Modelling and solving the VLSI problem

Combinatorial Decision Making
Module I Project Work

Aspromonte Marco
marco.aspromonte@studio.unibo.it

Romito Francesco
francesco.romito2@studio.unibo.it

February 2022

Abstract

The following report describes the implementation choices and the relative results about the VLSI problem. Every approach is tried with a python tester, in order to infer which is the best. There are two macro approaches, the Constraint Programming and SMT, the first developed in MiniZinc + Python, the second in Python. The results of the tests are computed using a MacBook Pro with an ordinary CPU (Intel Core i5 quad-core). In the text we indicate with numerical indexes between squared parenthesis the 7 aspects discussed into the project requirements.

1 Introduction

The **VLSI problem** is a very well-known and discussed problem in the Optimization topics, which solution can be reached with several approaches, based on trials and failures. At the beginning of the problem, we just have to deal with the given instances that describe the circuits that have to be placed in the plate, and also the plate itself [1]:

1. The given **fixed width** of the plate
2. The **number of circuits**
3. The **shape** of each circuit, described in terms of width (w) and height (h)

The purpose of the solver is to try many possible positions for each piece in order to find the best positions (x and y) which minimize the total height of the plate that we supposed to be, in theory, at infinity. The problem can be reduced in a minimization task of the following function[1]:

$$f(i) = max(y[i] + h[i])$$

2 Constraint Programming

The first model is divided into three main **CP-based** branches, each of them describes a different approach to find the solution, starting with the simpler model without considering different permutations in terms of rotation of the pieces and continuing with more complex ones that involve the rotation. Each model is also tested for all the instances with a python tester, useful also to describe, print and plot the results.

2.1 CP baseline model

2.1.1 Variables [1]

In the first model, called **'cp_solution.mzn'**, we define all the parameters that will be instantiated by the python tester from txts files. In according to the instances enunciated into the introduction, we have defined the following **input variables**:

- `array[OBJECTS] of int: piece_width` , which is an array containing the given **width of each circuit**
- `array[OBJECTS] of int: piece_height` which is an array containing the given **height of each circuit**
- `int: fixed_width` , which is the given **width of the plate**
- `int: num_circuits` , which is the given **number of circuits**
- `int: lower_limit` , which is computed as:

$$\sum_{i=0}^{num\_circuits} (piece\_width_i * piece\_height_i)/fixed\_width$$

- `set of int: OBJECTS = 1..num_circuits` , which is the instance of each circuit defined as object

After defining the input variables, we can define also the **output variables**, so we can compute and print them :

- `array[OBJECTS] of var 0..fixed_width: position_x` , which is the array of the computed **position along x** (width) axis of each piece
- `array[OBJECTS] of var 0..sum(piece_height): position_y` , which is the array of the computed **position along y** (height) axis of each piece
- `var lower_limit..sum(piece_height): plate_height` , which is the **height of the plate** we want to minimize, that is our objective function, defined with a lower bound (cited above) and an upper bound set as the sum of the pieces' heights.

For our implementation purposes, we have also defined some **support variables**:

- `array[OBJECTS] of int : ordered_ob = sort_by(OBJECTS, [-piece_height[ob]*piece_width[ob] | ob in OBJECTS]);`
  which is the array of **increasing sorted circuits** with respect to their area.
- `array[OBJECTS] of var 0..fixed_width: position_x_pi` that we use as the array of positions of each found solution but flipped on x axis.
- `array[OBJECTS] of var 0..sum(piece_height): position_y_pi` , that we use as the array of positions of each found solution but flipped on y axis.

All the search process is mainly defined by the following **Constraints**.

2.1.2 Global Constraints [3]

- **constraint cumulative(position_x, piece_width, piece_height, plate_height);**
  constraint cumulative(position_y, piece_height, piece_width, fixed_width);
  Cumulative constraints are used as **scheduling constraints** and require a set of positions (x or y) , the extension of each piece for both dimensions (width for x and height for y) , and the resource available for the opposite dimension, in fact for **position_x** we deal with the height of the plate (that should be minimized), and for **position_y** we deal with the fixed width of the plate.
- **constraint diffn(position_x, position_y,piece_width, piece_height);**
  The **diffn** constraint is instead used to manage the **non-overlapping** property. It forces circuits to be non-overlapping , given their origin on each dimension plus the extension on the respective one.

2.1.3 Domain Reduction Constraints [3]

Then, in according to the second project requirement, in order to fit properly the circuits inside the plate, the sum of all the pieces' position plus their extension along the x axis can be at most the fixed width of the plate, and in the second case the sum along the y axis can be at most the dynamic minimized value of the plate height; this is described in the following **domain reduction constraints**:

- with the following constraint we define a limit along the x axis: [2]

```
constraint forall(o in OBJECTS)(position_x[o] + piece_width[o] <= fixed_width);
```

- with the following constraint we define a limit along the y axis: [2]

```
constraint forall(o in OBJECTS)(position_y[o] + piece_height[o] <= plate_height);
```

- with the following constraint we always fix the biggest block in one of the four corner of the plate. The solver can freely choose one of them due to the \/ operator, therefore the other specular ones will be pruned by the **symmetry breaking constraints**.

```
constraint
(position_x[ordered_ob[1]] = 0  /\ position_y[ordered_ob[1]] = 0 )
                                  \/
(position_x[ordered_ob[1]] = 0  /\
position_y[ordered_ob[1]] = plate_height - piece_height[ordered_ob[1]] )
                                  \/
position_y[ordered_ob[1]] = 0 ) /\
(position_x[ordered_ob[1]] = fixed_width - piece_width[ordered_ob[1]]
                                  \/
(position_x[ordered_ob[1]] = fixed_width - piece_width[ordered_ob[1]]  /\
position_y[ordered_ob[1]] = plate_height - piece_height[ordered_ob[1]] );
```

2.1.4 Symmetry Breaking Constraints [4]

In order to reduce the search space size, we try also to include **symmetry breaking constraints** , since we want to avoid all the results that are already computed in a specular form and that would expand the search space making the computation time higher. To reach this goal we first have to instantiate our **support variables**, that we use to define a **symmetry relation**:

- if we want to obtain a solution that is symmetric on the y axis, the x has to be the same, while the specular position of y will be :

```
position_y_pi[ordered_ob[o]] == plate_height -
piece_height[ordered_ob[o]] - position_y[ordered_ob[o]]
```

- if we want to obtain a solution that is symmetric on the x axis, the y has to be the same, while the specular position of x will be :

```
position_x_pi[ordered_ob[o]] == fixed_width -
piece_width[ordered_ob[o]] - position_x[ordered_ob[o]]
```

- now we use these two support vectors to call the **lex_lesseq** constraint, in order to establish the lexicographic order between the standard and flipped positions, so we can avoid all the symmetric solutions. The symmetry breaking constraints are defined such that they do not create any conflict.[5]

```
constraint lex_lesseq(position_x_pi, position_x)
    /\     lex_lesseq(position_y_pi, position_y);
```

2.2 CP model handling rotation [6]

Up to now we have defined defined the model which is able to solve only the instances without considering the rotation of the pieces, so we want to extend the previous model allowing it to rotate the circuits such that it is able to find a solution with a reduced height with respect to the standard version with the appropriate instances. To deploy this feature we have defined a set of constraints which permit the swapping of height and width dimension for all pieces. All the **variables constraints** are inherited from the previous model (section 2.1), so here we describe only the extensive subsections.

2.2.1 Variables [1]

In order to implement this extension, the idea consists in adding one more **support variable** which consists in a boolean vector that expresses for each circuit if it is rotated or not.

- `array [1..num_circuits] of var 0..1 : rotation;`

Moreover we need to define other two **support width and height vectors** in which we instantiate the rotated (or not) shape of each circuit, as follows:

- `array[1..num_circuits] of var 0..max([max(piece_width),max(piece_height)]): piece_width_rotation;`
- `array[1..num_circuits] of var 0..max([max(piece_width),max(piece_height)]): piece_height_rotation;`

Then the plan is to assign to **piece_width_rotation** and **piece_height_rotation** the used dimension of each piece: if the respective value of the piece in the rotation vector is 0 , we keep them as they are, otherwise set the corresponding rotation vector value equal to 1 and assign to **piece_width_rotation** and **piece_height_rotation** the swapped values of width and height respectively, since they are now the used vectors which contains the shapes of each circuit. Here the code of such approach:

- `constraint forall(o in OBJECTS)(piece_height_rotation[o] = piece_height[o]* (1- rotation[o] ) + piece_width[o]*rotation[o]);`
- `constraint forall(o in OBJECTS)(piece_width_rotation[o] = piece_width[o]*(1- rotation[o] ) + piece_height[o]*rotation[o]);`

Since we want to avoid all the redundant results from the search space we set the rotation vector to 0 for all the squared pieces:

- `constraint forall(o in OBJECTS)(if piece_height[o] == piece_width[o] then (rotation[o] = 0)endif);`

2.3 CP model with Global Constraint 'geost_bb' [3]

The two previous models are based on a set of global constraints which work together to find a solution. However the MiniZinc suite offers also a single global constraint that allows to solve the packing problem handling rotations and the non overlapping constraints: this predicate is called **geost_bb**. Moreover due to the limits of the documentation about the language we have done a lot of preprocessing in python regarding the input parameters, in order to fit the attributes of the predicate in a valid form. Such parameters are generated in a file called 'data.dzn' in the **'cp'** folder. The **geost_bb** predicates accepts:

- **k** number of **dimensions**, that in our case is fixed to 2
- **rect_size**, that is a 1-d array which contains all the possible **valid shapes** , considering also the different orientations.
- **rect_offset** defines the **origin of the circuit** reference system, that in our case is fixed to (0,0)
- **coord**, that is the array in which we pack the **computed positions** for both dimensions for each circuit.

- **shape**, that is the array in which there are the references to the **allowed dimensions** for each circuit that are instead described in **in_shape**, that is a support array, where we find 1 element for the squared circuits, and two for the other ones.
- **kind**, that contains all the **assigned shape** to each already positioned circuit
- **[0,0] and [fixed_width,plate_height]**, that are the lower the upper **bounds** of the grid (plate).

In this third model we again inherited the symmetry breaking and domain reduction constraints (explained in sections 2.1.3 and 2.1.4)

2.4 Annotations

In the baseline CP model and in the CP geost model we have defined an int_search strategy with the following settings: [5]

- **variables**: the variables we want to find in order to satisfy the constraints, in this case **plate_height**
- **varchoice**: declares how to choice the next value to assign a value based on its domain, in this case we define **dom_w_deg** which chooses the variable with largest domain, divided by the number of attached constraints weighted by how often they have caused failure
- **strategy** : we set **indomain_min** because we want that to the block is assigned the smallest value of positions' domain

In the CP rotation model we have defined a sequential annotation of int_search that induces the solver to find solutions in the given input order. The parameters of the int_search annotations are defined respectively: [7]

- **variables** : **plate_height** for the first, **rotation** for the second
- **varchoice**: **dom_w_deg** for the first, **firts_fail** for the second
- **strategy** : we set **indomain_min** because we want that to the block is assigned the smallest value of positions' domain

In our models we have also implemented a **restart** technique in order to control how frequently a restart occurs. Restarts occur when a limit in nodes is reached, where search returns to the top of the search tree and begins again. In our case we have set the restart after $s[i] * 1000$ resources where s[i] is the ith number in the Luby sequence.

3 SMT model

This problem can be also solved using the Satisfiability Modulo Theories which is an extension of the Boolean SATisfiability problem, which we know that is a well-known NP-complete class of problems, which allows the possibility of using numbers and data structure (e.g. arrays) into the definition of the problem. To reach our purpose we have used a library in python called **Z3** developed by Microsoft. The approach followed for the implementation of the SMT-based model is similar to the one already used in CP, so developing a baseline model, and an improved one capable to handle the pieces' rotation.

3.1 SMT baseline model

3.1.1 Variables [1]

The first step is basically about importing the instances' **variables** into the python script. Therefore, in order to define the logical rules we have transposed such variables into the following notation:

- W , i.e. **fixed_width**, is the given **fixed width**
- H , i.e. **plate_height**, is the **height** of the plate we want to minimize
- n , i.e. **num_circuits** , is the **number of circuits**

- w , i.e. **piece_width**, is the **width** associated to the piece
- h , i.e. **piece_height**, is the **height** associated to the piece
- x , i.e. **position_x** , is the **vector of positions** along x axis
- y , i.e. **position_y** , is the **vector of positions** along y axis
- C , i.e. **biggest_c** , is the block with the **maximum area**

3.1.2 Rules

- Post the **Domain Reduction** rule such that all the solutions have to involve that the biggest block is placed in the bottom-left corner [1]:

$$(x[C] = 0 \wedge y[C] = 0)$$

  Despite to the CP model, we decided to fix the biggest block only in the chosen corner since in SMT there is a less optimized propagation technique, so even if we have a symmetry breaking rule that will avoid the other three specular combinations, the cost of the computation is way lower with this setting than the previous explained case in paragraph 2.1.3 [5].

- Post the **lower limit** for the minimized value as the ratio between the sum of the areas of each piece and the fixed width in this way we avoid that the algorithm explores solutions with height values that will never be able to solve the problem:

$$H \geq \sum_{i=0}^{n} (w[i] * h[i])/W$$

- Post the **Boundaries'** rules such that a piece must have a positive value of x and y and that the position of the external plus his dimension must be lower than the boundaries (**fixed_width** and **plate_height**) [2].

$$\bigwedge_{i=0}^{n} (x[i] \geq 0 \wedge x[i] + w[i] \leq W)$$

$$\bigwedge_{i=0}^{n} (y[i] \geq 0 \wedge y[i] + h[i] \leq H)$$

- Post the **Symmetry breaking** rule in order to avoid all the solutions that are specular to an already computed one, so far if we consider the horizontal and vertical flip of the board we prune three symmetric solutions with the lexicographic order. To do this we first define the **specular positions' vectors** in the following way [4]:

$$[x\_\pi[i] = W - w[i] - x[i] | i = 0....n]$$

$$[y\_\pi[i] = H - h[i] - y[i] | i = 0....n]$$

  Then we post the rule of **lexicographic order**:

$$lex\_lesseq(x\_\pi, x) \wedge lex\_lesseq(y\_\pi, y)$$

  Defining the function **lex_lesseq** as a recursive function which first checks if the values at the corresponding position in the arrays are the first less than the second. If it is not the case, the function sets the first return value as True and checks if they are equal. If it is not the case set also the second return value False (because it means that the value is greater than the second), concluding the check process with a False statement. After this check we need to verify also the other values in the arrays; we do this through the second part of the And logical port in the return statement, which recalls the function to the remaining part of the array and since we are using the And port they must be both True.

- Post the **non-overlapping** rule, putting in relationship each block with the others. Basically the rule is made by an OR port between four axioms [1]:
  - the piece j must finish before the starting of all the other blocks (in terms of the x axis);
  - the piece j must finish before the starting of all the other blocks (in terms of the y axis);
  - the piece j must start after the ending of all the other blocks (in terms of the x axis);
  - the piece j must start after the ending of all the other blocks (in terms of the y axis);

$$\bigwedge_{i=0,j=i+1}^{n} ((x[j] + w[j] \leq x[i]) \vee (y[j] + h[j] \leq y[i]) \vee (x[j] \geq x[i] + w[i]) \vee y[j] \geq y[i] + h[i]))$$

3.2 SMT Handling Rotation [6]

The baseline model can be also extended involving the rotation of the pieces, in this way we can explore solutions that possibly have a better minimization of the objective function. The rotation model inherits all the artifacts of the baseline model stated into 3.1.1 and 3.1.2 sections. To develop the rotation feature we have defined two additional rules that involve the usage of a rotation vector, called here **rot**, consisting of a vector of Boolean values :

- basically we admit the rotation only on **rectangular pieces**,

$$\bigwedge_{i=0}^{n} ((h[i] \neq w[i]) \rightarrow (rot[i] = 0 \vee rot[i] = 1))$$

  otherwise, so in the case of **squared pieces**, the rotation is forbidden, so the rotation value is set to 0:

$$\bigwedge_{i=0}^{n} ((h[i] = w[i]) \rightarrow rot[i] = 0)$$

- if a piece **is rotated**, the corresponding value on the rotation vector will be 1, and so its height and width values will be swapped, 0 otherwise:

$$\bigwedge_{i=0}^{n} ((rot[i] = 0) \rightarrow (w[i] = w_r[i] \wedge h[i] = h_r[i]))$$

$$\bigwedge_{i=0}^{n} ((rot[i] = 1) \rightarrow (w[i] = h_r[i] \wedge h[i] = w_r[i]))$$

4 Results

In this section we will explain the results obtained by each model, pointing out the differences between the models in terms of efficiency and number of solutions, trying to define also the best combination overall. Then , in order to have a graphical visualization of the solutions, in the python tests we have implemented a plot section where the two axes are the width and the height of the placed pieces.

4.1 CP results

The results that concern the CPs' models are important to define the role of some optimization's techniques we have developed in our project, since in some cases they are fundamental to reach the solution in less time and so, possibly, to solve more instances, in according to the time limit of 5 minutes,that if it is exceeded we consider the instance as unsolved.

4.1.1 Baseline Model

The baseline model is the one described in the section 2.1 which is able to solve the most number of instances between CP models. First we focused on a solving procedure only considering the minimization function without any kind of heuristic in the search, solving only **19/40** instances in the vanilla version. However , due to the NP-hardness of the problem, we influenced the search with the annotation, in which we specify the parameters defined in the paragraph 2.4. With this approach, we solved **28/40** instances. Then we also tried to solve with a **restart** approach, in particular with a Luby sequence, but it does not bring to exciting improvements with respect to the vanilla version, with **21/40 solved instances**. In the following histogram we show the performances defining the relationship between the solved instance and the computation time with the three approaches described above:

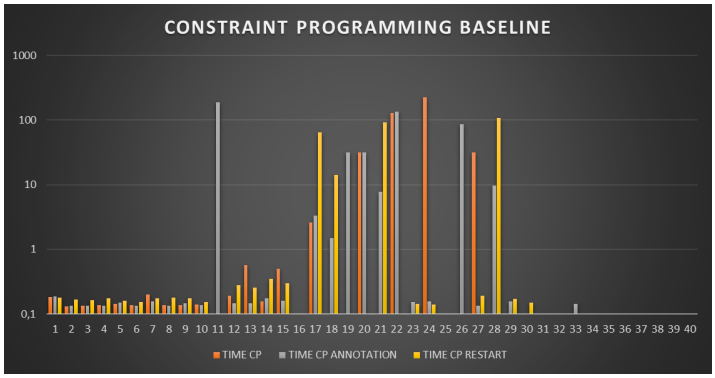

Figure 1: Baseline Model Performance

4.1.2 Rotation Model

The rotation model is the model described in the section 2.2 that is able to solve some of the instances possibly with a better minimized solution. The number of solved instances is on average less than the baseline model, since this kind of task requires more time due the rotation of the pieces, it solves just **13/40** instances indeed. However this model is able to obtain a better solution with respect to the first, if some pieces needs to be rotated to fit perfectly in a rectangle/square.

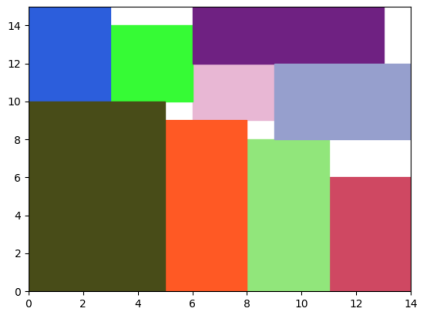

Figure 2: Baseline model solution (ins-7)

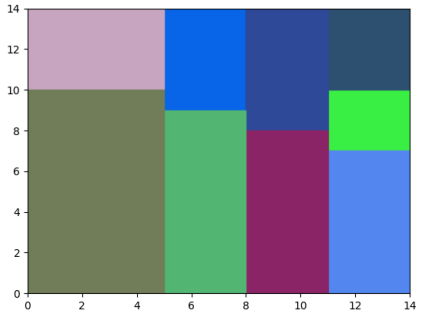

Figure 3: Rotation model solution (ins-7)

However it would be useless to rotate the pieces at each step because from the previous model we checked out that the most solutions did not need a rotation of the pieces; this brings also to a more restricted number of solved instances, so we have defined in the sequential search an annotation that forces the solver first to try the solution with the all not-rotated pieces first:

```
:: seq_search([
int_search([plate_height], dom_w_deg, indomain_min),
int_search(rotation, first_fail, indomain_min) ])
```

In such way we obtained **22/40** solved instances. The rotation model is interesting also because in some cases it is able to find a solution where the baseline model has failed :

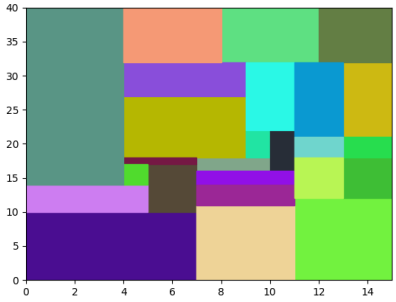

Figure 4: The 35 instance case

This is due to the capacity of the rotation model to find a permutation of pieces which perfectly fit a rectangle which consequently minimize the plate height, while the baseline one does not reach this solution, so it needs to prove that a non perfectly fitted plate is its best solution and doing this spends a lot of time , which leads to a failure due to the time bound.

In the following histogram we show the performances defining the relationship between the solved instance and the computation time with the three approaches described above:

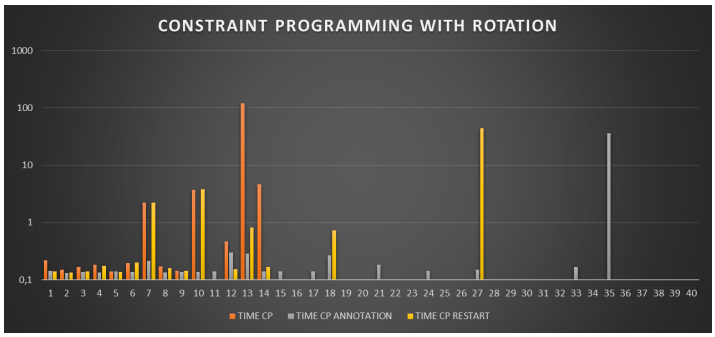

Figure 5: Rotation Model Performance

4.1.3 Geost Model

Because of Global Constraints are always the best choice, we also define a model with only one global constraint, **geost_bb** which should be optimized for the family of **Rectilinear Packing Problems**. However this is not true in our case, since even if we implemented such constraint and tried to optimize the research with annotations and restart techniques, this is the slower model and leads to less solved instance with respect to the other models, solving just **15/40** instances declaring this as the worst model. Therefore it is useful to define a python program that is able to fit the datas required by this constraint, described in the 2.3 section. However for the seek of completess we also show the following histogram with the Geost performances:

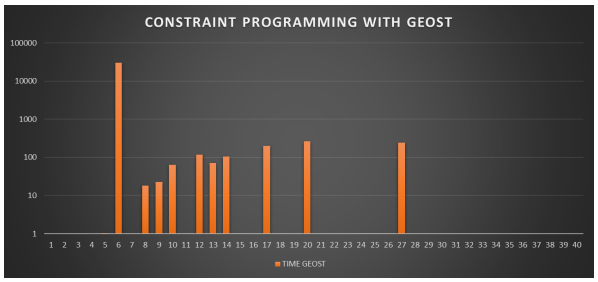

Figure 6: Geost Model Performance

4.2 SMT results

The results that concern the SMTs' models are interesting since as we can see, it is more performing than CP models without implementing any research heuristic (e.g. the annotation or restart technique). In general we know that all the models suffer the growth of instances , such that with a linear increasing of number of pieces, the space and time complexity of the problem increases exponentially, so from the results we can infer that **SMT has approximately constant growth** of the solving time, while the CP vanilla model has a **dramatic increasing** when the number of pieces rises; therefore, if we define some heuristic rule to induce the research in a specific branch of the search three, it is able to keep a constant low solution time considering the solved instances.

4.2.1 Baseline Model

The Baseline SMT model, that is described in the section 3.1, is the best model in terms of solved instances since it has computed **29/40 instances** in less time for the cases with few pieces. We can check such assertion in the following performance histogram:

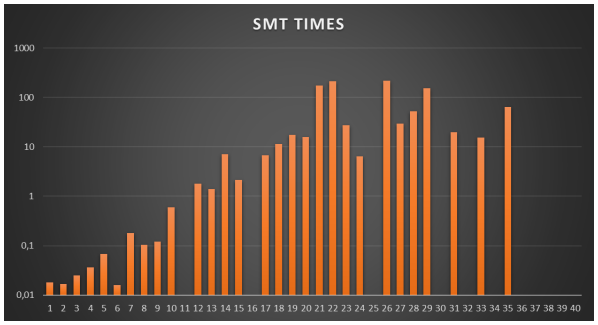

Figure 7: SMT Baseline Performance

The interesting aspect of SMT is that it is able to solve the maximum number of instances with respect to all the other models and their flavours, so with our implementation it is more performing than all the others in terms of solved instances, but in according to the previously introduced discussion about heuristic, it becomes worst in terms of time.

4.2.2 Rotation Model

The SMT rotation model, that is described in the section 3.2, is able to solve **18/40** instances. In according with the introduction about SMT results above, this model is able to solve more instances than the vanilla CP rotation model, but it is less performing than the CP rotation model with annotations. We can check such assertions in the following performance histogram:

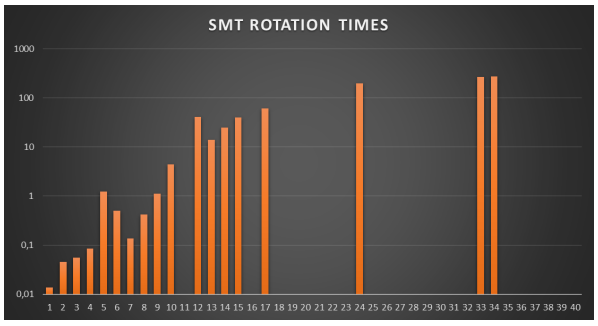

Figure 8: SMT rotation performance

5 Conclusions

The VLSI is a very difficult task that can be solved with different techniques, each of them with their pros and cons. We can say that different approaches can solve the same problem in similar time basing on which problem is purposed: the models which do not try to rotate the circuits have completed more tasks with respect to the other ones, since we have set a time limit and adding the rotation we make the problem heavily harder when we have to deal with several pieces of different dimensions. On the other hand we know that without rotation we can always expect a solution that is worst or equal to the respective one and in real scenarios a better model can be useful even if it takes more time to be computed. However, it is important to remark the importance of a good heuristic dealing with problems that belong to NP-hard class; this is particularly evident in the comparison between the CP and SMT rotation models, the best model in the rotation problem is the one with the annotation so with a heuristic indeed. At the end, summing up all the solutions we have found of all the models, we can conclude our task with **32/40** solved instances.

References

[Zeynep Kiziltan] Combinatorial Decision Making and Optimization slides

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

[Z3 Handbook] https://ericpony.github.io/z3py-tutorial/guide-examples.htm

[Peter Stuckey] https://www.coursera.org/lecture/advanced-modeling/2-4-3-rectilinear-packing-with-rotation-TCClS