-39	53.32100	60
ins-40	300	91

Rotation model:

Name	**Time**	l
ins-01	0.00000	8
ins-02	0.00000	9
ins-03	0.00000	10
ins-04	0.00000	11
ins-05	0.00100	12
ins-06	0.01500	13
ins-07	0.00100	14
ins-08	0.00400	15
ins-09	0.04000	16
ins-10	0.09000	17
ins-11	0.35300	18
ins-12	0.22000	19
ins-13	0.05700	20
ins-14	0.18500	21
ins-15	0.29600	22
ins-16	6.22400	23
ins-17	4.88600	24
ins-18	0.28500	25
ins-19	0.94800	26
ins-20	4.09300	27
ins-21	6.66000	28
ins-22	56.47800	29
ins-23	12.64100	30
ins-24	0.48600	31
ins-25	15.47000	32
ins-26	28.10500	33
ins-27	0.47400	34
ins-28	12.56400	35
ins-29	69.53500	36
ins-30	8.72400	37
ins-31	0.45700	38
ins-32	300	40
ins-33	3.96100	40
ins-34	0.55000	40
ins-35	2.68600	40
ins-36	0.99000	40
ins-37	300	61
ins-38	85.54800	60
ins-39	61.91800	60
ins-40	300	92

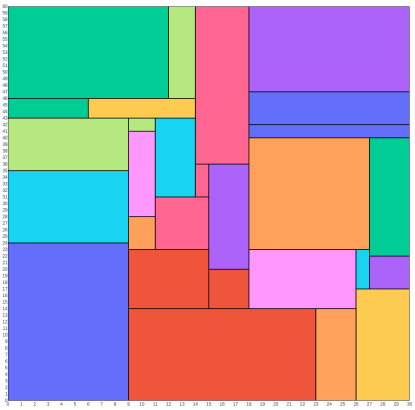


Figure 3: ins-39 with no rotation

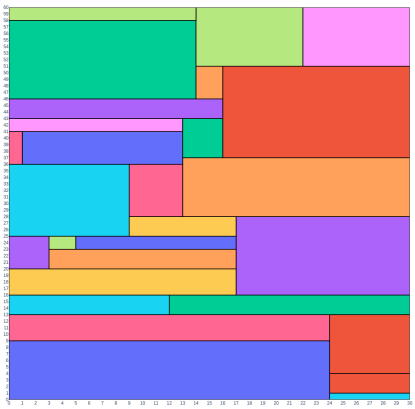


Figure 4: ins-39 with rotation

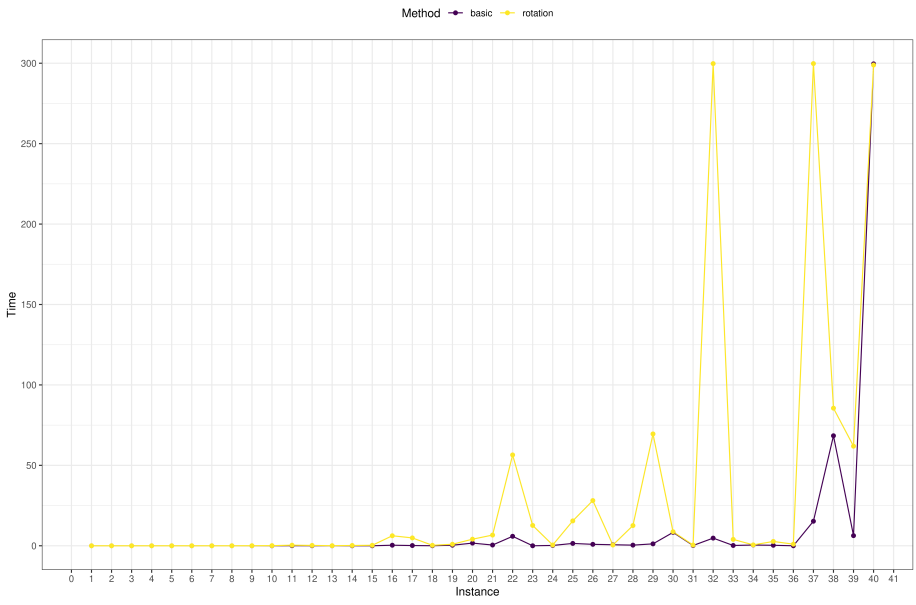


Figure 5: Comparison between basic and rotation models

References

[1] N. Nethercote. "MiniZinc: Towards a standard CP modelling language". In: *C. Bessiere, editor, Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, volume 4741 of LNCS, pages 529–543* (2002).

[2] Geoffrey Chu. "Chuffed - A lazy clause solver". In: *Data61, CSIRO, Australia* (2016).

[3] Laurent Perron and Vincent Furnon. "OR-Tools". Version 9.3. In: (2022). URL: `https://developers.google.com/optimization/`.