Optimization Project - Modelling and solving VLSI problem

Foschini Marco - marco.foschini3@studio.unibo.it,
Spurio Federico - federico.spurio@studio.unibo.it

July 2021

1 Introduction

The topic of the project is to model and solve the VLSI problem. The VLSI (Very Large Scale Integration) problem is about integrating circuits into silicon chips.
We have been asked to solve this problem with three different techniques: CP (Constraint Programming), propositional SATisfiability and its extension SMT (Satisfiability Modulo Theories).

2 Constraint Programming (CP)

2.1 Variables, constraints and objective function

The definition of the **variables** is somehow driven by the project specifications. As a matter of facts, in the VLSI instances we have to work on, the following variables are given: `max_width` (the maximum possible width of the silicon chip), `n_blocks` (the number of blocks/circuits to place) and the `heights` and the `widths` of each circuit.

```
int: n_blocks;                          %number of rectangles
set of int: BLOCKS = 1..n_blocks;

int: max_width;

array[BLOCKS] of int: height;           %array of heights
array[BLOCKS] of int: width;            %array of widths
```

We have to place inside an area with undefined height, a group of rectangles. So we have represented the position of the rectangles with the coordinates (x and y) of the bottom-left corner. Those coordinates are stored in the arrays `cornerx` and `cornery`

```
int: h = sum(height);                   %h as upper bound
int: w = sum(width);                    %w as upper bound
array[BLOCKS] of var 0..h: cornery;
array[BLOCKS] of var 0..w: cornerx;
```

The wording `0..h` and `0..w`, means that each coordinate x and y can only take values between 0 and the sum of all the heights (and correspondingly the sum of all the widths).

The fundamentals **constraints** that we have to apply such that the problem is solved are essentially two:

- Rectangle sizes must not exceed the given width of the silicon plate
- Rectangles must not overlap together

The first one can be represented as:

```
int: max_width;     %width of the silicon plate
constraint forall(b in BLOCKS)(cornerx[b]+width[b]<=max_width);
```

While the second one as:

```
constraint forall(b1,b2 in BLOCKS where b1<b2)
(cornerx[b1] + width[b1] <= cornerx[b2]  \/
 cornerx[b2] + width[b2] <= cornerx[b1]  \/
 cornery[b1] + heigth[b1] <= cornery[b2] \/
 cornery[b2] + heigth[b2] <= cornery[b1])
```

We can defined as **objective function** of the problem, the minimization of the height of the total area of the plate. In order to achieve that, we have defined the variable `makespan`, and set it as the highest point reached by the rectangles. Then, by minimizing the makespan variable, we will obtain the correct positions of the rectangles. So we will write:

```
var int: makespan = max(b in BLOCKS)(cornery[b] + height[b]);
solve minimize makespan;
```

2.2 Lower bound of height

In order to simplify the search, we can define boundaries for `makespan`. The upper bound can be simply expressed as the sum of all the heights. However, as the number of blocks increases, the search space grows. By introducing a lower bound, the execution time of many instances decreases dramatically. The lower bound, called `min_reach_h`, is computed as the summation of the areas of all rectangles divided by the width of the silicon plate, all rounded to the ceiling value:

$$min\_reach\_h = \left\lceil \frac{\sum_{i=1}^{n\_blocks} width[i] * height[i]}{max\_width} \right\rceil \tag{1}$$

Imposing a lower bound, not only shrink the search space, but helps to find the solution faster, given an appropriate search strategy. Indeed, we can impose to start the search with the lower value of `makespan` (explained in the Strategy Section 2.5)

```
var int: min_reach_h = ceil((sum(b in BLOCKS)(width[b] * height[b])
                            div max_width));
constraint makespan >= min_reach_h;
```

2.3 Global constraints

The constraints that we have defined before can be expressed by using the global one. **Global constraint** are more efficient and powerful, optimized for that particular job. To define that rectangles must not overlap we can use `diffn`, as shown above

```
constraint diffn(cornerx, cornery, width, height);
```

We can also use two `cumulative` constraint to express that rectangles must not exceed the given width and the height that we want to minimize.
The code will be:

```
constraint cumulative(cornery, height, width, max_width);
constraint cumulative(cornerx, width, height, makespan);
```

2.4 Symmetry breaking constraints

We have observed three types of symmetries present in the problem:

- Rectangles with the same width and height are interchangeable
- Rectangles that are adjacent along the x coordinate and have the same height are interchangeable
- Rectangles that are adjacent along the y coordinate and have the same width are interchangeable

By using the *lexicographical constraint* we are able to cut the search space by not considering solutions that are equivalent to others.
We will use in Minizinc the following code:

```
constraint forall(r1,r2 in BLOCKS where
                  r1 < r2 /\ width[r1] = width[r2] /\
                  height[r1] = height[r2])
                  (lex_less([cornerx[r1], cornery[r1]],
                            [cornerx[r2], cornery[r2]]));

constraint forall(r1,r2 in BLOCKS where
                  r1 < r2 /\ cornery[r1] = cornery[r2] /\
                  cornerx[r1] + width[r1] = cornerx[r2] /\
                  height[r1] = height[r2])
                  (lex_less([cornerx[r1]], [cornerx[r2]]));

constraint forall(r1,r2 in BLOCKS where
                  r1 < r2 /\ cornerx[r1] = cornerx[r2] /\
                  cornery[r1] + height[r1] = cornery[r2] /\
                  width[r1] = width[r2])
                  (lex_less([cornery[r1]], [cornery[r2]]));
```

2.5 Search strategy

To increase the performance of our algorithm, we added the following constraint:

```
array[BLOCKS] of var bool: R;

constraint forall(b in BLOCKS)(R[b] <-> width[b] > max_width / 2);

constraint forall(b in BLOCKS)(
  if R[b]
  then cornerx[b] = 0
  endif
);
```

It means that if a block has the width that is grater than the half of width of the circuit, then its *x-coordinates* must be 0.

We got very different results using two solvers: `Chuffed` and `Gecode`. For each of them we have taken different approaches and then compared the results to obtain the best search strategy for each of the solvers.

2.5.1 GECODE

First of all we have sorted the data in a decreasing order according 3 parameter:

- Area
- Width
- Height

Then for each sorting obtained we have applied 3 different search strategy that are:

```
solve minimize makespan;
```

```
solve :: seq_search([int_search([makespan],smallest,indomain_min),
                     int_search(cornery, input_order, indomain_min)
                     ])
minimize makespan;
```

```
solve :: seq_search([int_search([makespan],smallest,indomain_min),
                     int_search(cornerx, input_order, indomain_min)
                     ])
minimize makespan;
```

The first one is the default one. The second one will first try to set the height of the silicon plate to the smallest value (that will be the lower bound), then it will try to set the *y-coordinate* of each rectangle to the smallest value possible. The third one is similar to the second one, but it will try to set the *x-coordinates* instead of the *y-coordinates*. The solver will try to place the rectangles based on the order of the data in the input file. After that we have compared the results and we have tried to merge the best approaches to get less timeouts. In the end, the rectangles will be sorted according their heights and, given the ratio

between the sum of widths and the sum of heights, we will use one of the search strategies illustrated before.
In Minizinc we will use the following code:

```
int: h = sum(height);
int: w = sum(width);

var float: ratio = w/h;

solve ::
if ratio > 0.28 /\ ratio < 0.38
then seq_search([int_search([makespan], smallest, indomain_min),
                 int_search(cornerx, input_order, indomain_min)
                 ])
elseif ratio <= 0.28
then seq_search([int_search([makespan], smallest, indomain_min),
                 int_search(cornery, input_order, indomain_min)
                 ])
else seq_search([]) endif
minimize makespan;
```

In the end our algorithm will have 7 timeouts, further researches would be necessary to understand when to apply one of the possible sorts, in fact the number of timeout could be reduced to 5.

2.5.2 CHUFFED

First of all we have sorted the data in a decreasing order according 3 parameter:

- Area
- Width
- Height

Then for each sorting obtained we have applied 3 different search strategy that are:

```
solve minimize makespan;
```

```
solve :: seq_search([int_search([makespan],smallest,indomain_min),
                     int_search(cornery, input_order, indomain_min)
                     ])
minimize makespan;
```

```
solve :: seq_search([int_search([makespan],smallest,indomain_min),
                     int_search(cornerx, input_order, indomain_min)
                     ])
minimize makespan;
```

The first one is the default one. The second one will first try to set the height to the smallest value (that will be the lower bound), then it will try to set the *y-coordinate* of each rectangle to the smallest value possible. The third one is similar to the second one, but it will try to set the *x-coordinates* instead of the *y-coordinates*. The order on which rectangle is picked depends on which type of

sort we have done before.
By looking at the performances we noticed that sorting the rectangles by area in decreasing order and apply the third strategy lead us to have only 3 timeouts.

2.6 Rotation of rectangles

In case there's the possibility of rotating the rectangles, the global constraints used before are useless; as a matter of facts, we need to apply the `geost_bb` constraint.

```
predicate geost_bb(int: k,
                   array [int,int] of int: rect_size,
                   array [int,int] of int: rect_offset,
                   array [int] of set of int: shape,
                   array [int,int] of var int: x,
                   array [int] of var int: kind,
                   array [int] of var int: l,
                   array [int] of var int: u)
```

It will enforce the rectangles to stay in an area represented by the coordinates l and u. By minimizing the *y-coordinate* of `u` we will minimize the height of the area occupied.

2.7 Search strategy with rotation

The search strategy used with `Gecode` will need to be revisited because it doesn't make sense to sort by width in decreasing order in this specific case. One possible solution is to sort according the area and then try to merge the strategies as done before to get the best possible algorithm.
For `Chuffed`, we don't need to change anything, the data are already sorted according the area and the search strategy should work properly.

2.8 Benchmark

In these tables are contained the times taken by the solver to compute an instance with a particular strategy.
Strategy 1:

```
solve minimize makespan;
```

Strategy 2:

```
solve :: seq_search([int_search([makespan],smallest,indomain_min),
                     int_search(cornery, input_order, indomain_min)
                     ])
minimize makespan;
```

Strategy 3:

```
solve :: seq_search([int_search([makespan],smallest,indomain_min),
                     int_search(cornerx, input_order, indomain_min)
                     ])
minimize makespan;
```

Strategy 4:

```
solve ::
if ratio>0.28 /\ ratio< 0.38
then seq_search([int_search([makespan], smallest, indomain_min),
                 int_search(cornerx, input_order, indomain_min)
                 ])
elseif ratio<=0.28
then seq_search([int_search([makespan], smallest, indomain_min),
                 int_search(cornery, input_order, indomain_min)
                 ])
else seq_search([]) endif
minimize makespan;
```

Strategy 5 is **Strategy 4** but without the `lex constraint`.
Strategy 6 is **Strategy 3** but without the `lex constraint`.

The strategy 4 as said before is obtained by merging the past search strategies and let us have more instances resolved with `Gecode`. As you can see by adding the lex constraints we are able to solve two additional instances with `Gecode`, while one with `Chuffed`. Probably we get better result with the last solver due to the fact that it is based on lazy clause generation. The file *"benchmark.xlsx"* contains a more detailed description of all the approaches that we have taken.

3 SATisfiability

SAT is an abbreviation for the Boolean satisfiability problem: given a propositional formula the goal is to check if it is SATisfiable.

3.1 Variables, constraints and objective function

Similarly to what said in section 2.1, we have the **variables**: *n_blocks, max_width, width* and *height*.
The substantial difference between CP and SAT is that SAT can only check if an assignement is SATisfiable or not. So we cannot define an **objective function** like in CP. Therefore, we fixed, in addition to the maximum width, also the maximum height of the silicon plate. The maximum height is, for every instance, the one computed in section 2.2.
The definition of the variables representing the corner of the rectangles is not straightforward like the array defined in CP. Indeed, SAT works only on boolean variables. Hence, we could start from the CP model and, through encoding, obtain a SAT model.

3.1.1 Order encoding

There are many translation methods which encode a CSP into a SAT problem (direct encoding, log encoding, support encoding, etc.). Among them, **order encoding** fits well in our project, because explains, in a more natural way, the order relation of integers (*e.g.* useful for non-overlapping constraints).

For better understand our model, here an example of order enconding [1]: given a simple constraint $x_1 + 1 \leq x_2$ ($x_1, x_2 \in \{0, 1, 2, 3\}$), this is encoded into set of primitive comparisons as follows:

$$\neg(x_2 \leq 0), \quad (x_1 \leq 0) \vee \neg(x_2 \leq 1), \quad (x_1 \leq 1) \vee \neg(x_2 \leq 2), \quad (x_1 \leq 2) \tag{2}$$

Then those constraints are translated into formula of a SAT problem:

$$\neg px_{2,0}, \quad px_{1,0} \vee \neg px_{2,1}, \quad px_{1,1} \vee \neg px_{2,2}, \quad px_{1,2} \tag{3}$$

The wording $px_{i,c}$ means that $px_{i,c}$ is true when $x_i \leq c$.

So we have defined two 2D-matrices of boolean variables, px and py, with dimensions respectively $max\_width \times n\_blocks$ and $max\_height \times n\_blocks$, representing the encoded coordinates x and y of the left corner of each rectangle. Those matrices are not the only variables needed for solving the SAT problem; we need, in addition, two other matrices, with same dimension $n\_blocks \times n\_blocks$, called lr and ud. $lr_{i,j}$ is true if, given two rectangles r_i and r_j with $i \neq j$, r_i is placed to the left of r_j. Similarly, $ud_{i,j}$ is true if r_i is placed under r_j.

3.2 Constraints

For this SAT problem we have two main type of constraints:

- Constraints given by order encoding
- Non-overlapping constraints

Due to **order encoding**, we have 2-literal axiom clauses: For each rectangle r_i, and for all integer e and f such that $0 \leq e < max\_width - width_i$ and $0 \leq f < max\_height - height_i$:

$$\begin{aligned} \neg px_{i,e} \vee px_{i,e+1} \\ \neg py_{i,f} \vee py_{i,f+1} \end{aligned} \tag{4}$$

Then, for expressing the **non-overlapping constraints**, we have both 4-literal clauses and 3-literal clauses:

$$lr_{i,j} \vee lr_{j,i} \vee ud_{i,j} \vee ud_{j,i} \tag{5}$$

And, for each rectangle r_i, and for all integer e and f such that $0 \leq e < max\_width - width_i$ and $0 \leq f < max\_height - height_i$:

$$\begin{aligned} \neg lr_{i,j} \vee px_{i,e} \vee \neg px_{j,e+width_i} \\ \neg lr_{j,i} \vee px_{j,e} \vee \neg px_{i,e+width_j} \\ \neg ud_{i,j} \vee py_{i,f} \vee \neg py_{j,f+height_i} \\ \neg ud_{j,i} \vee py_{j,f} \vee \neg py_{i,f+height_j} \end{aligned} \tag{6}$$

With the aim of reduce the running time of every instance, we added two additional constraints: given two rectangles, if the sum of the *widths* (or *heights*) is greater then the *max_width* (respectively *max_height*), then the two rectangles cannot stay on the same *abscissa* (respectively *ordinate*), *i.e.* cannot stay side by side (or one over the other).
Expressed in propositional logic:

$$\begin{aligned} \neg lr_{i,j} \wedge \neg lr_{j,i} \quad &\text{if} \quad width_i + width_j > max\_width \\ \neg ud_{i,j} \wedge \neg ud_{j,i} \quad &\text{if} \quad height_i + height_j > max\_height \end{aligned} \tag{7}$$

3.3 Symmetry breaking constraints

In order to cut the search tree of the SAT problem, we use the first symmetry breaking constraint seen in section 2.4: rectangles with the same with and height are interchangeable. In SAT we encode this constraint as follows:

$$\begin{aligned} &\neg lr_{j,i} \\ &lr_{i,j} \vee \neg ud_{j,i} \end{aligned} \tag{8}$$

Those means that given two rectangles r_i and r_j with $i \neq j$, with same height and width, we impose that r_j must not be on the left of r_i and, either r_i is on the left, or r_j must be not under block r_i.

3.4 Benchmark

In these tables it is shown how much does the solver takes to compute the solution by using, or not using, the symmetry breaking constraints. Moreover, it is shown how, sorting according the area, would help in the search of the solution. We will have:

- **Annotation 1** will use the symmetry breaking constraints
- **Annotation 2** will not use the symmetry breaking constraints
- **Annotation 3** is similar to **Annotation 1** and will receive the data sorted by decreasing area

In the end the algorithm will use the first choice, because it resolves the largest number of instances.

4 Satisfiability Modulo Theories (SMT)

4.1 Variables, constraints and objective function

The variables defined for the SMT module are similar to those used in the previous parts. We have declared two arrays called `x` and `y`, containing the decision

variables that represent the *x-coordinate* and *y-coordinate* of each rectangle. Of course we want them to be positive so we have added the following constraints:

$$\forall i \in [1, n\_blocks](x_i \geq 0)$$

$$\forall i \in [1, n\_blocks](y_i \geq 0)$$

To solve successfully the problem we need to add two other type of constraints:

- Rectangle sizes must not exceed the given width of the silicon plate
- Rectangles must not overlap together

The first one will be encoded as:

$$\forall i \in [1, n\_blocks](x_i + width_i \leq max\_width)$$

While the second one as:

$$\begin{aligned} &\forall i \in [1, n\_blocks], \forall j \in [i+1, n\_blocks] \\ &(x_i + width_i \leq x_j) \vee \\ &(x_j + width_j \leq x_i) \vee \\ &(y_i + height_i \leq y_j) \vee \\ &(y_j + height_j \leq y_i) \end{aligned} \tag{9}$$

We have also defined another variable called `max_height` that will be our **objective function**; it represents the height of the rectangle that we want to minimize.

4.2 Lower bound of height

Similar to the other parts, we can define a lower bound with the equation (1) and we can use it in the following constraint:

$$max\_height \geq lower\_bound$$

4.3 Improve constraints

We can see that the second constraint is composed by many logical `or`; this lead us to think that it will decrease the efficiency of our algorithm. As a consequence we have rewritten the constraint in order to delete the logical `or`.
Firstly we need to define H and W as the size of the silicon plate, so they will be equal to:

- $W = max\_width$
- $H = lower\_bound$

Then we will declare two matrices of size *n_blocks* * *n_blocks* of boolean variables, called *xb* and *yb*. After that we will add to our solver the following 4 constraints:

$$\begin{gathered}\forall i \in [1, n\_blocks], \forall j \in [i+1, n\_blocks] \\ x_i + width_i \leq x_j + W * (xb_{ij} + yb_{ij}) \wedge \\ x_i - width_j \geq x_j - W * (1 - xb_{ij} + yb_{ij}) \wedge \\ y_i + height_i \leq y_j + H * (1 + xbi_j - yb_{ij}) \wedge \\ y_i - height_j \geq y_j - H * (2 - xb_{ij} - yb_{ij})\end{gathered} \tag{10}$$

However, after conducting some tests, we notice that the number of instances solved were decreased by 2. As a consequence, we kept the version with the constraint described with the logical `or`.

4.4 Symmetry breaking constraints

We have tried to translate the same symmetry breaking constraint used in Strategy Section 3.3.
The first one will be rewritten as:

$$\begin{gathered}\forall i \in [1, n\_blocks], \forall j \in [i+1, n\_blocks] \\ ((width_i + width_j > W) \implies ((x_i + width_i > x_j) \wedge (x_j + width_j > x_i)))\end{gathered} \tag{11}$$

While the second one:

$$\begin{gathered}\forall i \in [1, n\_blocks], \forall j \in [i+1, n\_blocks] \\ ((height_i + height_j > H) \implies ((y_i + height_i > y_j) \wedge (y_j + height_j > y_i)))\end{gathered} \tag{12}$$

However, these symmetry breaking constraints, do not lead the solver to solve more instances. Therefore, in the final code they will not be used.

4.5 Search strategy

To speed up the algorithm instead of adding the constraint:

$$max\_height \geq lower\_bound$$

We have added the following one:

$$max\_height = lower\_bound$$

In case the solver indicates that the problem is unsatisfiable, then we will increase the lower bound by 1 and run it again, so we will have a behaviour similar to the one of Minizinc. As a matter of facts, the solver will try to solve the problem by using the lower bound and, in a negative case, it will try with an higher value.

4.6 Rotation of rectangles

In case we rotate the rectangles, we will need to change some of the constraint defined before, but the work will almost remain the same.
Firstly we will have to modify the first constraint; to do that we will need to generate an array of boolean variables called r; each of the variable indicates if we have rotated the rectangle r_i by 90^0.
After that we can redefine the constraint as:

$$\forall i \in [1, n\_blocks](x_i + r_i * height_i + (1 - r_i) * width_i \leq max\_width)$$

While to satisfy the non-overlapping constraint, we will need only to declare Max as the maximum between M and H.
The constraint will be rewritten as:

$$\begin{gathered}
\forall i \in [1, n_blocks], \forall j \in [i+1, n_blocks] \\
x_i + r_i * height_i + (1 - r_i) * width_i \leq x_j + Max * (xb_{ij} + yb_{ij}) \wedge \\
x_i - r_j * height_j - (1 - r_i) * width_j \geq x_j - Max * (1 - xb_{ij} + yb_{ij}) \wedge \\
y_i + r_i * width_i + (1 - r_i) * height_i \leq y_j + Max * (1 + xb_{ij} - yb_{ij}) \wedge \\
y_i - r_j * width_j - (1 - r_j) * height_j \geq y_j - Max * (2 - xb_{ij} - yb_{ij})
\end{gathered} \tag{13}$$

4.7 Benchmark

In the end we have compared the two definitions of the non-overlapping constraint and the use of the symmetry breaking constraints, in fact:

- **Annotation 1** use the overlap constraint without the logical or
- **Annotation 2** use the overlap constraint with the logical or
- **Annotation 3** is almost similar to **Annotation 2** but there are also the symmetry breaking constraints

As show in the table, **annotation 2** gives us the best result so it will be the one used.

5 Tables

	Gecode: Instances sorted by height				
Instance	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
1	0.45	0.40	0.43	0.41	0.48
2	0.41	0.43	0.40	0.40	0.63
3	0.52	0.40	0.40	0.49	0.53
4	0.44	0.43	0.41	0.54	0.46
5	0.46	0.59	0.41	0.52	0.45
6	0.50	0.84	0.40	0.47	0.45
7	0.47	0.55	0.41	0.53	0.45
8	0.50	0.51	0.43	0.44	0.50
9	0.43	0.51	0.41	0.49	0.60
10	0.43	0.44	0.40	0.40	0.71
11	0.44	Timeout	0.62	0.43	1.15
12	11.84	0.44	0.44	10.42	0.67
13	0.87	0.45	0.44	0.72	0.86
14	3.77	0.43	0.47	2.84	0.66
15	0.94	0.43	0.41	0.43	0.48
16	0.53	Timeout	0.45	0.43	Timeout
17	12.96	0.53	0.43	0.43	0.46
18	1.77	0.47	0.50	0.49	0.49
19	Timeout	51.56	64.21	54.24	30.84
20	Timeout	1.21	64.23	52.48	32.63
21	Timeout	250.89	8.72	7.89	4.92
22	Timeout	1.31	238.90	199.01	126.75
23	5.39	269.62	0.48	0.44	0.46
24	0.91	2.33	1.98	1.73	1.16
25	Timeout	Timeout	Timeout	Timeout	Timeout
26	Timeout	0.50	Timeout	0.88	0.51
27	0.75	39.19	0.87	51.11	29.08
28	Timeout	0.45	10.22	11.43	6.48
29	Timeout	6.57	9.41	10.68	7.08
30	Timeout	Timeout	Timeout	Timeout	Timeout
31	6.02	Timeout	Timeout	7.96	25.79
32	Timeout	Timeout	Timeout	Timeout	Timeout
33	11.41	Timeout	Timeout	14.25	0.47
34	103.42	Timeout	Timeout	150.84	Timeout
35	184.12	Timeout	0.54	251.39	21.07
36	Timeout	Timeout	11.30	17.96	31.26
37	Timeout	Timeout	Timeout	Timeout	Timeout
38	Timeout	Timeout	Timeout	Timeout	Timeout
39	Timeout	Timeout	Timeout	Timeout	Timeout
40	Timeout	Timeout	Timeout	Timeout	Timeout
	Failure: 15	Failure: 14	Failure: 11	Failure: 7	Failure: 9

	Chuffed: Instances sorted by area			
Instance	Strategy 1	Strategy 2	Strategy 3	Strategy 6
1	0.40	0.41	0.39	0.40
2	0.39	0.40	0.39	0.39
3	0.40	0.39	0.39	0.40
4	0.40	0.49	0.40	0.39
5	0.43	0.50	0.42	0.40
6	0.42	0.47	0.41	0.41
7	0.43	0.53	0.42	0.40
8	0.48	0.46	0.45	0.41
9	0.47	0.49	0.45	0.41
10	0.50	0.42	0.60	0.42
11	Timeout	Timeout	43.22	36.11
12	7.34	0.50	0.55	0.54
13	4.43	0.53	0.54	0.55
14	7.79	0.52	0.68	0.51
15	18.09	0.52	0.50	0.54
16	Timeout	Timeout	1.21	1.13
17	Timeout	Timeout	0.67	0.69
18	Timeout	0.56	0.57	0.62
19	Timeout	1.59	49.64	43.42
20	Timeout	0.81	1.96	1.73
21	Timeout	4.89	2.88	2.88
22	Timeout	Timeout	2.96	3.35
23	Timeout	1.82	0.55	0.60
24	Timeout	0.62	0.72	0.73
25	Timeout	2.46	1.30	1.23
26	Timeout	6.02	80.59	81.03
27	Timeout	25.69	3.42	3.88
28	Timeout	2.68	4.65	4.53
29	Timeout	Timeout	8.93	9.31
30	Timeout	Timeout	Timeout	Timeout
31	101.21	Timeout	70.60	69.82
32	Timeout	Timeout	1.10	1.06
33	235.17	Timeout	50.61	43.29
34	Timeout	Timeout	39.26	115.22
35	83.56	2.67	2.91	5.61
36	Timeout	0.915	0.80	0.66
37	Timeout	Timeout	231.72	Timeout
38	Timeout	Timeout	Timeout	Timeout
39	Timeout	Timeout	15.15	13.38
40	Timeout	Timeout	Timeout	Timeout
	Failure: 23	Failure: 14	Failure: 3	Failure: 4

	SAT			
Instances	Annotation 1	Annotation 2	Annotation 3	Union of all strategy
1	0.06	0.65	0.06	
2	0.08	0.08	0.08	
3	0.14	0.13	0.12	
4	0.18	0.19	0.19	
5	0.26	0.27	0.26	
6	0.48	0.38	0.37	
7	0.38	0.39	0.38	
8	0.50	0.53	0.49	
9	0.54	0.55	0.53	
10	0.87	0.98	0.82	
11	1.63	1.67	4.29	
12	1.58	1.63	1.46	
13	1.35	1.36	2.70	
14	1.85	1.81	1.61	
15	1.88	1.89	1.97	
16	13.22	13.82	3.72	
17	7.87	7.98	3.54	
18	3.07	3.18	5.35	
19	32.25	33.93	13.49	
20	11.83	12.47	17.13	
21	18.08	18.31	8.36	
22	49.79	50.65	98.89	
23	13.19	13.66	4.25	
24	10.00	10.10	9.57	
25	32.70	20.41	48.48	
26	11.39	11.73	20.79	
27	16.25	16.50	21.04	
28	7.29	7.27	22.98	
29	7.32	7.32	7.90	
30	99.73	Timeout	Timeout	
31	10.69	10.67	11.15	
32	Timeout	Timeout	50.15	
33	17.20	16.97	17.13	
34	16.02	17.14	10.72	
35	10.49	13.25	7.84	
36	20.99	32.55	29.40	
37	22.56	84.32	137.11	
38	140.70	Timeout	Timeout	
39	58.02	56.03	99.10	
40	Timeout	Timeout	Timeout	
	Failure: 2	Failure: 4	Failure: 3	Failure: 1

	SMT		
Instances	Annotation 1	Annotation 2	Annotation 3
1	0.06	0.02	0.06
2	0.07	0.05	0.08
3	0.10	0.06	0.11
4	0.17	0.05	0.15
5	0.23	0.13	0.24
6	0.28	0.10	0.33
7	0.23	0.09	0.38
8	0.39	0.26	0.39
9	0.41	0.42	0.43
10	0.58	0.38	0.69
11	25.60	23.46	23.49
12	2.30	0.49	2.29
13	0.66	1.02	1.81
14	1.50	2.64	1.27
15	0.96	1.00	3.90
16	25.40	36.69	22.15
17	4.12	1.70	6.02
18	14.08	3.05	12.62
19	33.47	90.26	67.35
20	20.25	17.28	12.04
21	28.72	22.65	36.29
22	Timeout	160.47	282.83
23	16.23	13.09	36.06
24	2.49	3.25	3.63
25	83.59	Timeout	Timeout
26	133.56	186.93	8.40
27	2.68	21.61	7.58
28	49.19	58.10	62.87
29	73.82	23.65	22.44
30	Timeout	117.42	Timeout
31	3.70	7.94	8.29
32	Timeout	Timeout	Timeout
33	16.95	3.90	9.98
34	37.20	263.55	27.73
35	193.29	182.77	265.98
36	50.07	86.18	55.87
37	Timeout	Timeout	Timeout
38	Timeout	Timeout	Timeout
39	Timeout	54.49	Timeout
40	Timeout	Timeout	Timeout
	Failure: 7	Failure: 5	Failure: 7

References

[1] Soh, Takehide Inoue, Katsumi Tamura, Naoyuki Banbara, Mutsunori Nabeshima, Hidetomo. (2010). *A SAT-based Method for Solving the Two-dimensional Strip Packing Problem.* Fundam. Inform.. 102. 467-487. 10.3233/FI-2010-314.

[2] S. Banerjee, A. Ratna and S. Roy, *Satisfiability modulo theory based methodology for floorplanning in VLSI circuits* 2016 Sixth International Symposium on Embedded Computing and System Design (ISED), 2016, pp. 91-95, doi: 10.1109/ISED.2016.7977061.

[3] S. Martello *Packing problem in one or more dimensions* 2018 Winter School on Network Optimization http://www.or.deis.unibo.it/staff_pages/martello/Slides_Estoril_Martello.pdf

[4] Boschetti, Marco Antonio, and Lorenza Montaletti. *An Exact Algorithm for the Two-Dimensional Strip-Packing Problem* Operations Research, vol. 58, no. 6, 2010, pp. 1774–1791. JSTOR, www.jstor.org/stable/40984042

[5] Neuenfeldt Júnior, Alvaro. (2017). *The Two-Dimensional Rectangular Strip Packing Problem* 10.13140/RG.2.2.27201.04965.