Alma Mater Studiorum - Università di Bologna

Project work in Combinatorial Decision Making and Optimization

Very Large Scale Integration - VLSI Design

Stefano Poggi Cavalletti
stefano.poggi2@studio.unibo.it

Table of Contents

1 Introduction

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. 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.

Given a fixed-width plate and a list of rectangular circuits, the problem consists in deciding how to place the circuits on the plate in order to minimize the length of the final device. The combinatorial optimization problem will be modeled and solved with Constraint Programming (CP) and Satisfiability Modulo Theories (SMT). Moreover, in addition to a base model, it has been developed a model which takes into account the rotation of the circuits, in order to compare the solutions and their complexity.

2 Problem Description

The input consists of a set of 40 instances in a text format, each one containing the following data:

- width of the silicon plate w
- number of circuits to place n
- horizontal dimension x_i of the i-th circuit (for each circuit i from 1..n)
- vertical dimension y_i of the i-th circuit (for each circuit i from 1..n)

The output, instead, will be structured in the following way:

$w \quad h$
n
$x_0 \quad y_0 \quad \hat{x_0} \quad \hat{y_0}$
$x_1 \quad y_0 \quad \hat{x_1} \quad \hat{y_1}$
...
$x_n \quad y_n \quad \hat{x_n} \quad \hat{y_n}$

where h is the maximum height reached by the circuits configuration; x_i and y_i are respectively the horizontal and vertical dimensions of the i-th circuit; $\hat{x_i}$ and $\hat{y_i}$ are the coordinates of the left-bottom corner for the i-th circuit.

3 CP Modeling

The first part technique used to approach the problem was to model it with CP (Constraint Programming) using Minizinc. CP is a declarative programming paradigm where the solver tries to find a solution (if this exists) by assigning a value to each variable, in a way that all the defined constraints are satisfied.

Before proceeding with the model definition it is worth noting that the instances were provided in text files, hence it was necessary to first convert them into .dzn data files, which are readable by Minizinc.

3.1 Parameters and variables

First of all, all the parameters, decision variables and objective variables of the problem had to be defined.

In particular, we are given the following .dzn input data file:

$w = width$
$n = number\ of\ circuits$
$x\_sizes = [x_0,\ x_1,\ ...,\ x_n]$
$y\_sizes = [y_0,\ y_1,\ ...,\ y_n]$

where:

- w is the plate width
- n is the number of circuits to place in the plate

- *x_sizes* and *y_sizes* are two arrays with the horizontal and vertical sizes of the circuits, respectively (indexed from 1 to the number of circuits)

Therefore, the same **input parameters** have been defined in the model using the Minizinc syntax:

```
int: width;
int: n;
set of int: CIRCUITS = 1..n;

array[CIRCUITS] of int: x_sizes;
array[CIRCUITS] of int: y_sizes;
```

As for the **output parameters**, two parameters x and y were defined, indexed from 1 to n: each element i of the arrays contains the horizontal and vertical coordinates of the bottom-left corner of the i-th circuit, respectively:

```
array[CIRCUITS] of var 0..width-1: x
array[CIRCUITS] of var 0..sum(y_sizes)-1: y;
```

3.2 Domain reduction

The **domain reduction contraints** allow to make the model more efficient by reducing the variable domain. This is done by defining a range for each variable of the array x: the bottom left corner of each rectangle i cannot be placed farther along the x axis than the plate width minus its width, otherwise it would fall outside the plate. A similar constraint can be defined for the domain of the y variables: the bottom left corner of each rectangle i cannot be placed higher on the y axis than the height of the plate minus its height.

```
constraint forall(i in CIRCUITS) (x[i] <= width - x_sizes[i])::domain;
constraint forall(i in CIRCUITS) (y[i] <= height - y_sizes[i])::domain;
```

Moreover, with the same purpose of reducing variables domain, upper and lower bounds were defined for the plate's height:

- lower bound: the plate's height must be bigger than the height of the tallest rectangle

$$int: \ lowb = max(y\_sizes);$$

- upper bound: the plate's height must be below the sum of all the heights of each rectangle

$$int: \ upb = sum(y\_sizes);$$

The **objective variable** is *height*, which is the height of the plate: it is the variable to be minimised by the model during search. It ranges from the lower to the upper bound and is defined as:

```
var lowb..upb: height;
height = max([y[i] + y_sizes[i] | i in CIRCUITS]);
```

3.3 Constraints

One of the most important parts of the problem is related to the definition of multiple constraints and their propagation in order to remove inconsistent values from variables domain. The use

of **global constraints** is useful for the solution of this problem, since it allows to obtain more efficient solutions thanks to propagation algorithms. Two different global constraints were used: **cumulative** and **diffn**.

The cumulative constraint was taken from the modeling of scheduling problem: indeed it is usually used when we want you need to constraint the usage of shared resources by different tasks. In general, it requires that a set of tasks given by start times **s**, durations **d** and resource requirements **r** never require more than a global resource bound **b** at any one time. Our problem can be seen as the one where there is a fixed capacity resource (plate width/height) and each task (circuit) has its own start time (placement) and duration (width/height). The constraint is repeated twice, once along each axis.

```
constraint cumulative(y, y_sizes, x_sizes, width);
constraint cumulative(x, x_sizes, y_sizes, height);
```

The diffn constraint is is a non-overlapping constraint which holds if, for each pair of shapes, there exists a dimension in which their projections don't overlap. Given the vectors of x and y bottom-left coordinates and the vectors of x and y sizes, with the diffn constraint we prevent the circuits from overlapping.

```
constraint diffn(x, y, x_sizes, y_sizes);
```

Furthermore, **symmetry braking constraints** were applied to reduce the number of solutions: the solver may in fact explore many symmetric variants of the same solution. The basic idea behind symmetry breaking is to impose an order. In our case it was decided to place always the biggest circuit in the bottom left part of the plate. To do this we need to define an order of the circuits, which is done by sorting them in descending order considering their area, given by the product between *x_sizes* and *y_sizes*.

```
array [CIRCUITS] of  int: ord_circ = sort_by(CIRCUITS, area);
```

After that we use the global constraint *lex_lesseq* on the coordinates of the circuits: This requires that the array x is lexicographically less than or equal to array y. It compares them from first to last element, regardless of indices.

```
constraint symmetry_breaking_constraint(
    let {
        int: c1 = ord_circ[1], int: c2 = ord_circ[2]
    } in lex_lesseq([y[c1],x[c1]], [y[c2],x[c2]])
);
```

3.4 Search

Different combinations of search heuristics, for variables and domains, as well as restart strategies were employed to compare the model performances: *input_order*, *first_fail* and *dom_w_deg* for variables (which specify the order in which the values should appear in the search tree), and *indomain_min* for domain (which specify which choice of value for each variable will be explored next).

More specifically, *dom_w_deg* will 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. *indomain_min* will assign the variable its smallest domain value.

Also, restarting the search is useful to introduce randomness and break deterministic behaviour in searching solutions. In the model, different restart strategies were chosen:

- restart_constant(100)
- restart_linear(100)
- restart_geometric(1.5, 100)
- restart_luby(100)

3.5 Rotation model

In the case we decided to take rotation of the circuits into account we should perform some modifications to the model. This is done by introducing a boolean array indexed by the number of circuits which specifies if each circuit is rotated or not:

```
array[CIRCUITS] of var bool: rotation;
```

In case of rotation each rectangle will have its height and width swapped, therefore we need to update their actual values of widths and heights accordingly:

$$\forall i \; in \; 1..n \;\; x\_sizes\_rot = \begin{cases} y\_sizes[i] & \text{if rotation[i]} \\ x\_sizes[i] & \text{otherwise} \end{cases}$$

$$\forall i \; in \; 1..n \;\; y\_sizes\_rot = \begin{cases} x\_sizes[i] & \text{if rotation[i]} \\ y\_sizes[i] & \text{otherwise} \end{cases}$$

The objective function *height* and the constraints seen before are modified with the vectors x_sizes_rot and y_sizes_rot to take rotation into account. We also introduce a new constraint: a circuit cannot be rotated if its height is greater than the plate width.

```
constraint forall(i in CIRCUITS)(y_sizes[i] > width -> rotation[i]==false);
```

The output file is also modified so that the string *"rotated"* is printed next to the coordinates of the output circuits: this indicates that the relative circuit has been rotated.

As an example, a possible output for the instance 3 is the following:

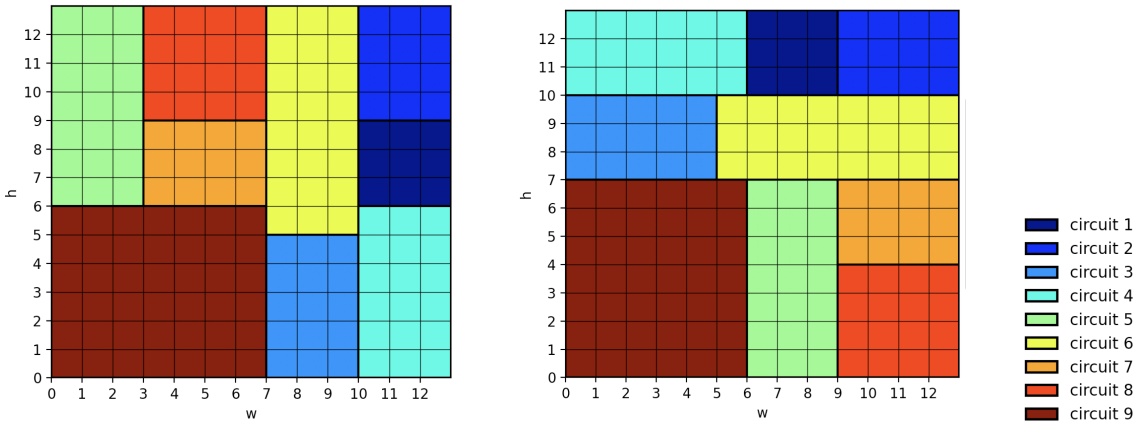
10 10
6
3 3 3 4
3 4 6 4 *rotated*
3 6 0 4
3 7 3 7 *rotated*
4 4 0 0
4 6 4 0 *rotated*

3.6 Results

Table 1: Solving times obtained by CP models (without and with rotation)

Instance	No rotation	Rotation
1	0,221	0,206
2	0,202	0,206
3	0,204	0,207
4	0,203	0,275
5	0,202	1,807
6	0,200	0,561
7	0,203	45,16
8	0,201	0,246
9	0,200	7,909
10	0,205	-
11	0,198	-
12	0,208	0,234
13	0,200	-
14	0,237	-
15	0,249	0,254
16	-	-
17	6,086	0,273
18	1,422	1,234
19	-	-
20	5,687	-
21	0,204	-
22	-	-
23	0,225	-
24	0,209	0,279
25	-	-
26	147,1	1,755
27	0,219	0,205
28	16,68	-
29	0,222	-
30	-	-
31	-	-
32	-	-
33	0,221	0,229
34	-	-
35	-	-
36	0,259	-
37	-	-
38	-	-
39	-	-
40	-	-

As shown in the table, the model without rotation is able to solve more instances within the time limit of 300 seconds, compared to the rotation model. Moreover, after performing different experiments with search heuristics and restart strategies, the best configuration turned out to be *input_order* and *indomain_min* with $restart\_geometric(1.5, 500)$.



(a) Solution without rotation

(b) Solution with rotation

(c) Legend of circuits

Figure 1: Example of a possible solution for the instance n. 6

4 SMT

Another approach to this problem consists in using SMT (Satisfiability Modulo Theories) that was chosen because of its expressive power due to first-order logic compared to SAT, which instead can be complex to formalize. The main drawback of SMT models is the worse efficiency, which however is compensated by the higher expressivity and scalability.

4.1 Variables

Modeling with SMT followed the same scheme that was seen for CP. First, the coordinates of the circuits have been encoded into two vectors x and y of integers with dimension given by the number of circuits. As previously seen, the arrays x_sizes and y_sizes contain the widths and heights of all the circuits, while *height* represents the maximum height of the plate and it is the objective function to minimize. The optimizer is called with the function `Optimize()` given by Z3 solver, whose aim is to minimize the objective function *height*.

As done before, the variable domain has been reduced to speed up the search: first of all, the horizontal coordinates x_i have to be greater than zero and their sum with the circuit widths must be below the plate width.

$$\forall i \in 1..n \quad x_i \geq 0 \ \wedge x_i + x\_sizes_i \leq w$$

Similarly, the vertical coordinates y_i have to be greater than zero and their sum with the circuit heights must be below the maximum plate height.

$$\forall i \in 1..n \quad y_i \geq 0 \ \wedge y_i + y\_sizes_i \leq height$$

4.2 Constraints

The main constraints are meant to tell the solver to respect the boundaries of the plate for the horizontal and vertical coordinates:

$$\forall i \in 1..n \quad x_i + x\_sizes_i \leq w$$

$$\forall i \in 1..n \quad y_i + y\_sizes_i \leq height$$

Furthermore, a similar definition of the global constraint *cumulative* (previously used in Minizinc for the CP solution) was given. Our problem was seen as a resource usage task where each circuit is an activity whose duration is the vertical height, its resources is equal to its horizontal width and the total number of resources is the plate width w. The same reasoning can be applied to the other axis.

This is defined in the following way:

$$cumulative(y, y\_sizes, x\_sizes, width)$$
$$cumulative(x, x\_sizes, y\_sizes, height_{max})$$

Moreover, a non-overlapping constraint is necessary to make sure that no pair of rectangles overlap:

$$\forall i, j \in 1..n, i \neq j \quad x_i + x\_sizes_i \leq x_j \vee x_j + x\_sizes_j \leq x_i \vee y_i + y\_sizes_i \leq y_i \vee y_i + y_j + y\_sizes_j \leq y_i$$

Symmetry braking constraints are the same as in CP: it has been decided to put the biggest circuit in the bottom left part of the plate, at x and y coordinates equal to zero.

4.3 Rotation model

If we allow the rotation of the circuits, we need to perform some modification to the model: we introduce two other vectors of variables *x_sizes_rot* and *y_sizes_rot* which contain the actual widths and heights considered, since in case of rotation the values of width and height will be swapped.

$$\forall i \in 1..n \quad (x\_sizes\_rot_i = x\_sizes_i \ \wedge \ y\_sizes\_rot_i = y\_sizes_i) \vee$$
$$\vee \ (x\_sizes\_rot_i = y\_sizes_i \wedge y\_sizes\_rot_i = x\_sizes_i)$$

It is also necessary to add another symmetry braking constraint due to the fact that, in case of rectangles which are squares (where width is equal to height), the solver will consider identical the solutions where it is rotated and where it is not rotated. With this constraint we force width and height to be the original ones, hence avoiding a possible rotation.

$$\forall i \in 1..n \quad x\_sizes_i = y\_sizes_i \implies (x\_sizes\_rot_i = x\_sizes_i \wedge y\_sizes\_rot_i = y\_sizes_i)$$

As previously done, the output file is modified to print the string *"rotated"* in case a circuit is rotated.

4.4 Search

The standard search method provided by Z3Py was used to compare the performance of the models: the class `Optimize` allows to get an assignment for each variable which minimizes the objective function through the method `Optimize.minimize(<minimization objective>)`, hence it is applicable to our problem where we want to minimize *height*.

4.5 Results

Table 2: Solving times obtained by SMT models (without and with rotation)

Instance	No rotation	Rotation
1	0,005	0,018
2	0,010	0,049
3	0,014	0,090
4	0,025	0,308
5	0,041	0,859
6	0,063	0,592
7	0,078	0,889
8	0,095	1,453
9	0,109	1,249
10	0,339	53,28
11	-	-
12	0,615	82,80
13	0,890	48,83
14	2,218	127,5
15	1,444	77,81
16	-	-
17	6,597	27,88
18	10,02	-
19	-	-
20	20,50	-
21	-	-
22	-	-
23	14,08	-
24	0,621	-
25	-	-
26	169,9	-
27	39,64	-
28	33,83	-
29	43,54	-
30	-	-
31	18,178	-
32	-	-
33	17,81	-
34	-	-
35	-	-
36	-	-
37	-	-
38	-	-
39	-	-
40	-	-

5 Conclusion

VLSI turned out to be a problem of non-negligible complexity: the goal was to find a good model through a combination of constraints and types of search in order to get the most efficient solution with as smaller solving times as possible.
In both cases with CP and SMT we were able to get satisfactory results, in particular in the base model which does not allow rotation (which adds further complexity). In particular, among the two techniques, CP managed to obtained the best results by solving a larger number of instances. In all the cases, as the number of circuits in the plate increases, the complexity raises in the same

manner, and the solver may not able to find a solution within the time limit of 300 seconds.

6 References

[1] The MiniZinc Handbook, https://www.minizinc.org/doc-2.3.0/en/

[2] Z3 API in Python, https://ericpony.github.io/z3py-tutorial/guide-examples.htm

[3] Programming Z3, Bjørner et al,
https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-bounded-model-checking