VLSI Report

Using CP & SMT models to solve VLSI problems of increasing difficulty: CP section.

Alessandro Maggio[1] and Serban Cristian Tudosie[2]

[1]alessandro.maggio5@studio.unibo.it
[2]serban.tudosie@studio.unibo.it

August 2021

CONSTRAINT PROGRAMMING

The most intuitive way to solve VLSI problem is using CP techniques. For this purpose we have used MiniZinc and several libraries and solver it has available, trying to achieve a good enough problem modelling to find a solution for all the 40 instances.

Problem formulation for CP

We noticed that the task has a strong analogy with the scheduling problem. Since we have to find positions for the given chips, we decided to use the global `cumulative` and `diffn` constraints of MiniZinc together with standard domain constraints. The following lexical conventions and equations show a rigorous way to explain our formulation.

Parameters:

- n number of chips
- w maximum plate width
- $chips\_w_i$ width of the i_th chip
- $chips\_h_i$ height of the i_th chip

Variables:

- h plate's height that must be minimized
- x_i bottom left x coordinate of the i_th chip
- y_i bottom left y coordinate of the i_th chip

The coordinates, after the assignment, represent a complete solution of a problem instance. The height variable is crucial to evaluate the solution quality.

Constraints:

$$0 \leq y_i \leq h - chips\_h_i \qquad i \in [1, n] \tag{1}$$

$$0 \leq x_i \leq w - chips\_w_i \qquad i \in [1, n] \tag{2}$$

$$\forall i, j.$$
$$y_i + chips\_h_i \leq y_j \bigvee y_j + chips\_h_j \leq y_i \tag{3}$$
$$\bigvee x_i + chips\_w_i \leq x_j \bigvee x_j + chips\_w_j \leq x_i$$

$$\forall j \sum_{i \mid y_i \leq j \leq y_i + chips\_h_i} chips\_w_i \leq w \qquad j \in [0, h) \tag{4}$$

$$min\_h \leq h \leq max\_h \tag{5}$$

Constraints Implementation

The first two constraints (1)(2) reduce the domain of the chips positions to only the places which avoid to make the chips exceed the plate's boundaries. Since in MiniZinc an array is declared such that all its elements must have the same domain range, it is not possible to operate the exact initialization expressed previously. However a quite accurate domain definitions are $0 \leq y_i \leq h - min(chips\_h)$ and $0 \leq x_i \leq w - min(chips\_w)$ combined with two extra constraints that emulate the desired behaviour (Code 1).

```
set of int: CHIPS = 1..n;
array[CHIPS] of var 0..w - min(chips_w): x_positions;
array[CHIPS] of var 0..max_h - min(chips_h): y_positions;
constraint max(i in CHIPS)(chips_h[i] + y_positions[i]) <= h;
constraint max(i in CHIPS)(chips_w[i] + x_positions[i]) <= w;
```

Code 1: Max Constraints

The third constraint (3) forces chips not to overlap. In order to achieve it, we preferred a MiniZinc library global constraint called *diffn*; in our case it can be formulate, according to our parameters and variables names, as

$$\texttt{diffn}(\{x_1 ... x_n\}, \{y_1 ... y_n\}, \{chips\_w_1 ... chips\_w_n\}, \{chips\_h_1 ... chips\_h_n\})$$

The cumulative constraint (4) is used along the vertical direction, exploiting a scheduler-like point of view, in our case formulated, still according to our parameters and variables names, as

$$\texttt{cumulative}(\{y_1...y_n\}, \{chips\_h_1...chips\_h_n\}, \{chips\_w_1...chips\_w_n\}, w)$$

MiniZinc offers an internal and efficient implementation of it and returns the y array. We tried to add a very similar constraint for the horizontal direction:

$$\texttt{cumulative}(\{x_1...x_n\}, \{chips\_w_1...chips\_w_n\}, \{chips\_h_1...chips\_h_n\}, h)$$

Sadly it reduces the performance, forcing to omit this second constraint. The last constraint (5) is useful in order to reduce the search space for h, providing to the solver the smallest possible range. The maximum possible value happens when all the chips are on a single column, so $max\_h = sum(chips_h)$. The minimum value is obtained when there are no empty spaces among the chips and all the plate width is fitted; mathematically it can be written as:

$$min\_h = \frac{\sum_{i \in [1,n]} chips\_w_i * chips\_h_i}{w}$$

Symmetry breaking constraint

There are two main axis of symmetry parallel to the plate's borders. Mirroring the disposition along the horizontal axis we encountered a problem: MiniZinc cannot iterate over h since it is not grounded yet and using max_h would be very computationally expensive since it includes a lot of empty space. Then we proceeded to break the symmetry along the vertical axis as shown in (Code 2) but it appears a critical disadvantage. Only few instances benefit from its introduction while most cases are solved faster without it.

```
constraint symmetry_breaking_constraint(
    lex_lesseq([x_positions[k] | k in CHIPS],
    [w - x_positions[k] - chips_w[ k] |k in CHIPS]));
```

Code 2: Symmetry breaking

Some other possible symmetries can be detected, specially in particular cases (i.e. square solutions), but often the effort to remove them is not worth.

Rotation case

Adding the possibility to rotate any chip, the problem grows significantly and it must be managed as efficiently as possible. We decided to introduce two new arrays for chips_w and chips_h in order to handle a rotation as a swap between the width and the height of a chip. The new arrays chips_w_true and chips_h_true are kept consistent with respect to the real chips' shapes using a boolean array. The constraint (Code 3) that achieves this consistency is shown in the following code fragment.

```
array[CHIPS] of var 0..max(max(chips_w), max(chips_h)):chips_w_true;
array[CHIPS] of var 0..max(max(chips_w), max(chips_h)):chips_h_true;

constraint forall(i in CHIPS)
((rotations[i] /\ chips_w_true[i] == chips_h[i] /\
chips_h_true[i] == chips_w[i])
xor (not(rotations[i]) /\ chips_w_true[i] == chips_w[i]
/\ chips_h_true[i] == chips_h[i]));
```

Code 3: Rotation

Search Strategy & Solver

The search strategy is a crucial part of Constraint Programming. Making many tries, we found that by searching first on y then on x and finally on h we reached the peak performance of this model. In order to implement it, we used search annotation as shown in (Code 4). As we can see, the solver chooses always the smallest value of the domain (`indomain_min`) in all the three cases. For the positions coordinates we used the `first_fail` strategy which imposes to pick the variable with the smallest domain. For h we noticed that there are no significant differences among several variable-selection policies because there are no points choice since `h` is a single variable.

```
      solve :: seq_search([
          int_search(y_positions, first_fail, indomain_min),
          int_search(x_positions, first_fail, indomain_min),
          int_search([h], smallest, indomain_min)])
          minimize h;
```

Code 4: Search

This construction for the solver settings let us to fully exploit the Chuffed solver capabilities. According to the documentation, the search annotations (enabled with Free Search command) results in a very good implementation. Empirically we noticed that, although Chuffed does not support effectively restart strategies yet, Gecode in every case shows longer time of computation even on simple problems. Lacking of an important resource like restarting with different seed during a single run, Chuffed precludes some interesting possible improvements.
Running different Solver and search heuristics we found that these settings shows the best performances even when rotation is allowed.

PERFORMANCES ANALYSIS

We collected data from all the 40 instances both in case of chips rotation allowed and not allowed. We set the cut-off time to 300 seconds, noticing as expected better performance in the base case. Nevertheless, the two cases have been succeeded in solving almost all the instances. Finally we can point out that there is a small fixed cost due to the starting time of MiniZinc, since it has be called from external Python code.
The following graph gives an overall idea of the general trend.

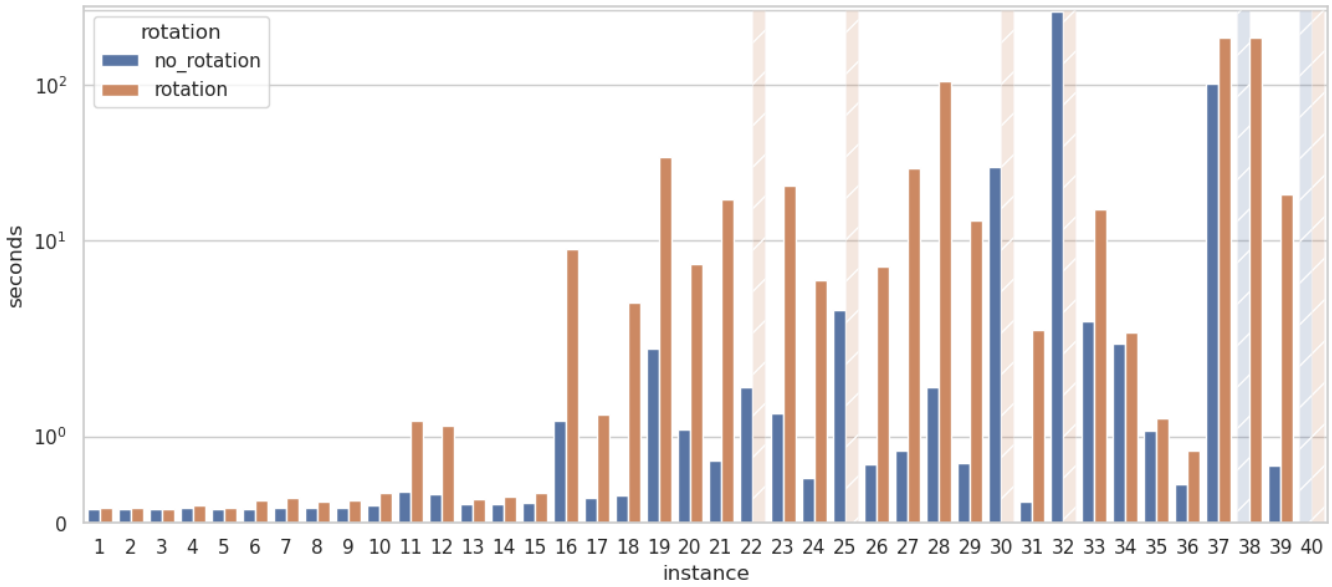

Figure 5: CP Solving Times

The collected data can be analysed more accurately in the following table (at 3 significant digits):

	CP	
Instance	No Rotation	Rotation
1	.161	.173
2	.163	.171
3	.164	.168
4	.170	.200
5	.168	.179
6	.168	.255
7	.172	.284
8	.180	.254
9	.177	.255
10	.200	.343
11	.369	1.17
12	.336	1.12
13	.221	.277
14	.222	.306
15	.228	.341
16	1.17	8.85
17	.296	1.25
18	.315	4.03
19	2.03	34.3
20	1.07	7.02
21	.718	18.4
22	1.57	-
23	1.26	22.3
24	.525	5.53
25	3.58	-
26	.678	6.83
27	.841	28.9
28	1.57	104
29	.692	13.4
30	29.4	-
31	.247	2.65
32	291	-
33	3.02	15.8
34	2.20	2.58
35	1.06	1.20
36	.451	.839
37	102	-
38	-	-
39	.669	19.7
40	-	-

Table 1: CP solving Time Results

Conclusions

Constraint Programming has shown very good possibility of customization both in model formulation and Solver settings. We believe that random restart strategies may greatly improve the overall behaviour letting the search process to backtrack avoiding frequent mistakes (for example using `dom_w_deg`). Longer time of execution would obviously lead to solve harder instances but it would not be interesting since nothing about the already solved ones would change in absence of random restarting policies, then we could not elaborate nothing more about CP. Finally it is not clear how to define a useful dual model which may actually improve the resolution; our SAT proposal can be easily translated in MiniZinc but the big growth of the number of variables make every effort pointless.
The reason of the bad MiniZinc adoption of illustrated SAT model is given by the fact that h would be replaced by max_h (and $max\_h >> h$) due of MiniZinc policy for iterating over variables. The SAT alternative model and other interesting comparison are included in the main Report (Github folder).