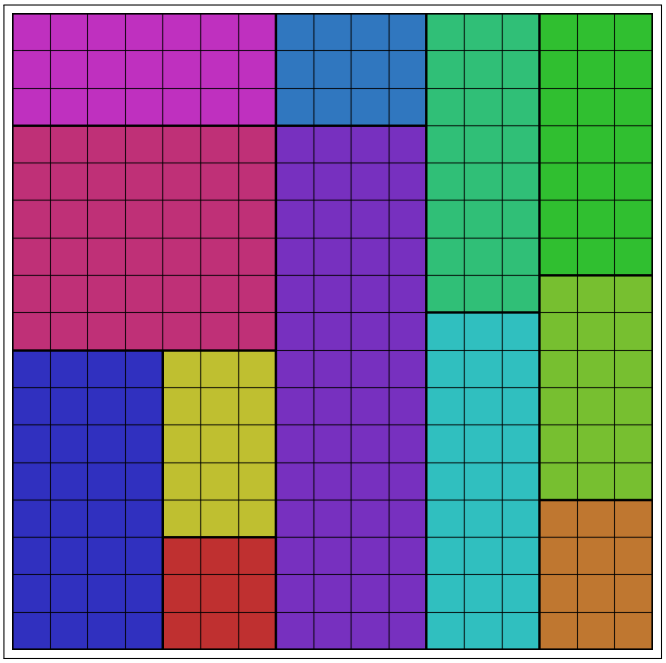
Very Large Scale Integration

Giacomo Berselli
giacomo.berselli@studio.unibo.it

Martino Mare Lakota Pulici
martinomare.pulici@studio.unibo.it

August 29, 2021



Contents

List of Figures

1 Introduction

Very Large Scale Integration (VLSI) refers to the real-world problem of fitting millions of circuits onto a single chip. The problem tackled in this project is a simplified version which deals with much smaller numbers and only rectangular circuits. The idea is to minimize the chip height, starting with a fixed chip width and a set of circuits. Two different versions of each problem are to be solved: in the first version, the circuits can not be rotated, while in the second version they can. Regarding the solving methods, three different minimization approaches are used and compared.

The first method is Constraint Programming, a paradigm which enables the user to declare constraints using some instantiated variables, leaving the burden of solving the problem to the machine. In particular, MiniZinc (a high-level, solver-independent, constraint modeling language) is used.

The other two methods both exploit the Z3 Theorem Prover, a cross-platform satisfiability solver developed by Microsoft. The two approaches which take advantage of the Z3 solver are Propositional Satisfiability and Satisfiability Modulo Theories. Propositional Satisfiability refers to a set of problems where the satisfiability of a boolean formula is verified. Typically, these formulas are given in Conjunctive Normal Form, i.e. using only conjunctions of disjunctions. Satisfiability Modulo Theories, on the other hand, is an extension of Propositional Satisfiability which does not limit its syntax to boolean atoms, using for example real numbers, lists, arrays and other data structures.

The basic variables used in this project are presented in Section 2. After this, the different solving approaches are explored in details in Sections 3 to 5. The same set of 40 instances is used with all methods, in order to provide a transparent and significative comparison of the different approaches.

2 Modeling

2.1 Variables

Each instance comes with 4 parameters describing the chip configuration:

- `chip_w`: the width of the chip;
- `n`: the number of circuits to fit on the chip;
- `inst_x`: a list containing the ordered widths of the circuits;
- `inst_y`: a list containing the ordered heights of the circuits.

In addition, 3 auxiliary variables are created:

- `min_index`: the index of the smallest circuit (see Section 2.2.3);
- `min_h`: the minimum possible height of the chip (see Section 2.4);
- `max_h`: the maximum possible height of the chip (see Section 2.4).

The results of the various solving procedures consist of 3 variables:

- `bl_x`: a list containing the ordered horizontal positions of the bottom-left corners of the circuits;
- `bl_y`: a list containing the ordered vertical positions of the bottom-left corners of the circuits;
- `chip_h`: the height of the chip.

2.2 Constraints

2.2.1 Boundaries consistency

Given the problem to fit all circuits onto the chip, the first obvious constraint is that of making the circuits not fall out of the chip. This is quite easy: it is sufficient to impose, for every circuit `k`, that both `bl_x[k]` and `bl_y[k]` are non-negative, and that the bottom right corner horizontal coordinate (`bl_x[k] + inst_x[k]`) is at most equal to `chip_w`.

2.2.2 Non-overlapping

The next step is to impose a non-overlapping constraint between each pair of circuits. In order to do this, the basic idea is to cycle through all the circuits and to check that the distance between every bottom-left corner pair is at least equal to the dimension of the circuit in at least one direction.

2.2.3 Symmetry breaking

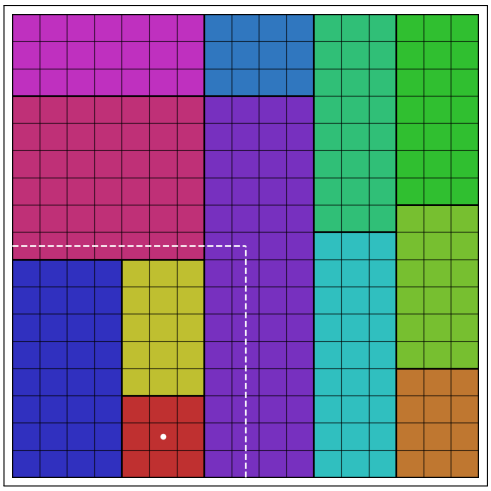
In order to eliminate symmetrical solutions, a simple symmetry breaking constraint is introduced. As shown in Fig. 1, each solution has four symmetrical configurations, where the circuits are located in symmetrical positions with respect to an horizontal and/or a vertical central symmetry axis.

In each of the configurations, the center of the smallest circuit (which has the index `min_index`) is highlighted with a white dot, while the bottom-left quadrant is indicated by a dashed white line. It can be noted that the white dot lies in the highlighted quadrant in only one of the 4 symmetrical configurations, with the exception of very rare cases (i.e. when it lies exactly on the line). Therefore, the chosen symmetry breaking constraint consists in forcing the white dot to lie in the bottom-left quadrant.

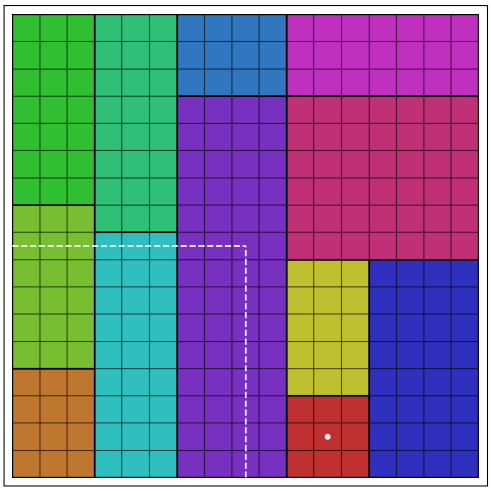
2.3 Rotation

The next step is to implement circuit rotations. Since the circuits are rectangular and nothing is known *a priori* about whether, keeping them in their base orientation, they would wedge perfectly without gaps, it may be possible that the minimal height of the chip is obtained by rotating some of the circuits. For example, the difference between the 2 solutions for Instance 10 is shown in Fig. 2. Even if, at least for the given instance, rotating does not actually reduce the chip height, it can still be observed that some circuits get rotated by the solver.

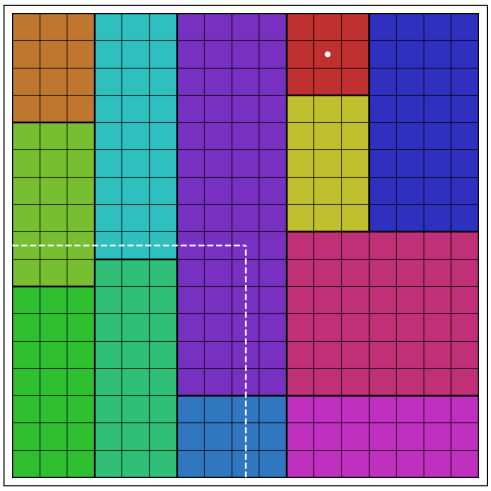
These rotations are handled differently in each implementation, but the basic idea is to offer both the normal and the rotated circuit to the solver and to impose an exclusive disjunction between the two orientations. In



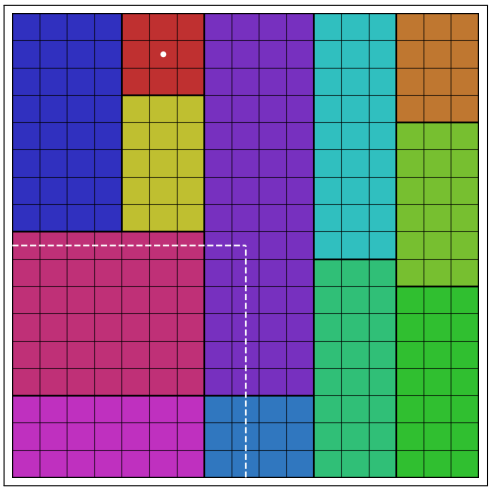
(a) Configuration 1.



(b) Configuration 2.

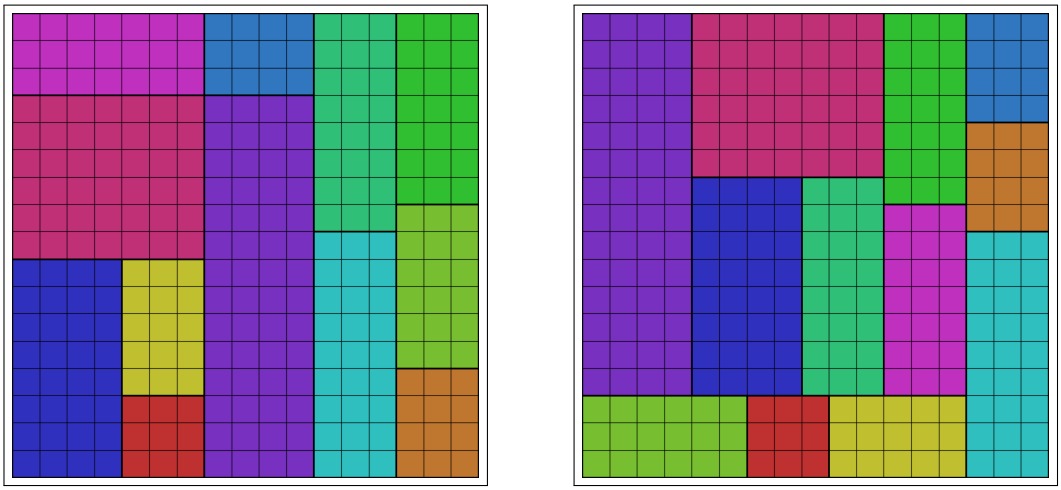


(c) Configuration 3.



(d) Configuration 4.

Figure 1: Symmetrical configurations of Instance 10, with a white dot in the center of the smallest circuit and a dashed white line indicating the bottom-left quadrant.



(a) Without rotations. (b) With rotations.

Figure 2: Solutions of Instance 10 with and without enabled rotations.

particular, it is useful to create 3 auxiliary variables:

- `rotated`: a boolean vector of flags signalling whether each circuit is rotated;
- `new_inst_x`: a list containing the actual horizontal dimensions of the circuits;
- `new_inst_y`: a list containing the actual vertical dimensions of the circuits;

This way, since boolean values are by nature exclusive, the dimensions of each circuit can be linked to the respective boolean value, effectively creating an exclusive disjunction between the two orientations.

2.4 Improvements

The main improvement of the model consists in adding boundaries to `chip_h`. The minimum height is fairly easy to evaluate, since in the best case scenario (i.e. if the circuits leave no gaps) `min_h = total_area / chip_w`, where `total_area` is calculated by simply adding up all the products of the circuits'

dimensions. Computing the maximum height, on the other hand, is more subtle. A naive approach consists in simply adding all heights together, considering the worst case as that where all circuits are stacked vertically. This is of course a reasonable upper boundary, but, as the number of circuits increases, `max_h` greatly overestimates the actual height.

A more clever solution can be reached with minimal preprocessing, reasoning as follows. If circuits are ordered by height and width, the following code can be run:

```
chip_cumulative = chip_w
k = 0
heights = []
while k < len(inst):
    if inst[k][0] <= chip_cumulative:
        chip_cumulative -= inst[k][0]
        heights.append(inst[k][1])
        del inst[k]
    else:
        k += 1
max_h += max(heights)
```

The reasoning is the following. The widths of the circuits are compared with the free horizontal space (`chip_cumulative`) and, if a circuit fits, the free space is decreased. When no more circuits fit on the line, the maximum height is taken among the located circuits and it is added to the total `max_h`. The process is then repeated for the next lines as long as there are circuits left. This way, a much more optimistic, but strictly valid, `max_h` is computed, as shown in Fig. 3.

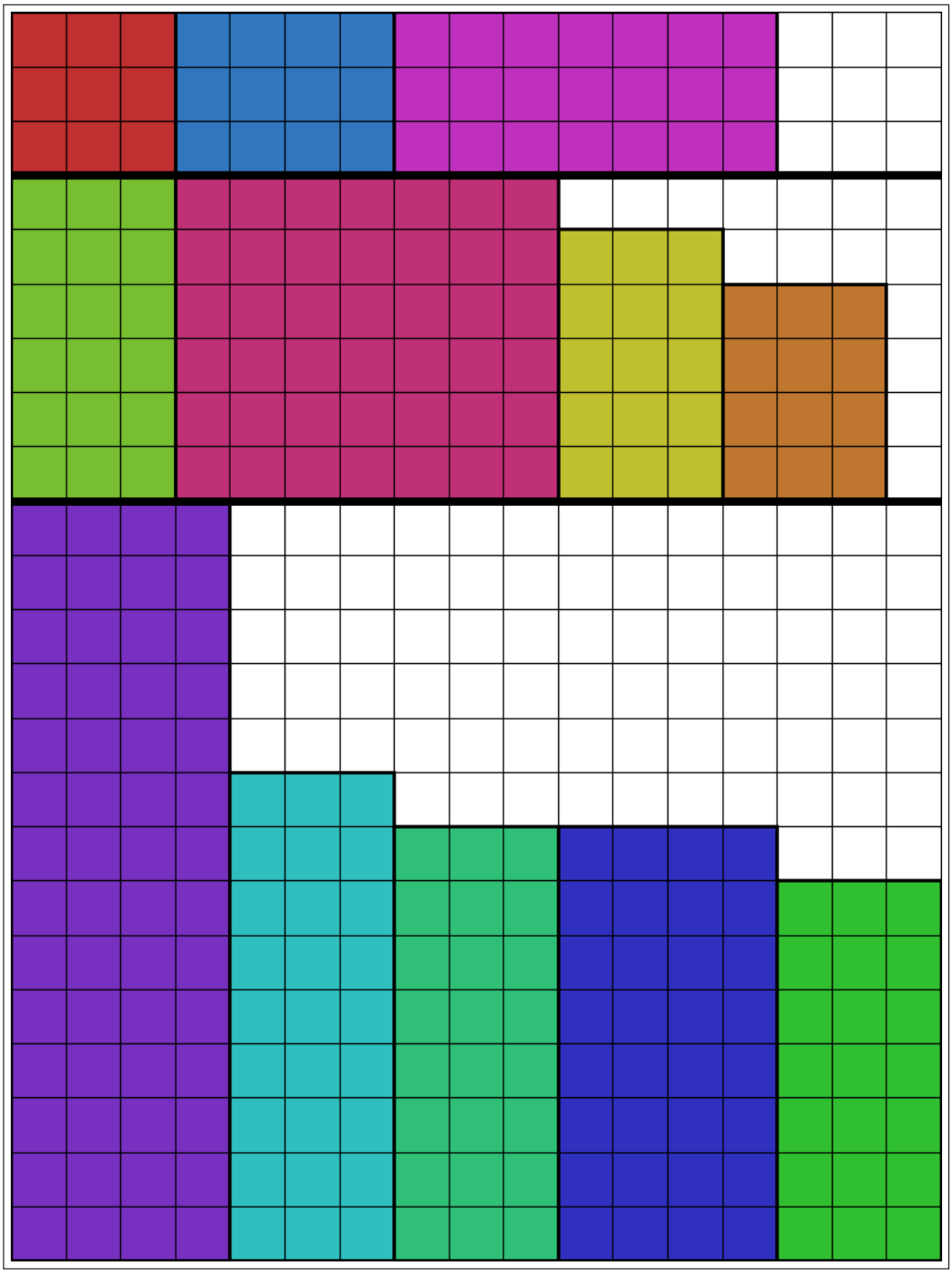


Figure 3: A graphical representation of the `max_h` algorithm for Instance 10.

3 Constraint Programming

The first paradigm used to solve the combinatorial problem of VLSI is Constraint Programming (CP), developed using MiniZinc as the modeling language. Two different models are used: `cp_normal.mzn`, which solves the problem by using exactly the instances defined as input in the project, and `cp_rotation.mzn`, which goes one step further, allowing also the rotation of the circuits themselves.

3.1 Variables

To start, `cp_normal.mzn` receives as input the variables `chip_w`, `n`, `inst_x`, `inst_y`, `min_h`, `max_h` and `min_index`, already defined in Section 2. Along with these, 3 additional variables are added to the output of the file:

- `bl_x`: an array of size `n` with domain `0..(chip_w - min(inst_x))`, containing the horizontal coordinates of the bottom-left corners of the circuits;
- `bl_y`: an array of size `n` with domain `0..(max_h - min(inst_y))`, containing the vertical coordinates of the bottom-left corners of the circuits;
- `chip_h`: a variable with domain `min_h..max_h`, which must be equal to the maximum circuit's upper edge, which represents the effective height of the chip and is the objective function to minimize.

3.2 Constraints

After having carried out a series of tests in which multiple constraints were developed and implemented together measuring the relative performances, the best model, consisting in four different constraints, was finally reached.

3.2.1 Boundaries consistency

This constraint ensures that no circuit is placed totally or partially outside of the chip. Since MiniZinc does not provide any similar default predicate,

the boundaries consistency constraint are implemented as follows:

```
constraint forall(i in RANGE) ((bl_x[i] + inst_x[i] <= chip_w)
↪  /\ (bl_y[i] + inst_y[i] <= chip_h));
```

Using a `forall` loop, it is verified that the sum of each circuit's bottom-left corner and its width or height is never greater than the width of the chip (`chip_w`) or its height (`chip_h`) respectively.

3.2.2 Non-overlapping

The non-overlapping constraint consists in imposing that the circuits within the chip do not overlap, ensuring that two different bottom-left corners do not have the same coordinates and that the circuits do not collide. This constraint is implemented using as a global constraint the predicate `diffn`, provided by MiniZinc, that performs exactly this check:

```
constraint diffn(bl_x, bl_y, inst_x, inst_y);
```

This constraint and the boundaries consistency one are necessary and sufficient to solve the problem well.

3.2.3 Cumulative

An implied constraint was also added to the model to improve propagation. The `cumulative` constraint is actually a scheduling constraint provided by MiniZinc and typically used to make sure that certain tasks do not overlap, but it adapts well to the VLSI problem:

```
constraint cumulative(bl_y, inst_y, inst_x, chip_w);
constraint cumulative(bl_x, inst_x, inst_y, chip_h);
```

The MiniZinc reference manual explains how the cumulative constraint "requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time." So, the first constraint requires that at any position along the horizontal axis, the required shared resource (i.e. the width of each circuit

`inst_x` in such position) of a circuit whose vertical coordinate is `bl_y` and whose height is `inst_y` never exceeds the width of the chip, `chip_w`. The second constraint instead requires that at any position along the vertical axis, the required shared resource (i.e. the height of each circuit `inst_y` in such position) of a circuit whose horizontal coordinate is `bl_x` and whose width is `inst_x` never exceeds the height of the chip, `chip_h`.

3.2.4 Symmetry breaking

Finally, a symmetry breaking constraint is added. Its purpose is to avoid generating solutions that are symmetrical to each other, especially to prune branches of the solution tree that lead to non-acceptable states. As explained in Section 2.2.3, the goal of the symmetry breaking constraint is to force the center of the smallest circuit to be located in the lower left quadrant. To do so, `min_index` is used to compute the center of the smallest circuit and to make sure that it lies in the lower-left quadrant, given by half the width and height of the chip:

```
constraint symmetry_breaking_constraint(((2 * bl_x[min_index] +
↪  inst_x[min_index]) <= chip_w) /\ ((2 * bl_y[min_index] +
↪  inst_y[min_index]) <= chip_h));
```

3.3 Rotation

The problem requirements do not provide the possibility to rotate the circuits to perform a better placement inside the chip optimizing the height. To add this feature a new model called `cp_rotation.mzn` is created, which is very similar to `cp_normal.mzn`, with just a few changes. A new array of booleans called `rotated` is created, where the `k`-th element is `True` if the `k`-th circuit must be rotated in the final result, `False` otherwise. Two other arrays of integers called `new_inst_x` and `new_inst_y` are created, which contain the actual horizontal and vertical dimensions:

```
array[RANGE] of var bool: rotated;
array[RANGE] of var int: new_inst_x;
```

```
array[RANGE] of var int: new_inst_y;
new_inst_x = [if rotated[k] then inst_y[k] else inst_x[k] endif
  ↪  | k in RANGE];
new_inst_y = [if rotated[k] then inst_x[k] else inst_y[k] endif
  ↪  | k in RANGE];
```

All previous constraints are then modified in such a way as to no longer use the default values of `inst_x` and `inst_y`, but rather the actual dimensions `new_inst_x` and `new_inst_y`. Finally, since it is not possible to predict if a circuit will be rotated or not in the final solution, the domains of the variables `bl_x` and `bl_y` are updated, setting as upper limit the minimum value present indifferently in `inst_x` or `inst_y`, called `min_size` and defined as:

```
int: min_size = min(inst_x ++ inst_y);
```

3.4 Search and Restart

MiniZinc provides to the users many different solvers and the possibility to use various type of search and restart algorithms. Among the natively available solvers, only 2 are based on CP: Gecode and Chuffed. The latter is used, because as explained by the reference manual of MiniZinc it is based on lazy clause generation and adapts techniques from SAT solving as activity-based search heuristic. In order to take full advantage of Chuffed's performance, the free search option is used: this allows the solver to switch between the search exploiting annotations and the activity-based one. Then, in order to optimize the solver at its best, an optimization level O5 is used: this performs root-node-propagation with Gecode and probe values of all variables at the root node. With this kind of settings, different types of variable choice annotations were tested. Search annotations in MiniZinc specify how to search in order to find a solution to the problem. The annotation is attached to the solve item, after the keyword `solve`. Six different types of annotations were tried:

- `(input_order, indomain_min)`: chooses the variable in order from array, assigns to each variable its smallest domain value;

- `(first_fail, indomain_min)`: chooses the variable with the smallest domain size, assigns to each variable its smallest domain value;
- `(dom_w_deg, indomain_min)`: chooses 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, assigns to each variable its smallest domain value;
- `(input_order, indomain_random)`: chooses variables in order from array, assigns to each variable a random value from its domain;
- `(first_fail, indomain_random)`: chooses the variable with the smallest domain size, assigns to each variable a random value from its domain;
- `(dom_w_deg, indomain_random)`: chooses 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, assigns to each variable a random value from its domain.

Any kind of depth-first search for solving optimization problems suffers from the problem that wrong decisions made at the top of the search tree can take an exponential amount of search to undo. One common way to avoid this problem is to restart the search from the top thus having a chance to make different decisions. MiniZinc includes annotations to control restart behaviour. These annotations, like other search annotations, are attached to the `solve` item of the model. The different restart annotations control how frequently a restart occurs. Restarts occur when a limit in nodes is reached, where search returns to the top of the search tree and begins again. Four different types of restart annotations were tried:

- `restart_constant(<scale>)`: where `<scale>` is an integer defining after how many nodes to restart;
- `restart_linear(<scale>)`: where `<scale>` is an integer defining the initial number of nodes before the first restart, the second restart gets twice as many nodes, the third gets three times, etc.;

- `restart_geometric(<base>,<scale>)`: where `<base>` is a float and `<scale>` is an integer, the `k`-th restart has a node limit of `<scale>` $*$ `<base>`$^{\texttt{k}}$;

- `restart_luby(<scale>)`: where `<scale>` is an integer, the `k`-th restart gets `<scale>` $*$ `L[k]` where `L[k]` is the `k`-th number in the Luby sequence, which looks like 1 1 2 1 1 2 4 1 1 2 1 1 2 4 8 ..., i.e. it repeats two copies of the sequence ending in $2^{\texttt{k}}$ before adding the number $2^{\texttt{k}+1}$.

Combining different types of search and restart strategies, it was found that the best possible combination is formed by `first_fail` as variable choice, `indomain_min` as variable constraint and `restart_luby(250)` as restart type.

3.5 Results

As shown in Fig. 4, the normal model without rotations solves all input instances except 40 with a maximum time of around 3 minutes. The addition of the symmetry breaking constraint allows to solve some instances that could not previously be solved, also speeding up the resolution time in general and reducing the number of nodes in the tree, proving that it works properly. By adding the ability to rotate the circuits in the solutions, slightly worse times are obtained instead. This behavior is predictable, since the solver must in this case perform many more combinations. Also, since each instance already allows for the minimum chip height without the rotation, adding the rotation does not lead to better results either. The model with rotations therefore solves all instances except for 22, 25, 30, 32, 37, 38, 39, and 40, with a maximum time of around 4 minutes.

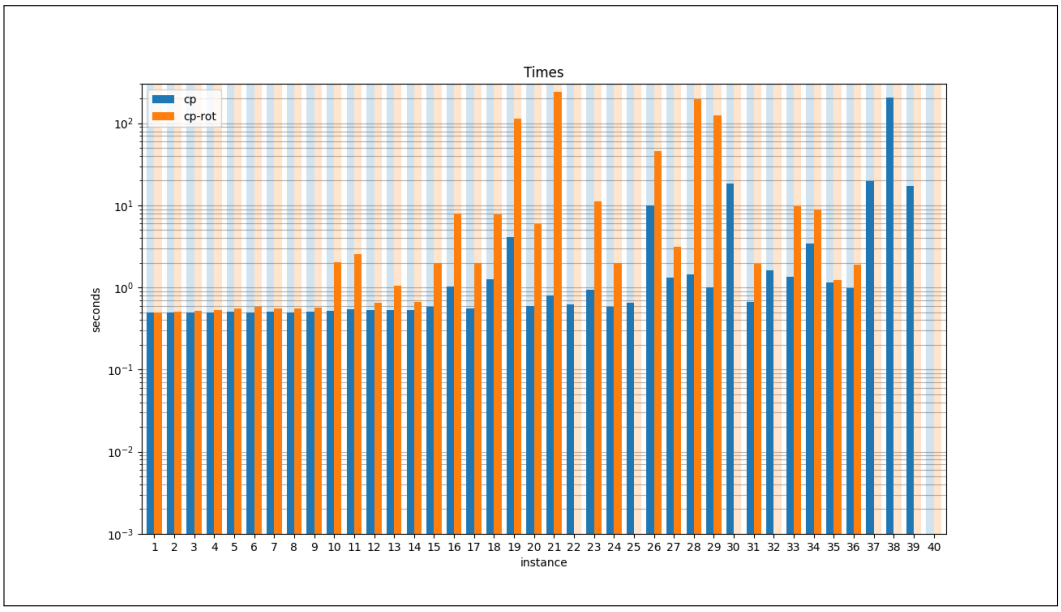


Figure 4: Constrain Programming times.

4 Propositional Satisfiability

The second paradigm used to solve the combinatorial problem of VLSI is Propositional SATisfiability (SAT), developed using Z3. Instances are solved both with and without rotations.

4.1 Variables

To start, the solver receives as input the variables `chip_w`, `n`, `inst_x`, `inst_y`, `min_h`, `max_h` and `min_index`, already defined in Section 2. Furthermore, in this case an `opt` variable is created, which contains the optimizer used by Z3 and to which all the constraints are added with an `add()` function. The solver also places logical conjunctions between constraints. Along with the already defined ones, 3 auxiliary variables are created:

- `chip`: a `chip_w` $\times$ `max_h` $\times$ `n` matrix of booleans where any element `chip[i][j][k]` is `True` if circuit `k` is present at coordinates `(i,j)`, `False` otherwise;
- `corners`: a `chip_w` $\times$ `max_h` $\times$ `n` matrix of booleans where any element `corners[i][j][k]` is `True` if the bottom-left corner of circuit `k` has coordinates `(i,j)`, `False` otherwise;
- `chip_h`: an integer which represents the effective height of the chip and is the objective function to minimize.

4.2 Constraints

After having carried out a series of tests in which multiple constraints were developed and implemented together measuring the relative performances, the best model was finally reached.

4.2.1 Structural

Contrary to the other paradigms, SAT requires some structural constraints. This is necessary because, dealing with boolean matrices, the concepts of

height and width occupied by each chip is not so simply manageable. The way it is done is by cycling first among all the circuits and creating a list of booleans `temp_corners`. Then, for every coordinate, another list of booleans `temp` is created. Next, all coordinates are cycled through again inside the higher-level cycle, in order to effectively have all possible pairs of coordinate. At this point, it is checked whether the new coordinates `ii` and `jj` would be part of the circuit if its coordinates were `i` and `j`: if this is the case, an implication constraint is added, signifying that, in case `(i,j)` become the location of the corner of chip `k`, then `chip[ii][jj][k]` must also be `True`. In addition, for every circuit, it is imposed that exactly one position must be occupied by the corner, as indicated by the last two lines of code:

```
for k in range(n):
    temp_corners = []
    for i in range(chip_w):
        for j in range(max_h):
            temp = []
            for ii in range(chip_w):
                for jj in range(max_h):
                    if (ii in range(i, i + inst_x[k]) and jj in
                     ↪  range(j, j + inst_y[k])):
                        temp.append(chip[ii][jj][k])
                    else:
                        temp.append(Not(chip[ii][jj][k]))
            opt.add(Implies(corners[i][j][k], And(temp)))
            temp_corners.append(corners[i][j][k])
    opt.add(at_least_one(temp_corners))
    opt.add(at_most_one(temp_corners))
```

4.2.2 Boundaries consistency

The boundaries consistency constraint is created by cycling through all the coordinates and all the circuits and imposing that, if corner `k` is at coordinates `(i,j)`, then both coordinates, summed with the circuit's dimensions, must

be at most equal to the dimensions of the chip:

```
for i in range(chip_w):
    for j in range(max_h):
        for k in range(n):
            opt.add(Implies(corners[i][j][k], And(i + inst_x[k]
             ↪  <= chip_w, j + inst_y[k] <= chip_h)))
```

4.2.3 Non-overlapping

Non-overlapping is achieved in a very straightforward way for both `chip` and `corners`. First, both dimensions are cycled and 2 lists of booleans are created. Then, it is imposed that only one circuit (or corner) can be present at each spatial position:

```
for i in range(chip_w):
    for j in range(max_h):
        temp_chip = []
        temp_corners = []
        for k in range(n):
            temp_chip.append(chip[i][j][k])
            temp_corners.append(corners[i][j][k])
        opt.add(at_most_one(temp_chip))
        opt.add(at_most_one(temp_corners))
```

4.2.4 Symmetry breaking

Symmetry breaking is achieved in the same way as when using the other approaches: it is imposed that the center of the smallest circuit lies in the bottom-left quadrant of the chip. In order to do so, the code first creates an empty list `symmetry_breaking` and then cycles all the spatial positions. At this point, it is imposed, using conjunctions, that the center of the smallest circuit lies in the bottom-left quadrant of the chip:

```
symmetry_breaking = []
for i in range(chip_w):
```

```
    for j in range(max_h):
        symmetry_breaking.append(And(corners[i][j][min_index],
        ↪ And(2 * i + inst_x[min_index] <= chip_w, 2 * j +
        ↪ inst_y[min_index] <= chip_h)))
opt.add(Or(symmetry_breaking))
```

Even if, at first glance, it may seem that using a conjunction is wrong, since of course the corner is not at all positions at the same time, the elements of `symmetry_breaking` are actually added as a disjunctive constraint, therefore effectively forcing the center *at least* once in the quadrant. But of course, the center can occupy only one position at a time, thanks to the non-overlapping constraint.

4.3 Rotation

To deal with rotation, a new list of booleans called `rotated` is created, where the `k`-th element is `True` if the `k`-th circuit must be rotated in the final result, `False` otherwise. Two other arrays called `new_inst_x` and `new_inst_y` are created as well, which contain the actual horizontal and vertical dimensions:

```
rotated = BoolVector("rotated", n)
new_inst_x = [If(rotated[k], inst_y[k], inst_x[k]) for k in
↪ range(n)]
new_inst_y = [If(rotated[k], inst_x[k], inst_y[k]) for k in
↪ range(n)]
```

All previous constraints are then modified in such a way as to no longer use the default values of `inst_x` and `inst_y` but the actual dimensions `new_inst_x` and `new_inst_y`.

4.4 Results

As shown in Fig. 5, SAT performs pretty badly compared to the other methods: the normal model without rotations only solves instances from 1 to 9, with the rotation model solving only the first 7.

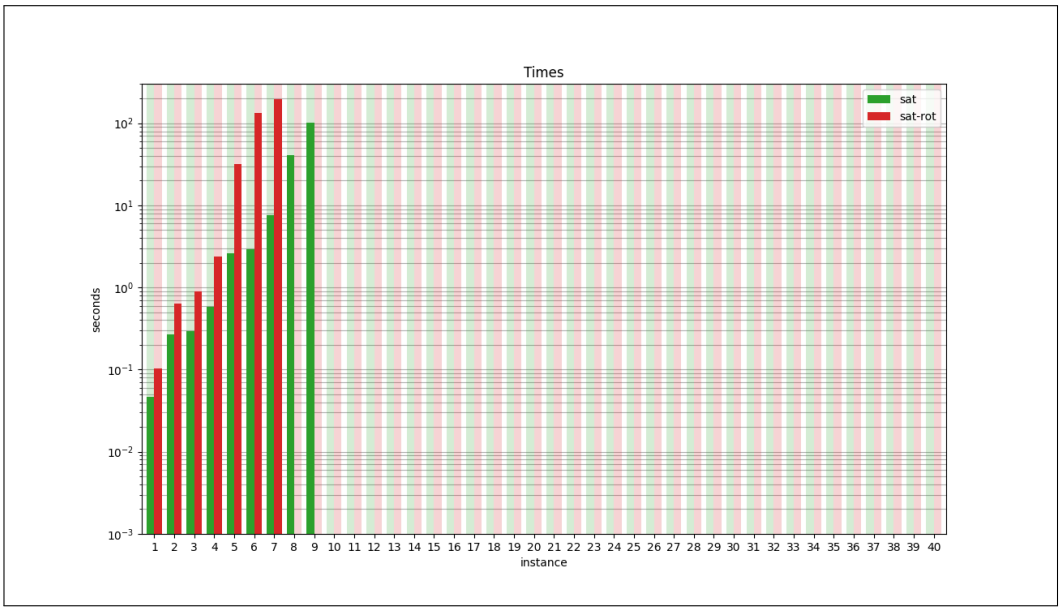


Figure 5: Propositional Satisfiability times.

5 Satisfiability Modulo Theories

The last solver used in the project is Satisfiability Modulo Theories (SMT). Once again, the instances are solved both in the standard way and by allowing circuits to be rotated.

5.1 Variables

To start, the solver receives as input the variables `chip_w`, `n`, `inst_x`, `inst_y`, `min_h`, `max_h` and `min_index`, already defined in Section 2. In addition, the usual `bl_x`, `bl_y`, and `chip_h` variables are created as well.

5.2 Constraints

The constraints implemented for SMT are the same used in CP, with the only difference that in this case the cumulative constraint was removed because after a series of tests it was found to worsen the general performance. In addition, since Z3 does not allow to predefine an initial domain for variables, it was necessary to add constraints limiting the domain of variables.

5.2.1 Boundaries consistency and domain reduction

As shown in the code, the idea of the constraints is the same as used in CP and SAT. Each circuit must not exceed the size limits of the chip:

```
for k in range(n):
    opt.add(bl_x[k] >= 0)
    opt.add(bl_x[k] + inst_x[k] <= chip_w)
    opt.add(bl_y[k] >= 0)
    opt.add(bl_y[k] + inst_y[k] <= chip_h)
```

As with CP, it is also possible to narrow the domain of `chip_h` between the previously computed `min_h` and `max_h` values in order to reduce the size of the node tree and speed up the search:

```
opt.add(chip_h >= min_h)
opt.add(chip_h <= max_h)
```

5.2.2 Non-overlapping

Since Z3 does not provide builtin predicates to define constraints like MiniZinc, the non-overlapping constraint is implemented by decomposing the constraint used in CP. The idea is to verify that for each pair of circuits within the chip, either the horizontal coordinates of the bottom corners cause the two circuits to be next to each other, or the vertical coordinates of the left corners are in a position such that the two circuits are on top of each other. If any of these four conditions are met, it is assured that the two circuits will not be overlapping, so there is a logical disjunctive operator between the four constraints:

```
for k in range(n):
    for l in range(k + 1,n):
        opt.add(Or((bl_x[k] + new_inst_x[k] <= bl_x[l]),
        ↪  (bl_x[k] >= bl_x[l] + new_inst_x[l]), (bl_y[k] +
        ↪  new_inst_y[k] <= bl_y[l]), (bl_y[k] >= bl_y[l] +
        ↪  new_inst_y[l])))
```

5.2.3 Symmetry breaking

The last constraint implemented is the symmetry breaking one, whose idea is explained in details in Section 2.2.3, while the implementation is similar to that of CP (Section 3.2.4).

```
opt.add(And(((2 * bl_x[min_index] + new_inst_x[min_index]) <=
↪  chip_w), ((2 * bl_y[min_index] + new_inst_y[min_index]) <=
↪  chip_h)))
```

5.3 Rotation

By changing the value of `rotation`, it is possible to allow the solver to search for a solution in which some circuits are rotated. To do this, a vector of booleans called `rotated` is created: the value of the `k`-th element indicates whether the `k`-th circuit should be rotated or not in the final solution. The arrays `new_inst_x` and `new_inst_y` then contain the actual dimensions of the circuits. The corresponding implementation is:

```
rotated = BoolVector("rotated", n)
new_inst_x = [If(rotated[k], inst_y[k], inst_x[k]) for k in
↪ range(n)]
new_inst_y = [If(rotated[k], inst_x[k], inst_y[k]) for k in
↪ range(n)]
```

5.4 Results

As shown in Fig. 6, the normal model without rotations solves all input instances until 18 with a maximum time of around 25 seconds. After that, the results worsen as the difficulty of the instances increases, only managing to solve Instances 20, 21, 23, 24, 26, 27, 28, 31, 33, and 36. The fact that some apparently more difficult instances are solved than others probably depends on the input order of the circuits. By adding the ability to rotate the circuits in the solutions, slightly worse times are obtained instead. This behavior is predictable, since the solver must in this case perform many more combinations. Also, since each instance already allows for the minimum chip height without rotations, adding them does not lead to better results either. The model with rotations therefore solves all instances until 15, together with a few other more difficult configurations. Surprisingly, however, rotations allow Instances 29, 34, and 35, which are not solved using the normal model, to be solved. Again, this behavior probably depends on how the circuits are ordered in the instances.

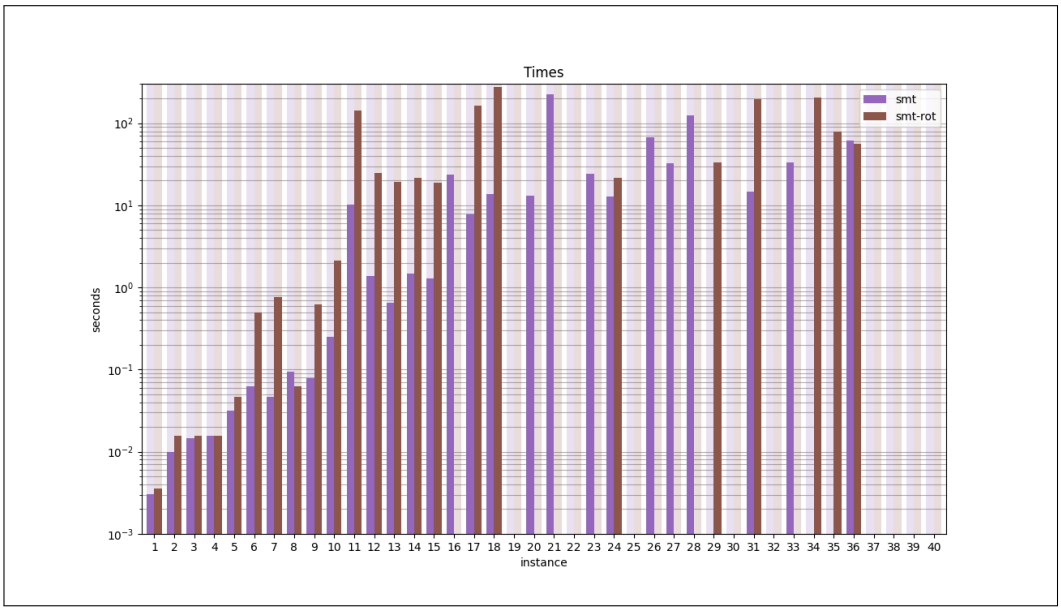


Figure 6: Satisfiability Modulo Theories times.

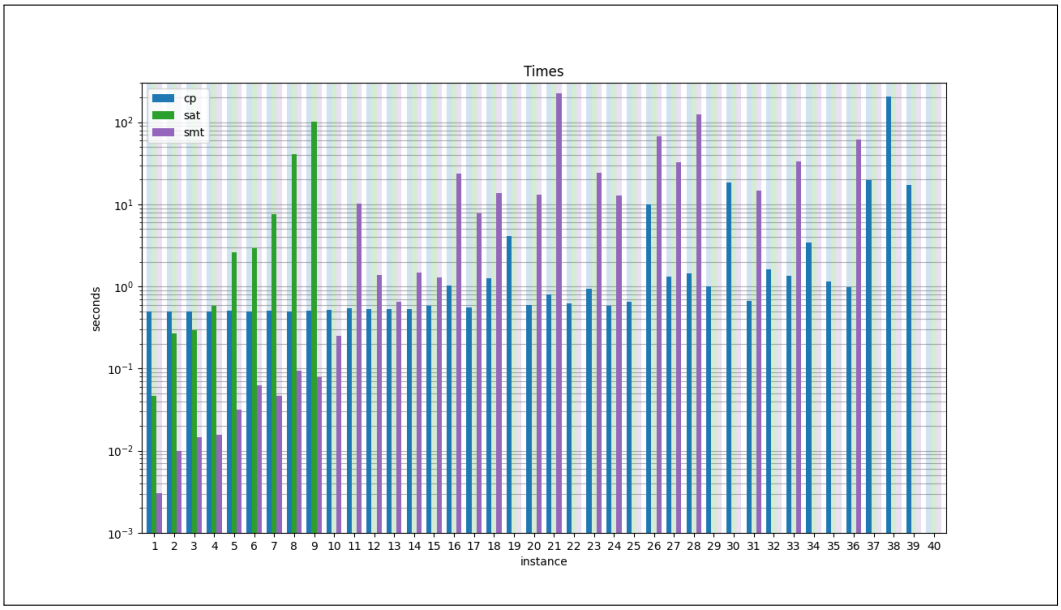


Figure 7: Times without rotations.

6 Results

6.1 Normal

The times shown in Fig. 7, despite generally increasing with the instance numbers, exhibit some interesting characteristics.

The first thing that can be noted is that the times achieved using SAT show the most regular behavior: they increase almost exponentially for the first 9 instances, exceeding the 5-minute limit on all successive inputs. This is reasonable, as the boolean matrices have dimensions `chip_w` $\times$ `max_h` $\times$ `n`: since all 3 dimensions of the matrices increase with the instance number, the exponential growth is predictable.

Looking at the SMT times, a pattern similar to that of SAT can be detected until Instance 11. After this entry, though, any regularity is lost. Even thouth all the times are quite high, they do not really seem to depend on the instance number, and even very similar instances show completely different behaviors. A possible explanation of this phenomenon may lie in the input order of the various circuits, or maybe in the way Z3 itself deals

with very large vectors.

The most surprising times, however, are the CP ones. This method is by far the best, being the only one which solves all instances but the last one (which is completely out of scale compared to the other 39), but complexity seems to have an impact even smaller than with SMT. In the beginning, CP is clearly the slowest method, falling short even of the very badly performing SAT. However, in contrast to the early growth of SAT and SMT times, CP stays stably below the second mark for most of the first 20 chips. Even when it starts to struggle with later instances, the times never grow as large as the SMT ones. This shows that, even if more time is needed in setting up the problem constraints properly, the solving part itself tends to run with much more constant and reliable times compared to the other methods.

6.2 Rotated

The behaviors of the different methods when rotations are enabled follow quite closely the trends illustrated in Section 6.1, with a general worsening. This general increase in the times is in line with the hypothesis, since the procedure of rotating inevitably takes some more time compared to simple translations. Again, SAT rapidly increases its solving time, exceeding the time limit after only 7 instances. CP starts with times similar to Fig. 7 and never increases too much: in general, it tends to either solve instances in a reasonable time, or not to solve them at all. SMT, on the other hand, exhibits the strangest behavior: it starts very fast, as usual, but never actually stabilizes as was the case without rotations. Instead, some random higher instances are sometimes solved, without an apparent pattern: again, it is suspected that the order in which the circuits are passed to the solver may influence its satisfiability more than expected. In conclusion, even when rotations are allowed, CP emerges as the clear winner: net of the usual slower start, its ability to solve very complicated instances in reasonable times makes it the most consistent and reliable method.

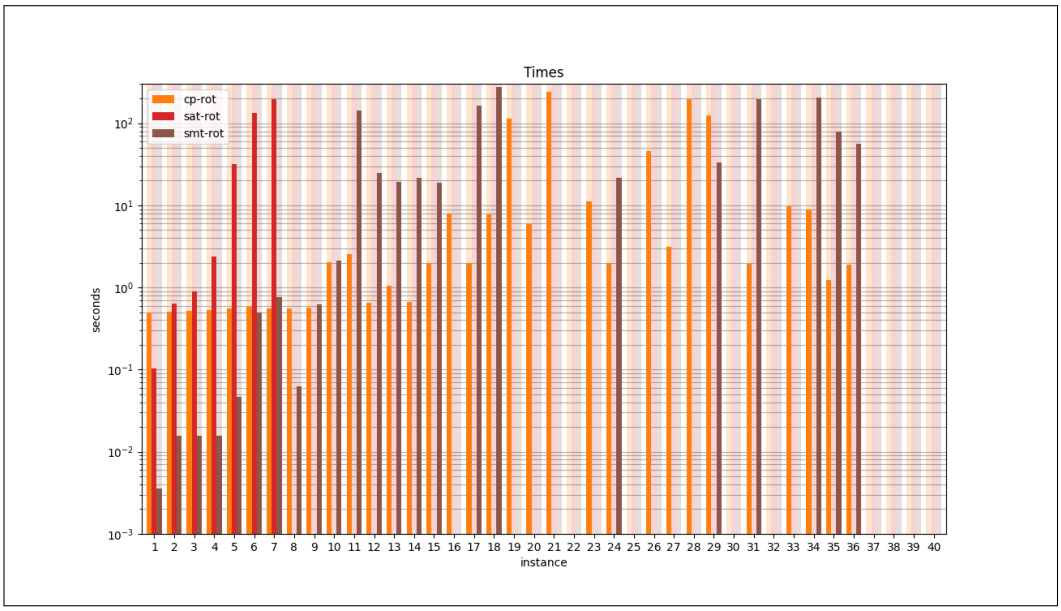


Figure 8: Times with rotations.