Very Large Scale Integration Project

Shamik, Basu
shamik.basu@studio.unibo.it | 0001035358

Pratiksha, Pratiksha
pratiksha.pratiksha@studio.unibo.it | 0001034021

December 5, 2022

Project report for

CDMO

Contents

1 INTRODUCTION

This report describes the solution of the VLSI problem using combinatorial approach. We will discuss four optimization techniques to solve the problem - Contraint Programming (CP), propositional SATisfiability (SAT), Satisfiability Modulo Theories (SMT), and Mixed-Integer Linear Programming (MIP).

Some of the common constraints are common for all the four techniques-

- Non overlapping and width and height(all models)
- Symmetry breaking in all except MIP

The CP implementation was majorly handled by Shamik Basu and Pratiksha Pratiksha has a small role in CP implementation. Whereas the MIP was majorly handled by Pratiksha and Shamik has a supporting role. The other two techniques SAT and SMT were completed by both of us together.

Some major issues while solving the problem with different techniques were-
CP- In CP, we faced issues in implementing the designing of the non overlapping constraint for the rotation case as we cannot use the global constraint for non overlapping constraint in rotation.
SAT- Encoding the primitive inequalities for constraining the width and height of overall plate and designing the three literal constraints for non overlapping constraint were difficult to implement in SAT model.
SMT- In SMT, almost everything is same as SAT so there were no separate issues while implementing the model. The only addition here is the boolean value for the rotation part.
MIP- While printing the output files for the rotation part of MIP model, the execution became very slow from instance 19 so we had to proceed with not printing the files for the rotation part. We were able to obtain till instance 18 but later we had to stop printing the output files to proceed.

The overall working time taken in completing the project was approximately 45 days.

2 CONSTRAINT PROGRAMMING - CP Model

The first technique to solve the VLSI problem is CP. We are using MiniZinc modelling language to solve the problem. The main idea behind this approach is that we will state the constraints and a solver is used to solve those constraints. The solution to a CP is a set of values to be assigned to the variables that satisfies all the constraints defined. We will solve the problem using CP in two versions: one without rotation and second will be with rotation. In the first one, the rotation of the chips is not allowed whereas in the second approach the chips can be freely rotated to find the final solution. In both the cases our main aim is to minimize the silicon plate's height while finding best optimum placement for the chips.

2.1 DECISION VARIABLE

For every chip, we declare 2 variables which contain all the possible values for x and y coordinates (bottom left corner):

- $y \epsilon [0, \sum_{n}^{i=1} height_i]$
- $x \, \epsilon \, [0, \, W]$
- $rotation_i \, = \, array \, [1..N] \, of \, var \, 0...1 :$

  where $rotation_i = 1$ if $circuit_i$ is rotated
  $= 0 otherwise$

OTHER VARIABLES:

- W = width of the silicon plate
- n = number of circuit boards
- width_i - width of the chip i
- height_i - height of the chip i

We have below arrays of integer for each circuit dimension:

The left bottom corner of the Bigger rectangular plate has been assumed to have co-ordinate (0,0).
This is the reference frame for measuring the co-ordinates of the circuit boards

2.2 OBJECTIVE FUNCTION

We stack the circuit boards upwards. So our main goal or objective is to reduce that stacking height. So we take the maximum height by adding the height of each circuit with the y co-ordinates.

We defined the objective function like adding the possible y-coordinates with the heights of the circuit boards as if we are stacking the boards on above another. Now in this way the Height is the maximum height. Now solver will try to minimize the Height using the constraints which are defined later in the project.

where ***height****- height of the circuits*

The objective function is H which we try to minimize:

$$max\{\sum_{n}^{i-1}(y_i + height_i)\}$$

2.3 CONSTRAINTS

We have used multiple constraints in the implementation of CP using both Chuffed and Gecode. All are mentioned below with detailed description.

2.3.1 CUMULATIVE CONSTRAINT

We apply Cumulative constraint to improve the constraint propagation. The below two cumulative constraints helps in finding a feasible solution. This is similar to a task scheduling problem:

$$constraint\ cumulative(x,\ width,\ height,\ Height):$$

where,

- circuit-boards = tasks
- x = start time of the task
- width = duration of task
- height = resource usage
- Height = the variable that we intend to minimize

$$constraint\ cumulative(y,\ height,\ width,\ W):$$

where,

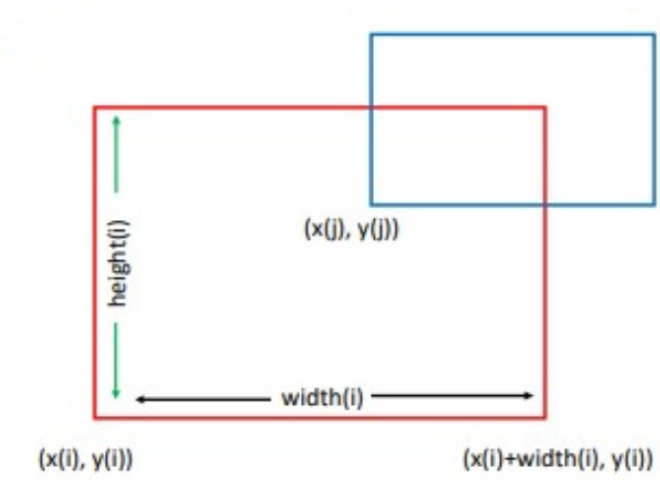
- circuit-boards = tasks
- y = start time of the task
- height = duration of task
- width = resource usage
- W = fixed

2.3.2 NO-OVERLAP CONSTRAINT

There are possibly four types of overlaps to occur. We will first discuss the cases and then after solving the cases, we will take a negation of all the equations to form the constraint.

OVERLAP SCENARIO:

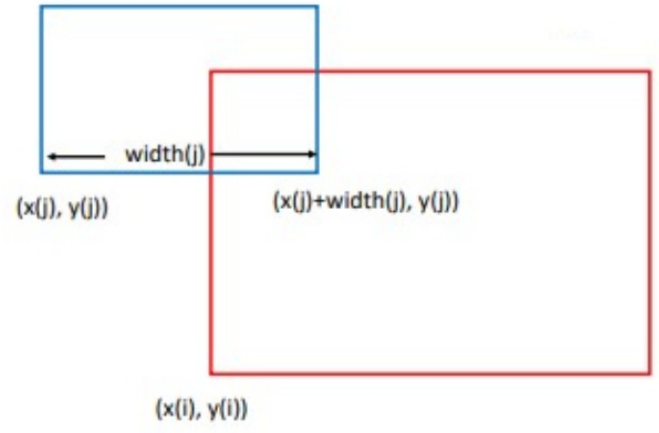
Case-1:



Here the red and the blue rectangle will overlap when:

$$x_i + width_i > x_j$$
$$y_i + height_i > y_j$$

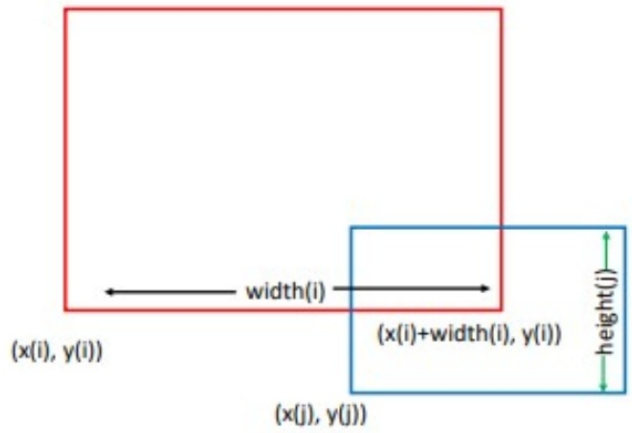
Case-2:



Here the red and the blue rectangle will overlap when:

$$x_i < x_j + width_j$$
$$y_i + height_i > y_j$$

Case-3:

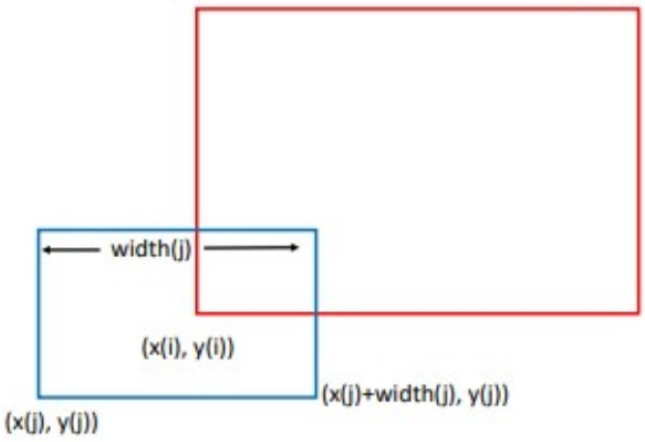


Here the red and the blue rectangle will overlap when:

$$x_i + width_i > x_j$$

$$y_i < y_j + height_j$$

Case-4:



Here the red and the blue rectangle will overlap when:

$$x_i < x_j + width_j$$

$$y_i < y_j + height_j$$

DERIVING THE CONSTRAINT:

So, to avoid overlap of circuits we need the no-overlap constraint and to achieve this we will consider the negation of above four cases and will get the below-

$$\neg(\, x_i + width_i > x_j \wedge y_i + height_i > y_j \wedge x_i < x_j + width_j \wedge y_i < y_j + height_j$$
$$\vee$$
$$x_i\, width_i <= x_j \vee y_i + height_i <= y_j \vee x_i >= x_j + width_j \vee y_i >= y_j + height_j$$
$$\vee$$
$$x_i <= x_j \vee y_i <= y_j \vee x_i >= x_j \vee y_i >= y_j)$$

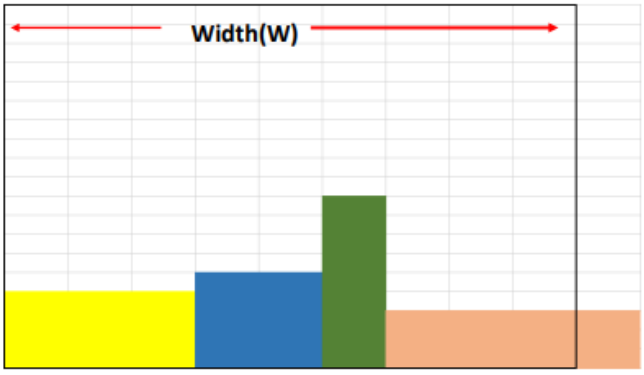
So, the final **Over-Lapping Constraint** is:

$$(x_i + width_i \leq x_j \vee x_j + width_j \leq x_i$$
$$y_i + height_i \leq y_j \vee y_j + height_j \leq y_i$$

There are two situations when the circuit boards are placed in such a manner that it can go beyond the maximum allowed capacity. We need some kind of mechanism to avoid such kind of mechanisms:

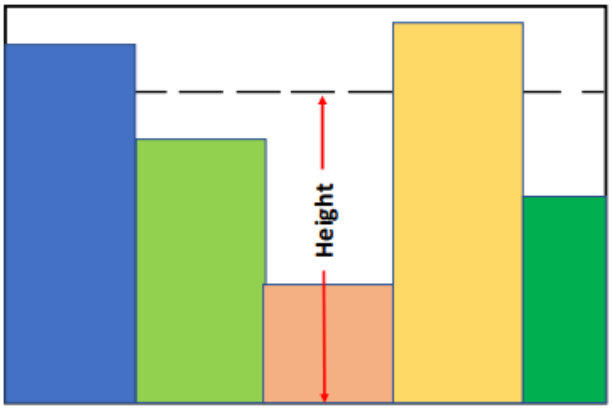
Case-1:

In this case, the circuits placement goes beyond the maximum possible width of the bigger rectangular plate.



Case-2:

In this case, the circuit placement goes beyond the maximum possible height of the bigger rectangular plate. This height is required to be minimized as part of our optimization problem.



If there is a violation then it means that there is at-least one circuit-board for which **x-coordinate** + **width** and/or **y-coordinate** + **height** is greater than **w** or **height** respectively. So, if we force the reverse inequality for ALL cases, we can avoid such issues.

$$x_i + width_i \leq W \qquad (\text{ where i} = [1..n])$$
$$y_i + height_i \leq height \qquad (\text{ where i} = [1..n])$$

The first equation ensures that the set of slabs can never go beyond the maximum possible width and the second equation ensures the same for the height (which we have to minimize).

2.3.3 GLOBAL CONSTRAINT

We used the diffn global constraint for the purpose of ensuring non-overlap because previous approach involves using **OR** which slows down constraint propagation. In our first attempt, we used a different approach as already explained and have re-written below for the ease of reference-

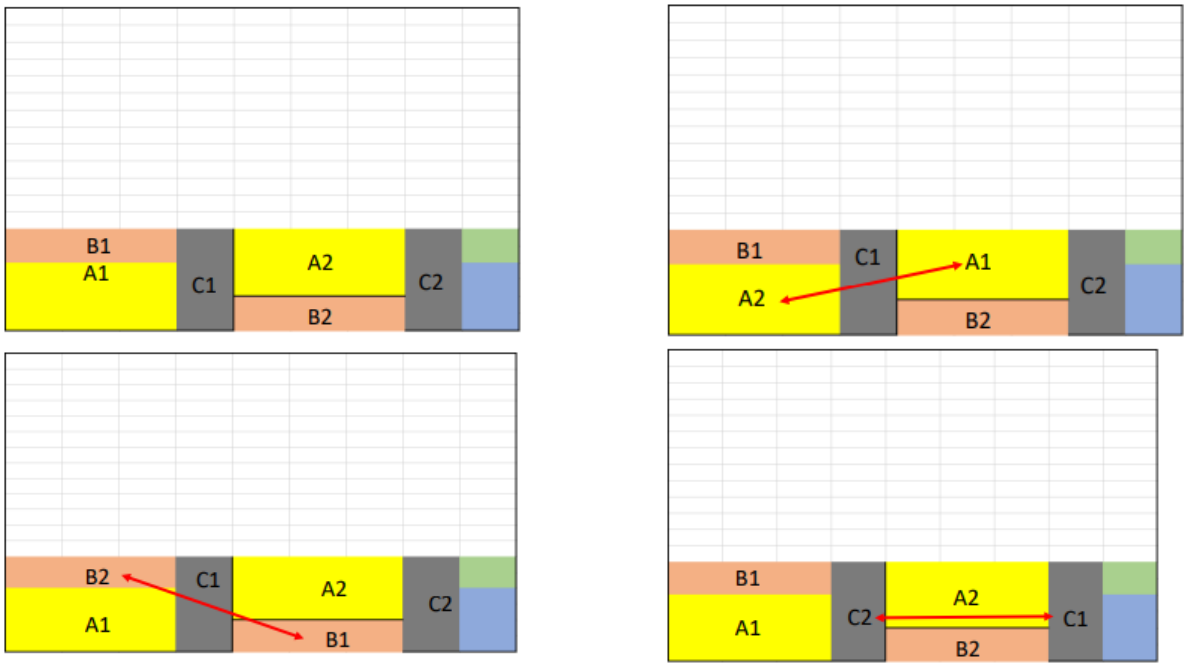
$$constraint\ diffn\ (x,\ y,\ width,\ height)\ :$$

In the predicate diffn[6], we have a constraint rectangle 'i' which is to be non-overlapping. Zero-width rectangles can still not overlap with any other rectangle.

2.3.4 SYMMETRY BREAKING CONSTRAINT

VLSI solution sets can have multiple symmetries. We use this constraint to reduce the solver's work by reducing the number of final solutions. To do this we will keep only the non-symmetric solutions. There are different types of symmetric solutions-

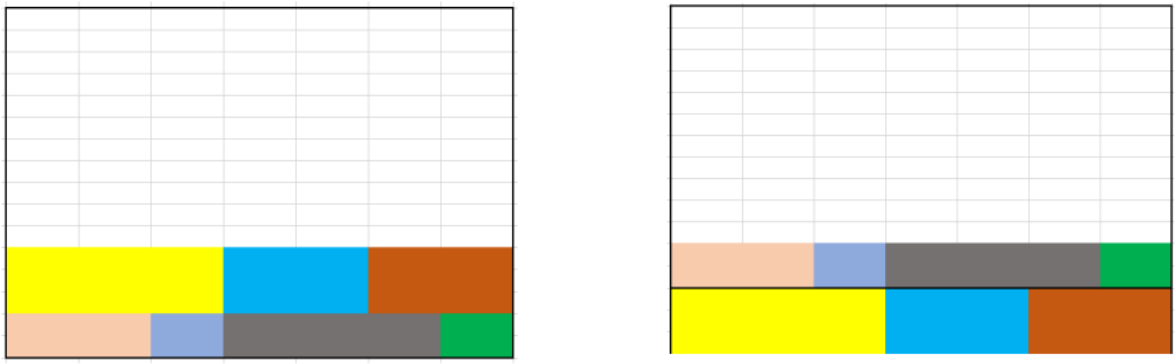
Type-1 - Diagonal Here we get similar solutions in diagonal form (as depicted in the below figures)



Here we intend to check if there are any two circuit boards of equal length and width. If we find any such case, then the Lexicographical constraint will be used to ensure there is no swapping. The Lexicographical constraint used here is lex_greater but lex_lesser can also be used.

$$if\ (width_i == width_j \wedge height_i == height_j) \quad (i \neq j)$$
$$then lex\_greater\ (x_i,\ y_i,\ x_j,\ y_j)$$

Type-2 - ROW SWAP Here we get similar solutions in horizontal form (as depicted in the below figures)



If there is swapping of rows ever, it would necessarily imply at least in one of the two scenario, at least one case, a smaller circuit board will be below a larger one. If however, all circuit boards are of exact same width but different heights, swapping of rows would take place. But of course, a large chunk of many sub-cases where swapping of rows might otherwise would have happened can be avoided for sure.

$$\begin{gathered} \forall_{i,j} \epsilon [1,\ n] \\ if\ width_i\ <\ width_j \qquad (\mathrm{i} \neq j) \\ then, lex\_greater((x_i,\ y_i),\ (x_j,\ y_j)) \end{gathered}$$

2.4 ROTATION

In this approach, we are allowed to rotate the circuits to get optimized solutions. In this case, the circuits are allowed to be rotated at an angle of 90°. After the rotation, the height of the circuit becomes the width and the width becomes the height. Here we have to make sure that for a given instance, both the rotated and the un-rotated circuit should not be present in the same plate. We are using all the constraints used in the non-rotation case with some modifications.

Here we try to compute the minimum of sum of heights if all the circuit-boards are being stacked up vertically. The logic is, if width is greater than the height of a particular circuit-board, we consider the height or else the width (like rotating the board) and like this if we keep on stacking on circuit-board on the top of another, the total height that we shall obtain is what we called the **min_sum_height**.

$$F(H) = \begin{cases} \sum\limits_{n}^{i=1} height_i,\ height_i\ <\ width_i \\ \sum\limits_{n}^{i=1} width_i,\ otherwise \end{cases}$$

$min\_sum\_height = F(H)$

$$y_i\ =\ array\,[1..N]\,of\ var\ 0...min\_sum\_height - 1$$

$$rotation_i\ =\ array\,[1..N]\,of\ var\ 0...1:$$

where $rotation_i = 1$ if $circuit_i$ is rotated
=0 otherwise

Now we add this to the width constraint from the no-rotation case. Here we are trying to ensure that under no circumstances, the set of slabs can go beyond the maximum permissible width.

$$(\ x_i\ +\ width_i\ )\,(\ 1\ -\ rotation_i\ )\ \leq\ Width$$

$$(\ x_i\ +\ height_i\ )\,rotation_i\ \leq\ Width \qquad \text{(this is the swapped scenario)}$$

Now we want to make sure that under no circumstances, the set of slabs can ever go beyond the maximum height (Height - the variable we are trying to minimize).
If the height and width are not swapped, then the equation should stay same and if they are swapped, we must use width instead of height.

$$(\, y_i \, + \, height_i \,)\,(\, 1 \, - \, rotation_i \,) \, \leq \, Height$$

$$(\, y_i \, + \, width_i \,)\, rotation_i \, \leq \, Height \qquad \text{(this is the swapped scenario)}$$

We are adding the rotation variable in the non-overlapping constraint. Here if the width and height are not swapped, the equation stays the same and if they are swapped, the height has to be used instead of width and vice-versa.

$$\begin{array}{c} (((x_i\, width_i)\,(\, 1 \, - \, rotation_i \,) \, \leq \, x_j) \wedge (x_i \, + \, height_i \,)\,(\, rotation_i \,) \, \leq \, x_j \,) \\ \vee(((x_j \, + \, width_j)\,(\, 1 \, - \, rotation_i \,) \, \leq \, x_i) \wedge (x_j \, + \, height_i \,)\,(\, rotation_j \,) \, \leq \, x_i \,)) \\ \vee(((y_i \, + \, height_i)\,(\, 1 \, - \, rotation_i \,) \, \leq \, y_j) \wedge (y_i \, + \, width_i \,)\,(\, rotation_i \,) \, \leq \, y_j \,)) \\ \vee(((y_j \, + \, height_j)\,(\, 1 \, - \, rotation_j \,) \, \leq \, y_i) \wedge (y_j \, + \, width_j \,)\,(\, rotation_j \,) \, \leq \, y_i \,)) \end{array} \qquad (\text{i} \neq \text{j})$$

Now we are redefining the objective function. If there is no rotation, the equation remains the same as the previous one but in case of rotation, we consider the width instead of height. The equation is mentioned below-

$$max\{[((y_i \, + \, height_i)\,(1 \, - \, rotation_i)) \, + ((y_i \, + \, width_i)\,(rotation_i))]\}$$

2.5 VALIDATION

We have specified a search strategy. A search strategy decides how the search should be done. Below we have described the search strategy for the solvers we have used.

We observe from the above tables and plots that time taken increased for chuffed when we applied the symmetry breaking constraint but remained same for gecode (for both rotation and no-rotation case).

Also we can observe from the tables that for the rotation part, gecode takes more time than chuffed for both with and without symmetry breaking constraint. Whereas for the no-rotation part, there is no much difference for the with symmetry breaking constraint. For the implementation without symmetry breaking constraint, chuffed has taken less time than gecode.

2.5.1 CHUFFED

Chuffed is based on lazy clause generation. Lazy clause generation is a hybrid approach to constraint solving that combines features of finite domain propagation and Boolean satisfiability. Chuffed adapts some techniques from SAT solving and can often be fasters than aa lot of other CP solvers.

We have implemented both no-rotation and rotation using chuffed and below are the details of the functions. The main aim here is to minimize the Height. The two cases (with and without rotation) are implemented as follows-

NO-ROTATION

$$int\_search(x_i,\ random\_order,\ indomain\_min,\ complete)$$

ROTATION

$$int\_search(rotation,\ random\_order,\ indomain\_min,\ complete)$$

2.5.2 GECODE

Gecode supports integer, set and float. We are using Gecode because it supports many constraints and search annotations that are not present in Minizinc standard library.
Below is the explanation of rotation and non-rotation function aimed at minimizing Height using GECODE.

NO-ROTATION

$$int\_search(x_i,\ dom\_w\_deg,\ indomain\_random,\ complete)$$

ROTATION

$$int\_search(rotation,\ dom\_w\_deg,\ indomain\_random,\ complete)$$

- **x_i** = 1D array (also decision variable) passed to intsearch
- **rotation** = 1D array (also decision variable) passed to intsearch
- **random order** = choose the variable with the smallest value of domain size divided by weighted degree, which is the number of times it has been in a constraint that caused failure earlier in the search
- **dom_w_deg** = Choose the variable with largest domain, divided by the number of attached constraints weighted by how often they have caused failure
- **indomain_min** = assign the variable its smallest domain value
- **indomain_random** = assign the variable a random value from its domain
- **complete** = it refers to an exhaustive exploration of the search space

EXPERIMENTAL DESIGN

We have implemented our model using both chuffed and gecode on no-rotation and the rotation case. We did not used the global constraint in the non-rotation part and were facing difficulties in the propagation but after adding the global constraint-diffn in the non-rotation part we got more instances solved. But in rotation part it wasn't possible to use the diffn so we had to explicitly accommodate the rotation constraints in the non-overlapping constraints. We have also used the Implied constraint for better propagation.
At last we have integrated the minizinc file using the python API to make the experiment reproducible and also set the timeout int the script. The model files are no_rotation_with_sym.mzn for using chuffed. no_rotation_with_sym_gecode.mzn for using gecode. rotation_with_sym.mzn for using chuffed with rotation and rotation_with_sym_gecode.mzn for using gecode with rotation. In order to check with and without symmetry we manually commented out the symmetry breaking constraints in each of these files.

For reproducibility execute :

- python CP_Integration.py
- provide model path:
- specify solver:
- specify instance folder path:

Hardware specification-
Processor 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
Installed RAM 8.00 GB (7.79 GB usable)
System type 64-bit operating system, x64-based processor

Software Specification-
Python 3.9.7
MiniZinc to FlatZinc converter, version 2.6.0, build 473934157
z3 0.2.0 0.2.0
z3-solver 4.11.2.0 4.11.2.0

Time Limit-
30 seconds

EXPERIMENTAL RESULTS

First we have implemented the model in Chuffed and Gecode without the symmetry breaking constraint and later with the constraint. Below are the histograms showing difference between time taken by with and without symmetry in chuffed and gecode Implementation for no-rotation case.

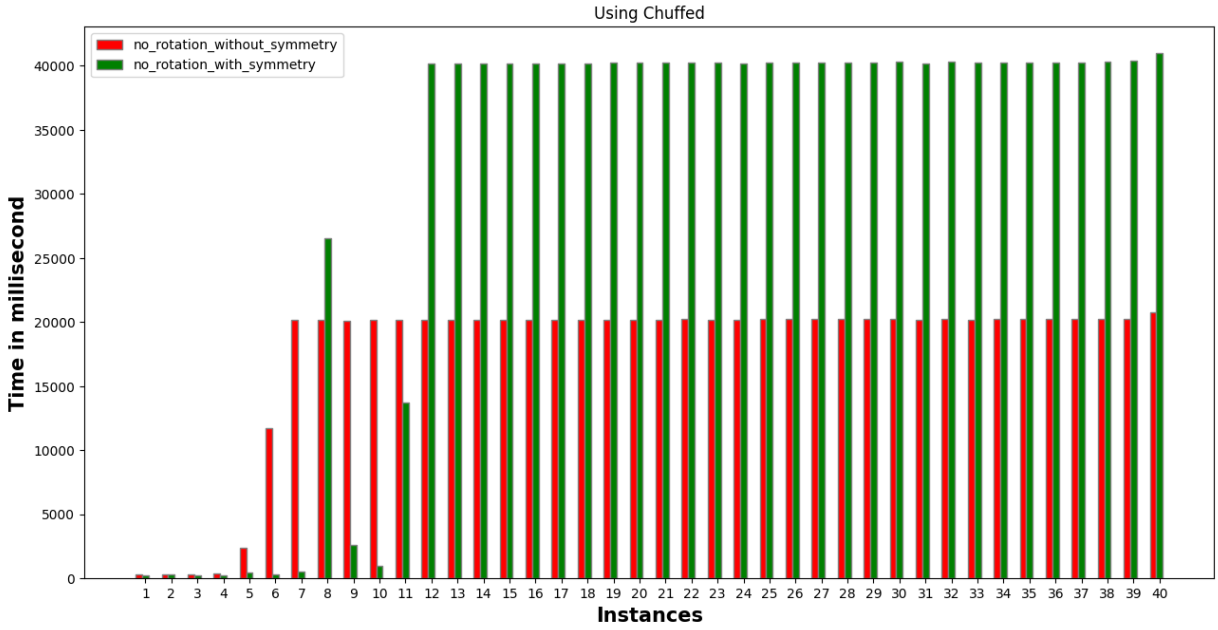
NO-ROTATION				
ID	**Chuffed w/o SB**	**Chuffed + SB**	**Gecode w/o SB**	**Gecode + SB**
1	269.03	252.89	249.01	271.22
2	269.08	263.93	283.14	272.01

Table 1 continued from previous page

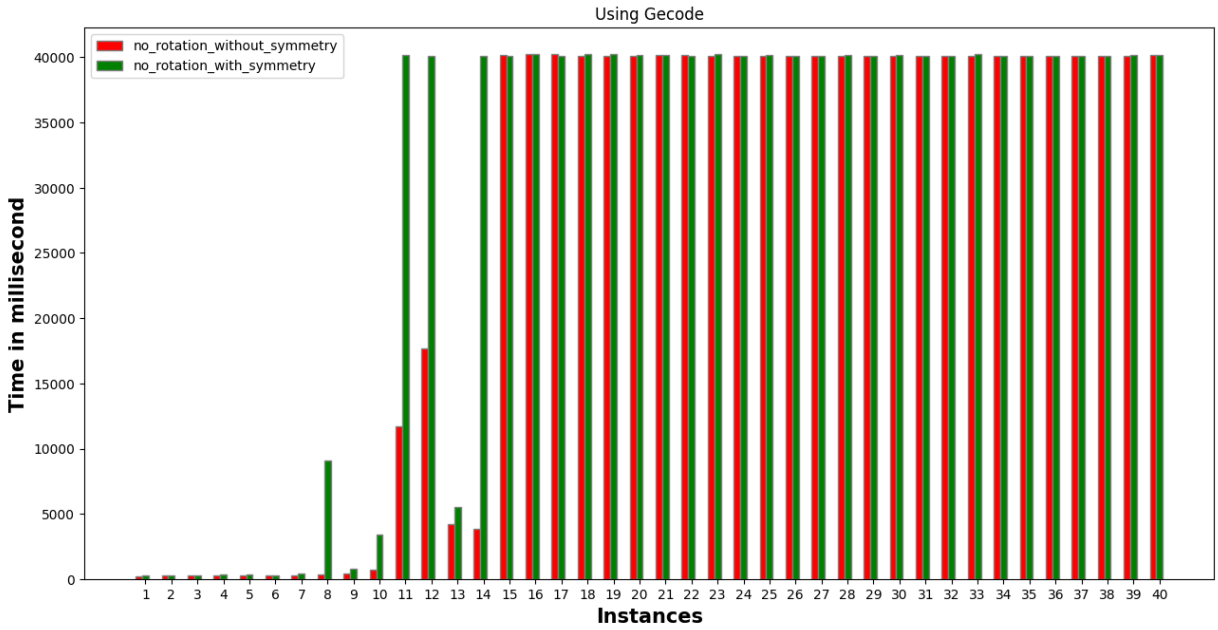
3	284.52	250.18	276.54	293.7
4	396.66	260.16	300.45	345.35
5	2348.44	440.82	281.97	377.78
6	11682.68	326.46	296.34	259.05
7	20145.07	517.72	317.61	450.68
8	20138.5	26525.42	348.85	9062.96
9	20116.76	2599.84	459.73	785.42
10	20144.37	982.75	729.56	3448.03
11	20135.81	13744.49	11697.63	40134.95
12	20149.02	40142.3	17707.17	40112.46
13	20139.74	40138.23	4245.47	5565.5
14	20142.74	40150.57	3877.85	40127.06
15	20167.6	40168.23	40139.53	40083.97
16	20162.69	40174.2	40252.36	40239.77
17	20163.79	40179.91	40245.82	40121.77
18	20170.77	40172.23	40119.33	40249.94
19	20185.49	40218.4	40118.7	40243.01
20	20189.29	40239.21	40119.51	40135.31
21	20165.96	40252.26	40128.49	40146.95
22	20205.85	40247.98	40130.7	40112.39
23	20178.32	40216.45	40121.42	40229.77
24	20170.95	40184.51	40111.49	40107.07
25	20257.34	40259.67	40089.65	40131.56
26	20224.22	40224.38	40104.1	40115.09
27	20202.09	40213.64	40125.3	40105.64
28	20202.09	40229.61	40100.24	40144.81
29	20217.07	40216.32	40109.36	40120.43
30	20230.01	40295.68	40103.41	40140.56
31	20163.76	40167.36	40103.79	40117.07
32	20251.12	40329.92	40100.13	40110.77
33	20176.57	40216.4	40093.27	40241.76
34	20203.49	40228.73	40105.4	40106.1
35	20202.64	40249.25	40101.73	40118.25
36	20210.43	40249.03	40100.38	40126.83
37	20230.75	40270.46	40097.12	40107.61
38	20248.33	40314.82	40104.45	40126.66
39	20261.58	40367.07	40106.9	40131.84
40	20723.48	41003.43	40133.06	40146.59

ID - Instance number time - millisecond

No Rotation - With and Without Symmetry | Chuffed



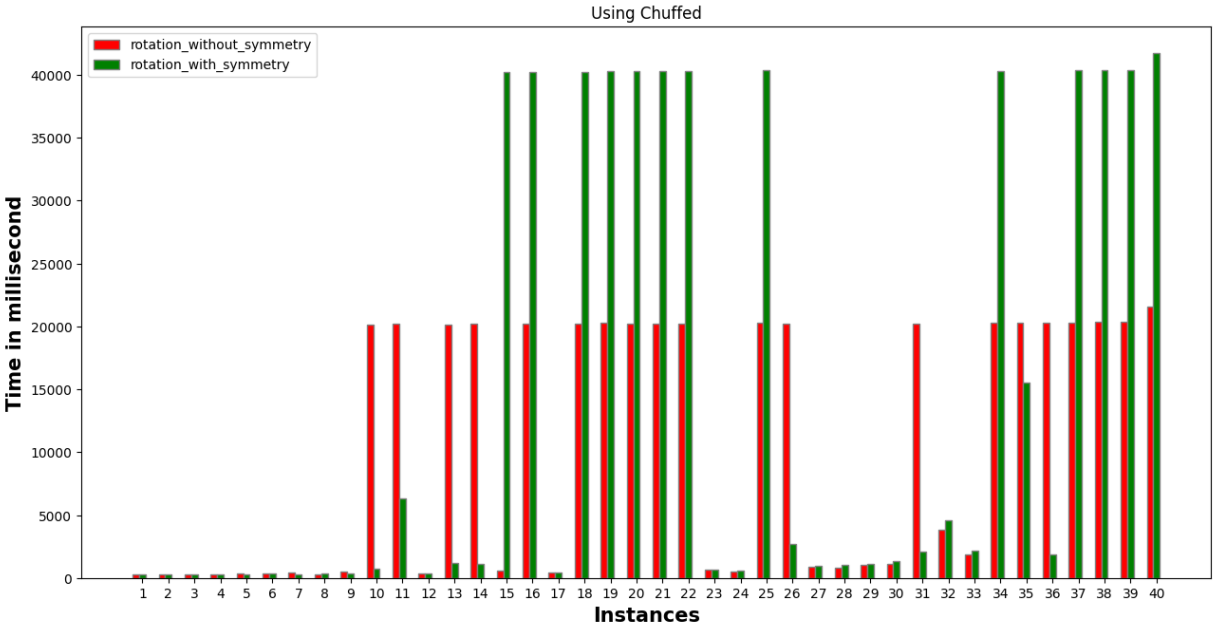
No Rotation - With and without Symmetry | Gecode



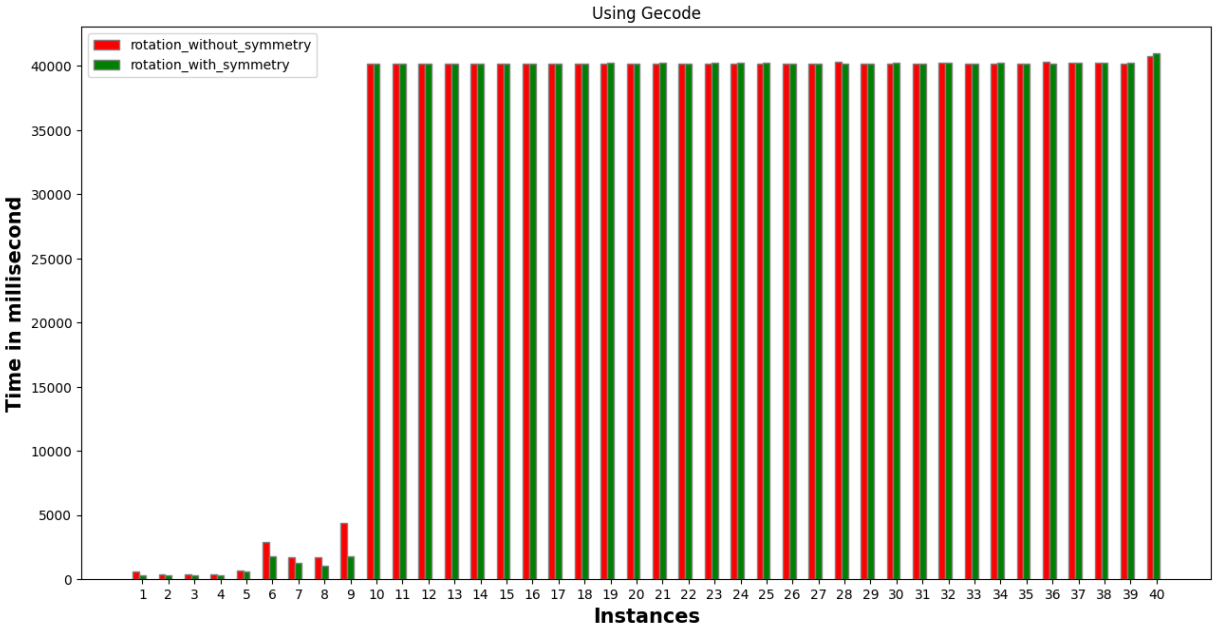
Now, these are the timings for the rotation case.

ROTATION				
ID	**Chuffed w/o SB**	**Chuffed + SB**	**Gecode w/o SB**	**Gecode + SB**
1	297.98	295.76	568.41	287.01
2	304.06	273.46	378.18	272.4
3	298.1	296.58	397.56	315.58
4	307.66	316.86	407.55	321.05
5	345.98	331.05	649.17	570.33
6	413.3	362.58	2887.72	1804.28
7	425.25	318.2	1673.04	1293.95
8	316.97	356.67	1719.3	1067.78
9	507.83	377.5	4342.42	1786
10	20150.95	759.77	40144.92	40144.67
11	20215.97	6375.13	40154.48	40177.75
12	393.52	360.65	40156.03	40168.03
13	20172.32	1209.53	40150.6	40191.41
14	20188.68	1154.15	40144.63	40208
15	588.27	40179.55	40154.27	40195.18
16	20198.43	40214.05	40155.03	40162.5
17	459.42	468.33	40145.13	40164.89
18	20211.3	40242.46	40159.9	40182.35
19	20268.79	40271.48	40185.27	40218.86
20	20229.1	40319.8	40174.43	40205.81
21	20242.32	40262.19	40189.6	40219.88
22	20253.53	40277.2	40189.09	40199.51
23	660.6	681.52	40161.72	40224.93
24	545.89	606.19	40164.75	40213.55
25	20303.04	40392.59	40209.46	40215.94
26	20252.24	2685.36	40181.13	40207.41
27	943.44	997.11	40162.61	40195.42
28	863.75	1072.48	40299.53	40199.47
29	1073.09	1159.5	40181.85	40203.83
30	1122.75	1376.72	40209.51	40281.75
31	20244.71	2144.73	40179.49	40182.83
32	3823.26	4632.2	40240.85	40236.81
33	1891	2174.29	40169.55	40178.8
34	20265.02	40298.46	40192.87	40216.95
35	20265.56	15572.85	40207.62	40209.46
36	20261.96	1858.45	40302.57	40192.33
37	20312.14	40375.98	40231.25	40224.7
38	20366.73	40386.39	40219.62	40213.55
39	20351.96	40371.9	40208.87	40225.19
40	21576.16	41736	40800.26	41003.99
ID - Instance number Time - millisecond				

Rotation - With and without Symmetry | Chuffed



Rotation - With and without Symmetry | Gecode



We observe from the above tables and plots that time taken increased for chuffed when we applied the symmetry breaking constraint but remained same for gecode (for both rotation and no-rotation case).

3 SATisfiability - SAT Model

The second technique to solve the VLSI problem is SAT. SAT is an abbreviation for the Boolean satisfiability problem: given a propositional formula the goal is to check if it is SATisfiable.
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 this project, because it 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 encoding [1]: given a simple constraint $x1 + 1 \leq x2$ ($x1,\ x2 \in$ 0, 1, 2, 3), this is encoded into set of primitive comparisons as follows:

$$\neg(x_2 \leq 0),\ (x_1 \leq 0) \vee \neg(x_2 \leq 1),\ \ (x_1 \leq 1) \vee \neg(x_2 \leq 2), (x_1 \leq 2)$$

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

$$\neg px_{2,0},\ px_{1,0} \vee \neg px_{2,1},\ px_{1,1}, \vee \neg px_{2,2},\ px_{1,2}$$

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 $max\_width * n\_blocks$ and $max\_height * n\_blocks$, representing the encoded coordinates x and y of the left **bottom** 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 $n\_blocks * nblocks$, 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 .

3.1 DECISION VARIABLE

In order to make them positive so we have added the following constraints:

$$\begin{aligned} px_{i,e} &= x - coordinate\ of\ all\ the\ chips \\ py_{i,f} &= y - coordinate\ of\ all\ the\ chips \\ lr_{i,j} &= true\ if\ chip\ i\ placed\ left\ of\ chip\ j \\ ud_{i,j} &= true\ if\ chip\ i\ placed\ lower\ to\ chip\ j \end{aligned}$$

3.2 OBJECTIVE FUNCTION

The substantial difference between CP and SAT is that SAT can only check if an assignment is SATisfiable or not. So here 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 of instance, **is the ratio of sum of the area the circuit boards to the width of the plate.** 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.

3.3 CONSTRAINTS

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

- Constraint given by order encoding
- Non-overlapping constraints

Due to **order encoding**, we have 2-literal axiom clauses (We are using this to implement the width and height constraints): 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 - $height + i$:

$$\neg px_{i,e} \vee px_{i,e+1}$$
$$\neg py_{i,f} \vee py_{i,f+1}$$

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

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

And, for each rectangle r_i, and for all integere and f such that $0 \leq$ e $<$ max_width - $width_i$ and 0 $\leq$ f $<$ max_height - $height_i$

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

With the aim of reducing 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:

$$\neg lr_{i,j} \wedge \neg lr_{j,i} \quad if \quad width_i + width_j > max\_width$$
$$\neg ud_{i,j} \wedge \neg ud_{j,i} \quad if \quad height_i + height_j > max\_height$$

3.3.1 SYMMETRY BREAKING CONSTRAINTS

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

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

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 VALIDATION

We observe from the below tables and histograms that the time taken is mostly same for all the instances even after using the symmetry breaking constraint. A very few instances have huge difference in time taken in getting optimized solution for both the cases (with and without symmetry breaking constraint). Three instances timed out when we implemented our model without symmetry breaking but two of them got solved after implementing the constraint and only one got timed out.

EXPERIMENTAL DESIGN

We have added a driver code for the experimental purposes for the SAT implementation and for the reproducability for the experiments where we have also set the timeout as 300 seconds. We are getting consistent results after running the experiments several times. For reproducibility execute: >python ' . Sat.py '
>Specify instance folder path

For implementing SAT, we have used order encoding and broke any inequality in primitive inequality (every constraint is an inequality). The primitive inequalities are like normal encoding and one inequality can be broken into various primitive inequalities. We have encoded these primitive inequalities into Boolean variables and then we apply the constraints. Here the width of the plate is constant and we have taken the height as a best possible height. We have used 3 and 4 literal constraints for taking care of the non-overlapping constraints. Wherever we get a case of unsatisfiability, we increase the height by 1 and again check the satisfiability. After observing the performance of the model on instances, we have implemented symmetry breaking but there wasn't much improvement.

Hardware Specification-
Processor 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
Installed RAM 8.00 GB (7.79 GB usable)
System type 64-bit operating system, x64-based processor

Software Specification- Python 3.9.7
MiniZinc to FlatZinc converter, version 2.6.0, build 473934157
z3 0.2.0 0.2.0
z3-solver 4.11.2.0 4.11.2.0

Time out- 300 seconds

EXPERIMENTAL RESULTS

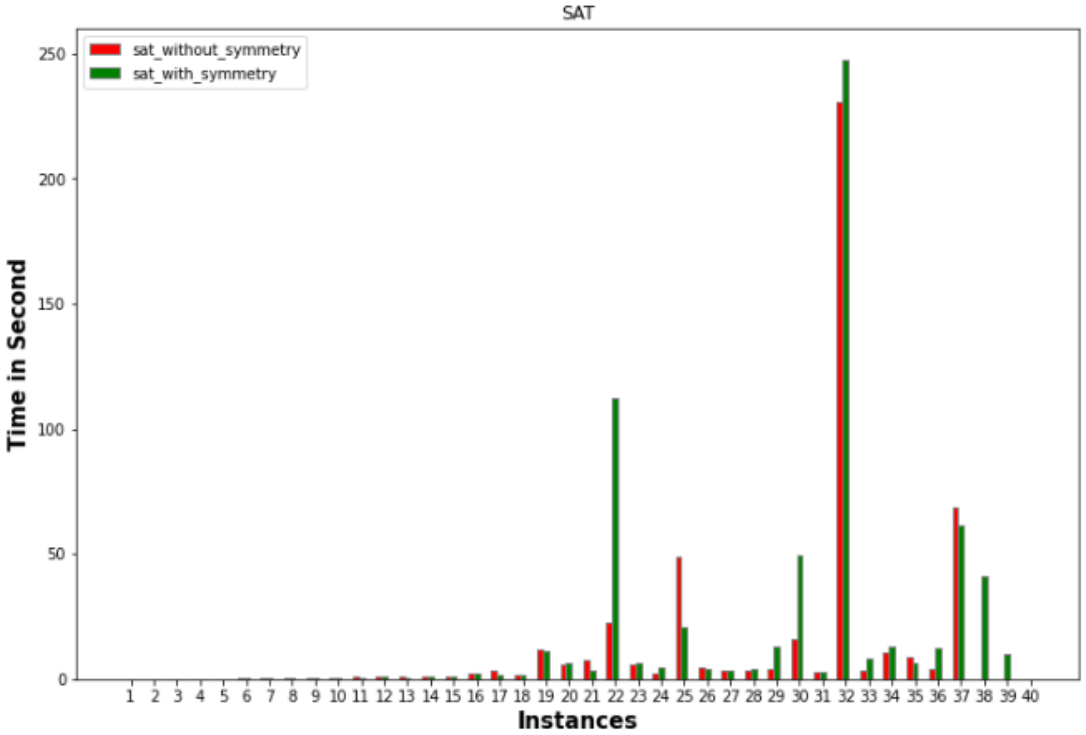
In these tables it is shown how much does the solver takes to compute the solution. We first implemented the model for the no-rotation case without the symmetry breaking constraint and later implemented the constraint and checked the performance. Moreover, it is shown how, sorting according the area, would help in the search of the solution.

NO-ROTATION		
ID	**Without SB**	**With SB**
1	0.03	0.03
2	0.05	0.05
3	0.06	0.05
4	0.09	0.09
5	0.14	0.14
6	0.2	0.18
7	0.21	0.25
8	0.28	0.25
9	0.31	0.28
10	0.5	0.41
11	1.01	0.77
12	0.94	0.83
13	0.78	0.65
14	1	0.88
15	1.08	0.92
16	2.33	2.07
17	3.63	1.51
18	1.94	1.52
19	12.1	11.1
20	5.75	6.45
21	7.83	3.73
22	22.55	112.53
23	5.9	6.43
24	2.51	4.94
25	48.93	20.55
26	4.42	4.03
27	3.46	3.59

Table 3 continued from previous page

28	3.6	3.78
29	4.07	12.98
30	15.95	49.69
31	2.72	2.71
32	230.42	247.61
33	3.18	8.41
34	10.81	13.12
35	9.06	6.34
36	3.85	12.61
37	68.69	61.59
38	TIMEOUT	40.91
39	TIMEOUT	10.26
40	TIMEOUT	TIMEOUT
ID - Instance number; Time - second		

SAT - With and without Symmetry



We observe from the above table and histograms that the time taken is mostly same for all the instances even after using the symmetry breaking constraint. A very few instances have huge difference in time taken in getting optimized solution for both the cases (with and without symetry breaking constraint).

Note- The last three instances timed out for without symmetry case.

4 SATisfiability Modulo Theories - SMT Model

The third technique to solve the VLSI problem is SMT. SMT is concerned with the study of SATisfiability of formulas using some theories. The language of SAT solvers if Boolean logic but for SMT it is FOL (Fist Order Logic).

4.1 DECISION VARIABLES

The variables defined for the SMT module are similar to those used in the previous parts (SAT). We have declared two arrays called x and y, containing the all possible co ordinates of the circuit boards.
In order to make them positive so we have added the following constraints:

$$\forall_i \epsilon [1,\ n\_blocks](x_i \geq 0)$$
$$\forall_i \epsilon [1,\ n\_blocks](y_i \geq 0)$$

rotation variable = whether the chip was rotated (only for the rotated part)

4.2 OBJECTIVE FUNCTION

We have declared the **max_height** which is the sum of the areas of each chip / the plate's width.

4.3 CONSTRAINTS

To solve successfully[2] 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:

$$\forall_i \epsilon [1,\ n\_blocks](x_i + width_i \leq max\_width)$$

While the second one as:

$$\begin{gathered}\forall_i \epsilon\,[1,\ n\_blocks], \forall_j \epsilon\,[i+1,\ n\_blocks] \\ (x_i + width_i \leq x_j) \vee \\ (x_j + width_j \leq x_i) \vee \\ (y_i + height_i \leq y_j) \vee \\ (y_j + height_j \leq y_i)\end{gathered} \tag{9}$$

4.3.1 LOWER BOUND OF HEIGHT

We have taken the $max\_heigth = lowerbound$. Then if the model is not satisfied then we increase the max_heigth by 1

4.3.2 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:

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

[2]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:

$$\begin{gathered}\forall_i \epsilon\,[1,\ n\_blocks], \forall_j \epsilon\,[i+1,\ n\_blocks] \\ x_i + width_i \leq x_j + W * (xb_{ij} + ybi, j) \wedge \\ x_i - width_j \geq x_j + W * (1 - xb_{ij} + ybi, j) \wedge \\ y_i + height_i \leq y_j + H * (1 + xb_{ij} - ybi, j) \wedge \\ y_i - height_j \geq y_j - H * (2 - xb_{ij} - ybi, j)\end{gathered}$$

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.

4.3.3 SYMMETRY BREAKING CONSTRAINTS

[2]We have tried to translate the same symmetry breaking constraint of SAT.
The first one will be rewritten as:

$$\forall_i \epsilon [1,\ n\_blocks], \forall_j \epsilon [i+1,\ n\_blocks]$$
$$((width_i + width_j > W) \Rightarrow ((x_i + width_i > x_j) \wedge (x_j + width_j > x_i)))$$

While the second one:

$$\forall_i \epsilon [1,\ n\_blocks], \forall_j \epsilon [i+1,\ n\_blocks]$$
$$((height_i + height_j > H) \Rightarrow ((y_i + height_i > y_j) \wedge (y_j + height_j > y_i)))$$

4.4 ROTATION

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°.

After that we can redefine the constraint as:

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

While to satisfy the **non-overlapping constraint**, we will need only to declare Max as the maximum between M and H. Here we have to use the non-overlapping constraint with logical OR. The constraint will be rewritten as:

$$\forall_i \epsilon [1,\ n\_blocks], \forall_j \epsilon [i+1,\ n\_blocks]$$
$$x_i + r_i * height_i + (1 - r_i) * width_i \leq x_j + Max * (xb_{i,j} + yb_{i,j}) \wedge$$
$$x_i - r_i * height_j - (1 - r_i) * width_j \geq x_j - Max * (1 - xb_{i,j} + yb_{i,j}) \wedge$$
$$y_i + r_i * width_i + (1 - r_i) * height_i \leq y_j + Max * (1 + xb_{i,j} - yb_{i,j}) \wedge$$
$$y_i - r_j * width_j - (1 - r_j) * height_j \geq y_j - Max * (2 - xb_{i,j} - yb_{i,j})$$

4.5 VALIDATION

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. 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.

From the histograms and tables in the RESULT section, we can observe that our model was able to solve the maximum number of instances in the no-rotation case when the symmetry breaking constraint was not applied. The addition of symmetry breaking constraint has reduced the number of solved instances for no-rotation. In addition, when symmetry breaking constraint is applied to the rotation case, we get most of the instances unsolved (they all timed out).

EXPERIMENTAL DESIGN

The implementation of SMT is a lot similar to SAT. The only addition here is the addition of the rotation case for which we have we had to modify some constraints but the motive of the constraint remained the same.

Hardware Specifications:

Processor 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
Installed RAM 8.00 GB (7.79 GB usable)
System type 64-bit operating system, x64-based processor
AND
Processor Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz 2.20 GHz **Installed RAM** 8.00 GB
System type 64-bit operating system, x64-based processor

Software Specifications:
Python 3.9.7
MiniZinc to FlatZinc converter, version 2.6.0, build 473934157

Time Out: 300 seconds

EXPERIMENTAL RESULTS

We have implemented the SAT model for both no-rotation and the rotation case. First we implemented without the Symmetry breaking constraint and later with the constraint to observe the difference in the timings. Below are timings for different instances.

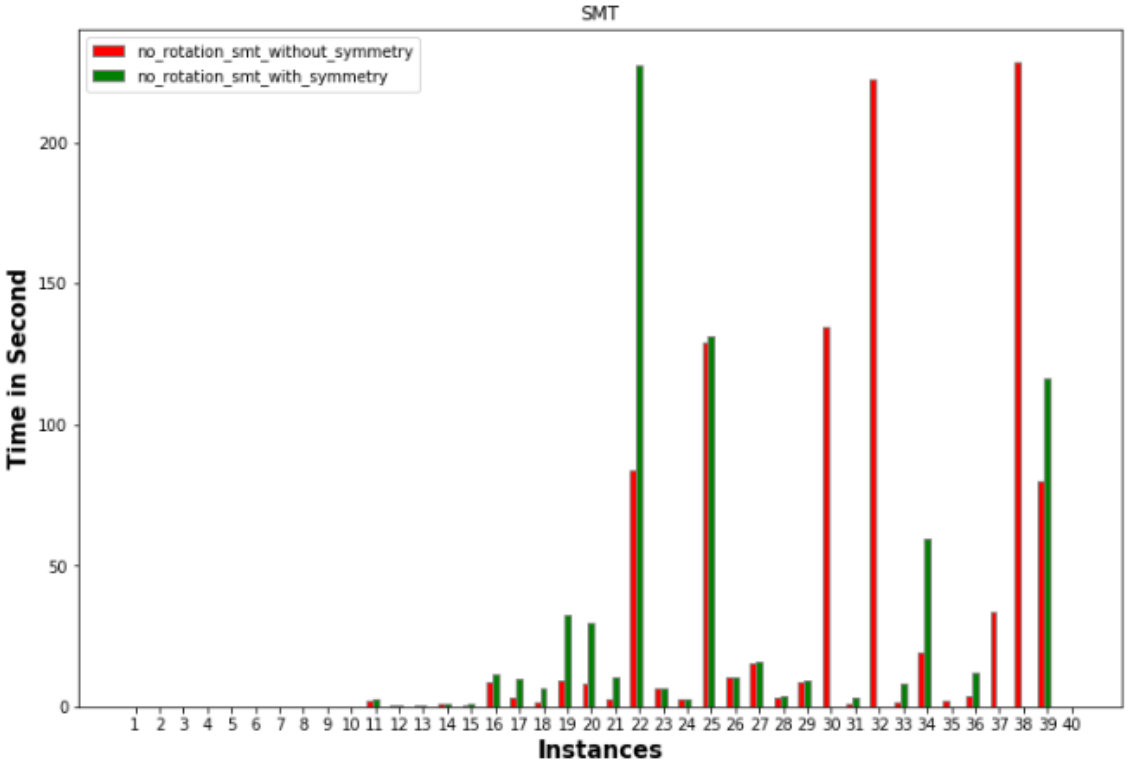
ID	No-Rotation w/o SB	No-Rotation + SB	Rotation w/o SB	Rotation + SB
1	0	0.02	0.08	0.02
2	0.02	0.03	0.03	0.05
3	0.02	0.03	0.06	0.06
4	0.03	0.05	0.26	0.3
5	0.04	0.05	0.11	1.01
6	0.08	0.11	0.11	0.14

Table 4 continued from previous page

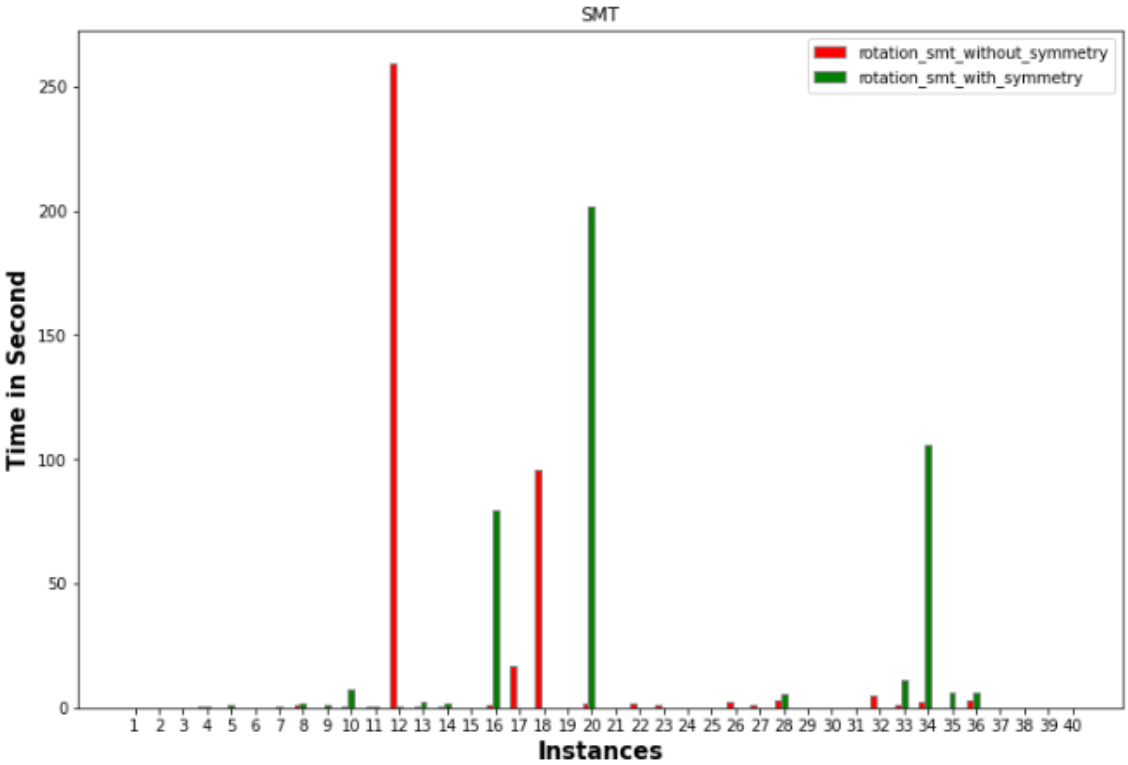
7	0.08	0.08	0.17	0.38
8	0.09	0.13	1.23	1.49
9	0.07	0.09	0.17	0.9
10	0.11	0.13	0.74	7.56
11	2.34	2.46	0.71	0.47
12	0.42	0.47	259.6	0.36
13	0.35	0.42	0.44	2.52
14	0.72	0.86	0.64	1.64
15	0.6	0.72	TIMEOUT	TIMEOUT
16	8.5	11.53	0.96	79.72
17	3.45	10.03	16.94	TIMEOUT
18	1.67	6.29	95.98	TIMEOUT
19	9.08	32.6	TIMEOUT	TIMEOUT
20	8.04	29.48	1.51	201.47
21	2.72	10.41	TIMEOUT	TIMEOUT
22	83.77	227.14	1.81	TIMEOUT
23	6.43	6.73	1.1	TIMEOUT
24	2.75	2.87	TIMEOUT	TIMEOUT
25	129.05	131.03	TIMEOUT	TIMEOUT
26	10.16	10.35	2.29	TIMEOUT
27	15.46	15.93	1.26	TIMEOUT
28	3.38	3.73	2.81	5.81
29	8.86	9.18	TIMEOUT	TIMEOUT
30	134.75	TIMEOUT	TIMEOUT	TIMEOUT
31	1.18	2.99	TIMEOUT	TIMEOUT
32	222.38	TIMEOUT	4.86	TIMEOUT
33	1.78	8.17	1.36	10.95
34	19.3	59.62	2.53	105.79
35	2.12	TIMEOUT	TIMEOUT	6.39
36	3.69	12.23	2.81	6.36
37	33.66	TIMEOUT	TIMEOUT	TIMEOUT
38	228.69	TIMEOUT	TIMEOUT	TIMEOUT
39	80.02	116.21	TIMEOUT	TIMEOUT
40	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT

ID - Instance number; Time - second

No Rotation - With and without Symmetry



Rotation - With and without Symmetry



5 MIXED INTEGER PROGRAMMING - MIP Model

The fourth technique to solve the VLSI problem is MIP. MIP is an abbreviation for Mixed-Integer Linear Programming. In MIP, the variables are integers and the objective function and equations are linear. To implement our model in MIP, we have used gurobi.

5.1 DECISION VARIABLES

x : the possible x-coordinates of the circuit boards
y : The possible y-coordinates of the circuit boards

OTHER VARIABLES: We have used the below variables in the solution for MIP :
n : number of circuit boards
dx : The width of each circuit boards
dy : The height of each circuit boards
width : width of the rectangular plate
safe_height = height*(1.3)
h_s : the all possible height of individual circuit boards
b : binary variable (used for reification)
best_height : sum of the area of all chips divided by plate width

5.2 OBJECTIVE FUNCTION

Here we have first encoded the constraints using reification. Then we linearized them using Big M method.We have fixed in addition to the maximum width, also the maximum height of the silicon plate. The maximum height is, for every of instance, **is the ratio of sum of the area the circuit boards to the width of the plate.**

The objective function is H which we try to minimize. We have selected the lower bound and the upper bound for the H as follows:

$$H \leq H\_s_i + (safe\_height - min(dy)) * (1 - b_i) : LowerBound$$
$$H \geq H\_s_i : UpperBound$$

5.3 CONSTRAINT

We have used the same non-overlapping and width contraints mentioned in the above section (CP, SMT). In addition to those, we have added and modified few more constraints to find the optimum solution using MIP. Here we have first encoded the constraints using reification. Then we linearized them using Big M method

5.3.1 Restricting the W and H

$$x_i + dx_i \leq width$$
$$y_i + dy_i == H\_s_i$$

5.3.2 Non-Overlapping Constraints

$$\begin{aligned}(x_i + dx_i \leq x_j \vee (dx_i + width)\,(1 - b_0)) \\ \wedge\,(x_j + dx_j \leq x_i \vee (dx_j + width) * (1 - b_1)) \\ \wedge\,(y_i + dy_i \leq y_j \vee (dy_i + safe\_height) * (1 - b_2)) \\ \wedge\,(y_j + dy_j \leq y_i \vee (dy_j + safe\_height) * (1 - b_3))\end{aligned} \qquad (\text{i} \neq \text{j})$$

Either the difference between the 2 board's x co-ordinates is $\leq$ width of the board or the plate width. Similarly, for height the difference should be either board height or the safe_height.

5.4 ROTATION

In order to consider rotation we have introduced a binary variable R. All the above constraints remain same and we have to add the rotation part of width and height with multiplying them with the rotation_i value. It is similar to what we did in CP.

$$x_i + (1 - rotation_i) * dx_i + (dy_i - dx_i) * rotation_i \leq width$$
$$H\_s_i - y_i * (1 - rotation_i) + (dy_i - dx_i) * rotation_i == dy_i$$

5.4.1 Non-Overlapping Constraints

$$
\begin{aligned}
&(x_i \ + \ dx_i * b_0 * (1 - rotation_i) \ + \ b_0 * dy_i * rotation_i \ \leq \ width * (1 - b_0) \ ) \\
&\wedge (x_j \ + \ dx_j * b_1 * (1 - rotation_i) \ + \ b_1 * dy_j * rotation_i \ \leq \ width * (1 - b_1) \ ) \\
\wedge &(y_i \ + \ b_2 * dy_i * (1 - rotation_i) \ + \ b_2 * rotation_i * dx_i \ \leq \ y_j \ + \ safe\_height \ - b_2 * safe\_height) \\
\wedge &(y_j \ + \ b_3 * dy_j * (1 - rotation_i) \ + \ b_3 * rotation_i * dx_j \ \leq \ y_i \ + \ safe\_height \ - b_3 * safe\_height)
\end{aligned}
$$

5.5 Implied Constraint

$$
\begin{aligned}
H \ &\geq \ best\_height \\
H \ &\leq \ safe\_height
\end{aligned}
$$

5.6 VALIDATION

The performance of MIP model is quite impressive. It takes very less time as compared to the other models. There is no symmetry breaking constraint in MIP and thus we have only two cases to analyse. We observe that the rotation case solves the instances faster than the no-rotation case for almost all the instances.

EXPERIMENTAL DESIGN

For implementing MIP, in addition to all the constraints in SMT, we have introduced a Boolean variable (b) encode the inequality constraints. And as MIP is non-linear, addition of this variable (b) allows us to linearize the problem by using the big m trick

Hardware Specifications:
Processor Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz 2.20 GHz **Installed RAM** 8.00 GB **System type** 64-bit operating system, x64-based processor

Software Specifications:
Python 3.9.7
gorobi 9.5.2

EXPERIMENTAL RESULTS

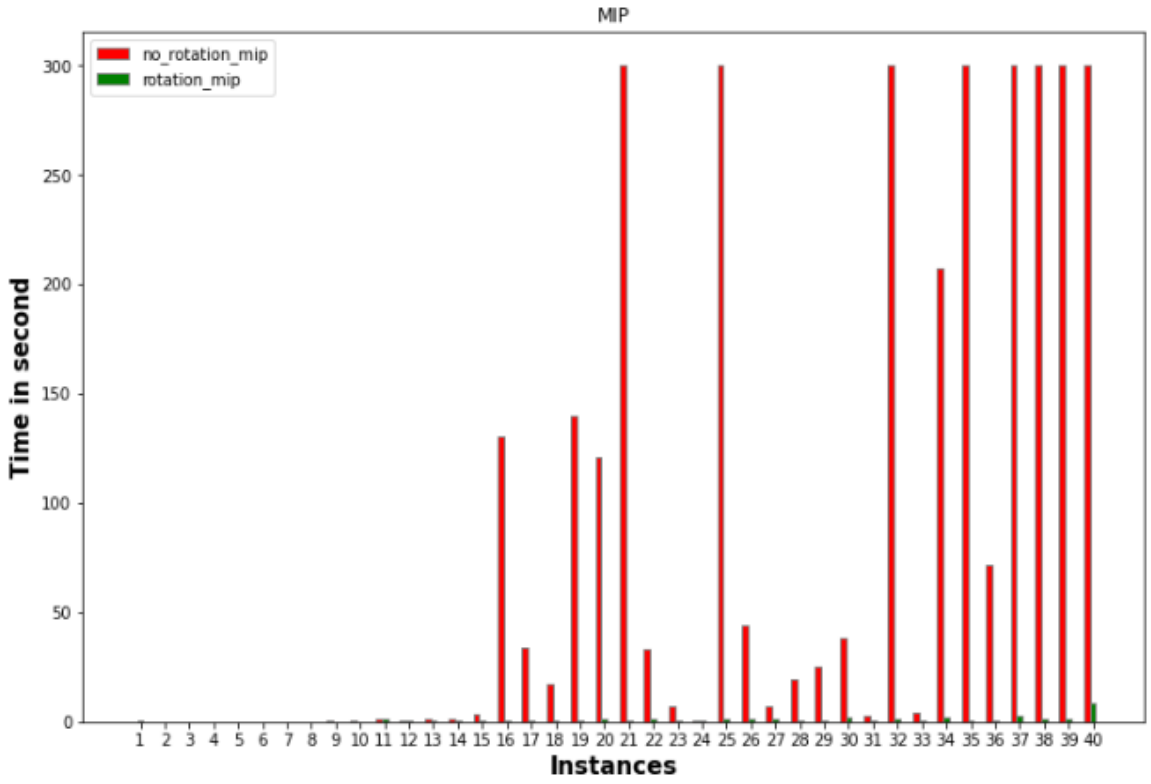
We have first implemented the model for without rotation case and later allowed rotation of the chips. Below are the timings for each instance-

ID	No-Rotation	Rotation
1	0.16	0.47
2	0.05	0.05
3	0.04	0.06
4	0.06	0.08
5	0.18	0.07
6	0.07	0.11
7	0.14	0.11
8	0.07	0.11
9	0.64	0.12
10	0.59	0.15
11	0.99	1.08
12	0.89	0.24
13	1.24	0.36
14	1.62	0.29
15	3.2	0.24
16	130.16	0.39
17	33.7	0.3
18	17.6	0.65
19	139.81	0.47
20	121.34	1.37
21	300.27	0.57
22	33.11	1.08
23	7.37	0.28
24	0.56	0.27
25	300.2	1.34
26	44	1.3
27	7.12	0.95
28	19.38	0.52
29	24.95	0.54
30	38.27	2.05
31	2.84	0.54
32	300.2	1.44
33	3.93	0.53
34	207.59	1.8
35	300.18	0.42
36	71.89	0.82
37	300.25	2.71
38	300.28	1.24
39	300.22	1.38

Table 5 continued from previous page

40	300.53	8.26
ID - Instance number; Time - second		

No Rotation - With and without Symmetry



From the above, we observe that the instances take a lot of time when we do not allow rotation. When rotation is allowed, the time decreases to almost negligible as compared to the no-rotation case.

6 CONCLUSION

Below is the comparative table and plot of best timings of all the four models. We can see that overall CP takes more time than other models to solve all the instances but it is the only model which is consistent throughout. Whereas MIP although not consistent has the best performance as it takes very less time as compared to the other models for solving most of the instances.

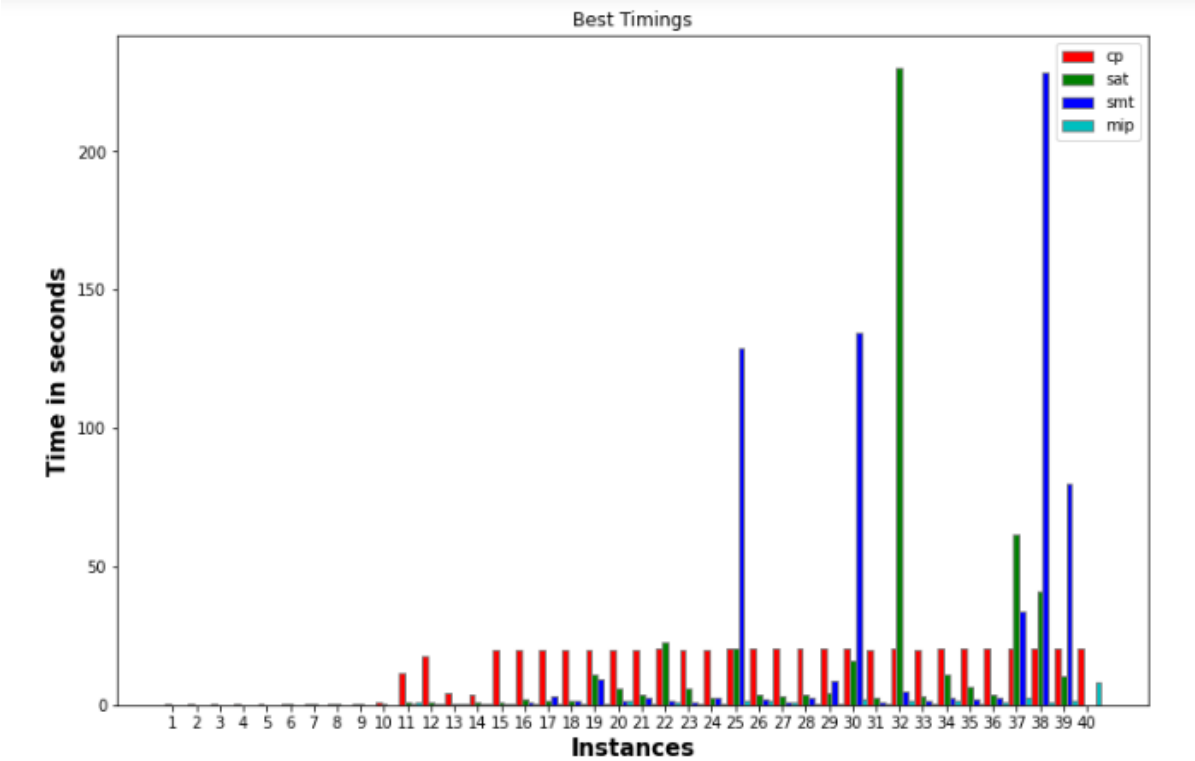
In all the four implementation, CP was the easiest and is also quite flexible as we just have to declare the objective function and optimize it. For the SAT and SMT model, we have to take a

best possible height and then try to compute the satisfiability. MIP had a different approach than all the other and we just linearized the SAT constraint using the big-m trick.

ID	CP (s)	SAT (s)	SMT (s)	MIP (s)
1	0.24	0.03	0	0.16
2	0.26	0.05	0.02	0.05
3	0.25	0.05	0.02	0.04
4	0.26	0.09	0.03	0.06
5	0.28	0.14	0.04	0.07
6	0.25	0.18	0.08	0.07
7	0.31	0.21	0.08	0.11
8	0.34	0.25	0.09	0.07
9	0.45	0.28	0.07	0.12
10	0.72	0.41	0.11	0.15
11	11.69	0.77	0.47	0.99
12	17.70	0.83	0.36	0.24
13	4.24	0.65	0.35	0.36
14	3.87	0.88	0.64	0.29
15	20.16	0.92	0.6	0.24
16	20.16	2.07	0.96	0.39
17	20.16	1.51	3.45	0.3
18	20.17	1.52	1.67	0.65
19	20.18	11.1	9.08	0.47
20	20.18	5.75	1.51	1.37
21	20.16	3.73	2.72	0.57
22	20.20	22.55	1.81	1.08
23	20.17	5.9	1.1	0.28
24	20.17	2.51	2.75	0.27
25	20.25	20.55	129.05	1.34
26	20.22	4.03	2.29	1.3
27	20.20	3.46	1.26	0.95
28	20.20	3.6	2.81	0.52
29	20.21	4.07	8.86	0.54
30	20.23	15.95	134.75	2.05
31	20.16	2.71	1.18	0.54
32	20.25	230.42	4.86	1.44
33	20.17	3.18	1.36	0.53
34	20.20	10.81	2.53	1.8
35	20.20	6.34	2.12	0.42
36	20.21	3.85	2.81	0.82
37	20.23	61.59	33.66	2.71
38	20.24	40.91	228.69	1.24
39	20.26	10.26	80.02	1.38
40	20.72	TIMEOUT	TIMEOUT	8.26

Table 6 continued from previous page
ID - Instance number; TIME = ms-milliseconds, s-seconds

Best Timings of Different Models



7 REFERENCE

[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 J´unior, Alvaro. (2017). *The Two-Dimensional Rectangular Strip Packing Problem* 10.13140/RG.2.2.27201.04965.

[6] https://www.minizinc.org/doc-2.5.3/en/lib-globals.html