VLSI: VERY LARGE SCALE INTEGRATION

CDMO REPORT - CONSTRAINT PROGRAMMING

Lorenzo Pratesi
lorenzo.pratesi2@studio.unibo.it

Artificial Intelligence

University of Bologna

June 2022

Contents

1 Introduction

VLSI (Very Large Scale Integration) refers to the problem of integrating circuits into silicon chips. 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 plate is minimized. We will address two variants of the problem; in the first each circuit must be placed in a fixed orientation with respect to the others, while in the second case each circuit could be also rotated by 90° degrees. All the models have been implemented in MiniZinc (1), a constraint programming modelling language and solved using two solvers: Cuffed and Gecode. In Section 2 we will discuss how the problem is formally defined by means of its inputs and how the outputs will be structured. in Section 3 we will see how different models are implemented and all their constraints. In Section 4 we will describe all the search strategy and solver's configurations. In Section 5 we will see how models performed all over the instances provided.

2 Problem Description

The input of the problem is structured as follows:

- w: width of the plate;
- n: number of circuits;
- m: a $n \times 2$ matrix where m_{i1} and m_{i2} are the width and the height of the i-th circuit respectively.

The aim of the problem is to fill the plate with all the circuits by minimizing the total height without exceeding its width w.

The instances of the problem are structured in the following way:

w
n
$m_{11}\ m_{12}$
$m_{21}\ m_{22}$
...
$m_{n1}\ m_{n2}$

The outputs of the problem will be structured in the following way:

$w\ h$
n
$x_1\ y_1\ m_{11}\ m_{12}$
$x_2\ y_2\ m_{21}\ m_{22}$
...
$x_n\ y_n\ m_{n1}\ m_{n2}$

where x_i and y_i are the coordinates of the bottom-left corner of the i-th circuit.

Note that, if rotation of circuits is allowed, the dimension of a circuit could be inverted.

3 Modelling

We are now going to introduce the models and the constraints used to address the VLSI problem. Three types of models have been defined:

- **Complete**: the model that uses all the constraints described below, without allowing rotation of circuits;
- **Final**: the model without using the channeling and symmetry breaking constraint and without allowing rotation of circuits;

- **Rotation**: like the Final model, but allowing rotation.

3.1 Variables

For each circuit i, two variables x_i and y_i are defined, representing the coordinates of the bottom-left corner of the circuit.

A variable h is also defined, representing the maximum height of the area of the plate covered by the circuits. The aim of the solver is to minimize this variable.

If rotation is allowed, for each circuit i a variable r_i is defined, representing the circuit orientation. Two functions have been also defined:

$$get_{dx}(i, r) = (1 - r_i)m_{i1} + r_i m_{i2}$$

$$get_{dy}(i, r) = (1 - r_i)m_{i2} + r_i m_{i1}$$

They return the correct size of the circuit i given the circuit orientation r_i

3.1.1 Bounds

In order to define the bounds of the variables, we must first define some constants:

- $area_{min} = \sum_{i=1}^{n} m_{i1}m_{i2}$
- $min_h = max(max(m_{02}, ..., m_{n2}), \lfloor area_{min}/w \rfloor)$
- $max_h = \sum_{i=1}^{n} m_{i2}$

If rotation is allowed:

- $area_{min} = \sum_{i=1}^{n} m_{i1}m_{i2}$
- $min_h = max(max(min(m_{01}, m_{02}), ..., min(m_{n1}, m_{n2})), \lfloor area_{min}/w \rfloor)$
- $max_h = \sum_{i=1}^{n} max(m_{i1}, m_{i2})$

The constant $area_{min}$ represents the minimum area of the plate covered by the circuits, The constant min_h represents the minimum height possible. The intuition behind its definition is that the minimum height must be either the maximum height among all the circuits or the height of the plate as it was filled perfectly with $area_{min}$. The constant max_h represents the maximum height possible.

Let's define the bounds of the variables:

- $\forall_{i \in \{1..n\}}\ 0 \leq x_i \leq w - min(m_{01}, ..., m_{n1})$
- $\forall_{i \in \{1..n\}}\ 0 \leq y_i \leq max_h - min(m_{02}, ..., m_{n2})$
- $min_h \leq h \leq max_h$

If rotation is allowed:

- $\forall_{i \in \{1..n\}}\ 0 \leq x_i \leq w - min(min(m_{01}, m_{02}), ..., min(m_{n1}, m_{n2}))$
- $\forall_{i \in \{1..n\}}\ 0 \leq y_i \leq max_h - min(min(m_{01}, m_{02}), ..., min(m_{n1}, m_{n2}))$
- $min_h \leq h \leq max_h$
- $\forall_{i \in \{1..n\}}\ r_i \in \{0, 1\}$

3.2 Ordering

As we will discuss in the following section, one of the heuristics that could be used for choosing the variables is following the order of the input. One possible idea to improve the search speed is to enforce the solver to choose a pre-defined variable ordering based on the sizes of the circuits. For instance, if we want to order the circuits so that the first ones are those with the biggest area, then we can define an array of indexes as

```
order = reverse(arg_sort([m[i, 1]*m[i, 2] | i in 1..n]))
```

Different sorting strategies could be taken into account, the best one (based on the instances results) was order by the width of each circuit, and if the width is the same consider also the height in the same way. The following ensure the ordering described:

- `reverse(arg_sort([m[i, 1] * max(w + 1, max_h+1) + m[i, 2] | i in 1..n]))`
- ```
  reverse(arg_sort([get_dx(i, r) * max(w + 1, max_h+1) + \
      get_dy(i, r) | i in 1..n]))
  (if rotation is allowed)
  ```

Probably that is the best ordering strategy because it induces the solver to follow a greedy approach towards filling first the width of the plate, in addition to fail first.

3.3 Dual Model

We are now going to introduce the Dual model of the model with the variable described above. For simplicity, it has been defined only without allowing rotation.

3.3.1 Variables and Bounds

We can represent the dual model of this problem as an bi-dimensional array b, where each axis represents a coordinate of the plate. $b_{ij} = k$ means that in the position of the plate indicated by i and j there is the circuit k. More formally:

$$\forall_{i \in \{0,..,w-1\}, j \in \{0,..,max_h-1\}} \; b_{ij} = k \qquad k \in \{0, ..., n\}$$

Note that k can assume also 0 as value, that represents the absence of circuits.

3.4 Constraints

3.4.1 No over w/h constraint

We must ensure that all the circuits do not exceed the plate both horizontally and vertically. The following constraint ensure this behavior:

$$\forall_{i \in \{1..n\}} \; x_i + m_{i1} \leq w \land y_i + m_{i2} \leq h$$

If rotation is allowed:

$$\forall_{i \in \{1..n\}} \; x_i + get_{dx}(i, r) \leq w \land y_i + get_{dy}(i, r) \leq h$$

Basically, the sum of each circuit's coordinates and its corresponding size must be less or equal the maximum bound h or w.

3.4.2 No overlap constraint

In order to ensure that each circuit will not be placed one on top of the other (so to not overlap) for each pair of circuits (i, j) we must be sure that $x_i + m_{i0} \leq x_j \lor y_i + m_{i1} \leq y_j$. If one of those conditions holds,
  ```

it means that one circuit is placed in front of the other along the considered axis, so they do not overlap. MiniZinc provides the `diffn` constraint, which does exactly what is stated above. In particular the `diffn` constraint is defined as `diffn(x, y, dx, dy)` where x and y are arrays of integer variables representing the bottom-left coordinates of the circuits, while dx and dy are arrays of integer variables representing the sizes of the rectangles. For this particular problem, the constraint has been implemented as:

- `diffn(x, y, [m[i,1] | i in 1..n], [m[i,2] | i in 1..n])`
- `diffn(x, y, [get_dx(i, r) | i in 1..n], [get_dy(i, r) | i in 1..n])`
  `(if rotation is allowed)`

3.5 Cumulative constraint

The cumulative constraint is a MiniZinc global constraint defined as `cumulative(s, d, r, b)` where:

- s is an array of integers, representing the starting time of each job
- d is an array of integers, representing the time duration of each job
- r is an array of integers, representing the resources needed for each job;
- b is an integer representing the maximum amount of resources that could be allocated at each time t.

For our problem, this constraint could be used as an implied constraint. In particular, if we think:

- the two axis of the plate (x and y) as temporal lines,
- each circuit as a job and its coordinates as a starting time in the relative axis,
- the circuit's size along a particular axis as duration and the other one as a resource
- the axis bound as b.

we can define one cumulative constraint per axis and obtain the the expected results. This two constraints have been implemented as:

- `cumulative(x, [m[i,1] | i in 1..n], [m[i,2] | i in 1..n], h)`
  `(for x axis)`
- `cumulative(y, [m[i,2] | i in 1..n], [m[i,1] | i in 1..n], w)`
  `(for y axis)`

if rotation is allowed:

- `cumulative(x, [get_dx(i, r) | i in 1..n], [get_dy(i, r) | i in 1..n], h)`
  `(for x axis)`
- `cumulative(y, [get_dy(i, r) | i in 1..n], [get_dx(i, r) | i in 1..n], w)`
  `(for y axis)`

3.6 Symmetry breaking constraint

There are two different types of symmetries in this problem: horizontal and vertical flip of the plate (and both at the same time). In order to break them, one can simply place a circuit in the bottom-left corner of the plate. In this way we indirectly ensure that the solver will not consider symmetrical options. Three considerations here:

- the circuit placed in that position is the first that follows the ordering criterion described in Section 3.2;
- placing a circuit a priori in that position does not affect the optimality of the problem, but affects the search strategy of the solver. The symmetry constraint in some instances led to worse performance with respect to the model without it;
- if we try to break other symmetries in the plate (we will call them "local"), i.e. those without considering the circuit placed in its bottom-left corner, we could end in sub-optimal solutions. Refer to Appendix A for an analysis of a sub-optimal case.

The following constraint ensure the aforementioned behavior:

$$x_{order_0} = 0 \land y_{order_0} = 0$$

3.7 Channeling

The idea for the channeling constraint is to ensure better propagation by joining the dual and the standard model. In order to do so, we must ensure that:

$$\forall_{i,j,k}\, b_{ij} = k \Leftrightarrow x_k <= i \land x_k + m_{k1} > i \land y_k <= j \land y_k + m_{k2} > j$$

$$i \in \{0, ..., w-1\}, j \in \{0, ..., max_h - 1\}, k \in \{1, ..., n\}$$

Note that k never assumes 0 as value, because the absence of circuits is not supported by the standard model, so the solver is free to assign 0 in the dual model where the above condition is never met.

4 Search strategies

We used two different types of solver, Chuffed and Gecode. For each one, different heuristic strategies have been experimented. Some heuristic annotations are supported only by Gecode. For the Cuffed solver, the free search option has been also experimented. In the following, all the heuristics used for solving the problem will be described:

4.1 Variable choosing heuristics

- `dom_w_deg`: only available for Gecode, 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;
- `input_order`: choose the variable in order from the array, following the order described in 3.2;
- `fail_first`: choose the variable with the smallest domain size.

4.2 Domain constrain heuristics

- `indomain_min`: assign the variable its smallest domain value;
- `indomain_random`: only available for Gecode, assign the variable a random value from its domain.

4.3 Restart heuristics

The following heuristics are implemented only for Gecode and are ignored by Chuffed:

- `restart_luby(250)`: luby restart with scale 250;
- `restart_linear(100)`: linear restart with scale 100;
- `restart_none`: no restart.

5 Results

In this section we will describe the results obtained on the instances provided. We are going to see how different solvers, heuristics, and models performed on some samples. Neither one of the model was able to solve instance 40. The model that resulted the fastest is the Final model. Table 1 shows the timing statistics using different solvers and search configuration on the Final model. The fastest configuration resulted to be the one with Chuffed as solver with free search active, and default search heuristics (input_order, indomain_min, restart_none). Table 2 shows the solving statistics of the Final model over instance 13, one of the instances solved using all the configurations. Note how the number of failures is low when following the variable order and assigning its smallest domain value. Table 3 and Figure 1 shows a comparison between Final, Complete and Rotation models by means of solving times statistics. For all the experiments we used the fastest resulting configuration for the Final model (Chuffed with free search). We can clearly see that the Final model outperformed in almost all the instances the Complete model. The Rotation model had some difficulties in solving some particular instance.

Table 1: Timing statistics with different solvers and search configurations for Final model

Solver	Free Search	Heuristic variable	Heuristic value	Restart heuristic	Solved instances	Max solving time	Avg solving time
Chuffed	Yes	input_order	indomain_min	restart_none	**39**	**27.99**	**1.7385**
Chuffed	Yes	first_fail	indomain_min	restart_none	**39**	154.03	6.41
Chuffed	No	input_order	indomain_min	restart_none	30	146.97	15.3703
Chuffed	No	first_fail	indomain_min	restart_none	19	242.85	29.9763
Gecode	No	dom_w_deg	indomain_min	restart_luby	13	11.49	1.3031
Gecode	No	dom_w_deg	indomain_min	restart_linear	13	189.89	16.4292
Gecode	No	dom_w_deg	indomain_min	restart_none	16	278.92	25.6825
Gecode	No	dom_w_deg	indomain_random	restart_luby	31	59.96	10.3187
Gecode	No	dom_w_deg	indomain_random	restart_linear	30	200.52	24.364
Gecode	No	dom_w_deg	indomain_random	restart_none	15	97.11	10.85
Gecode	No	first_fail	indomain_min	restart_luby	13	10.78	1.2192
Gecode	No	first_fail	indomain_min	restart_linear	12	21.22	1.9217
Gecode	No	first_fail	indomain_min	restart_none	15	95.32	11.0633
Gecode	No	first_fail	indomain_random	restart_luby	30	219.21	15.8233
Gecode	No	first_fail	indomain_random	restart_linear	30	275.77	31.7947
Gecode	No	first_fail	indomain_random	restart_none	16	33.69	5.6994
Gecode	No	input_order	indomain_min	restart_luby	19	253.43	26.4763
Gecode	No	input_order	indomain_min	restart_linear	16	77.64	8.47
Gecode	No	input_order	indomain_min	restart_none	25	245.25	28.8436
Gecode	No	input_order	indomain_random	restart_luby	24	271.47	26.0567
Gecode	No	input_order	indomain_random	restart_linear	23	245.92	35.3461
Gecode	No	input_order	indomain_random	restart_none	15	267.03	29.4433

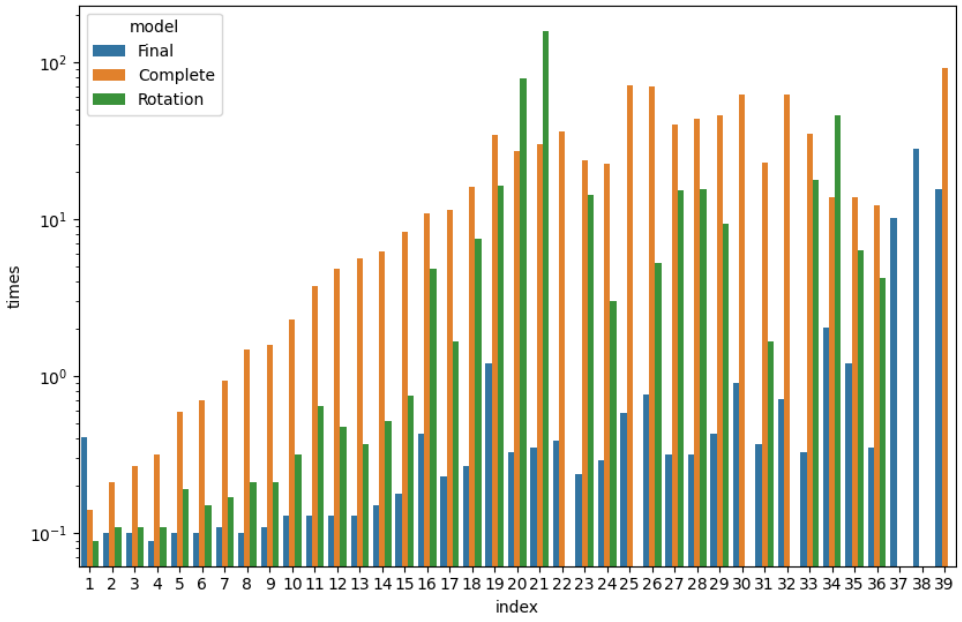
Table 2: Solving statistics on instance 13 (ins-13.txt) for Final model

Solver	Free Search	Heuristic variable	Heuristic value	Restart heuristic	Failures	Restarts	Propagations
Chuffed	Yes	input_order	indomain_min	restart_none	169	87	218723
Chuffed	Yes	first_fail	indomain_min	restart_none	522	92	326101
Chuffed	No	input_order	indomain_min	restart_none	169	86	216575
Chuffed	No	first_fail	indomain_min	restart_none	2033	87	607482
Gecode	No	dom_w_deg	indomain_min	restart_luby	24716	118	339135
Gecode	No	dom_w_deg	indomain_min	restart_linear	61417	119	793566
Gecode	No	dom_w_deg	indomain_min	restart_none	8986	0	121389
Gecode	No	dom_w_deg	indomain_random	restart_luby	5731	61	74078
Gecode	No	dom_w_deg	indomain_random	restart_linear	10985	61	115506
Gecode	No	dom_w_deg	indomain_random	restart_none	164066	0	1740154
Gecode	No	first_fail	indomain_min	restart_luby	27801	118	378533
Gecode	No	first_fail	indomain_min	restart_linear	82007	124	1029633
Gecode	No	first_fail	indomain_min	restart_none	12034	0	163310
Gecode	No	first_fail	indomain_random	restart_luby	10081	67	109883
Gecode	No	first_fail	indomain_random	restart_linear	69294	83	752506
Gecode	No	first_fail	indomain_random	restart_none	135361	0	1349451
Gecode	No	input_order	indomain_min	restart_luby	486	87	26963
Gecode	No	input_order	indomain_min	restart_linear	486	87	26963
Gecode	No	input_order	indomain_min	restart_none	794	0	17809
Gecode	No	input_order	indomain_random	restart_luby	54053	121	574319
Gecode	No	input_order	indomain_random	restart_linear	60248	78	591076
Gecode	No	input_order	indomain_random	restart_none	1910443	0	21066470

Table 3: Timings statistics for different models

	Solved instances	Max solving time	Avg solving time
Final	39	27.99	1.7385
Complete	37	91.87	22.5254
Rotation	32	158.06	12.8166

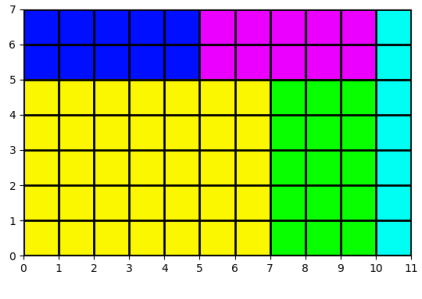
Figure 1: Solving times grouped by all the instances



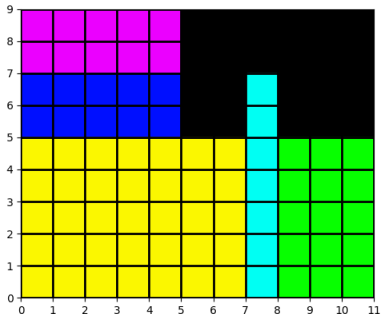
References

[1] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

Figure 2: Local symmetry



(a) Optimal configuration



(b) Sub-optimal configuration

A Local symmetry analysis

Figure 2a, shows an optimal configuration for those circuits. Suppose we decide to try a different configuration by keeping the yellow block in the same position but trying to invert the position of the cyan and green circuits. A possible optimal follow-up configuration is the one showed in Figure 2b, and it is clearly sub-optimal with respect to the previous one.