Very Large Scale Integration problem

Combinatorial and Decision Making
Optimization project

Farinola Francesco - francesco.farinola@studio.unibo.it

March 2022

Contents

1 Problem description

The trend of integrating circuits onto silicon chips is referred to as VLSI (Very Large Scale Integration). The smartphone is a typical example. The modern trend of lowering transistor sizes has pushed the integration of more and more functionalities of smartphone circuitry into a single silicon die (i.e. plate), allowing engineers to put more and more transistors into the same area of silicon. This allowed the contemporary cellphone to evolve into a formidable tool, shrinking from the size of a big brick-sized machine to a device small enough to easily carry in a pocket or handbag, with a video camera, touchscreen, and other advanced functions.

We have been tasked with designing the VLSI of the circuits that define your electrical device: Given a fixed-width plate and a list of rectangular circuits, determine how to arrange them on the plate so that the finished device's length is reduced (increasing portability). Each circuit must be positioned in a fixed orientation with regard to the others in order for the device to function correctly, hence it cannot be rotated.

2 Hardware and Software Specifications

The machine on which the experiments were conducted has:

- Intel Core i5-9300H
- 24 GB of RAM
- NVIDIA GeForce GTX 1660Ti

The project was developed on Windows 11 Pro using the PyCharm IDE 2021.3 with Python version 3.8.10. The following libraries have been used:

- *minizinc == 0.5.0*
- *z3-solver == 4.8.14.0*
- *matplotlib==3.5.1*
- *numpy==1.21.4*

3 Constraint Programming approach

3.1 Variables

- **Input variables**

  Namely, the variables used to obtain inputs from any instance *.txt* file:

  - $plate_w$: an integer to store the plate width
  - n: an integer to store the number of circuits
  - $CIRCUITS$: a set of integers to enumerate the circuits
  - $width$ and $height$: two arrays of integers to store widths and heights of each circuit

```
int: plate_w; %Width of the plate
int: n; % Number of total circuits
set of int: CIRCUITS = 1..n; % Enumerate circuits
array[CIRCUITS] of int: width;  % Widths of the circuits
array[CIRCUITS] of int: height; % Heights of the circuits
```

- **Variable to minimize**

  Namely, the height of the plate which will be the objective function to minimize. In order to reduce its domain, this variable has been defined between lower and upper bounds which are respectively the max height of a single circuit and the sum of all the circuits' height.

$$\max(height) \leq plate_h \leq \sum_i height_i$$

```
var max(height)..sum(height): plate_h;
```

- **Output variables**

  Namely, the solution coordinates c_x and c_y for each circuit. Also the domain of this variables have been reduced since:

$$c_x + \min(width) \leq plate_w$$

$$c_y + \min(height) \leq \sum_i height_i$$

```
array[CIRCUITS] of var 0..plate_w - min(width): c_x;
array[CIRCUITS] of var 0..sum(height) - min(height): c_y;
```

3.2 Constraints

- **Implied constraints**

  As suggested by the professor in the project specification, if we draw a horizontal line and sum the horizontal sides of the traversed circuits, the sum can be at most $plate_w$. A similar property holds if we sum the vertical sides, the sum can be at most $plate_h$:

$$\forall i \in \{1, \ldots, n\} \quad c_{x,i} + width_i \leq plate_w \wedge$$
$$c_{y,i} + height_i \leq plate_h$$

```
constraint forall(i in CIRCUITS) (c_x[i] + width[i] <= plate_w /\
                                  c_y[i] + height[i]<= plate_h );
```

- **Global constraints:**

  MiniZinc comes with a library of global constraints that may be used to define models. These constraints are high-level modeling abstractions for which many solvers construct specialized, efficient inference methods. We will make use of two different type of global constraints, as suggested by prof. Peter J. Stuckey in [1]:

  - ***diffn***: usually occurs in scheduling problems but it can be adapted to the current problem. This constraint ensures no overlapping over two-dimensional rectangles. In particular, given two circuits at positions $(c_{x,i}, c_{y,i})$ with dimensions $(width_i, height_i)$ these do not overlap.

$$diffn(c_x, c_y, width, height)$$

  - ***cumulative***: as said in [2], can be used as a necessary but not sufficient condition for the two-dimensional case of the *diffn* constraint. A first (respectively second) *cumulative* constraint is obtained by forgetting the y-coordinate (respectively the x-coordinate) of the origin of each rectangle occurring in a *diffn*

constraint. This constraints will not ensure the packing. They are just redundant constraints to improve propagation, hence solving.

$$cumulative(c_x, width, height, plate_h)$$

$$cumulative(c_y, height, width, plate_w)$$

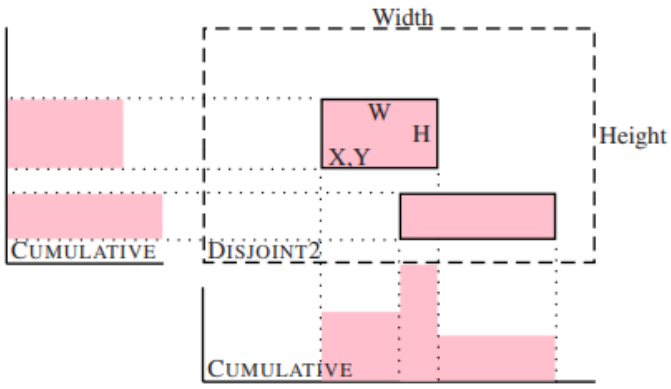

Figure 1: Two-dimensional case for rectangles placement using cumulative and diffn global constraints. Credit [3]

```
constraint diffn(c_x,c_y,width,height);
constraint cumulative(c_x, width, height, plate_h);
constraint cumulative(c_y, height, width, plate_w);
```

3.3 Symmetry Breaking

- **Circuits of the same size**: these kind of circuits are completely interchangeable since they have same width and height. So we define an array for each circuit to gather all the identical circuits and then we set an ordering between them, so they cannot be exchanged and the search space is pruned.

$$\forall i \in \{1, \ldots, n\}$$
$$\text{Let } I = \{j | j \in \{i+1, \ldots, n\}$$
$$\text{where } width_i = width_j \wedge height_i = height_j\}$$
$$\forall j \in I, \quad lex\_less([c_{y,j-1}, c_{x,j-1}], [c_{y,j}, c_{x,j}])$$

```
constraint symmetry_breaking_constraint(forall (i in CIRCUITS) (
let { array[int] of int: identical = [j | j in i+1..n
                                       where width[j] = width[i] /\
                                             height[j] = height[i]]
} in if length(identical)>1 /\ min(identical)=i then
  forall (j in 1..length(identical)-1)
  (lex_less([c_y[identical[j-1]], c_x[identical[j-1]]],
            [c_y[identical[j]],   c_x[identical[j]] ]))
  else true endif));
```

- **Force biggest circuit to be on the bottom-left corner wrt the second one:** by looking at [4], we know that placing bigger circuits is more constraining that placing smaller ones since they consume more area. So it might be better to order the circuits by decreasing area and force the biggest one to be on the bottom-left corner:

$$\forall i \in \{1, \ldots, n\}$$
$$\text{Let } O = \text{sorted}(-height_i * width_i)$$
$$lex\_less([c_{y,O1}, c_{x,O1}], [c_{y,O2}, c_{x,O2}])$$

```
array[CIRCUITS] of int : ordered_circuits =
sort_by(CIRCUITS, [ - height[i]*width[i] | i in CIRCUITS]);

constraint symmetry_breaking_constraint(
  let { int: i=ordered_circuits[1], int: j=ordered_circuits[2] }
  in lex_less([c_y[i],c_x[i]], [c_y[j],c_x[j]]));
```

- **Dominance constraint:** [5][3] this is a special case of symmetry breaking since it removes feasible but dominated assignments. In general, a rectangle placement is dominated if it leaves a gap in which all rectangles that can individually fit can also be packed together in the gap without protruding from it. With this constraint, we force circuits to be partially adjacent to each other or to the borders:

$$\forall i \in \{1, \ldots, n\}, \quad c_{x,i} \in \{0\} \cup W \quad \wedge \quad c_{y,i} \in \{0\} \cup H$$
$$\text{where } W = \{width_j + c_{x,j} | \forall j \in \{1, \ldots, n\}, j \neq i\}$$
$$\text{and} \quad H = \{height_j + c_{y,j} | \forall j \in \{1, \ldots, n\}, j \neq i\}$$

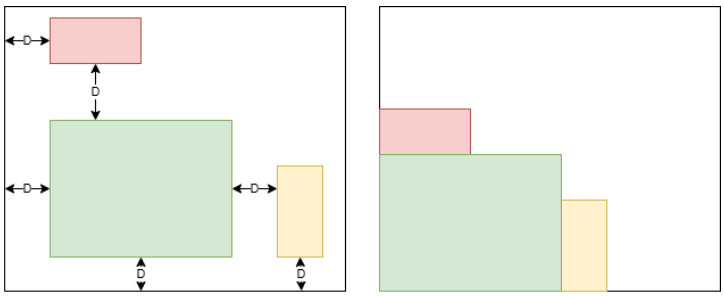

Figure 2: All the distances D (*gaps*) are not allowed since with this constraint each circuit has to be placed at the border or adjacent to another circuit. Figure A: pruned positioning. Figure B: positioning respecting the constraint . (*Figures made by me* - Note that rectangles are not forced to be at the same time near the border and adjacent to other rectangles but only one has to be satisfied)

```
constraint forall(i in CIRCUITS)(
   member([0] ++ [width[j] + c_x[j] |j in CIRCUITS where j != i],
          c_x[i]) /\
   member([0] ++ [height[j] + c_y[j] |j in CIRCUITS where j != i],
          c_y[i]));
```

3.4 Rotation

The circuits cannot be rotated due to the problem constraints. This indicates that a $n\ x\ m$ circuit cannot be positioned in the silicon plate as a $m\ x\ n$ circuit. Let's consider a general example in which rotation is permitted.

3.4.1 Additional variables

In order to allow rotation, 3 additional variables have been defined:

- *rotation:* Boolean array indicating whether the *i-th* circuit has to be rotated (True) on not (False)
- $real_w$*:* Array of integers indicating the actual width of a circuit - if the corresponding *i-th* value in the array *rotation* is True then we switch

width with height.

$$\forall i \in \{1, \ldots, n\} \quad real_w = \begin{cases} height_i & \text{if } rotation_i = T \\ width_i & \text{otherwise} \end{cases}$$

- $real_h$*:* Array of integers indicating the actual height of a circuit - if the corresponding *i-th* value in the array *rotation* is True then we switch height with width.

$$\forall i \in \{1, \ldots, n\} \quad real_h = \begin{cases} width_i & \text{if } rotation_i = T \\ height_i & \text{otherwise} \end{cases}$$

```
array[CIRCUITS] of var bool : rotation;
array[CIRCUITS] of var int: real_width =
        [if rotation[i] then height[i] else width[i] endif | i in CIRCUITS];
array[CIRCUITS] of var int: real_height =
        [if rotation[i] then width[i] else height[i] endif | i in CIRCUITS];
```

3.4.2 Constraints

In order for the model to work correctly we have to make some modifications on the previously defined **suggested constraints** and **global constraints** in which we simply substitute the original width (height) with the real width (real height):

```
constraint forall(i in CIRCUITS) (c_x[i] + real_width[i] <= plate_w /\
                                  c_y[i] + real_height[i] <= plate_h);
constraint diffn(c_x,c_y,real_width,real_height);
constraint cumulative(c_x, real_width, real_height, plate_h);
constraint cumulative(c_y, real_height, real_width, plate_w);
```

If a circuit can be rotated and its height is greater than the maximum chip width, it is obvious that it cannot be rotated. This may be redundant but it is better to define it anyway:

$$\forall i \in \{1, \ldots, n\} \quad height_i > plate_w \implies rotation_i = F$$

3.4.3 Additional symmetry breaking

To prune the search space and avoid symmetric assignments, i have defined two more constraints as circuits now can be rotated and there are many particular cases.

- **Do not rotate squares:** circuits with same width and height if rotated will have always the same dimension so we prune this assignment:

$$\forall i \in \{1, \ldots, n\} \quad width_i = height_i \implies rotation_i = F$$

- **Do not rotate both circuits if their inverted sizes are the same:** this is more complex to explain with words so we make an example. If we have two circuits of size $3x2$ and $2x3$, and we rotate both of them they are interchangeable so we prune this possibility:

$$\begin{aligned}
&\forall i \in \{1, \ldots, n\}, \quad \forall j \in \{1, \ldots, n \mid j \neq i\}, \\
&width_i = height_j \land height_i = width_j \implies \\
&rotation_i = F \land rotation_j = F \quad \lor \\
&rotation_i = T \land rotation_j = F \quad \lor \\
&rotation_i = F \land rotation_j = T
\end{aligned}$$

3.5 Search

MiniZinc does not have a default setting for how we wish to look for solutions. This leaves the search entirely up to the underlying solver. However, in other cases, notably for combinatorial integer issues, we may want to describe how the search should be conducted. This necessitates that we provide a search strategy to the solver.

I have tried different strategies, with the help of some literature [3] using different search strategies for both variables and values.

I first tried doing an integer search over the variable to minimize $plate_h$. The idea is that starting the search from the lower bound of the objective function would lead us directly to the optimal solution. So we search from variables domain starting from the minimum and test different variable selection strategies:

```
solve
%  :: int_search([plate_h], input_order, indomain_min)
   :: int_search([plate_h], first_fail, indomain_min)
%  :: int_search([plate_h], dom_w_deg, indomain_min)
  minimize plate_h;
```

The best results obtained were over the *first_fail* variable selection strategy, which was able to solve with optimal solution many instances but not the most difficult ones. **Restart** in these strategies cannot be applied since

the search is deterministic and a restart would lead to exploring the same exactly search space [6].

So, to apply restart and add some randomization to the search space, i changed the value selection strategy to *indomain_random* and tried different restart strategies to check whether difficult instances could be solved:

```
solve
% *** Search integer values with different strategies ***
  :: int_search([plate_h], first_fail, indomain_min)
%  :: restart_constant(500)
%  :: restart_linear(200)
  :: restart_geometric(1.5, 700)
%  :: restart_luby(500)
  minimize plate_h;
```

According to [3], an alternative way would be to assign all of the x variables first, followed by any of the y variables. The advantage is that once the x values are fixed, there are few, if any, options for the y values, lowering the effective depth of the search tree to n. Unfortunately, if this strategy fails, it will result in extensive backtracking (thrashing), making finding a solution all but impossible. Anyway, it is worth giving it a try:

```
solve
  :: int_search(c_x, first_fail, indomain_min)
  :: int_search(c_y, first_fail, indomain_min)
  minimize plate_h;
```

3.6 Results

Remember that we used a timeout of 300s, so each execution that exceeds this time interval got aborted.

We first show results obtained using the value search strategy `indomain_min` over `plate_h`. With this kind of search, the first solutions found is the optimal one, so we choose the variable search strategy with the minimum mean time for solving the instances.

By using this kind of search, we were able to solve 31 out of 40 instances with the optimal solution using all strategies (Figure 3) but with different mean times:

- `input_order`: 17,918 s
- `first_fail`: 16,592 s
- `dom_w_deg`: 17,169 s

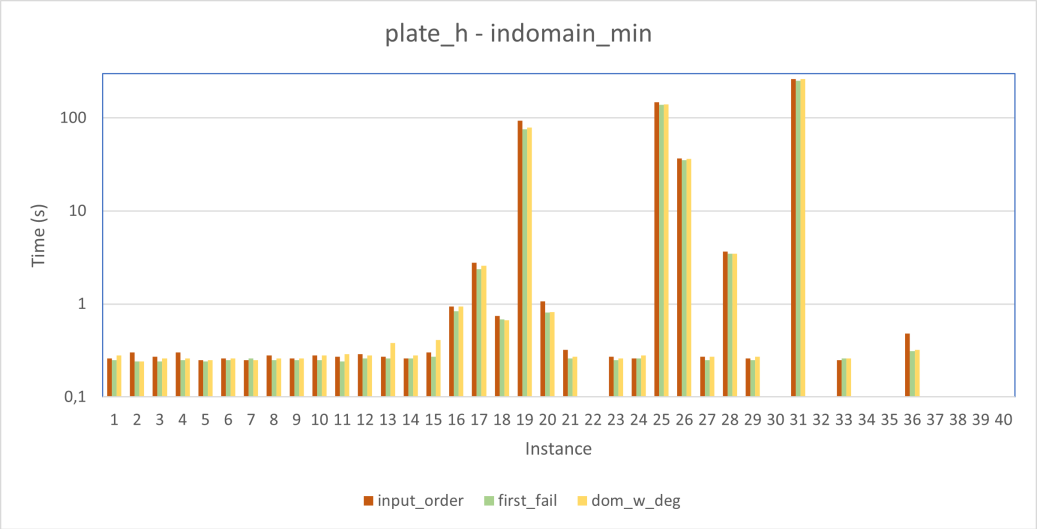

Figure 3: Histogram of execution times for solving each instance with different variable selection strategies using an integer search over plate_h and value selection strategy indomain_min

Next, we show the results of searches done by applying some randomization. In particular, we used the `first_fail` variable selection strategy and `indomain_random` as value selection strategy. In Table 1 we can see all the instances that did not get solved with optimality in any of the restart strategy used and their respective value for the objective function (plate_h).

From the table, we can tell that using `restart_geometric(1.5,700)` we got the best results, solving all the instances, 30 out of 40 with an optimal solution within a time limit of 300 seconds and when finding a non-optimal solution we get the lowest objective function value except for the Instance 40 which is the most difficult to solve.

In conclusion, with randomization and geometric restart we are able to solve $\sim$ 36 with decent results, while when solving without randomization and restart we were able to solve only 31 with an optimal result and the rest did not get solved.

	No restart		Constant		Linear		Geometric		Luby	
	Opt	Plate_h	Opt	Plate_h	Opt	Plate_h	Opt	Plate_h	Opt	Plate_h
11	Yes	18	No	20	No	19	Yes	18	No	19
16	No	25	No	24	No	24	No	24	No	24
18	-	-	No	28	Yes	25	Yes	25	Yes	25
19	Yes	26	No	27	No	27	Yes	26	No	27
21	Yes	28	No	31	No	29	Yes	28	Yes	28
22	No	30	No	31	No	31	No	31	No	31
25	-	-	Yes	32	Yes	32	No	33	Yes	33
26	No	158	Yes	33	Yes	33	Yes	33	Yes	33
27	No	147	Yes	34	Yes	34	Yes	34	Yes	34
29	No	155	Yes	36	Yes	36	Yes	36	Yes	36
30	No	67	No	41	No	38	No	38	No	38
31	No	54	Yes	38	Yes	38	Yes	38	Yes	38
32	-	-	No	40	No	40	No	40	No	40
34	No	41	No	41	No	41	No	41	No	41
35	No	41	No	41	No	41	No	41	No	41
37	-	-	No	63	No	63	Yes	60	No	62
38	-	-	No	64	No	62	No	61	No	61
39	-	-	No	62	No	62	No	61	No	62
40	-	-	No	415	No	415	No	501	No	415

Table 1: List of instances not solved or solved with non-optimality using First Fail and different Restart strategies

When solving with the **X then Y strategy**, as expected we did not get good results. Only 17 instances got solved with an optimal solution while the other 23 got a bad value for the objective function (Figure 4).

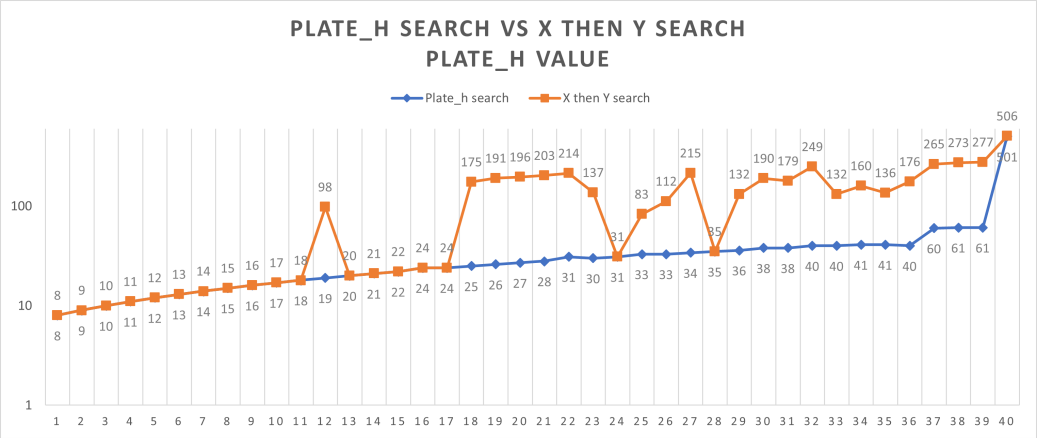

Figure 4: Scatter plot of the plate_h values obtained with the best search method (plate_h, first_fail, indomain_random and restart geometric) vs the X then Y strategy

Finally, the following results refer to the best search strategy found (plate_h, first_fail, indomain_random and restart geometric) applied to the model where rotation is allowed (Figure 5).

When rotation is allowed the search space greatly increase and this is why we are not able to solve most of the instances (only 18 out of 40).

We can conclude that the best combination of model/search strategy which was able to solve optimally 30 instances and get good results for the others (except 37, 38, 39, 40) uses the no rotation model and performs an integer search over plate_h using first_fail, indomain_random and geometric restart with parameters 1.5 and 700.

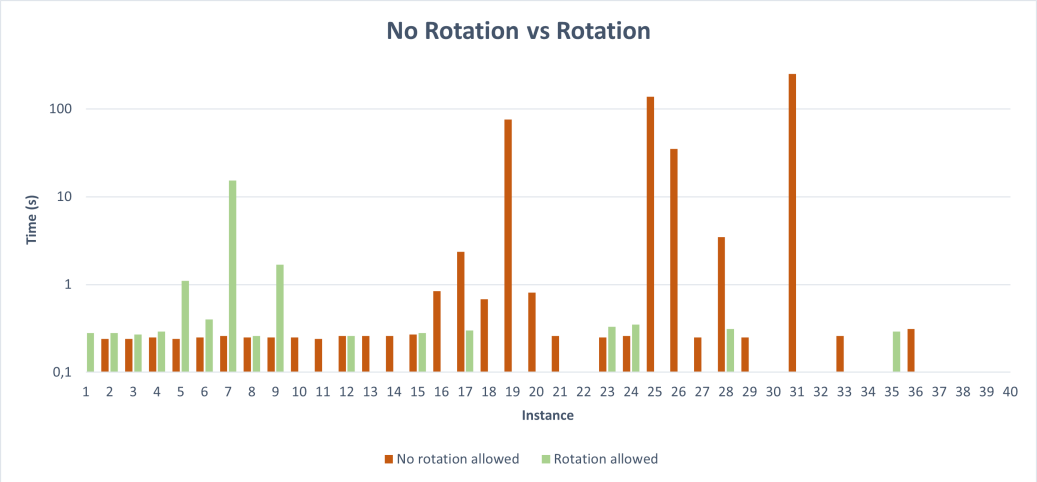

Figure 5: Histogram of execution times for solving each instance when rotation is allowed and when it is not allowed

4 SMT

Satisfiability modulo theories (SMT) is the problem of determining if a mathematical formula is satisfiable in computer science and mathematical logic. It extends the Boolean satisfiability problem (SAT) to more sophisticated formulae including real numbers, integers, and/or data structures like lists, arrays, bit vectors, and strings. The term comes from the fact that these statements are interpreted inside ("modulo") a particular formal theory in first-order logic with equality (often disallowing quantifiers). SMT solvers like Z3 are programs that attempt to solve the SMT problem for a certain set of inputs.

4.1 Variables

The model encoding is similar to the CP model.

As input, we will have:

- an integer `plate_w` indicating the width of the plate
- an integer `n` indicating the number of circuits
- two arrays of integers `width` and `height` indicating the widths and height of the circuits

As output, we will have:

- two `IntVector` - array of integers - `x` and `y` indicating the coordinates (x, y) of each circuit
- an `Int` - integer - `plate_h` indicating the height of the plate, which is the variable to minimize.

We will also have an additional variable `area` indicating the indexes of the circuits ordered by decreasing area. We will use this variable to constrain the biggest circuit to be at the bottom-left of the second biggest one.

4.2 Constraints

We will try to define using first-order logic most of the constraints used in the CP model.

- **Bound coordinates:** With this constraint we force coordinates to get values between a lower (0) and upper bound (maximum width/height):

$$\bigwedge_{i=1}^{n} \quad (x_i \geq 0 \quad \wedge \quad x_i + width_i \leq plate_w) \tag{1}$$

$$\bigwedge_{i=1}^{n} \quad (y_i \geq 0 \quad \wedge \quad y_i + height_i \leq plate_h) \tag{2}$$

- **Diffn - no overlap:** Thanks to [7], i was able to decompose the global constraint Diffn from MiniZinc and adapt it to my use case:

$$\begin{aligned} \bigwedge_{i=1}^{n} \quad \bigwedge_{j=i+1}^{n} \quad & (x_i + width_i \leq x_j \quad \vee \\ & y_i + height_i \leq y_j \quad \vee \\ & x_j + width_j \leq x_i \quad \vee \\ & y_j + height_j \leq y_i) \end{aligned} \tag{3}$$

- **Force adjacency:** with this constraint we will force circuits to be adjacent to each other OR to the border. To do so, we define first a set which includes the coordinate 0 and the coordinates that may be occupied by other rectangles and then force to satisfy at least one of those. This is similar to the *cumulative* global constraint with the concept but we do not force a circuit to be on the coordinate 0. I tried to adapt a version of the *cumulative* constraint with the help of [7] but this did now work properly as some instances would be UNSAT. It has been implemented as:

$$\bigwedge_{i=1}^{n} \quad (\bigvee_{j=1}^{|W|} \quad (x_i = W_{i,j}))$$

$$\text{where } \forall i \in \{1, \ldots, n\}, W_i = \{0\} \cup \{x_j + width_j | \forall j \in \{1, \ldots, n\}, j \neq i\} \tag{4}$$

$$\bigwedge_{i=1}^{n} \quad (\bigvee_{j=1}^{|H|} \quad (y_i = H_{i,j}))$$

$$\text{where } \forall i \in \{1, \ldots, n\}, H_i = \{0\} \cup \{y_j + height_j | \forall j \in \{1, \ldots, n\}, j \neq i\} \tag{5}$$

- **Symmetry breaking - Same dimension circuits**: we want circuits with the same dimension to not be interchangeable so we define:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} (width_i = width_j \wedge height_i = height_j) \implies ((x_i < x_j \wedge y_i \leq y_j) \quad \vee \quad (x_i \leq x_j \wedge y_i < y_j)) \tag{6}$$

- **Symmetry breaking - Place biggest circuit before**: we want to impose lexicograhic order between the first biggest circuit and the second biggest one by placing the former always at the bottom-left w.r.t the latter. Unfortunately, we cannot force the biggest circuit to be placed at coordinate (0,0) as this somehow makes some instances UNSAT. To do this, we first define an array which contains the indexes of the circuits ordered in descending order by area and then, define the follow:

$$\begin{aligned} &((x_{area_1} < x_{area_2}) \wedge (y_{area_1} \leq y_{area_2})) \quad \vee \\ &((x_{area_1} \leq x_{area_2}) \wedge (y_{area_1} < y_{area_2})) \end{aligned} \tag{7}$$

4.3 Rotation

When allowing rotation, we need to add variables, constraints and make some modifications to our model:

Additional variables:

- a boolean array (`BoolVector`) `rotation` containing bool values indicating whether the circuit in *i-th* position should be rotated or not
- an array of integers `w` indicating the actual width of the *i-th* circuit:

$$\forall i \in \{1, \ldots, n\}, \quad w_i = \begin{cases} height_i & \text{if } rotation_i = True \\ width_i & \text{otherwise} \end{cases}$$

- an array of integers `h` indicating the actual height of the *i-th* circuit:

$$\forall i \in \{1, \ldots, n\}, \quad h_i = \begin{cases} width_i & \text{if } rotation_i = True \\ height_i & \text{otherwise} \end{cases}$$

Additional constraints:

- **Do not rotate squares**: for breaking symmetry we do not want squared circuits to be rotated as they will have the same size:

$$\bigwedge_{i=1}^{n} \; (width_i = height_i) \implies rotation_i = False \tag{8}$$

- **Do not rotate both circuits**: for breaking symmetry of two circuits whose sizes are inverted and, by rotating them, they would be interchangeable:

$$\begin{aligned} \bigwedge_{i=1}^{n} \; \bigwedge_{j=1, j\neq i}^{n} & (width_i = height_j) \wedge (height_i = width_j) \implies \\ & ((rotation_i = False \wedge rotation_j = False) \vee \\ & (rotation_i = True \wedge rotation_j = False) \vee \\ & (rotation_i = False \wedge rotation_j = True)) \end{aligned} \tag{9}$$

4.4 Search & Results

Z3Py provides the standard search method with a class `Optimize` which allows us to get an assignment for each variable such that it minimize or maximize the objective function through the respective method

`Optimize.minimize(obj)` or `Optimize.maximize(obj)`.

For our specific problem, we want to minimize the plate height so we will call the method `Optimize.minimize(plate_h)`.

We can add assertions to our model by calling the method `Optimize.add(<list of assertions>)` and then check if the model is SAT, UNSAT or unknown (aborting optimal solution finding if time out is exceeded) with the method `Optimize.check()`.

If an optimal solution is found (`check()` method returns SAT) then we can get the assignments of the solution through the `model()` and `evaluate()` methods.

In Figure 7 we can see the elapsed time for solved instances with SMT without and with allowed rotation. In the original model we are able to solve with an optimal solution 22 instances out of 40 while with rotation we were able to solve only the first 10 instances.

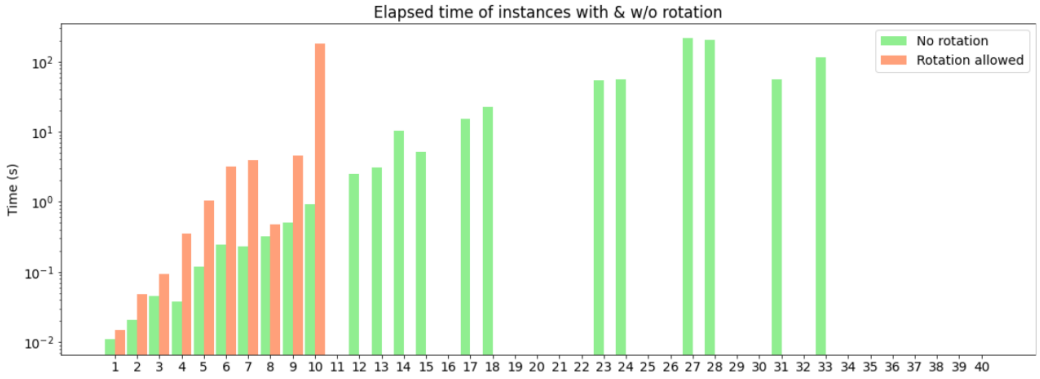

Figure 6: SMT results with (Green) and without (Red) rotation allowed.

5 SAT

SAT refers to the Boolean Satisfiability problem. In other words, it examines if the variables in a given Boolean formula can be consistently substituted by the values True or False so that the formula evaluates to True. If this is true, the formula is said to be SATisfiable.

The substantial difference between SMT and SAT is that we cannot define an objective function to minimize or maximize but we can check only whether a formula is SAT or not. Furthermore, we cannot use quantifiers, variables constants or functions. SAT encodings are less expressive than SMTs but, nowadays, SAT solvers can solve formulas with millions of variables and clauses efficiently.

The SAT encoding i'm going to present is an adaption of the work done in [8] by *Takehide Soh et al* for the 2D-SPP with some changes.

5.1 Variables

The following values are given as input:

- $plate_w$: the maximum plate width
- n : the number of circuits
- $widths$: the width of each circuit
- $heights$: the height of each circuit

Let c_i and $c_j \in R(i \neq j)$ be two circuits and e and f any integers. Then the SAT encoding uses four kinds of atoms:

- $lr_{i,j}$ is True if circuit c_i is placed at the left to the circuit c_j, False otherwise
- $ud_{i,j}$ is True if circuit c_i is placed at the downside w.r.t. to the circuit c_j, False otherwise
- $x_{i,e}$ is True if circuit c_i is placed at less or equal to e
- $y_{i,f}$ is True if circuit c_i is placed ar less or equal to f

The value of $x_{i,e}$ is True when the circuit i occupies the position e in the plate grid. So by getting the lowest True value we get the bottom-left corner coordinate x of the circuit i. Same goes for the y coordinate.

When translating the encoding to the solution, we scan the x and y arrays from the lower bound and when we find a True value we break the loop for the current circuit and get the coordinate position.

Furthermore, when searching for a solution using a SAT encoding, we want to find as first the **optimal solution**. In order to do so, we need to fix the value of the plate height ($plate_h$) and find a feasible assignment to all the atoms we defined before and which satisfies all the constraints we are going to define.

To find the optimal value of $plate_h$, we calculate the **sum of the areas of circuits** and **divide it by the plate width**. By getting the **upper bound** of this value we can approximate the optimal value of the plate height.

$$\text{optimal } plate_h = \lceil \frac{1}{plate_w} \sum_i width_i * height_i \rceil$$

5.2 Constraints

The constraints defined for this problem are encoded as axioms that are added to the solver object and are the following:

- **Reduce domain**: namely, we do not want circuits to take positions that overlap with borders, so we set as True all the atoms that would exceed the borders.

$$\begin{aligned} &\forall i \in \{1, \ldots, n\}, \\ &\forall e \in \{plate_w - width_i, \ldots, plate_w\} \qquad x_{i,e} = True \\ &\forall f \in \{plate_h - height_i, \ldots, plate_h\} \qquad y_{i,f} = True \end{aligned}$$

- **No overlapping**: In order to ensure no overlapping and the order encoding we need three types of axioms:
  - **2-literal clauses**: for each circuit c_i and predefined integer e and f such that $0 \leq e < plate_w - width_i$ and $0 \leq f < plate_h - height_i$ we have:

$$\neg x_{i,e} \vee x_{i,e+1}$$

$$\neg y_{i,f} \vee y_{i,f+1}$$

– **3-literal clauses**: for each circuits c_i and c_j with $i < j$ and predefined integer e and f such that $0 \leq e < plate\_w - width_i$ and $0 \leq f < plate\_h - height_i$:

$$\neg lr_{i,j} \vee x_{i,e} \vee \neg x_{j,e+width_i}$$
$$\neg lr_{j,i} \vee x_{j,e} \vee \neg x_{i,e+width_j}$$
$$\neg ud_{i,j} \vee y_{i,f} \vee \neg y_{j,f+height_i}$$
$$\neg ud_{j,i} \vee y_{j,f} \vee \neg y_{i,f+height_j}$$

To ensure no overlapping and assignments of atoms it was added four more clauses to each of the previous with inverted values of i and j, and are:

$$\neg lr_{i,j} \vee \neg x_{j,width_i-1}$$
$$\neg lr_{i,j} \vee x_{i,plate_w-width_i-1}$$
$$\neg ud_{i,j} \vee \neg y_{j,height_i-1}$$
$$\neg ud_{i,j} \vee y_{i,plate_h-height_i-1}$$

where $plate_w - width_i - 1$ and $plate_h - height_i - 1$ represent the maximum value that the coordinate x or y can assume if the first circuit is at the right side or upside w.r.t. the second circuit.

– **4-literal clauses**: for each circuits c_i and c_j with $i < j$ we have:

$$lr_{i,j} \vee lr_{j,i} \vee ud_{i,j} \vee ud_{j,i}$$

so for each couple of circuits only one of these is True to ensure an ordering.

The following constraints guaranteed symmetry breaking and prune the search space:

- **Largest circuit**: With this constraint, we lookup for the circuit with the maximum area c_{max} and then force the domain of the other circuits to be at right side or upside w.r.t to it.

If for each other circuit c_i, $width_i > \lfloor \frac{plate_w - width_{max}}{2} \rfloor$ we set the atom:

$$\neg lr_{i,max}$$

If for each other circuit c_i, $height_i > \lfloor \frac{plate_h - height_{max}}{2} \rfloor$ we set the atom:

$$\neg ud_{i,max}$$

- **Large circuits** When we have large circuits, if the sum of their sizes exceeds the maximum size of the plate we can force the direction variable to be false in both circuits.

  For each couple of circuits c_i, c_j if $width_i + width_j > plate_w$ we set:

  $$\neg lr_{i,j}$$
  $$\neg lr_{j,i}$$

  For each couple of circuits c_i, c_j if $height_i + height_j > plate_h$ we set:

  $$\neg ud_{i,j}$$
  $$\neg ud_{j,i}$$

- **Same sized circuits**: Circuits c_i, c_j with the same dimensions can fix their positional relation by adding the clauses:

  $$\neg lr_{i,j}$$

  $$lr_{i,j} \vee \neg ud_{j,i}$$

5.3 Rotation

If we allow rotation of the circuits by 90 degrees, we need an auxiliary variable r of boolean variables and make some modifications to the model.

First, we cannot use symmetry breaking constraints since they may interfere with rotation and make the problem UNSAT, so we must remove them.

Then, when allowing rotation, we need to modify the first constraint which reduces the domain of the x and y related atoms. Since the circuit can

be rotated, if its height is less than its width, the circuit can assume bigger values of x. The same goes with the y variable.

Also, we would need to modify the overlapping constraints and allow more combinations which would greatly increase the search space and so, the search time.

I tried implementing those constraints but did not have success since the complexity of overlapping formulas greatly increases. Finally, all these modifications would be worth only to solve the last instance (40), in fact without allowing rotation the encoding of the problem successfully solved all the instances except the last one with decent execution times. I will leave the model without allowing rotation.

5.4 Results

The SAT encoding of the problem, without allowing rotation of circuits, produced the following results:

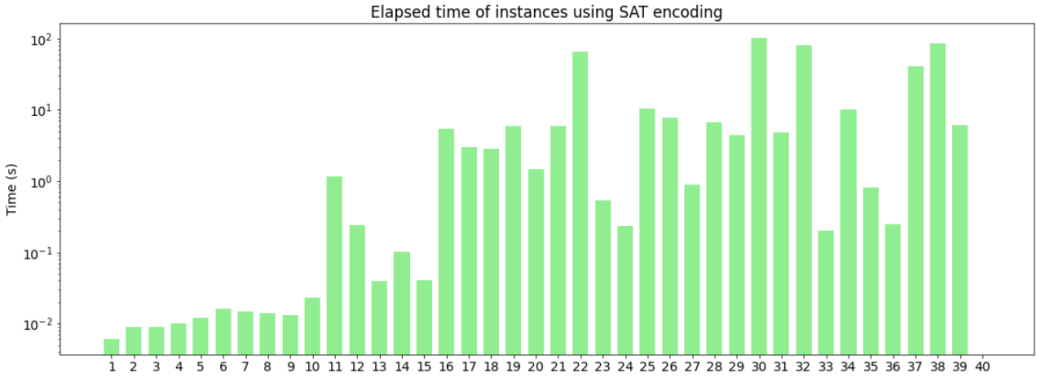

Figure 7: SAT results without rotation allowed.

With this model, we were able to solve the first 39 instances out of 40 with a mean time of execution of 11.56 seconds. Several runs where made to check whether the problem could always solve all the instances and sometimes happened that the instance 38 was not always solved.

However, despite the low expressiveness of propositional logic, this happens to be the best approach to solve the VLSI problem with smaller execution times w.r.t. the other approaches.

References

[1] Peter J. Stuckey. *Square Packing lesson - Coursera*. URL: https://www.coursera.org/lecture/advanced-modeling/2-4-1-square-packing-oXHGs?utm_source=link&utm_medium=page_share&utm_content=vlp&utm_campaign=top_button.

[2] *Global Constraint Catalog*. URL: https://sofdem.github.io/gccat/gccat/Cdiffn.html.

[3] Helmut Simonis and Barry O'Sullivan. "Search Strategies for Rectangle Packing". In: *Principles and Practice of Constraint Programming*. Ed. by Peter J. Stuckey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 52–66. ISBN: 978-3-540-85958-1.

[4] Eric Huang and Richard Korf. "New Improvements in Optimal Rectangle Packing". In: Jan. 2009, pp. 511–516.

[5] E. Huang and R. E. Korf. "Optimal Rectangle Packing: An Absolute Placement Approach". In: *Journal of Artificial Intelligence Research* 46 (Jan. 2013), pp. 47–87. ISSN: 1076-9757. DOI: 10.1613/jair.3735. URL: http://dx.doi.org/10.1613/jair.3735.

[6] *Search annotations - 2.5.4 Restart*. URL: https://www.minizinc.org/doc-2.5.5/en/mzn_search.html.

[7] *Hakank - My Z3/Z3Py page*. URL: http://hakank.org/z3/.

[8] Takehide Soh et al. "A SAT-Based Method for Solving the Two-Dimensional Strip Packing Problem". In: *Fundam. Inf.* 102.3–4 (Aug. 2010), pp. 467–487. ISSN: 0169-2968.