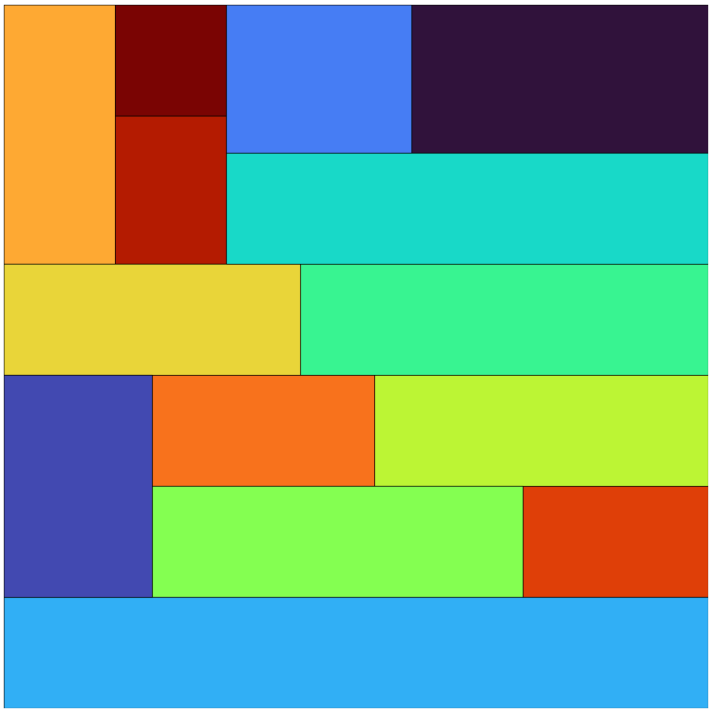
Combinatorial Decision Making and Optimization Project

VLSI Design

Daniele Morotti daniele.morotti@studio.unibo.it

Andrea Valente andrea.valente6@studio.unibo.it



Università di Bologna
Alma Mater Studiorum

Contents

1 Introduction

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.

We have to consider two variants of the problem. In the first, each circuit must be placed in a fixed orientation with respect to the others. This means that, an $n \times m$ circuit cannot be positioned as an $m \times n$ circuit in the silicon plate. In the second case, the rotation is allowed, which means that an $n \times m$ circuit can be positioned either as it is or as $m \times n$. In our case we decide to implement a solution with the following techniques:

- **Constraint Programming**;
- **Satisfiability Modulo Theories**;
- **Mixed-Integer Linear Programming**.

1.1 Format of the instances

Instance format An instance of VLSI is a text file consisting of lines of integer values. The first line gives w, which is the width of the silicon plate. The following line gives n, which is the number of necessary circuits to place inside the plate. Then n lines follow, each with x_i and y_i, representing the horizontal and vertical dimensions of the i-th circuit. For example, a file with the following lines:
9
5
3 3
2 4
2 8
3 9
4 12
describes an instance in which the silicon plate has the width 9, and we need to place 5 circuits, with the dimensions 3×3, 2×4, 2×8, 3×9, and 4×12.

For the Minizinc model we decide to implement a Python script that convert the .txt files into .dzn files in order to read directly the file from Minizinc.

Solution format Where to place a circuit i can be described by the position of i in the silicon plate. The solution should indicate the length of the plate l, as well as the position of each i by its $\hat{x_i}$ and $\hat{y_i}$, which are the coordinates of the left-bottom corner i. This could be done by for instance adding l next to w, and adding $\hat{x_i}$ and $\hat{y_i}$ next to x_i and y_i in the instance file. To exemplify, the solution of the previous instance could look like:
9 12
5
3 3 4 0
2 4 7 0
2 8 7 4
3 9 4 3
4 12 0 0
which says for instance that the left-bottom corner of the 3×3 circuit is at (4, 0).

Thus we create a .txt file for each solved instance following the above instructions and we also create a directory that contains all the plots for the solved instances (also for the non-optimal solutions).

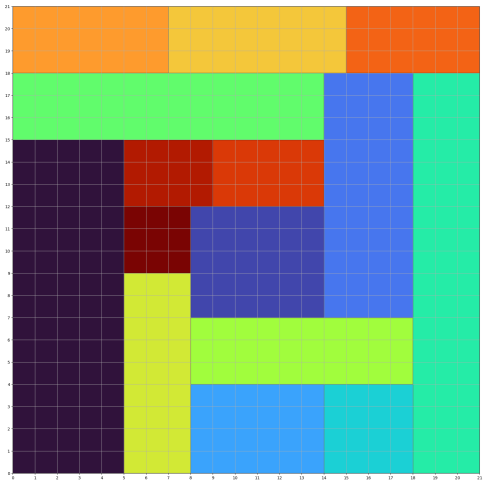


Figure 1: Example of a solved instance

1.2 Scripts and different models

For each model, after the implementation of the main constraints, we have considered different parameters in order to find the best set of options that allows us to get the maximum number of optimal solutions.

In order to simplify the testing part we wrote a **Python** script for each model that can be run using different available parameters and it gives as output the different files we have explained before.
The plots and the *.txt* files in the */out* directory of the delivered project have been generated considering the best set of parameters (i.e. they are the best solutions we were able to find) and more details about the different implementations and tests are delivered in the sections below.

We took inspiration from some scientific papers that we read online and which can be found in the references at the end of this report.

2 CP model

Constraint Programming is a declarative programming paradigm for stating and solving combinatorial optimization problems. The user models a decision problem by formalizing the unknowns, their possible values and the relations between them.
For this project we implemented our model in **Minizinc** [1] and we ran it with two different constraint solvers included in the Minizinc binary distribution, **Gecode** and **Chuffed**.

2.1 Variables and domains

First of all we need to decide the variables that we need for the encoding and then we define their domains and the relations with the constants and the other unknowns.

We take as input a .dzn file containing:

- `W`, the width of the plate where we have to place the circuits;
- `N`, the number of circuits;
- the `widths` and the `heights` of the circuits

and we save them into variables with those names.
As variables we decided to create two arrays `coord_x` and `coord_y` that contain the x and y coordinates of the circuits and the variable `l` which represents the height of the plate that we want to minimize.

The variables in `coord_x` can assume values between 0 and `W - min(widths)`, because the largest allowed coordinate is when the circuit with the lowest width is placed next to the right side of the plate.
The same reasoning should be done for the `coord_y` array but the domain is between 0 and `l - min(heights)`.
The lower bound for the variable `l` is the maximum value between the maximum height and the sum of the areas of all the circuits divided by W (i.e. no spaces between the circuits, thus the optimal solution). We need to take the maximum between the two values because if there is a circuit much higher than the others then the formula is not correct. In Minizinc the previous statements look like:

```
% Inputs
int: W;
int: N;
array[1..N, 1..2] of int: CIRCUITS;
% Extract widths and heights from matrix
array[1..N] of int: widths = [CIRCUITS[i, 1] | i in 1..N];
array[1..N] of int: heights = [CIRCUITS[i, 2] | i in 1..N];
% Variables
int: l_low=max(max(heights),
               ceil(sum([(widths[i]*heights[i])|i in 1..N])/W));
int: l_up = sum(heights);
var l_low..l_up: l;
% x and y coordinates of the circuits
array [1..N] of var 0..W-min(widths): coord_x;
array [1..N] of var 0..l_up-min(heights): coord_y;
```

2.2 Constraints

The main constraints of the problem are:

- forbid the circuits to exceed the plate;
- avoid overlapping between the different circuits.

The first one can be written using a global constraint from the Minizinc library [2]. In general, it's better to use global constraints as solvers adopt some algorithms that allow to have a very efficient propagation.

In this case we use `diffn(x, y, dx, dy)` that constrain each rectangle `i`, given by its origins `x[i]`, `y[i]` and sizes `dx[i]`, `dy[i]` to be non overlapping with the other rectangles.

```
include "globals.mzn"
constraint diffn(coord_x, coord_y, widths, heights);
```

In order to constrain the circuits between 0 and the maximum width W and between 0 and `l` we can simply write:

```
constraint forall(i in 1..N)(
    coord_x[i]+widths[i] <= W /\ coord_y[i]+heights[i] <= l
);
```

At this point we need to add some other constraints to reduce the search space and improve the performance. We decided to use the `cumulative(s, d, r, b)` global constraint, it checks that a set of tasks given by start times `s`, durations `d`, and resource requirements `r`, never requires more than a global resource bound `b` at any one time. The conversion from these variables to ours is straightforward, we consider as `s` our x/y coordinates, as `d` the widths/heights, as `r` the heights/widths and as global resources we use `l`/`W`.
At any time we cannot exceed the `W` and the `l` bounds if we sum all the circuits in a certain position, therefore this constraint seems to fit well in the model. These may be considered as *implied constraints* because thanks to the 2 main constraints the circuits already have to meet these properties. Adding these constraints our model is able to solve almost all instances with low execution times.

```
constraint cumulative(coord_x, widths, heights, l);
constraint cumulative(coord_y, heights, widths, W);
```

2.2.1 Symmetry breaking

VLSI design problem has many symmetries:

1. reflection over the x axis;
2. reflection over the y axis;
3. circuits with the same dimensions.

In order break the third symmetry we impose an ordering constraint using `lex_lesseq` on the two circuits with the same dimensions.

```
predicate symm_breaking_same() =
      forall(i, j in 1..N where i < j /\ widths[i]==widths[j]
                /\ heights[i]==heights[j])(
            lex_lesseq(
                [coord_x[i], coord_y[i]],
                [coord_x[j], coord_y[j]]
            )
      );
```

For the first two constraints we wrote a predicate that computes the reflected position for each circuit with respect to the x and the y axes and then we impose an ordering constraint between one circuit and its reflected version.

```
predicate symm_breaking_axes() =
  let {
      array[1..N] of var int: new_x =
                  [W - coord_x[i] - widths[i] | i in 1..N];
      array[1..N] of var int: new_y =
```

```
                    [l - coord_y[i] - heights[i] | i in 1..N]
  } in lex_lesseq(coord_x, new_x)/\lex_lesseq(coord_y, new_y);
```

2.3 Considering the model with rotation allowed

We also need to consider a model that allows the rotation of the different circuits. In this case we need some additional arrays:

- a `rot` boolean array, that stores 'true' for each circuit that is rotated and 'false' otherwise;
- a w_real array of integers that contains the actual width for each circuit (e.g. if the i-th circuit has been rotated then w_real_i is equal to $heights_i$);
- a h_real array of integers that contains the actual height for each circuit (e.g. if the i-th circuit has been rotated then h_real_i is equal to $widths_i$);

We add these lines of code in Minizinc:

```
array [1..N] of var bool: rot;
array [1..N] of var 1..max(max(widths), max(heights)): w_real=
    [if rot[i] then heights[i] else widths[i] endif |
        i in 1..N];
array [1..N] of var 1..max(max(widths), max(heights)): h_real=
    [if rot[i] then widths[i] else heights[i] endif |
        i in 1..N];
```

We can keep all the constraints of the basic model but for the new ones we have to consider w_real and h_real instead of widths and heigths because we must compute the bounds using the actual widths and heights of the circuits.

2.3.1 New constraints for rotation

Given the new array `rot` we should add some more constraints to reduce the search space and break symmetries. The first constraint that we add to the model set rot_i `= false` for all the circuits i that have a height larger than `W`, obviously the rotation is not possible and we avoid that the model tries some combinations of values.

```
constraint forall(i in 1..N)(
      if heights[i] > W then rot[i]=false endif
);
```

To break the symmetry of squared circuits we inserted a constraint that set rot_i `= false` for each circuit i with $widths_i == heights_i$, because it's not useful to rotate a square piece.

```
constraint forall(i in 1..N)(
      if widths[i] == heights[i] then rot[i]=false endif
);
```

2.4 Minizinc Search

By default in MiniZinc there is no declaration of how we want to search for solutions. This leaves the search completely up to the underlying solver but sometimes, particularly for combinatorial integer problems, we may want to specify how the search should be undertaken.
A search strategy determines which choices to make and it can affect the performance and the number of found solutions as you will see in the following comparisons. We decided to perform a sequential search that consider different variables, the `l`, the `coord_x` array and the `coord_y` array. We first considered the `l` because in our opinion is more important to find a good set of values for the height and then the solver should work better on searching coordinates. For each `int_search` we have to specify the

variable choice annotation and the constraint choice. In our tests we compared these different variable choice annotations [3]:

- **first fail**, choose the variable with the smallest domain size;
- **input order**, choose in order from the array;
- **smallest**, choose the variable with the smallest value in its domain;
- **dom_w_deg**, 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.

and we constrained the variables with **indomain min**, that assigns the variable its smallest domain value and **indomain random** that assigns the variable a random value from its domain (Chuffed does not support it).
We also tried the **restart luby** annotation with Gecode considering **indomain random** and **dom_w_deg** such that the solver doesn't get stucked in a non-productive area.
As said in the introduction we run our model with Gecode and Chuffed and we noticed from the beginning that Chuffed performs way much better than Gecode, moreover with the **free search** option both solvers work better.

2.5 Tests and results

2.5.1 Basic model

First of all we can compare the different execution times of Gecode using different Variable Order Heuristics. In the legend, "im" means `indomain_min`, "rand" means `input_random` and "luby" means `luby restart` with 200 as value. For each different run (blue, orange, green) we print the bar only if the solver was able to find the optimal solution within the time limit of 300 seconds. In the following chart you can find the results up to the 30-th instance because we can have a meaningful comparison without the last ten instances.

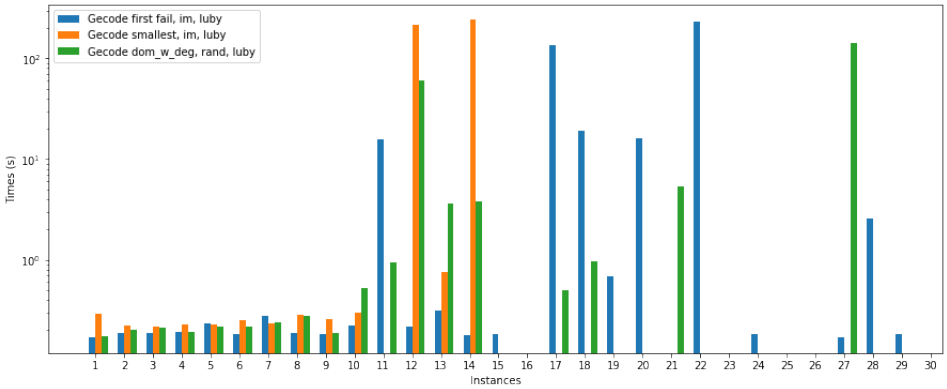


Figure 2: Comparison between the different VOHs with Gecode solver

Using the `free search` option we noticed that the execution time decreases by a few second but the number of optimal solutions found remained the same. Solver could be quicker with this option because is allowed to ignore search annotations. Given that the results with Gecode was not satisfactory we ran the model with Chuffed solver and it outperforms Gecode both in the number of optimal solutions found and in the execution times.
As you can see from figure 3, using **Chuffed** solver with `free search`, `first fail` choice for the variables and `indomain min` for the choice of the value, we are able to get the optimal solutions for 39 instances out of 40 with an average time of 6.59 seconds (only for the last instance we didn't find an optimal solution).
Using **Gecode** solver with `first fail` choice for the variables and `indomain min` for the choice of the value, we are able to get the optimal solutions for 28 instances out of 40 with an average time of 19.19 seconds.
In general Chuffed performed well with different types of VOHs as you can check from the figure 4.

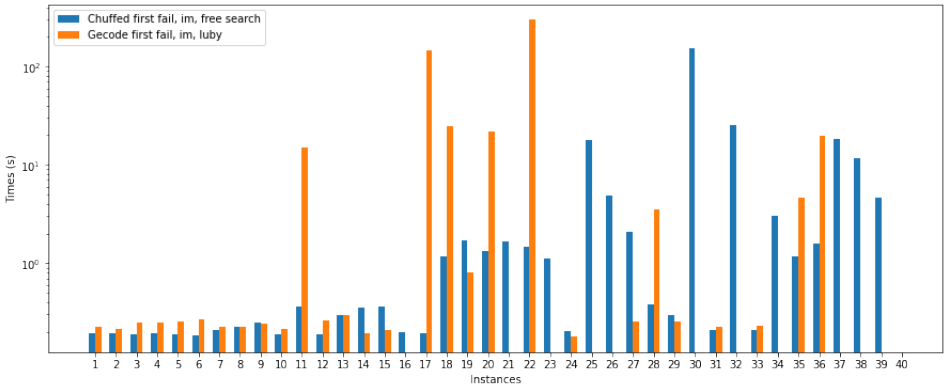


Figure 3: Comparison between Chuffed and Gecode

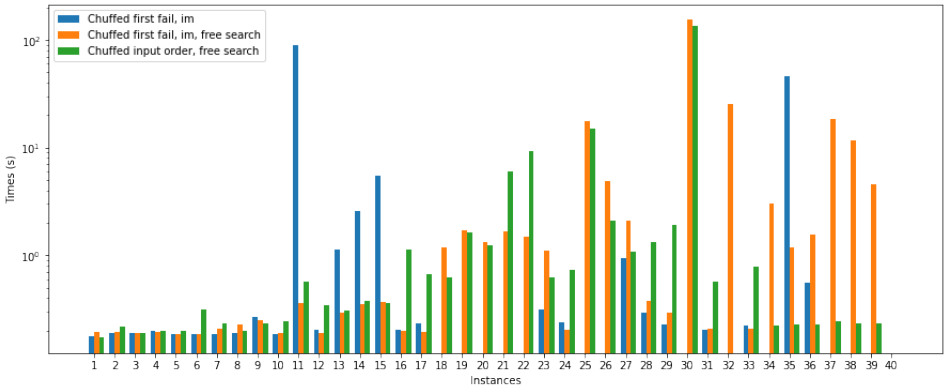


Figure 4: Comparison between different VOHs with Chuffed

2.5.2 Rotation model

In this case it doesn't make sense to consider Gecode because is not able to find the optimal solutions for instances after the 15-th.
Running **Chuffed** solver with `free search`, `first fail` choice for the variables and `indomain min` for the choice of the value, we are able to get the optimal solutions for 32 instances out of 40 with an average time of 11.47 seconds.
In the following graph you can see the execution times using Chuffed on the rotation model with the first fail strategy and the dom_w_deg one.

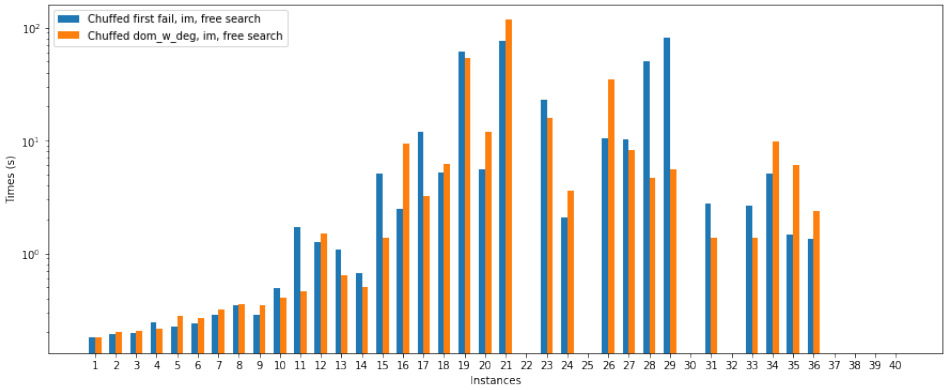


Figure 5: Comparison between first fail and dom_w_deg heuristics on the rotation model

3 SMT model

Satisfiability Modulo Theory concerns the study of the satisfiability of formulas with respect to some **background theories**, in our case we test the model always considering the *LIA (Linear Integer Arithmetic)* logic [4], that fits perfectly with our implementation.

There are a lot of solvers that can be used to run the model and usually they also have a specific language in which you can write but we decide to implement the model with SMT-LIB [5], a standard library for encoding SMT problems. This gives us the opportunity to create a **solver-independent** model that can be run by different solvers without changing the code (using the same .smt2 file created by a Python script).
We tested the model with **z3** [6] (a solver created by Microsoft) and **CVC4** [7] using the Python package PYSMT, [8] that allows you to install different solvers and it gives you a Python interface to run them. The CVC4 installation doesn't work on Windows therefore we ran all the SMT tests on Linux.

3.1 Variables and domains

The first step of the implementation is the choice of the variables that we need for the encoding. Obviously we need an `Int` variable for the `l`, the height to minimize, then we need a pair of integer variables x_i and y_i for each circuit `i` that represent the coordinates of the bottom left point. In the following equations we consider `N` as the number of circuits, `W` as the width of the plate, `widths` and `heights` as the widths and the heights of the given circuits and all these values are given as input to the solver. The values of the coordinates are within the following domain:

$$\begin{aligned} x_i \in \{0, W - min(widths)\},\ \forall i \in [1, N], \\ y_i \in \{0, l\_up - min(heights)\},\ \forall i \in [1, N] \end{aligned} \tag{1}$$

indeed we know for the x coordinates that the minimum value that they can assume is 0 and the maximum is if the circuit with the minimum width is placed next to the right side of the plate. The same reasoning can be applied for the domain of the y coordinates but as maximum value we hypothesized the sum of all the heights of the circuits (`l_up` in this case).
Note that in SMT-LIB the coordinates are represented by the variables `coord_xi` and `coord_yi` instead of x_i and y_i. For example the previous constraints on the domain have been written in the .smt2 file with python in the following way:

```
lines = []
lines += [f"(assert (and (>= coord_x{i} 0) (<= coord_x{i} {W-min(widths)})))"
            for i in range(N)]
lines += [f"(assert (and (>= coord_y{i} 0) (<= coord_y{i} {l_up-min(heights)})))"
            for i in range(N)]
```

It's also important to give a lower and an upper bound to the height `l` because we need to minimize it. A reasonable lower bound [9] for this variable is:

$$l\_low = max(max(heights), \lfloor \frac{\sum_{i=1}^{N} widths_i * heights_i}{W} \rfloor \tag{2}$$

because the optimal case is when there is no space between the circuits but if a very high circuit is given the minimum lower bound is the height of that circuit.

As upper bound we consider a very large value that is the sum of all the heights of the circuits because in the worst case all the circuits need to be stacked.

$$l\_up = \sum_{i=1}^{N} heights_i \tag{3}$$

After that we have computed the two bounds we can set the constraint in SMT-LIB as in the example before, if you want to see the .smt2 file you can find it in the `/SMT/src/` directory of the project.

3.2 Constraints

Up to now we have set only the constraints on the domains but it's not enough in order to solve successfully this problem.

3.2.1 Main constraints

The two main constraints are the one that forbids the circuits to exit from the allowed area and then the **no overlapping constraints** to prevent the circuits from overlapping [10].

The first constraint can be added considering the following inequalities:

$$\begin{aligned} x_i + widths_i <= W,\ \forall i \in [1, N] \\ y_i + heights_i <= l,\ \forall i \in [1, N] \end{aligned} \tag{4}$$

And the no overlapping constraints are the following ones:

$$\begin{gathered} x_i + widths_i <= x_j \\ \vee \\ y_i + heights_i <= y_j \\ \vee \\ x_i - widths_j >= x_j \\ \vee \\ y_i - heights_j >= y_j,\ \forall i, j \in [0, N] s.t. i < j \end{gathered} \tag{5}$$

We put an OR relation between all the inequalities because we want that for each pair of circuits i and j at least one inequality is satisfied.
The first represents the situation in which i is to the left of j, the second is when i is below j, the third is when i is to the right of j and the last one is when i is above j.

3.2.2 Cumulative constraints

In order to get a better solution we decide to implement also in SMT the cumulative constraints used in the Minizinc model. The idea is that this problem can also be seen as a scheduling problem in which the coordinates represent the starting times, the widths/heights represent the duration of the task, the heights/widths represent the resource requirements and W, l are the global resources that cannot exceeded.
We can write two constraints, one considering the height and the other considering the maximum width of the plate:

$$\sum_{i | x_i <= heights_j < x_i + widths_i} y_i \ <= l, \quad \forall j \in [1, N] \tag{6}$$

The one that consider the y-axis is equal to:

$$\sum_{i | y_i <= widths_j < y_i + heights_i} x_i \ <= W, \quad \forall j \in [1, N] \tag{7}$$

3.2.3 Symmetry breaking

We try to add some symmetry breaking constraints in order to remove some symmetric solutions and reduce the search space improving the solver's ability to find a solution.
There are 3 possible symmetries:

- **horizontal symmetry**, if we invert the positions of the circuits from left to right and vice versa;
- **vertical symmetry**, if we invert the positions of the circuits from top to bottom and vice versa;
- two circuits with the **same size**.

In the Minizinc implementation we used a global constraint to manage the first two cases but in this SMT implementation we notice a worsening in the model's capability. Thus for avoiding the first two symmetries we decide to put the circuit with the bigger area in the position (0, 0) of the plate.
In order to avoid the third one we wrote a constraint that impose an order between the coordinates of the equal circuits:

$$(widths_i == widths_j \land heights_i == heights_j) \implies (x_i <= x_j \land y_i <= y_j)\,,\ \forall i,j \in [1,N]\ s.t.\ i < j \quad (8)$$

3.3 Considering the model with rotation allowed

As said before we also need to consider a model that allows the rotation of the different circuits. In this case we need some additional variables for each circuit:

- a rot_i boolean variable, that represent if the circuit i is rotated or not;
- a w_real_i variable, that is the actual width of the circuit i (e.g. if the i-th circuit has been rotated then w_real_i is equal to $heights_i$);
- a h_real_i variable, that is the actual height of the circuit i (e.g. if the i-th circuit has been rotated then h_real_i is equal to $widths_i$);

Given the new variables we must add some constraints to make the model work correctly. The first two constraints that we add in the SMT-LIB model are related to the explanation given before, if a circuit has been rotated then its width and height need to be inverted:

```
lines += [f"(assert (ite rot{i} (= w_real{i} {heights[i]}) (= w_real{i} {widths[i]})))"
            for i in range(N)]
lines += [f"(assert (ite rot{i} (= h_real{i} {widths[i]}) (= h_real{i} {heights[i]})))"
            for i in range(N)]
```

Obviously we keep all the constraints stated in the previous sections but instead of considering the original $widths$ and $heights$ now we compute the assertions with w_real and h_real variables.

3.3.1 New constraints for rotation

We can decrease the search space imposing the variable $rot_i = false$ for all the circuits i in which the height is greater than the maximum width W, indeed the solver cannot rotate a circuit if it doesn't respect the boundary constraint on the W.

$$(heights_i > widths_i) \implies rot_i = false,\ \forall i \in [0,N] \quad (9)$$

A symmetry breaking for circuits with the same width and height can be inserted, because it doesn't make sense to rotate a square.

$$(heights_i == widths_i) \implies rot_i = false,\ \forall i \in [0,N] \quad (10)$$

3.4 Searching for optimality

As stated before, SMT-LIB doesn't support the optimization of a target function so we had to manually implement the search for the optimal solution.
We decided to test two different algorithms to see which one performs better:

- Lower bound search
- Binary search on domain

3.4.1 Lower bound search

This algorithm is very symple, since we know that the lower bound of the height `l` is the optimal solution we start the search imposing the constraint `l == lower_bound` such that if it exists it would be found without searching for others. If the solver returns 'unsat' then we increase by one the lower_bound and we start the search again, obviously we remove from the assertions stack the previous constraint. We

can write the pseudocode for this function as follows:

Algorithm 1 Lower bound search pseudocode

```
l_guess ← lower_bound
solver.push()
solver.add_constraint(l == l_guess)
while ¬solver.solve() do
    l_guess ← l_guess + 1
    solver.pop()
    solver.push()
    solver.add_constraint(l == l_guess)
end while
```

The other algorithm is more complex, it tries to guess a new bound for `l` summing the lower and upper bound, at the beginning set as the lower and upper bound of the `l` and imposing a constraint $l < l\_guess$ if it found a solution it continues with this procedure otherwise it increases the lower bound up to the guessed l plus one. The algorithm stops when the lower bound is greater than the upper bound. It also needs to manage the stack of assertions in order to remove a constraint if it leads to a failure. We should check a lot of different things that we don't represent in the code below, the following pseudocode is only to better understand the reasoning behind this function. We took inspiration from the slides on OMT of Roberto Sebastianini [11].

Algorithm 2 Binary search pseudocode

```
low ← lower_bound
up ← upper_bound
l_guess ← upper_bound
l_optimal ← upper_bound
while low < up do
    up_backup = l_guess
    l_guess ← ⌈(low + up)/2⌉
    solution ← solver.solve()
    if ¬solution then
        solver.pop()
        up ← up_backup
        low ← l_guess + 1
    else
        l_optimal ← solution
        up ← l_guess
        solver.pop()
        solver.push()
        solver.add_constraint(l < up)
    end if
end while
```

3.5 Used Solvers

We decided to use z3 and CVC4 solvers because we think they are the most efficient ones. Both could be run by Pysmt after having installed them (I remind you that CVC4 installation fails on Windows).
Both solvers are based on lazy/DPLL(T) paradigm but z3 implements a module called *Simplifier* that simplifies the input formulas by applying standard algebraic reduction rules and contextual simplifications, and probably it enhances the solver performance [12].
In our case the solvers take as input the .smt2 files that we create with Python and after the parsing they try to find a solution, as execution time we consider only the actual time for solving the instance (in any case the creation of the file requires little time).
In the following figures of the charts all the instances that reach an execution time of 300 seconds are the ones for which the solvers didn't find an optimal solution within the time limit.

3.5.1 z3

With z3 and applying the binary search our basic model (without rotations) is able to compute the optimal solution for 34 out of 40 instances with an average time of 7.8 seconds (on the solved instances). It is not able to get the optimal solution for the instances 30, 32, 37, 38, 39 and 40.
Using low bound search we can solve the same instances but with an average time of 6.95 seconds, the performance is a little better, more in the simpler instances, probably because during the binary search the solver can learn some assumptions that may help in subsequent resolutions. From now on in all the other comparisons we used the binary search method.

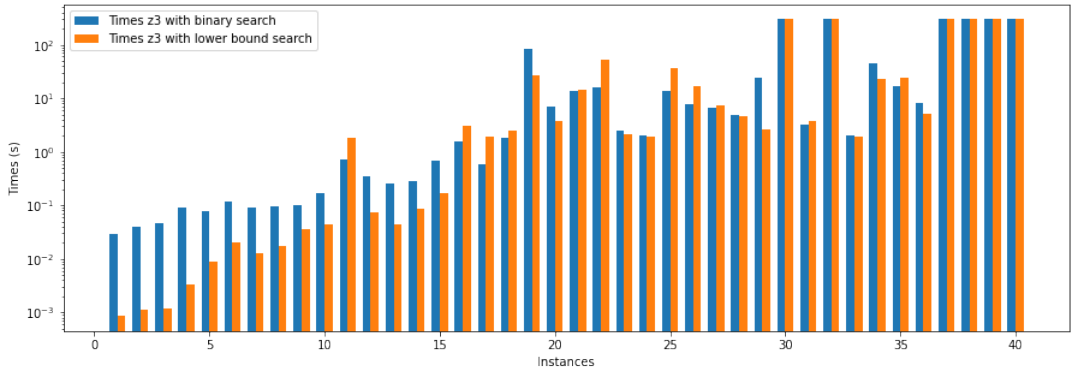


Figure 6: Comparison between the execution times of z3 using the binary and the lower bound search

If we consider the model wherein the rotations are allowed z3 is able to find the optimal solutions only for 25 instances. As in previous models, also SMT performs worse with rotations, probably because the search space is bigger and the problem is more complex. Indeed also the average time for finding the optimal solutions is increased to 46 seconds.

3.5.2 CVC4

CVC4 solver has worse performances with respect to z3 and it finds less optimal solutions. With the binary search strategy it is able to find the optimal solution for 29 instances out of 40 with an average time of 40 seconds. Using the lower bound search the solver is only able to compute the optimal solution for 28 instances out of 40. It is much quicker with an average time of 28 seconds but it cannot solve the same number of instances.

Considering the model with the rotations CVC4 can only return the optimal solution for 17 instances, it performs much worse than the z3 solver.

3.5.3 Comparison between the two solvers

As we can deduct from the previous data, z3 solver performs better than CVC4, it solves a larger number of instances returning the optimal solution and it is faster than CVC4.

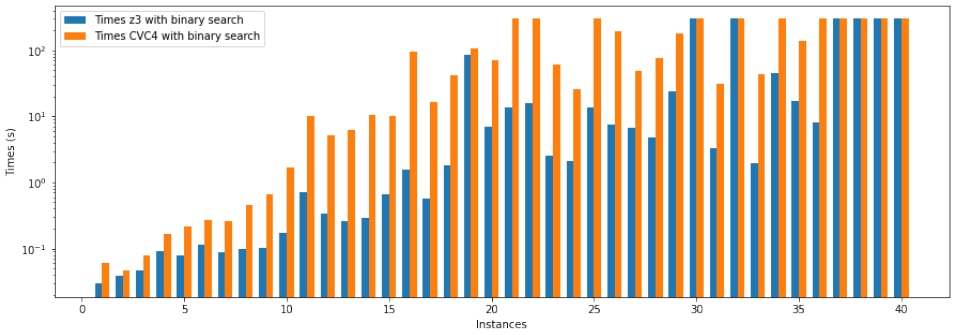


Figure 7: Comparison between the execution times of the best solve for z3 and CVC4

In this second chart we can see the differences between z3 and CVC4 execution times with rotation model, however both solvers didn't work well with this model.

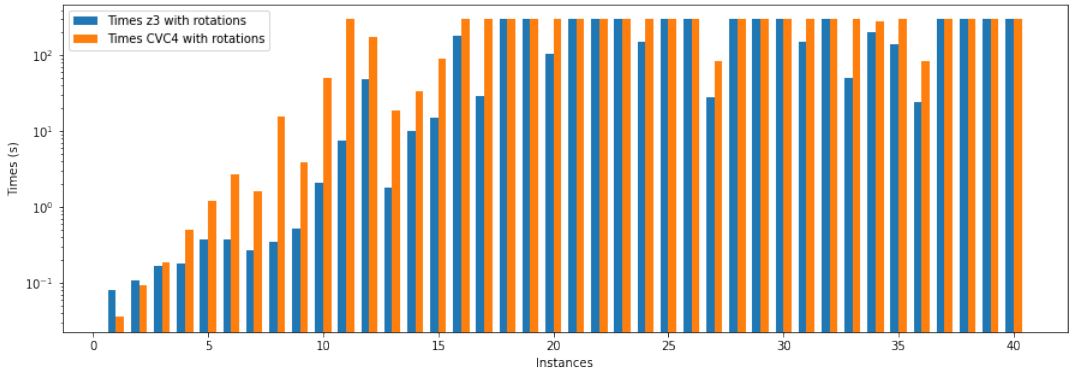


Figure 8: Comparison between the execution times of z3 and CVC4 on the rotation model

3.6 Conclusions

We can conclude that z3 is more effective than CVC4 with our model and the search for the optimal value doesn't affect so much the performance of the solvers. In order to improve the execution time and the number of optimal solutions found we should try a different logic, for example saving the different variables in some arrays instead of using a lot of variables.
The images below represent the solutions for the instance 20 solved with and without rotations allowed.

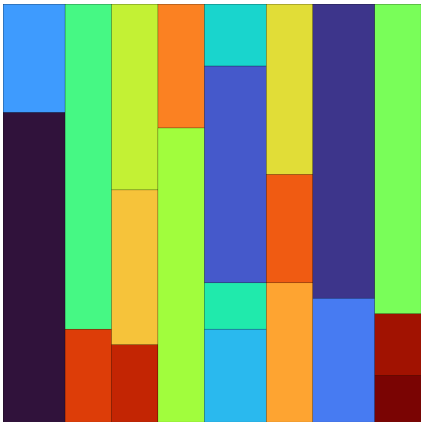


Figure 9: Solution without rotations

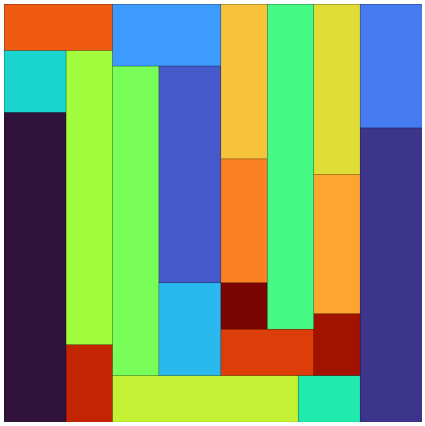


Figure 10: Solution with rotations

4 LP Model

Linear Programming (LP) is an operation research (OR) tool based on a set of constraints expressed as a system of linear (in)-equalities, where the purpose is to optimize a linear function. For our case, we consider a Mixed-Integer Programming (MIP) approach, where variables are integers.

MIP problems can be implemented in different ways and many solvers are available. We decided to implement two different models using both the Python library **PuLP**[13] and also **Minizinc**[1], this allows us to build **solver-independent** models, and for our case we choose two solvers for testing: **MOSEK**[14] and **CPLEX Optimizer**[15].

As previously mentioned, we propose two different models for emphasizing respectively the time needed for solving instances and the number of different instances solved. The former has been built with the Python library *PuLP* and it is solved with both CPLEX and MOSEK, the latter is developed in MiniZinc and solved with only *CPLEX*.

4.1 PuLP Model

The PuLP model is very similar to CP and SMT models, for this reason only a briefly explanation will be given. We propose both the model without rotation and the one allowing circuit to be rotated by 90°.

We decided to use the *PuLP* for three reasons:

- We have already developed Python scripts when solving instances with other tools (SMT, CP), which we can also use in MIP
- The library offers many solvers out of the box, for example **CPLEX**, **MOSEK** and **Gurobi**, therefore only the installation of the solver is required from an user side. In this way we can also configure more solvers than the ones we are proposing and only a small change in code is required
- PuLP has simplifying tools for writing the model, manipulating their variables and obtain the computed solution, at the same time taking advantage of the Python language

The parameters given as input are:

- `N`: the number of circuits
- `W`: the width of the plate
- `widths` and `heights`: widths and heights of the given circuits

4.1.1 Variables and domains

For the encoding of decision variables, we only need **integer variables**. They are:

- `l`: the height to minimize
- x_i and y_i: a pair of variables representing the coordinates of the bottom left point of the `i`-th circuit.

It is important to give a lower and an upper bound to the height `l` because we need to minimize it. A reasonable lower bound[9] for this variable is:

$$l\_low = max(max(heights), \lceil \frac{\sum_{i=1}^{N} widths_i \cdot heights_i}{W} \rceil \tag{11}$$

because the optimal case is when there is no space between the circuits, that is a plate perfectly covered by circuits, but if a high circuit is given, then the lower bound can be increased because we are sure that a perfect plate cannot exists, resulting in a smaller domain, which is always preferred.

For the upper bound we consider a very large value that is the sum of all the heights of the circuits because in the worst case all circuits need to be stacked one above the other.

$$l\_up = \sum_{i=1}^{N} heights_i \tag{12}$$

Therefore, the height is constrained as:

$$\begin{aligned} l\_low \leq l \\ l \leq l\_up \end{aligned} \tag{13}$$

In addition, the domain of coordinates are:

$$\begin{aligned} x_i \in [0, W - widths_i],\ \forall i \in [1, N], \\ y_i \in [0, l\_up - heights_i],\ \forall i \in [1, N] \end{aligned} \tag{14}$$

in fact, for the x coordinate the minimum value is 0, because we are referring to the bottom left point, and the maximum is the point in the case the circuit is placed to the right side of the plate.
The same reasoning can be applied for the y coordinate, but in this case the maximum value is the upper bound of the height, namely `l_up`.

In Python we are referring to these variables as `l_low`, `l_up`, `l`, `coord_x_i`, `coord_y_i`. They are described using the PuLP library in this way

```
l = LpVariable("l", lowBound=l_low, upBound=l_up, cat=LpInteger)
coord_x = [LpVariable(
    f"coord_x_{i}", lowBound=0, upBound=int(W - widths[i]), cat=LpInteger)
    for i in set_N]
coord_y = [LpVariable(
    f"coord_y_{i}", lowBound=0, upBound=int(l_up - heights[i]), cat=LpInteger)
    for i in set_N]
```

It can be seen that the `LpVariable` class defines the variable name, lower and upper bounds and the variable type, that is **integers**. Python list comprehension over $[1, N]$ is used to define coordinates variables, which are accessed as lists later in the program.

4.1.2 Constraints

Two more constraints are needed: the **y-coordinate constraint** and the **no overlapping constraint**.

The former guarantees that the y-axis coordinate never exceeds the available plate, that is

$$y_i + heights_i <= l,\ \forall i \in [1, N] \tag{15}$$

This is needed only for the y-axis because `l` is a variable, that is the height we are trying to minimize, thus the plate height varies when solving the instance. The same does not apply to the width of the plate, which is fixed.

The no overlapping constraint prevents circuits from overlapping one on each other and it is described as

$\forall i, j \in [1, N]$ s.t. $i < j$:

$$\begin{aligned} x_i + widths_i \leq x_j + \delta^1_{i,j} \cdot W \\ x_j + widths_j \leq x_i + \delta^1_{j,i} \cdot W \\ y_i + heights_i \leq y_j + \delta^2_{i,j} \cdot l\_up \\ y_j + heights_j \leq y_i + \delta^2_{j,i} \cdot l\_up \\ \delta^1_{i,j} + \delta^1_{j,i} + \delta^2_{i,j} + \delta^2_{j,i} \leq 3 \end{aligned} \tag{16}$$

The first two constraints represent the case when i is to the left of j or vice versa. The third and the forth represent when i is below j or vice versa.

As observed in other tools, this constraints is an OR condition between the four inequalities, because we want for each pair of circuits i and j that at least one inequality is satisfied.

In general, to encode an OR condition, as the no overlapping one, two things must be considered:

- $\delta^k_{i,j} \in \{0,1\}$: it is a binary variable to exploit the **big-M constant trick**, in this case it ensures at least one inequality, such as $x_i + widths_j \leq x_j$, holds.
- `W` and `l_up`: M constants, which are big enough to have the inequality holding when $\delta^k_{i,j}$ is 1.

For this reason, at least one $\delta^k_{i,j} = 0$, hence the **M constant** is not considered and the condition to avoid the overlap between two circuits holds.

In Python $\delta^k_{i,j}$ are represented in this manner

```
set_C = range(2)
delta = LpVariable.dicts(
    "delta", indices=(set_N, set_N, set_C), cat=LpBinary, lowBound=0, upBound=1)
```

Other than this, an addition constraint can be added, in fact two circuits cannot be put aside if they are greater than the width of the place, therefore we want to only consider the y-axis constraints, while the left and right are set in advance, knowing they cannot hold. This can be represented as follows

$\forall i, j \in [1, N]$ s.t. $i < j \wedge widths_i + widths_j > W$:

$$\begin{aligned} \delta^1_{i,j} &= 1 \\ \delta^1_{j,i} &= 1 \end{aligned} \tag{17}$$

Finally, the circuit with the larger area has been put in a fixed position, that is **(0,0)**, in order to break the symmetry of the model. This gives overall better performance.

Being k the index of the circuit with the larger area, then we have the following constraints:

$$\begin{aligned} x_k &= 0 \\ y_k &= 0 \end{aligned} \tag{18}$$

The optimization function to minimize is

$$min\ l \tag{19}$$

4.1.3 Considering the model with rotation allowed

Other than the classic problem, an alternative model with circuits both in their actual size and also turned by 90° has been modelled. In this case we need an additional variable for each circuit:

- rot_i: it is a binary variable, holds if the i-th circuit should be considered as rotated, which size is $height_i \times width_i$.

In Python it is described as

```
rotation = LpVariable.dicts("rot", indices=set_N, lowBound=0, upBound=1, cat=LpBinary)
```

Given the new variable, we must change some constraints to obtain a working model.

The upper bound of the height changes because for each circuit the maximum value between the height and the width of that must be considered, that is

$$l\_up = \sum_{i=1}^{N} max(widths_i, heights_i) \tag{20}$$

The same applies for coordinates, which are represented as

$$\begin{aligned} x_i &\in [0, W - min(widths_i, heights_i)],\ \forall i \in [1, N], \\ y_i &\in [0, l\_up - min(heights_i, widths_i)],\ \forall i \in [1, N] \end{aligned} \tag{21}$$

The first two constraints added are needed to check the boundaries of the actual size of the circuit. Even though in the model without rotation we did not consider the constraint for the x-axis, in this case it is mandatory because the domain constraint we impose do not take into consideration the actual size of the circuit. They are added as follows

$$\begin{aligned} x_i + widths_i \cdot (1 - rot_i) + heights_i \cdot rot_i <= W,\ \forall i \in [1, N], \\ y_i + heights_i \cdot (1 - rot_i) + widths_i \cdot rot_i <= l,\ \forall i \in [1, N] \end{aligned} \tag{22}$$

A new constraint is added to decrease the search space imposing the variable $rot_i = 0$ for all the circuits i in which the height is greater than the maximum width W, indeed the solver cannot rotate a circuit if it doesn't respect the boundary constraint on the W.

$$(heights_i > W) \implies rot_i = 0,\ \forall i \in [0, N] \tag{23}$$

A symmetry breaking for circuits with the same width and height can be added, because they should not be rotated

$$(heights_i = widths_i) \implies rot_i = 0,\ \forall i \in [0, N] \tag{24}$$

The no overlapping constraint considers the actual size of the circuit, as pointed out in 21, that is

$\forall i, j \in [1, N]$ s.t. $i < j$:

$$\begin{aligned} x_i + widths_i \cdot (1 - rot_i) + heights_i \cdot rot_i \leq x_j + \delta^1_{i,j} \cdot W \\ x_j + widths_j \cdot (1 - rot_j) + heights_j \cdot rot_j \leq x_i + \delta^1_{j,i} \cdot W \\ y_i + heights_i \cdot (1 - rot_i) + widths_i \cdot rot_i \leq y_j + \delta^2_{i,j} \cdot l\_up \\ y_j + heights_j \cdot (1 - rot_j) + widths_j \cdot rot_j \leq y_i + \delta^2_{j,i} \cdot l\_up \\ \delta^1_{i,j} + \delta^1_{j,i} + \delta^2_{i,j} + \delta^2_{j,i} \leq 3 \end{aligned} \tag{25}$$

4.2 Positions and covering model

Noticing the poor performances of the PuLP model in MIP, with respect to models developed in SMT and CP, we propose a different approach, following the positions and covering (P&C) methodology[16].

Before describing the procedure, we implement this model in Minizinc and not directly in Python using PuLP, because it depends heavily on the performance of *for-loop*, which are necessary in the first step.

We tried the representation using the PuLP library, but we observe poorer performance in this model than in the first one, for this reason, because we have already used Minizinc from Python (CP tool), we decided to go for the latter implementation. Nonetheless, the PuLP model can be found in the source file **create_model.py** located in the **MIP/src** folder.

The model assumes an initial height `l` for perfect packing, or a better lower bound if known. We decided to choose the one previously computed as `l_low`.
Then a set of valid positions is generated, representing the location where circuits can be placed in the plate. Finally an integer linear programming (ILP) set-covering formulation for this decision problem is solved. The solution is feasible if the perfect packing is possible, otherwise the height is increased by one and these steps are repeated, until a solution is found. This two-step methodology is described in 11.

4.2.1 Positions stage

In the first phase, after assuming an initial height `l`, a set of valid positions for each circuit is computed.

First of all, the plate of fixed height has $W \cdot l$ **tiles**: these are all the possible positions which can contain circuits. Then, a circuit can be in a **valid position**, with respect to the bottom left point, only if the following condition holds

$$V(i) = \{\lfloor j \rfloor \mid j \in [0, l - heights_i] \times [0, W - widths_i]\},\ \forall i \in [1, N] \tag{26}$$

Each position is encoded in a unique way among all circuits, that is

$$\forall i \in [1, N],\ V(i) \cap V(j) = \emptyset,\ \forall j \in [1, N] \wedge i \neq j \tag{27}$$

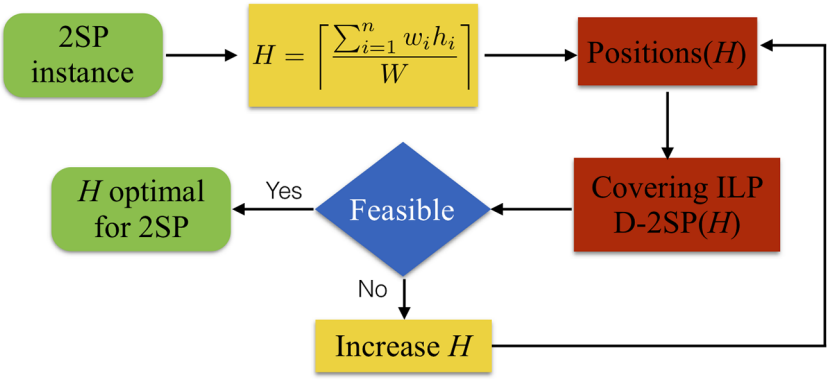


Figure 11: Positions and covering methodology

After having all possible positions for each circuit, a correspondence matrix is computed, where rows are all **valid positions** and columns are **tiles**, in such a way

$$\begin{aligned} J &= \bigcup_i V(i) \\ P &= [1, W \cdot l] \\ \forall i \in J, j \in P,\ c_{i,j} &= \begin{cases} 1 \text{ if position } i \text{ covers tile } j \\ 0 \text{ otherwise} \end{cases} \end{aligned} \tag{28}$$

A position is said to *cover* a tile if the circuit which the position belongs to, located in that specific point with respect to its bottom left point, contains the tile.

These computations can be clearly seen in the file **model_base.mzn** located in *MIP/src*

4.2.2 Covering stage

In this section is explained the ILP model, which has the following decision variables

$$\forall i \in [1, N], j \in V(i),\ x_{i,j} = \begin{cases} 1 \text{ if circuit } i \text{ is placed in valid position } j \\ 0 \text{ otherwise} \end{cases} \tag{29}$$

Then, the first inequalities added are

$$\sum_{i \in [1,N]} \sum_{j \in V(i)} c_{j,p} x_{i,j} \leq 1,\ \forall p \in P \tag{30}$$

which constrain the solution to have each tile occupied by at most one circuit. In fact, even though position indexes are unique to each other, given a column of the correspondence matrix it is possible to know all positions which can occur in a given tile.

Next, another constraint is mandatory for each circuit to have exactly one position covered. This is expressed by the following inequalities

$$\sum_{j \in V(i)} x_{i,j} = 1,\ \forall i \in [1, N], \tag{31}$$

The optimization function is: $min\ z = 0$, given that only a satisfiable assignment is searched and solved.

These constraints are represented in Minizinc as follows

```
constraint forall (p in set_P)
    (sum (i in set_I, j in valid_i[i])
        (corr_matrix[j,p] * place[i,j]) <= 1);
```

```
constraint forall (i in set_I)
    (sum (j in valid_i[i]) (place[i,j]) == 1);
constraint forall (i in set_I)
    (sum (j in set_J diff valid_i[i]) (place[i,j]) == 0);
```

Variables used in MiniZinc are:

- `l_bound`: l
- `set_I`: $[0, N]$
- `valid_i[i]`: $V(i)$
- `corr_matrix[j,p]`: $c_{j,p},\ \forall j \in V(i),\ p \in P$
- `place[i,j]`: $x_{i,j},\ \forall i \in [1, N],\ j \in V(i)$

4.2.3 Model with rotation allowed

A model with rotation allowed has not been represented, essentially because we want to compare the models proposed in their encoding without rotation for better evaluation, which can be seen in the next section. Another thing is that this model relies on the performance implementation of the correspondence matrix, which is worsening considering also rotation circuits.

4.3 Results

Result for both models are presented in the following section, with an additional comparison between the two models.

4.3.1 PuLP

For the first model described in Python with the PuLP library we run tests on both **CPLEX** and **MOSEK**.

CPLEX is a solver released by *IBM* and it is considering one of the best solvers available, second to only Gurobi[17]. MOSEK is another solver which we propose for a quantitative comparison for the same model. However it performs worse than CPLEX in all tests we have performed.

We have academic licenses for both solvers. Tests are run on a laptop in Windows 11 equipped with an Intel i7-8550U with 16GB of ram.

Both solvers are not able to solve instances 19, 21, 25, 26, 28, 30, 32, 34, 35, 36, 37, 38, 39, 40. MOSEK offers the worst performance overall, being unable to solve also some initial instances, such as 14 and 15.

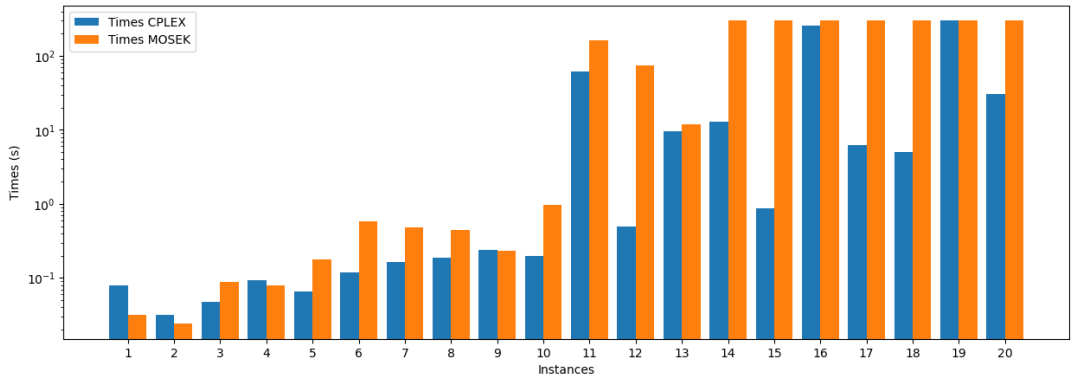


Figure 12: Comparison between the execution times of CPLEX and MOSEK

Comparisons that we propose in this section are:

- MOSEK vs CPLEX with the base model and solver default configurations

- Base model vs rotation model using CPLEX with default configuration
- CPLEX vs CPLEX with symmetry breaking with the base model

A comparison between MOSEK and CPLEX is shown for the first 20 instances in 12. It can be seen that CPLEX is far better than MOSEK, both in time needed for solving and for the number of instances solved. MOSEK fails to solve instances after the 13th, while CPLEX solves more instances and with less time.

For this reason we proceed with remaining tests with only the CPLEX solver.

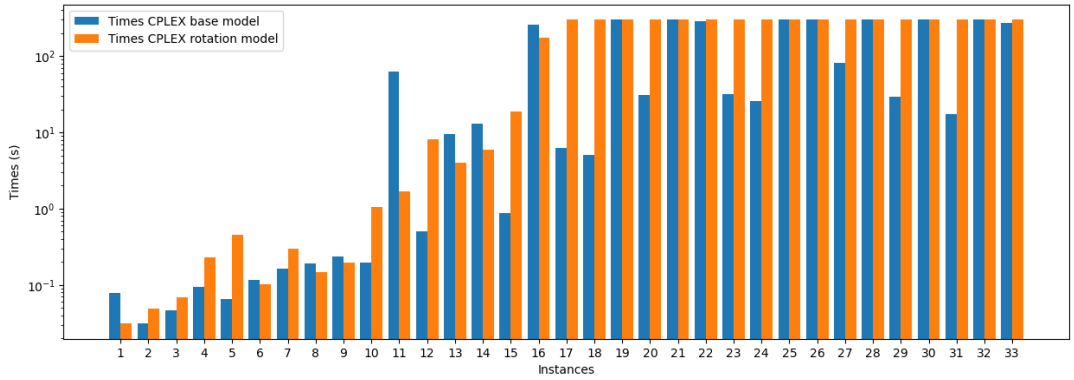


Figure 13: Comparison between the execution times of CPLEX in the base and rotation models

We solved the model described in 4.1 with CPLEX in default configuration. As can be noticed in 13, it is able to solve 26 instances, with an average time of *43,50 seconds*. However, considering a time limit of 90 seconds, 23 instances are solved with an average time of *13.67 seconds*.

In the same figure, a comparison with the model with allowed rotation is shown. It is solved with the same configuration, to better analyze differences. The rotation model solves only the first 16 instances, with an average time of *2,75 seconds* in the first 15 instances, better than the base model which requires almost *6 seconds*.
On the other hand, it cannot solve more instances, resulting in a poorer model than the base one. This is because the first instances have at most 16 circuits and a smaller area, in this way the rotation model is more powerful. The increasing number of circuits and height of the optimal plate decrease the efficiency.

Finally, we have done tests for a configuration of CPLEX with a more aggressive symmetry breaking[18], passing the option `set preprocessing symmetry 5`, but we do not noticed any gain in performance.

4.3.2 Comparing PuLP and P&C models

The positions and covering model explained in 4.2 is realized in Minizinc and tested with CPLEX with the default configuration. The PuLP model uses the same configuration explained before.

The P&C model is the best model, in fact as can be seen from 14 it solves **34 instances**, with a mean time of *55.50 seconds* for the solved instances. However, as highlighted at the start of the chapter 4, the main reasons we have developed two models is that the PuLP model has a lower mean time for solved instances, while P&C solves 8 instances more.

The most time-consuming step is the flattening time of Minizinc, due to the preprocessing stage for the generation of valid positions for each circuit, which takes pseudo-polynomial[16] time in the size of the input, while the solve time for the decision problem is of just few seconds. This can be seen in 14 because its times are not decreasing until last instances.

Solver	N. instances	Mean time	Total time
PuLP	26	43.50 seconds	1130 seconds
P&C	34	55.51 seconds	1887 seconds

Table 1: Compare between solved instances of PuLP and P&C models

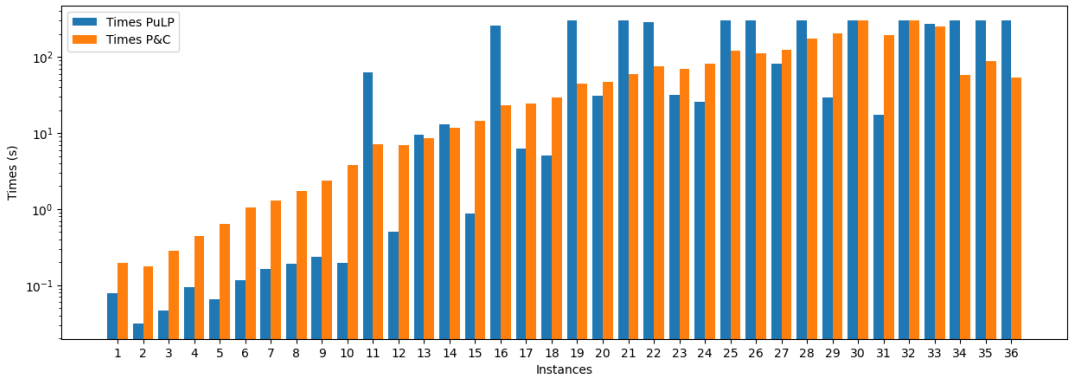


Figure 14: Comparison between the PuLP model and the P&C model

In Table 1 are compared the results between two solvers, supporting the explanations given so far. In addition, the total time needed to the P&C model to solve the exact same instances of the PuLP model is 1180 seconds, confirming that the time needed is greater.

4.4 Conclusions

The model realized with Minizinc is more effective than the PuLP one, it solves 34 instances in contrast to 26. The rotation model does not have the same efficiency as the base one, it fails instances after the 16th, however the time needed for the solved ones is lower than the model where rotations are not allowed.

References

[1] M. University, "Minizinc." https://www.minizinc.org/index.html.

[2] Peter J. Stuckey, Kim Marriot, Guido Tack, "The minizinc handbook." https://www.minizinc.org/doc-2.6.4/en/predicates.html#global-constraints, 2020.

[3] Peter J. Stuckey, Kim Marriot, Guido Tack, "The minizinc handbook." https://www.minizinc.org/doc-2.5.5/en/mzn_search.html, 2020.

[4] "Smt-lib logics." https://smtlib.cs.uiowa.edu/logics.shtml.

[5] Cesare Tinelli. Clark Barret, Pascal Fontaine, "The smt-lib standard, version 2.6," 2021. https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[6] "z3 solver." https://github.com/Z3Prover/z3.

[7] "Cvc4 solver." https://cvc4.github.io/.

[8] "Pysmt package." https://github.com/pysmt/pysmt.

[9] Imed Kacem. Abdelghani Bekrar, "An exact method for the 2d guillotine strip packing problem," *Hindawi*, p. 20, 2009. https://www.hindawi.com/journals/aor/2009/732010/.

[10] Suchismita Roy. Suchandra Banerjee, Anand Ratna, "Satisfiability modulo theory based methodology for floorplanning in vlsi circuits," *arxiv*, p. 8, 2016. https://arxiv.org/ftp/arxiv/papers/1709/1709.07241.pdf.

[11] R. Sebastianini, "Optimization modulo theories, an introduction." https://alexeyignatiev.github.io/ssa-school-2019/slides/rs-satsmtar19-slides.pdf, 2019.

[12] A. Hofler, "Smt solver comparison." https://spreadsheets.ist.tugraz.at/wp-content/uploads/sites/3/2015/06/DS_Hoefler.pdf, 2014.

[13] "Pulp package." https://github.com/coin-or/pulp.

[14] "Mosek." https://www.mosek.com/.

[15] "Cplex." https://www.ibm.com/products/ilog-cplex-optimization-studio.

[16] N. M. Cid-Garcia and Y. A. Rios-Solis, "Exact solutions for the 2d-strip packing problem using the positions-and-covering methodology," *PLOS ONE*, vol. 16, Jan. 2021.

[17] L. M. Hvattum, A. Løkketangen, and F. Glover, "Comparisons of Commercial MIP Solvers and an Adaptive Memory (Tabu Search) Procedure for a Class of 0-1 Integer Programming Problems," p. 14.

[18] "Cplex ibm documentation." https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/icos/22.1.0?topic=cplex-list-parameters.