

VLSI - Very Large Scale Integration

Combinatorial Decision Making and Optimization Module 1

Dahesh, Parsa (dahesh.parsa@studio.unibo.it)

Granata, Ludovico (ludovico.granata@studio.unibo.it)

Persiani, Simone (persiani.simone2@studio.unibo.it)

Academic year 2020-2021

Abstract

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. A typical example is the smartphone. The modern trend of shrinking transistor sizes, allowing engineers to fit more and more transistors into the same area of silicon, has pushed the integration of more and more functions of cellphone circuitry into a single silicon die (i.e. plate). This enabled the modern cellphone to mature into a powerful tool that shrank from the size of a large brick-sized unit to a device small enough to comfortably carry in a pocket or purse, with a video camera, touchscreen, and other advanced features.

1 Constraint Programming

1.1 Variables

1.1.1 Problem parameters

First, let's define the problem parameters:

- *width*, the width of the board;
- *n_circuits*, defines the total number of circuits;
- *hor_dim* and *ver_dim*, arrays that store the horizontal and vertical dimensions of each circuit (e.g. the height and width of circuit *i* are stored in *hor_dim[i]* and *ver_dim[i]*).

We also sorted the circuits by their total area in order to exploit this ordering in the searching process (see section 1.3). Therefore, in many of the constraints that will be described in the following section, it is assumed that the circuits are sorted.

1.1.2 Decision variables

Now, let's define the decision variables:

- *X_pos*, *x* coordinate of the bottom left corner of each circuit;
- *Y_pos*, *y* coordinate of the bottom left corner of each circuit.

The goal is to minimize and find the optimal *height* of the board while fitting all the given circuits.

It is important to properly define the domain of each decision variable in order to improve the performance of the searching process. This has to be done without excluding any valid solution from the search space. Therefore, we have that:

$$height \in [min_height, max_height]$$

where:

- $min_height = (\sum_i ver_dim_i * hor_dim_i) / width$, i.e. the sum of the circuit areas divided by the board's width;
- $max_height = \sum_i ver_dim_i$, i.e. the sum of the vertical dimension of each circuit.

and

$$\begin{aligned} X_pos &\in [0, width - \min(hor_dim)] \\ Y_pos &\in [0, max_height - \min(ver_dim)] \end{aligned}$$

1.2 Constraints

In this section, we will discuss all the constraints that we have tested including the less successful ones. We will later on compare the different variants to establish the best model.

1.2.1 Main Constraints

The border constraints check that no circuit is partially or completely positioned outside of the grid and are implemented as follows:

```
constraint forall (c in CIRCUITS)
  (X_pos[c] + hor_dim[c] <= width)::domain;
constraint forall (c in CIRCUITS)
  (Y_pos[c] + ver_dim[c] <= height)::domain;
```

Then we introduce the `diffn` global constraint to avoid circuits overlapping each other:

```
constraint diffn(X_pos, Y_pos, hor_dim, ver_dim)::domain;
```

1.2.2 Implied constraints

To improve the searching process we added an implied constraint, namely the `cumulative` global constraint. The definition of `cumulative` is the following:

“It requires that a set of tasks given by start times s , durations d , and resource requirements r , never require more than a global resource bound b at any one time.”

We can use it by imagining that the start times are the positions X_pos (resp. Y_pos) of our circuits, the durations are the horizontal dimensions hor_dim (resp. vertical dimensions ver_dim), the resource requirements are the vertical dimensions ver_dim (resp. horizontal dimensions hor_dim) and the global resource is the $width$ (resp. $height$). We implemented them as:

```
constraint cumulative(X_pos, hor_dim, ver_dim, height);
constraint cumulative(Y_pos, ver_dim, hor_dim, width);
```

1.2.3 Symmetry breaking constraints

The first symmetry breaking constraint consists in placing the *largest circuit on the bottom left* of the board. In this way, we remove the solution in which the board is rotated by 180° or reflected along the central horizontal/vertical line (see figure 1). This circuit has index 1 since the circuits are sorted by dimension. The constraint is implemented as follows:

```
constraint X_pos[1] == 0 /\ Y_pos[1] == 0;
```

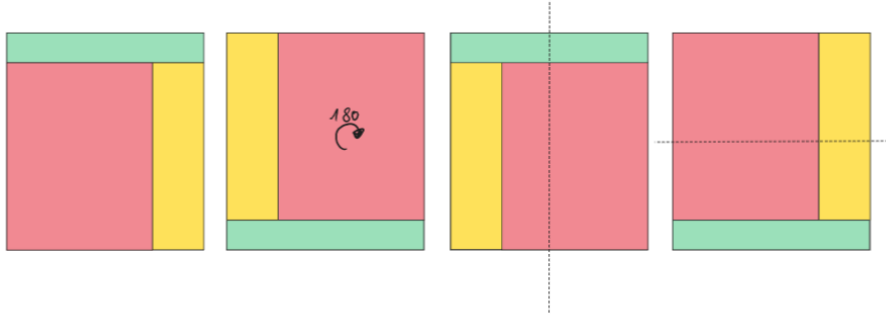


Figure 1: Largest block on the bottom left

The second constraint imposes a lexicographic ordering to break symmetries on the X and Y axis. To do so, we first have to obtain the symmetric coordinates of each circuits:

$$X_pos_sym_i = width - X_pos_i - hor_dim_i$$

$$Y_pos_sym_i = height - Y_pos_i - ver_dim_i$$

Then we can apply the `lex_lesseq` constraints:

```
constraint symmetry_breaking_constraint (
    lex_lesseq(X_pos, X_pos_sym)
);
constraint symmetry_breaking_constraint (
    lex_lesseq(Y_pos, Y_pos_sym)
);
```

The last symmetry breaking constraint is called `two_stack`. In the vertical variant, given two blocks with the same width that are placed one on top of the other, the larger block – namely the one with the greater vertical dimension – will be placed on the bottom as shown in fig. 2.

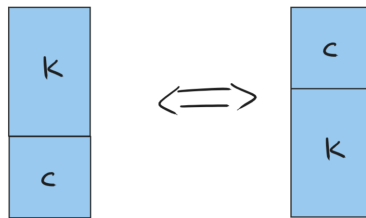


Figure 2: `two_stack_ver`

Respectively, the same approach is applied to the horizontal variant.

To check whether two circuits are stacked, we implemented the following predicates:

```

predicate two_stack_ver(int: i, int: j) = (
  X_pos[i] == X_pos[j] /\ hor_dim[i] == hor_dim[j] /\
  ver_dim[i] + ver_dim[j] ==
    max([Y_pos[i]+ver_dim[i], Y_pos[j]+ver_dim[j]])
    - min([Y_pos[i], Y_pos[j]])
);

predicate two_stack_hor(int: i, int: j) = (
  Y_pos[i] == Y_pos[j] /\ ver_dim[i] == ver_dim[j] /\
  hor_dim[i] + hor_dim[j] ==
    max([X_pos[i]+hor_dim[i], X_pos[j]+hor_dim[j]])
    - min([X_pos[i], X_pos[j]])
);

```

For example, in the `two_stack_ver` case, two circuits `i` and `j` are vertically stacked if besides having the same width and x coordinate, the sum of their vertical dimensions equals to the difference between the upper side of the top circuit and the bottom of the second circuit.

To these two constraints, we added a second layer of circuit stacking which follows the same reasoning of the previously mentioned constraints and addresses the issue showed in fig. 3.

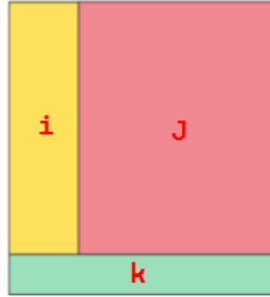


Figure 3: `three_stack_ver`

The stacking constraints were implemented as:

```

constraint symmetry_breaking_constraint (
  forall ( i,j in CIRCUITS where i<j /\ two_stack_ver(i,j)) (
    Y_pos[i] <= Y_pos[j]      % Larger block on the bottom

    % Note: the circuits are ordered by total area so
    % circuit i is larger than circuit j with i<j

    % Three stack layer:
    % we already checked if circuits i and j are stacked.
    % Now we check for circuit k

    /\ forall( k in CIRCUITS where i!=k /\ j!=k /\
      ver_dim[k] == ver_dim[i] + ver_dim[j] /\
      hor_dim[k] + hor_dim[i] ==
      max([X_pos[k]+hor_dim[k],X_pos[i]+hor_dim[i]]) -
      min([X_pos[k],X_pos[i]]) ) (
        X_pos[k] < X_pos[i] /\ X_pos[k] < X_pos[j]
      )
    )
  );

constraint symmetry_breaking_constraint (
  forall ( i,j in CIRCUITS where i<j /\ two_stack_hor(i,j)) (
    Y_pos[i] <= Y_pos[j]

    /\ forall( k in CIRCUITS where i!=k /\ j!=k /\
      hor_dim[k] == hor_dim[i] + hor_dim[j] /\
      ver_dim[k] + ver_dim[i] ==
      max([Y_pos[k]+ver_dim[k],Y_pos[i]+ver_dim[i]]) -
      min([Y_pos[k],Y_pos[i]]) ) (
        Y_pos[k] < Y_pos[i] /\ Y_pos[k] < Y_pos[j]
      )
    )
  );

```

Note: when using the largest circuit on the bottom left constraint, we check that in the stacking constraints the indexes i,j,k are NOT equal to 1 (i.e. the index of the largest circuit). This is done to avoid excluding valid solutions.

1.3 Searching

Search annotations have an important role in improving the performances. We compared different searching strategies on the same model: to do so, we selected a mini-batch of 5 arbitrarily-chosen instances over which we ran each searching configuration

and obtained the following results:

Variable	Choice	Restart	Solved	Avg solving time [ms]
input_order	random	luby	5/5	10064
first_fail	random	luby	5/5	11253
smallest	random	luby	4/5	25087
dom_w_deg	random	luby	5/5	1133
dom_w_deg	min	luby	5/5	2001
dom_w_deg	median	luby	3/5	1076
dom_w_deg	split	luby	5/5	1235
dom_w_deg	random	geometric	5/5	1163
dom_w_deg	random	linear	5/5	25376
dom_w_deg	random	constant	4/5	34407

Table 1: Mini-batch of instances: 5, 8, 10, 12, 14. The best configuration is highlighted in bold.

In addition to the best configuration we also added the `relax_and_reconstruct` annotation for X_{pos} .

1.4 Results

We tried the following combinations of constraints with the best searching strategy found in the previous section because the X, Y s.b. constraint and *largest circuit on the bottom left* are not compatible with each other:

- ① X, Y s.b. constraint + `two_stack` + `three_stack`
- ② largest circuit on the bottom left + `two_stack` + `three_stack`

Finally, we obtained the following results:

Model	Solved instances	Average solving time
1	32/40	4.502 s
2	32/40	8.370 s

The model with the best performances is number ①.

1.5 Rotations

It is possible to make some small changes to the models to allow the blocks to be rotated. The idea is to add an array of boolean variables that tells if a block is rotated or not. We can define the horizontal and vertical dimension of each block in the following way:

$$\begin{aligned}
\bigwedge_i rotated_i \longrightarrow (dim1_i = ver_dim_i \wedge dim2_i = hor_dim_i) \wedge \\
\neg rotated_i \longrightarrow (dim1_i = hor_dim_i \wedge dim2_i = ver_dim_i)
\end{aligned} \tag{1}$$

where $dim1_i$ and $dim2_i$ indicate the horizontal and vertical dimensions of the rotation model, while hor_dim_i and ver_dim_i are constants read from the input instance files (i.e. they represent the original non-rotated dimensions of the i^{th} block).

2 SMT

2.1 Variables

First, let's define the problem parameters:

- *width*, the width of the board;
- *n_circuits*, defines the total number of circuits;
- *hor_dim* and *ver_dim*, the height and width of each circuit.

Now, let's define the decision variables:

- *circuitx*, *x* coordinate of the bottom left corner of each circuit;
- *circuity*, *y* coordinate of the bottom left corner of each circuit.

The goal is to minimize and find the optimal *height* of the board.

2.2 Constraints

2.2.1 Non-overlapping Constraint

The first constraint that we implemented is the non-overlapping constraint which, as the name suggests, guarantees that no circuit is overlapping inside the grid. It is developed following the MiniZinc documentation and it's implemented as follows:

$$\bigwedge_{i < j} (x_i + \text{hor_dim}_i \leq x_j) \vee (y_i + \text{ver_dim}_i \leq y_j) \vee (x_j + \text{hor_dim}_j \leq x_i) \vee (y_j + \text{ver_dim}_j \leq y_i) \quad (2)$$

where:

- *x* and *y* are the circuit coordinates;
- *hor_dim_i* and *ver_dim_i* are the circuit dimensions.

2.2.2 Border Constraints

The border constraints check that no circuit is partially or completely positioned outside of the grid. It's implemented as follows:

$$\bigwedge_i (0 \leq \text{circuitx}_i) \wedge (\text{circuitx}_i + \text{hor_dim}_i \leq \text{width}) \wedge (0 \leq \text{circuity}_i) \wedge (\text{circuity}_i + \text{ver_dim}_i \leq \text{height}) \quad (3)$$

2.2.3 Symmetry Breaking Constraint

While trying to break as many symmetries as possible in the model, we had to balance the effectiveness with the complexity of the constraints to be implemented, as complex constraints can require a combinatorial amount of time to be generated. Having evaluated many other options, we settled only on positioning the biggest circuit in the bottom-left corner of the grid, namely $(n, m) = (0, 0)$.

It is sufficient to implement the following constraints:

$$circuitx_k = 0 \wedge circuity_k = 0 \quad (4)$$

where:

$$k = \underset{i}{\operatorname{argmax}} hor_dim_i \cdot ver_dim_i$$

2.3 Rotation model

The rotation model introduces a new boolean decision variable called *rotated* which is *true* in case the circuit is rotated w.r.t. its starting orientation, *false* otherwise. To correctly use the horizontal and vertical dimension variables in the previously mentioned constraints, we implemented the following constraint:

$$\begin{aligned} \bigwedge_i rotated_i &\longrightarrow (hor_dim_i = dim2_i \wedge ver_dim_i = dim1_i) \wedge \\ \neg rotated_i &\longrightarrow (hor_dim_i = dim1_i \wedge ver_dim_i = dim2_i) \end{aligned} \quad (5)$$

where *dim1* and *dim2* are constants read from the input instance files, that represent the non-rotated width and height of each circuit.

3 SATisfiability

3.1 Model setup

In our proposed solution, each instance file is initially parsed in order to extract the following constants:

- $n_{circuits}$, the amount of circuits to be packed inside the grid;
- w_{grid} , the width of the grid;
- w_i and h_i with $i \in [0, n_{circuits})$, the width and height of each circuit.

Considering that one of the assigned goals is the minimization of the packing height and that –at the same time– we are interested in reducing the amount of variables and related constraints for performance reasons, we chose to limit the height of the grid to the optimal packing height as computed here:

$$h_{opt} = \frac{\sum_{i=0}^{n_{circuits}-1} w_i \cdot h_i}{w_{grid}} \quad (6)$$

h_{opt} is the packing height in case no gap is left between the circuits and each of them is aligned in such a way that they collectively reach the lowest possible height. We have found that this choice is not too big of a limitation for the solver, as each provided instance is solvable within this constraint.

3.2 The variables

We started modelling the problem by defining the Boolean variables needed to represent its domain. After having considered a couple of other options, we chose to identify the position of each circuit inside the grid with the pair of coordinates in which its bottom-left corner can be placed. This led us to the definition of the following variables:

$$v_{i,n,m}$$

, where $i \in [0, n_{circuits})$ is the index of the circuit, while n and m represent (respectively) the horizontal and vertical coordinates of the bottom-left corner of the circuit. $v_{i,n,m}$ is true if the bottom-left corner of the i -th circuit is placed at the point (n, m) inside the grid.

3.2.1 Rotation model

In order to consider also the 90°-rotated version of each circuit, the following variables have to be defined:

$$r_i$$

, where $i \in [0, n_{circuits})$ is the index of the circuit. r_i is true if the i -th circuit is rotated by 90°.

3.3 Positioning constraints

Each circuit must be placed in some position inside the grid and –at the same time– it cannot be placed in more than one position. Such a statement can be encoded in SAT by looping over all the possible (n, m) positions for the i -th circuit, in order to apply the following constraint:

$$\bigwedge_i \text{exactly_one}(i) = \bigwedge_i \left[\left(\bigvee_{(n,m)} v_{i,n,m} \right) \wedge \left(\bigwedge_{(n,m) \neq (p,q)} \neg (v_{i,n,m} \wedge v_{i,p,q}) \right) \right] \quad (7)$$

An additional constraint is implicitly enforced by limiting the possible positions to only those that result in the circuit being completely contained inside the grid. In this way, the grid boundaries are always respected:

$$\mathcal{H}_i = [0, w_{grid} - w_i], \mathcal{V}_i = [0, h_{opt} - h_i]$$

$$(n, m), (p, q) \in \mathcal{H}_i \times \mathcal{V}_i$$

3.3.1 Rotation model

In the variant of the model that takes into consideration also the possibility to rotate the circuits by 90° , the previous constraint has to be split in two as follows:

$$\neg r_i \longrightarrow \bigwedge_i \text{exactly_one}(i) \quad (8)$$

$$r_i \longrightarrow \bigwedge_i \text{exactly_one}(i) \quad (9)$$

In case of equation 8 the domain of possible positions is $\mathcal{H}_i \times \mathcal{V}_i$, while in case of equation 9 the domain must be revised as follows (since rotating by 90° a circuit means swapping its dimensions):

$$\mathcal{H}_i^R = [0, w_{grid} - h_i], \mathcal{V}_i^R = [0, h_{opt} - w_i]$$

$$(n, m), (p, q) \in \mathcal{H}_i^R \times \mathcal{V}_i^R$$

For each squared circuit, we force it not to be rotated in order to avoid unnecessary work for the solver:

$$\bigwedge_{w_i=h_i} \neg r_i \quad (10)$$

3.4 Non-overlapping constraints

No pair of circuits should ever overlap inside the grid. Since this kind of constraint is inherently symmetrical, we apply it to every possible pair of circuits (i, j) with $i < j$ (so to be sure not to apply the same constraint to the same two blocks twice). We chose to encode this constraint in SAT by reasoning as follows: if the i -th circuit is placed at (n, m) , then there are some positions (p, q) in which the j -th circuit cannot be placed, as in such cases it would overlap with i .

$$\bigwedge_{i < j} v_{i,n,m} \longrightarrow \neg v_{j,p,q} \quad (11)$$

Such forbidden positions are determined by considering both the dimensions of the j -th circuit and those of the i -th. The resulting forbidden range is then intersected with the actual positioning domain of the j -th circuit.

$$\begin{aligned} (n, m) &\in \mathcal{H}_i \times \mathcal{V}_i \\ p &\in [n - w_j + 1, n + w_i - 1] \cap \mathcal{H}_j \\ q &\in [m - h_j + 1, m + h_i - 1] \cap \mathcal{V}_j \end{aligned}$$

3.4.1 Rotation model

When rotations are allowed, the previous constraint has to be split into four variants (one for each possible combination of r_i and r_j values) and both the domain for (n, m) and the forbidden ranges for (p, q) have to be re-computed:

$$\bigwedge_{i < j} v_{i,n,m} \wedge \neg r_i \wedge \neg r_j \longrightarrow \neg v_{j,p,q} \quad (12)$$

$$\bigwedge_{i < j} v_{i,n,m} \wedge \neg r_i \wedge r_j \longrightarrow \neg v_{j,p,q} \quad (13)$$

$$\bigwedge_{i < j} v_{i,n,m} \wedge r_i \wedge \neg r_j \longrightarrow \neg v_{j,p,q} \quad (14)$$

$$\bigwedge_{i < j} v_{i,n,m} \wedge r_i \wedge r_j \longrightarrow \neg v_{j,p,q} \quad (15)$$

For example, in case of equation 14:

$$\begin{aligned} (n, m) &\in \mathcal{H}_i^R \times \mathcal{V}_i^R \\ p &\in [n - w_j + 1, n + h_i - 1] \cap \mathcal{H}_j \\ q &\in [m - h_j + 1, m + w_i - 1] \cap \mathcal{V}_j \end{aligned}$$

3.5 Symmetry-breaking constraints

While trying to break as many symmetries as possible in the model, we had to balance the effectiveness with the complexity of the constraints to be implemented, as complex constraints can require a combinatorial amount of time to be generated. Having evaluated many other options, we settled on the following ideas:

1. positioning the biggest circuit in the bottom-left corner of the grid, namely $(n, m) = (0, 0)$;
2. when two circuits with the same horizontal dimensions are vertically stacked on top of each other, the bigger one should stay in the bottom;
3. when two circuits with the same vertical dimensions are horizontally stacked next to each other, the bigger one should stay on the left.

For point 1) is sufficient to implement this simple constraint:

$$v_{k,0,0} \tag{16}$$

$$k = \operatorname{argmax}_i w_i \cdot h_i$$

For points 2) and 3), we chose to implement a set of constraints very similar to those used for avoiding overlaps, in which we loop over every possible pair of circuits (i, j) with $i < j$ (so to be sure not to apply the same constraint to the same two blocks twice). We chose to encode such statements in SAT by reasoning as follows, assuming that the i -th circuit is placed at (n, m) :

1. if the i -th circuit is bigger than (or equal in size to) the j -th circuit, then:
 - (a) if $w_i = w_j$, then (p, q) cannot be such that the two circuits are vertically stacked with i on top of j ;
 - (b) if $h_i = h_j$, then (p, q) cannot be such that the two circuits are horizontally stacked with i on the right of j ;
2. if the i -th circuit is smaller than the j -th circuit, then:
 - (a) if $w_i = w_j$, then (p, q) cannot be such that the two circuits are vertically stacked with j on top of i ;
 - (b) if $h_i = h_j$, then (p, q) cannot be such that the two circuits are horizontally stacked with j on the right of i .

$$\bigwedge_{i < j} v_{i,n,m} \longrightarrow \neg v_{j,p,q} \tag{17}$$

For example, in case of situation 2.b) (if $n + w_i \in \mathcal{H}_j$):

$$(n, m) \in \mathcal{H}_i \times \mathcal{V}_i$$

$$(p, q) = (n + w_i, m)$$

3.5.1 Rotation model

The symmetry-breaking constraints shown above become more complex when considering also the possibility to rotate the circuits by 90° . All of the above 4 different situations must be split into other 4 subcases (one for each possible combination of r_i and r_j values):

$$\bigwedge_{i < j} v_{i,n,m} \wedge \neg r_i \wedge \neg r_j \longrightarrow \neg v_{j,p,q} \quad (18)$$

$$\bigwedge_{i < j} v_{i,n,m} \wedge \neg r_i \wedge r_j \longrightarrow \neg v_{j,p,q} \quad (19)$$

$$\bigwedge_{i < j} v_{i,n,m} \wedge r_i \wedge \neg r_j \longrightarrow \neg v_{j,p,q} \quad (20)$$

$$\bigwedge_{i < j} v_{i,n,m} \wedge r_i \wedge r_j \longrightarrow \neg v_{j,p,q} \quad (21)$$

For example, in case of situation 2.b) with $r_i \wedge \neg r_j$ (if $n + h_i \in \mathcal{H}_j$):

$$\bigwedge_{i < j} v_{i,n,m} \wedge r_i \wedge \neg r_j \longrightarrow \neg v_{j,p,q}$$

$$(n, m) \in \mathcal{H}_i^R \times \mathcal{V}_i^R$$

$$(p, q) = (n + h_i, m)$$

4 Results and Tables

Here are the tables with the final results¹ (timeout is set to 5 minutes, results are computed as the average of 3 runs):

Standard model							
Ins.	CP	SMT	SAT	Ins.	CP	SMT	SAT
1	0.295s	0.016s	0.294s	21	1.162s	timeout	timeout
2	0.297s	0.018s	0.572s	22	1.377s	timeout	timeout
3	0.300s	0.032s	2.158s	23	2.071s	8.942s	timeout
4	0.311s	0.039s	3.363s	24	0.655s	7.399s	timeout
5	0.307s	0.046s	6.707s	25	timeout	timeout	timeout
6	0.316s	0.071s	11.254s	26	2.020s	57.975s	timeout
7	0.308s	0.070s	11.510s	27	0.901s	20.394s	timeout
8	0.320s	0.100s	15.745s	28	0.956s	53.038s	timeout
9	0.317s	0.086s	23.150s	29	2.261s	65.314s	timeout
10	0.352s	0.332s	45.351s	30	timeout	timeout	timeout
11	22.743s	timeout	timeout	31	0.861s	8.249s	timeout
12	0.452s	0.688s	170.593s	32	timeout	timeout	timeout
13	0.409s	0.541s	timeout	33	0.668s	21.927s	timeout
14	0.435s	1.772s	timeout	34	timeout	timeout	timeout
15	0.456s	0.983s	timeout	35	40.135s	timeout	timeout
16	9.154s	timeout	timeout	36	43.464s	timeout	timeout
17	1.377s	5.479s	timeout	37	timeout	timeout	timeout
18	0.565s	6.516s	timeout	38	timeout	timeout	timeout
19	1.045s	timeout	timeout	39	timeout	timeout	timeout
20	1.400s	133.834s	timeout	40	timeout	timeout	timeout

¹Run with CPU: AMD Ryzen 5 3600; GPU: NVIDIA GeForce GTX 1660 Super; RAM: 16GB; OS: Windows 10

Rotation model							
Ins.	CP	SMT	SAT	Ins.	CP	SMT	SAT
1	0.300s	0.019s	1.902s	21	timeout	timeout	timeout
2	0.311s	0.027s	3.998s	22	timeout	timeout	timeout
3	0.315s	0.047s	11.905s	23	timeout	timeout	timeout
4	0.318s	0.149s	23.888s	24	2.910s	timeout	timeout
5	0.325s	0.476s	186.306s	25	timeout	timeout	timeout
6	0.352s	0.781s	270.830s	26	timeout	timeout	timeout
7	0.446s	1.045s	timeout	27	7.723s	timeout	timeout
8	0.350s	0.826s	137.327s	28	timeout	timeout	timeout
9	18.657s	2.784s	timeout	29	timeout	timeout	timeout
10	1.774s	133.357s	timeout	30	timeout	timeout	timeout
11	timeout	timeout	timeout	31	timeout	timeout	timeout
12	2.256s	timeout	timeout	32	timeout	timeout	timeout
13	1.739s	timeout	timeout	33	timeout	timeout	timeout
14	timeout	timeout	timeout	34	timeout	timeout	timeout
15	3.192s	timeout	timeout	35	timeout	timeout	timeout
16	timeout	timeout	timeout	36	timeout	timeout	timeout
17	timeout	timeout	timeout	37	timeout	timeout	timeout
18	timeout	timeout	timeout	38	timeout	timeout	timeout
19	timeout	timeout	timeout	39	timeout	timeout	timeout
20	timeout	timeout	timeout	40	timeout	timeout	timeout