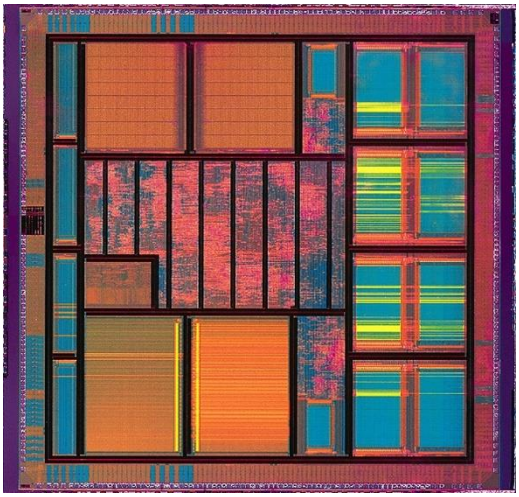
Alma Mater Studiorum - Università di Bologna

VLSI (Very Large Scale Integration) Design with CP, SAT and SMT technologies



Combinatorial Decision Making and Optimization - Module 1

2020-2021

Sandeep Kumar Kushwaha

sandeep.kushwaha@studio.unibo.it

Zarmina Ursino

zarmina.ursino@studio.unibo.it

Jyoti Yadav

jyoti.yadav@studio.unibo.it

Contents

Introduction

Description of the Problem VLSI (Very Large-Scale Integration) refers to the process of integrating circuits into silicon chips. This process is used in assembling the modern electrical motherboards, and circuits as they are efficient and space saving. 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 cell phone circuitry into a single silicon die (i.e., plate). This enabled the modern cell phone 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. While placing the circuits multiple factors are considered:

- Power dissipation of the chips on the board,
- Distance between the chips based upon their working interdependencies.
- Operating temperature of the chips,
- Dimension of the chips to be placed on the board.

As the combinatorial decision and optimization expert, it's assigned to design the VLSI of the circuits defining the electrical device given a fixed-width plate and a list of rectangular circuits, deciding how to place them on the plate so that the length of the final device is minimized (improving its portability). For the device to work properly, each circuit must be placed in a fixed orientation with respect to the others, therefore it cannot be rotated.

1. CP

Constraint programming is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, operations research, algorithms, graph theory and elsewhere. The basic idea in constraint programming is that the user states the constraints and a general-purpose constraint solver is used to solve them. In our VLSI problem, by the use of Minizinc [1], we are going to minimize the total height to place all the circuits in the plate. Their position is defined with the decision variables x and y, which represent the (x, y) coordinates of the bottom-left corner of each piece.

1.1. Model 1

In this model we implement a basic modelling, with the variables, the constraints, and the objective function. First, we define the decision variables as two array of coordinates which represent the bottom-left corner of each circuit.

```
array[N_PIECES] of var 0..fixed_width-min(horizontal_dim): x;
int: upper_bound = sum(vertical_dim);
array[N_PIECES] of var 0..upper_bound: y;
```

The ‘horizontal_dim’ and the ‘vertical_dim’ are the dimension of each piece.

```
array[N_PIECES] of int: horizontal_dim;
array[N_PIECES] of int: vertical_dim;
```

The other variables are defined as:

```
int: fixed_width;
int: n_circuits;
set of int: N_PIECES = 1..n_circuits;
```

Two constraints are implemented: the first to point out that all circuits must be inside the plate and thus they can’t exceed the fixed width; the second specifies that they must not overlap.

```
constraint forall (i in N_PIECES)(
  x[i] + horizontal_dim[i] <= fixed_width
);
constraint forall(i,j in N_PIECES where i < j) (
  x[i] + horizontal_dim[i] <= x[j] \/ x[j] + horizontal_dim[j] <= x[i] \/
  y[i] + vertical_dim[i] <= y[j] \/ y[j] + vertical_dim[j] <= y[i]
);
```

At last, we optimise the model by minimizing the height of the highest circuit.

```
var int: plate_height = max(i in N_PIECES)(y[i] + vertical_dim[i]);
solve minimize plate_height;
```

1.2. Model 2

If we draw a horizontal line and sum the horizontal sides of the traversed circuits, the sum can be at most the value of the 'fixed_width'. A similar property holds if we draw a vertical line. Therefore, we improved the previous model by creating two functions '*at_most_width*' and '*at_most_height*', whose computations are considered additional constraints.

In these functions we calculate dynamically the available space by summing the horizontal and vertical dimensions. Next, when we call them as constraint, we imply that all local widths and all local heights are less than the fixed given width and the plate height.

```
function var int: at_most_width(int: loc_h,
                                array[int] of var int: y,
                                array[int] of int: h,
                                array[int] of int: v) =
                                sum(i in N_PIECES) (if loc_h < y[i]+v[i] /\
loc_h >= y[i] then h[i] else 0 endif);

function var int: at_most_height(int: loc_w,
                                 array[int] of var int: x,
                                 array[int] of int: v,
                                 array[int] of int: h) =
                                 sum(i in N_PIECES) (if loc_w < x[i]+h[i]
/\ loc_w >= x[i] then v[i] else 0 endif);

constraint forall(i in 1..upper_bound) (
  at_most_width(i, y, horizontal_dim, vertical_dim) <= fixed_width
);
constraint forall(i in 1..width) (
  at_most_height(i, x, vertical_dim, horizontal_dim) <= plate_height
);
```

1.3. Model 3

We used global constraint [2], to represent high-level-modelling abstraction to obtain an efficient algorithm from the modelling and solving points of view. We used the predicates:

- '*diffn'* : it holds if for each shape of circuit there exists a dimension in which their projections doesn't overlap.

```
constraint diffn(x,y,horizontal_dim,vertical_dim);
```

- '*cumulative'*: to describe the cumulative resource usage

```
constraint cumulative(x, horizontal_dim, vertical_dim, plate_height) /\
           cumulative(y, vertical_dim, horizontal_dim, fixed_width);
```

Moreover, we used the search annotation ::int_search which means we should search by selecting from the array '*plate_height*', the variable with the smallest current domain (this is the *first_fail* rule), try setting it to its smallest possible value (*indomain_min* value selection) and the strategy supported is '*complete*', meaning an exhaustive exploration of the search space .

1.4. Model 4

To improve the search efficiency in CP, we exploited symmetry thanks to 'Symmetry Breaking Constraints' [2]. Thus, we added extra constraints imposing an ordering on the symmetric objects and only assignments satisfying the order are considered.

First, when two adjacent blocks share the same *x* dimension (so one of them is on top of another) or the same *y* dimension (so one of them is to the side of the other) they can be swapped.

```
constraint forall (i,j in N_PIECES where i != j) (
  y[i]==0 /\ y[j]==0 /\ x[i] < x[j] -> horizontal_dim[i] >= horizontal_dim[j]
);
constraint forall (i,j in N_PIECES where i != j) (
   x[i]==0 /\ x[j]==0 /\ y[i] < y[j] -> vertical_dim[i] >= vertical_dim[j]
);
```

Another symmetry is the ‘3 block symmetry’, which involves 2 adjacent circuits *i* and *j* with the same vertical dimension and both are adjacent to a bigger circuit, whose horizontal dimension is *xi +xj*. Thus the 2 circuits can be swapped with the bigger one and, analogously, we do the same if *i* and *j* share the same horizontal dimension.

```
constraint forall (i,j in N_PIECES where i != j) (
   y[i] == y[j] /\ vertical_dim[i] == vertical_dim[j] /\ horizontal_dim[i] >
horizontal_dim[j] -> x[i] < x[j]
 );

constraint forall (i,j in N_PIECES where i != j) (
   x[i] == x[j] /\ horizontal_dim[i] == horizontal_dim[j] /\ vertical_dim[i] >
vertical_dim[j] -> y[i] < y[j]
 );
```

Then the last symmetry is dealt. This one forces the first biggest block to be the bottom left of the second biggest block.

```
constraint forall (i in N_PIECES, j in N_PIECES where i < j) (
    horizontal_dim[i]==horizontal_dim[j] /\ vertical_dim[i]==vertical_dim[j] -
> y[i]*fixed_width+x[i] < y[j]*fixed_width+x[j]
 );
```

1.5. Model 5

In this last model, we allow the Rotation of the circuits: a *nxm* circuit is positioned as an *mxn* circuit in the silicon plate. Thus, the following constraint is implemented:

```
array[N_PIECES] of var int: h;
 array[N_PIECES] of var int: v;

constraint forall(i in N_PIECES) (
   h[i] == vertical_dim[i] /\ v[i] == horizontal_dim[i]
               );
```

1.6. Results and Performances

1.6.1. Compilation of models

In the python file ‘converter.py’ all the text files instances are ‘converted’ in Minizinc data files with the ‘.dzn’ extension and saved into the ‘instances_dzn’ folder. To run all the models described these data files are selected. In the bash file ‘compiler.sh’, the 5 Minizinc files are compiled, for instances 1, 2, 3, 4, 5, 6, 7, whose outputs are written for each model into the ‘out’ folder thanks to the command line [3]:

```
echo "time minizinc --solver gecode model$i.mzn $in --output-to-file $out --
time-limit 300000"
time minizinc --solver gecode src/model$i.mzn $in --output-to-file $out --time-
limit 300000
```

In this section, the results of the five models implemented are compared in terms of execution time. It can be seen that the first three models are not very efficient as the number of circuits increases, the execution time increases considerably; in fact, the symbol '--' indicates that the process have exceed a time limit of 5 minutes, hence they have been aborted . So, between model 4 and model 5, the former seems to be a good compromise.

Ins/Model	**Model-1**	**Model-2**	**Model-3**	**Model-4**	**Model-5**
Ins-1	423 msec	489 msec	638 msec	599 msec	605 msec
Ins-2	439 msec	514 msec	631 msec	658 msec	643 msec
Ins-3	449 msec	595 msec	723 msec	715 msec	667 msec
Ins-4	627 msec	574 msec	679 msec	721 msec	707 msec
Ins-5	2s 769msec	718 msec	796 msec	816 msec	820 msec
Ins-6	36s 588msec	716 msec	828 msec	868 msec	854 msec
Ins-7	49s 805msec	755 msec	849 msec	987 msec	892 msec
Ins-8	--	1s 23 msec	1s 72msec	1s 42msec	1s 119msec
Ins-9	--	911 msec	989 msec	1s 74msec	1s 12msec
Ins-10	--	1s 379msec	1s 110 msec	1s 233msec	2s 726msec
Ins-15	--	--	--	2s 36msec	16s 247msec
Ins-20	--	--	--	58s 4msec	--

Table 1, Comparison among different models

1.6.2. Plots

After compiling our Minizinc models via the compiler.sh file, we developed the graphical representation of the solutions in the python file *'plotter.py*' for each model's output file. Here we made use of the '*matplotlib.patches.Rectangle'* library [4] which helps us define a rectangle through an *xy* anchor point and through its width and height. For simplicity we show below the solutions for the first three instances even though solutions have been developed for more than half of the instances, as can be seen from Table 1.

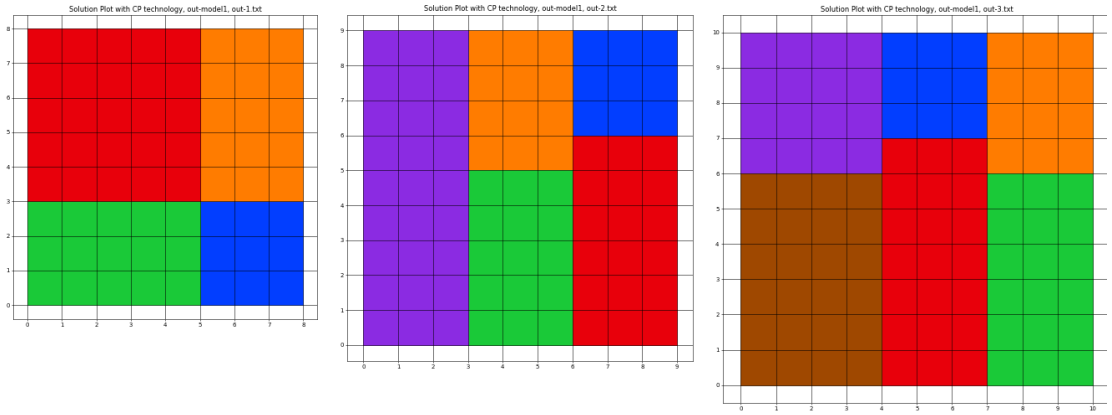


Figure 1 , Model 1 – output 1, 2, 3

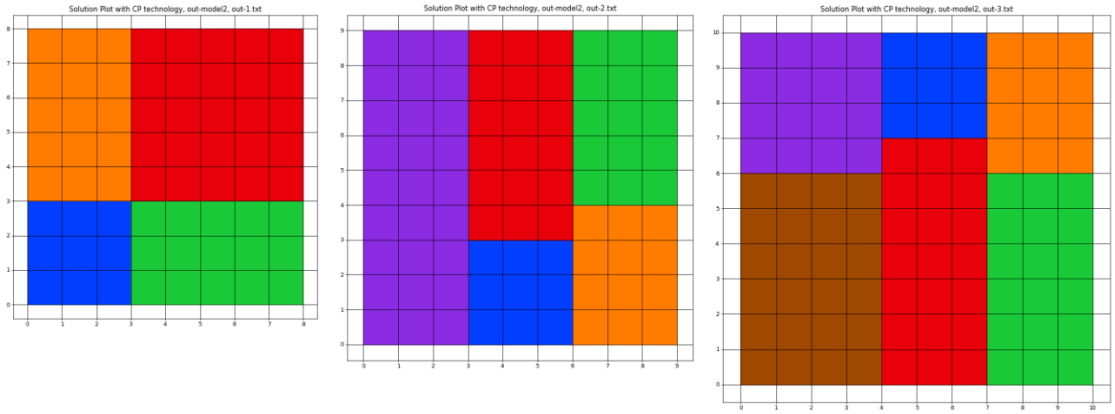


Figure 2 Model 2 - output 1, 2, 3

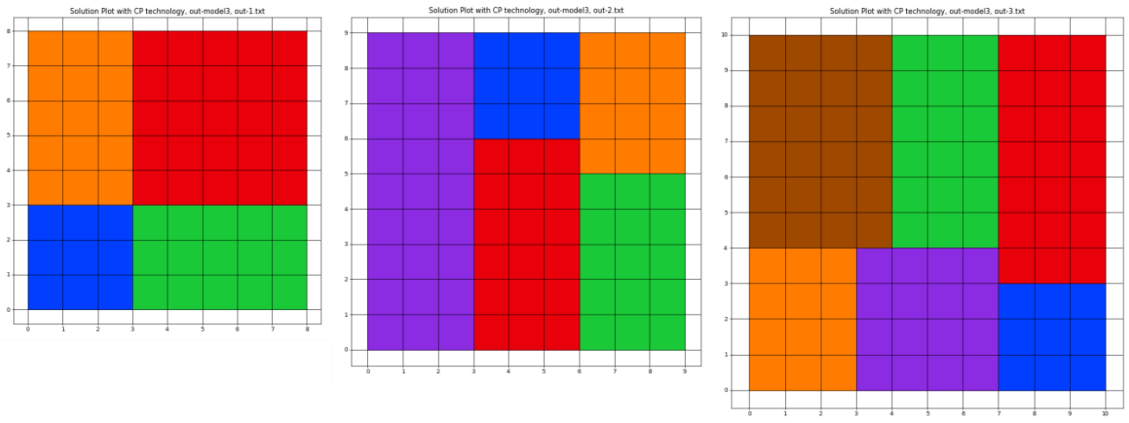


Figure 3 Model 3 – output 1, 2, 3

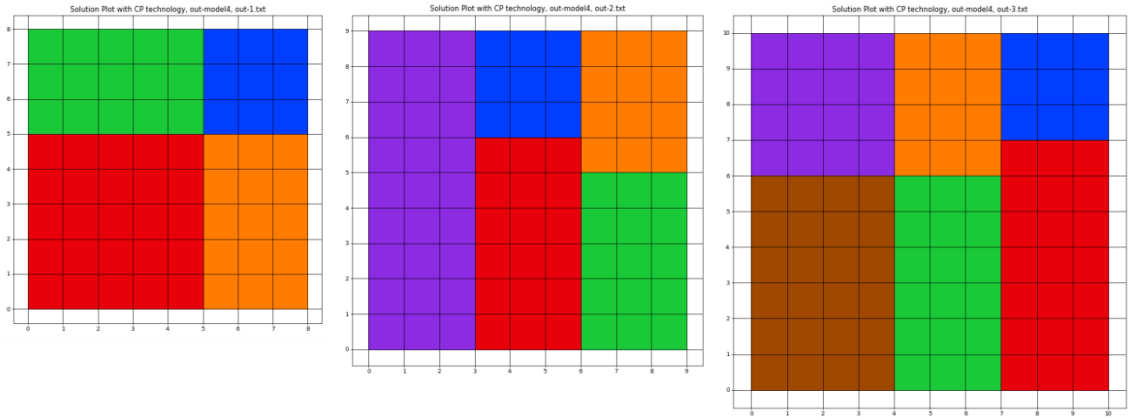


Figure 4 Model 4 – output 1, 2, 3

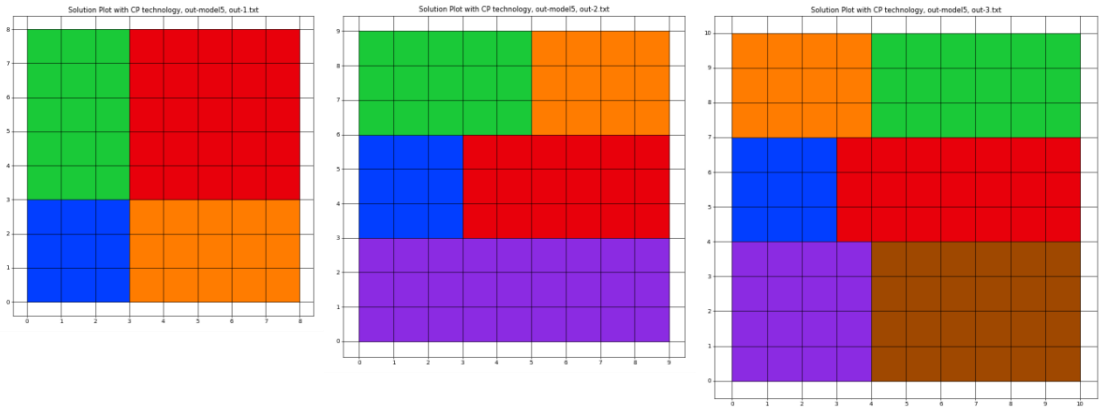


Figure 5 Model 5 – output 1, 2, 3

1.6.3. Hardware used

Our experiments were conducted on the machine with those specifications:

- Intel(R) Core(TM) i7-8550U CPU, 1.80GHz -1.99 GHz
- NVIDIA GeForce MTX 150
- The version of the interpreter used is Python 3.9.0 and Minizinc solver is Gecode 6.3.0

2. SAT

SAT refers to the Boolean Satisfiability problem where we want to determine if there exists an interpretation that satisfies a given Boolean formula. In other words, it establishes if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to true.

Satisfiability problem is to decide given a SAT formula is satisfiable or not. If the CNF SAT formula is satisfiable/consistent we would like to compute a satisfying assignment. A SAT solver is a tool that takes as input as CNF formula and outputs either a satisfying Boolean assignment to the variables used in the CNF formula is consistent or UNSAT if it is not. These solvers are typically binaries which accept input in the form of a text file with the CNF formula and write the relevant output to the console. We will describe the implementation of the VLSI (Very Large-Scale Integration) problem with the help of Z3.

Z3 is a state-of-the-art theorem prover developed at Microsoft Research and used in many real-world applications such as: software/hardware verification and testing, constraint solving, analysis of hybrid systems, security, and geometrical problems. A SAT solver is an algorithm that establishes satisfiability of formulas by taking them as input and returning either SAT if it finds a combination of values that can satisfy them or UNSAT otherwise.

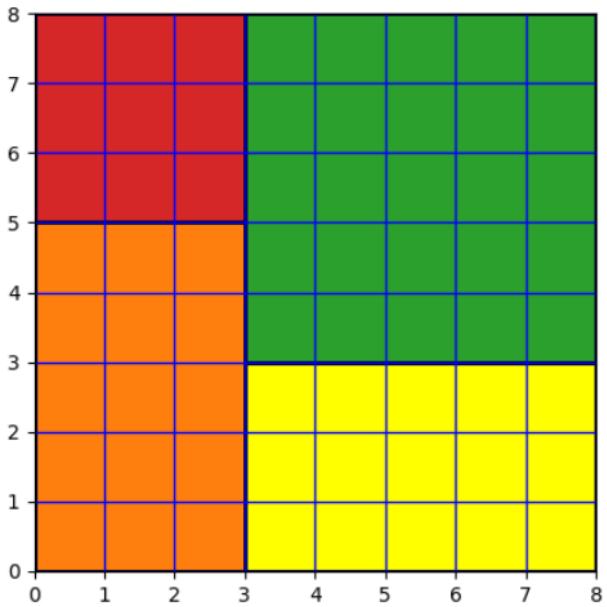


Figure 6, Output of an instance with w=8 and n=4

2.1. Input and Output

The inputs to be fed to the program consists of a set of 40 text files. Each text file contains the following information:
• The width of the silicon chip (width).
• The number of circuits (n).
• The width and height of each individual circuit (Wc, Hc ∀c in 1, ..., n).

$$
\begin{aligned}
8 \quad &\leftarrow width \\
4 \quad &\leftarrow n \\
3\ 3 \quad &\leftarrow W1 \ and \ H1 \\
3\ 5 \quad &\leftarrow \mathrm{W}2 \ and \ H2 \\
5\ 3 \quad &\leftarrow W3 \ and \ H3 \\
5\ 5 \quad &\leftarrow W4 \ and \ H4
\end{aligned}
$$

The program should read each text file and output the following information:
• The width and height of the silicon chip (width, height).
• The number of circuits (n).
• The width, height, lower left x-position, and lower left y-position of each individual circuit (Wc, Hc, Pxc, Pyc ∀c in 1, .., n).

$$
\begin{aligned}
8\ 8 \quad &\leftarrow width \ and \ height \\
4 \quad &\leftarrow n \\
3\ 3\ 0\ 0 \quad &\leftarrow W1\ , H1, Px1 \ and \ Py1 \\
3\ 5\ 0\ 3 \quad &\leftarrow \mathrm{W}2\ , H2, Px2 \ and \ Py2 \\
5\ 3\ 3\ 0 \quad &\leftarrow W3\ , H3, Px3 \ and \ Py3 \\
5\ 5\ 3\ 3 \quad &\leftarrow W4\ , H4, Px4 \ and \ Py4
\end{aligned}
$$

2.2. Decision Variables

Let's define some input variables:

- circuit widths is an array of the given widths (Wc ∀c having the range from 1, .., n) of all circuits.
- circuit heights is an array of the given heights (Hc ∀c having the range from 1, .., n) of all circuits.
- max height is the calculated upper bound of the silicon chip height. It is the sum of all the heights of the circuit for that instance.

$$maxHeight = \sum_{c=1}^{n} Hc$$

- min height is the calculated lower bound of the silicon chip height. W here Ac is the calculated area of the cth circuit.

  Ac = Wc ∗ Hc

$$minHeight = \frac{\sum_{c=1}^{n} Ac}{Width}$$

Following we specify that the output variable height ranges from min height to max height. To find the solution with the smallest height the SAT model will be used with increasing values of height.

The SAT model is defined as the 3D Boolean array plate[i][j][c]. The first and second dimensions (i, j) represent the coordinates of the plate and range from 1 to width and 1 to height respectively, the third-dimension c represents each circuit position and ranges from 1 to n.

plate = [[[Bool(f"plate_{i}_{j}_{c}") for c in range(n)] for j
→ in range(height)] for i in range(width)].

2.3. Constraints

- First constraint is the Non overlapping where At most one circuit can occupy each place of the plate.

$$\bigwedge_{i=1}^{width} \bigwedge_{j=1}^{height} \bigwedge_{0<c1<c2\leq n} \neg\Big(plate\ i,j,c1\ \wedge\ plate\ i,j,c2\Big)$$

- Every circuit must be placed in its complete form in at least one of its possible positions.

one possible position

$$\bigwedge_{c=1}^{n} \bigvee_{i=1,j=1}^{width(temp),height(temp)} \bigwedge_{ii=i}^{i+Wc} \bigwedge_{jj=j}^{j+Hc} plate\ (c,ii,jj),$$

$$\text{width(temp)} = \text{width} - \text{Wc} + 1,$$

$$\text{height(temp)} = \text{height} - \text{Hc} + 1$$

2.4. Symmetry

In order to further analyse the problem in order to understand if, from an erroneous solution, there are similar solutions that we can deuce as unsatisfiable as the are permutation or symmetrical of the erroneous one. This technique is called symmetry.
In symmetry breaking constraints placing the biggest circuit always on the coordinates (0,0). In this way, the biggest circuit is forced to be always at the bottom left corner of the plate and the smallest in the last. This reduces the number of symmetric solutions.

2.5. Rotation

For rotation we rotate the circuit by 90 degrees. Here we interchange the positions of width and height of the circuit.
A better solution consists in adding to the model a list of Boolean variables that is the same length as the number of circuits and keeps track of whether the circuit is rotated or not.

rotation = [Bool(f"r_i") for i in range(n)]

Given this variable, it's easy to find the actual width and height of each circuit (if the circuit is rotated width and heights need to be swapped) and using actual Wc and actual Hc instead of Wc and Hc in the constraints.

$$\forall c \text{ in } 1..n \text{ actual } Wc = \begin{cases} Hc, & \text{if rotation}[c] \text{ is true} \\ Wc, & \text{otherwise} \end{cases}$$

$$\forall c \text{ in } 1..n \text{ actual } Hc = \begin{cases} Wc, & \text{if rotation}[c] \text{ is true} \\ Hc, & \text{otherwise} \end{cases}$$

2.6. Results and Performances

2.6.1. Execution

We were able to solve 11/40 instances with the time constraint of 300 seconds.

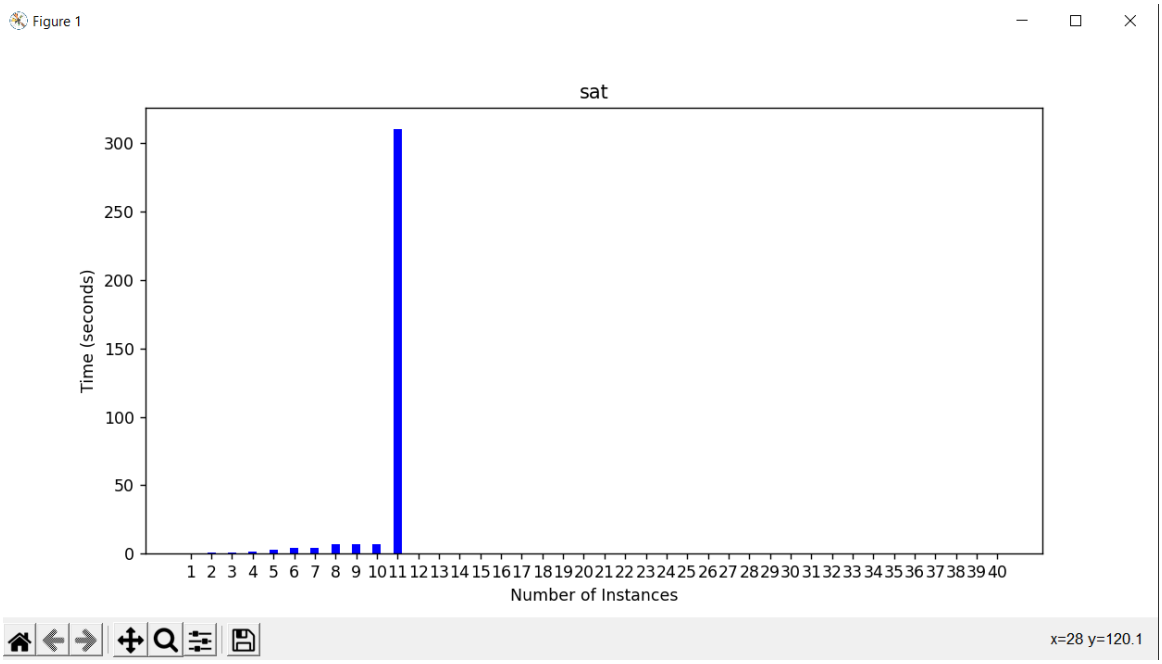


Figure 7, Execution time of instances

2.6.2. Hardware Used

Our experiments were conducted on the machine with those specifications:

- AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx
- NVIDIA GeForce GTX 1660 Ti with Max-Q Design GDDR6 @6GB (192 Bits)

The version of the interpreter used is Python 3.9.0 with z3-solver

2.7. Drawback of SAT

SAT solvers are very popular for their efficiency and performances. Their problem lies in the expressive power; encoding of SAT models can be complex and translation of problems in propositional logic can be expensive. Some problems are naturally expressed in logics rather than propositional logic.

3. SMT

Satisfiability modulo theories (SMT) is the problem of determining whether a mathematical formula is satisfiable. It generalizes the Boolean satisfiability problem (SAT) to more complex formulas involving real numbers, integers, and/or various data structures such as lists, arrays, bit vectors, and strings. The name is derived from the fact that these expressions are interpreted within ("modulo") a certain formal theory in first-order logic with equality (often disallowing quantifiers). In our project we have used Z3 Theorem Prover developed by Microsoft Research. Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers. Its main applications are extended static checking, test case generation, and predicate abstraction.

3.1. Modelling process

3.1.1. Constraints on the value of h

The chips should remain in limits of the board dimensions assumed. The value of minimum h is calculated by the following equation:

$$h_{min} = \lceil \frac{\sum_{i=0}^{n} x_i * y_i}{w} \rceil$$

$$h_{max} = \begin{cases} \sum_{i=0}^{n} y_i & when\, rotation\, is\, disabled \\ \sum_{i=0}^{n} x_i & when\, rotation\, is\, enabled \end{cases}$$

While execution of the code, if *h* is insufficient to accommodate all the chips in the PCB, we increment the value of *h* until constraints are satisfied.

3.1.2. Constraints of the maximum coordinate value for the chips on the coordinate system

The constraints on choosing the choosing the values of p_{xi} and p_{yi} are as follows:

$$(px_i + x_i + 1 \leq w) \wedge (px_i + x_i \geq 0) \wedge (py_i + y_i \geq 0) \wedge (py_i + y_i + 1 \leq h)$$

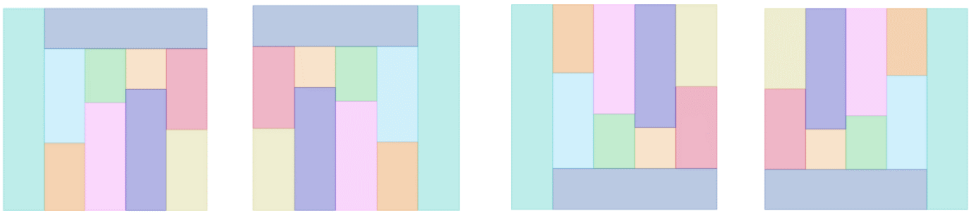
3.1.3. Constraints on the uniqueness of calculated coordinates

The calculate p_{xi}, p_{yi} should unique and shouldn't match with any other coordinate value of chip ie. p_{xj}, p_{yj}.

$$\forall_i \forall_j \in \{0..n\} : \quad (p_{xi} \neq p_{xj}) \wedge (p_{yi} \neq p_{yj})$$

3.1.4. Handling cases of symmetries during chip placement

While placing the chips, we can encounter multiple solutions, chances are the solutions can be symmetrical and hence, they are repetitive and not unique. The followings are the possible repetitions that can be obtained for the original image:



Original image Horizontal flip Vertical flip 180* rotation

Figure 8

To handle these situations, we will begin with placing the largest chip first, as it won't restrict the other chip's placement and will be more effective during the later stages. So, we sort the chips in size order and place the biggest chip first and smallest in the end.

3.1.5. Overlapping Constraints

To avoid the possible overlapping of the chips, we will add constraint on the coordinates for the chips in such a way that, considering the horizontal axis, sum of the x-coordinate & width of chip should be always less than or equal to the x-coordinate of other chips. Same can be implemented to the vertical dimensions.

$$\forall_i \forall_j \in \{0..n\} \wedge i < j : (p_{xi} + x_i \leq p_{xj}) \vee (p_{xj} + x_j \leq p_{xi}) \vee (p_{yi} + y_i \leq p_{yj}) \vee (p_{yj} + y_j \leq p_{yi})$$

3.1.6. Considering Rotational Cases

When we consider the rotation of the chips, we add more options during the placement of the chips on the PCB. In such cases, we will have additional *x* & *y* coordinates denoted by x_{ir}, $y_{ir.}$

$$x_{ir} = \begin{cases} x_i & when\, rotation\, is\, disabled \\ y_i & when\, rotation\, is\, enabled \end{cases}$$

$$y_{ir} = \begin{cases} y_i & when\, rotation\, is\, disabled \\ x_i & when\, rotation\, is\, enabled \end{cases}$$

3.2. Search Methods

We used z3 to solve the handle the cases and constraints applied to our problem. We are optimizing the model by reducing the number of possible solutions using constraints and hence the search space is reduced as more constraints are applied. In the base model, we are considering the chips in their original alignment. Whereas in the general model, we have considered those cases where the chips can be rotated as well.

3.3. Results observed

While running the models on multiple instances, we saw that that the base model performs well in the beginning when the complexity is less. But when more complex instances are provided, the base model struggles to provide solutions whereas the time taken to solve instances increases as we move forward with the instances. Whereas with rotation the general model, can find more solutions but in the beginning instances, the results take time. Whereas for complex cases, the results are better than the base models.

References

[1] The Handbook Minizinc, https://www.minizinc.org/doc-2.3.0/en/
[2] Propagation Global Constraint and Symmetry Breaking Constraints from Slides
[3] The Minizinc Command Line Tool,
https://www.minizinc.org/doc2.5.0/en/command_line.html
[4] Matplotlib Patches Rectangles,
https://matplotlib.org/stable/api/_as_gen/matplotlib.patches.Rectangle.html
[5] A New Method to Encode the At-most – One constraint into SAT,
"https://www.researchgate.net/publication/301455290_A_New_Method_to_Encode_the_At-Most-One_Constraint_into_SAT"

Bibliography

- *Symmetry Declarations for MiniZinc*, National ICT Australia, Victoria Laboratory, Department of Computer Science and Software Engineering, University of Melbourne, Australia https://people.eng.unimelb.edu.au/pstuckey/minisym/paper.pdf
- *"Scalable, High-Quality, SAT-Based Multi-Layer Escape Routing"*, Sam Bayless, Holger H. Hoos, and Alan J. Hu, University of British Columbia, Canada
- "SAT-SMT some examples", Dennis Yurichev, November 19 2021, https://sat-smt.codes/SAT_SMT_by_example.pdf
- Z3 issue, https://github.com/dvc94ch/pycircuit/issues/3