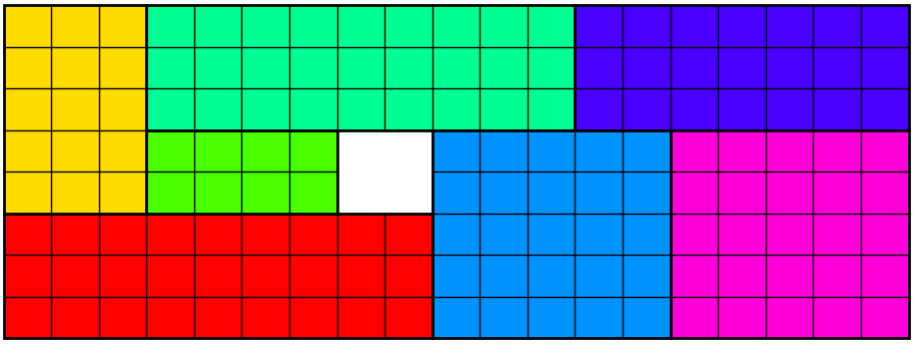
Very Large Scale Integration

Project Work in Combinatorial Decision Making and Optimization



Riccardo De Matteo - riccardo.dematteo@studio.unibo.it
Alessandro Stockman - alessandro.stockman@studio.unibo.it
August 2021

Contents

1 Introduction

Very large-scale integration (**VLSI**) is the process of creating an integrated circuit (IC) by integrating circuits into silicon chips. With the possibility of shrinking more and more the transistor sizes, the need to pack them in smaller areas has grown accordingly. For this reason, developing smart methods for spacial arrangements of the different components on the chips has become crucial for developing modern devices. The aim of this project is to develop optimization strategies (using established technologies), that given a fixed plate-width and a list of rectangular circuits, is able to place them on the plate so that the length of the final device is minimized.

2 CP

2.1 Model

As usual, we modelled the problem by considering the bottom left corner of each circuit as its identifying coordinate. The following variables are used:

2.1.1 Input variables

- An integer `plate_width`: the width of the board
- A set of integers `circuits` which spans the interval from 1 to the total number of circuits to be placed
- Two arrays `widths` and `heights` that store for each circuits both the width and the height

```
int: plate_width;
int: num_circuits;
set of int: circuits = 1..num_circuits;

array[circuits] of int: widths;
array[circuits] of int: heights;
```

2.1.2 Output variables

- Two arrays of variables `x` and `y` that will contain respectively the x-coordinate and the y-coordinate of the bottom-left corner for each rectangle
- the variable `plate_height` which store the minimum height that the board should have in order to accommodate every rectangle of the specific instance

```
array[circuits] of var 0..plate_width-min(widths): x;
array[circuits] of var 0..upper_bound-min(heights): y;

var lower_bound..upper_bound: plate_height;
```

2.1.3 Objective function

- Solve minimize `plate_height`

2.2 Reducing variables domain

A constraint programming solver basically finds a solution by smartly scanning the search space. For this reason reducing the domain of the variables should provide a significant improvement for the model.

- For each variable of the array `x` we defined a possible range. In particular the bottom left corner of each rectangle cannot be placed farther on the right along the x axis than the width of board minus its width; otherwise it would fall outside the board.
- A similar constraint on the domain of the `y` variables can be defined: the bottom left corner of each rectangle cannot be placed higher on the y axis than the current considered height of the plate minus its height.

```
constraint forall(c in circuits) (x[c]<=plate_width-widths[c])::domain;
constraint forall(c in circuits) (y[c]<=plate_height-heights[c])::domain;
```

- We also defined an upper and lower bounds for the `plate_height`:
  - Lower bound: The total height of the board clearly cannot be smaller than the height of the tallest rectangle
  - Upper bound: The total height of the board cannot exceed the sum of all the heights of each rectangle since that obviously a trivial solution

```
var lower_bound..upper_bound: plate_height;
int: upper_bound = sum(heights); % Plate height upper bound: circuits height sum
int: lower_bound = max(heights); % Plate height lower bound: max circuit height
```

2.3 Constraints

2.3.1 Main Constraint

The most straightforward constraint is the one limiting each rectangle to appear inside the board without falling outside the perimeter

```
constraint forall(c in circuits) (
    x[c] + widths[c] <= plate_width /\ y[c] + heights[c] <= plate_height
);
```

2.3.2 Global Constraint

Another tool that solvers employ in solving a CSP/COP is constraint propagation, which consist in examining the constraints in order to remove inconsistent values from the domain of the variables. Solving our problem, which involves multiple constraints, necessitates that propagation algorithms interact between them and that each could wake up when necessary to propagate again an already propagated constraint. For this reason we thought that using Global constraint would be beneficial to the solution of our problem, since they embed specialized propagation algorithms that can be potentially much more efficient than a generalized propagation. Moreover they allow an easier expression of the statement with respect to the model. We therefore deployed two types of global constraints:

- `cumulative` [1]: We borrowed this constraint from the modeling of scheduling problem; in fact it is usually used when we want to constraint the usage of shared resources, by different tasks. Each task is usually associated with a start time, a duration and a resource requirement. We can see our problem as one where there is a fixed capacity resource (`plate_width/height`) and each task (`circuit`) has its own start time (`placement`) and duration (`width/height`).The constraint is repeated twice, once along each axis.

- diffn [2] : The `diffn` constraint occurs in placement and scheduling problems. It is a two-dimensional non-overlapping constraint which holds if, for each pair of shapes, there exists a dimension in which their projections doesn't overlap. In this way we prevented the circuits from overlapping.

```
constraint diffn(x, y, widths, heights);
constraint cumulative(x, widths, heights, plate_height);
constraint cumulative(y, heights, widths, plate_width);
```

2.4 Symmetry Breaking Constraints

Symmetry is very common in constraint satisfaction and optimisation problems. The modelling technique for dealing with symmetry is called symmetry breaking, and in its simplest form, involves adding constraints to the model that rule out all symmetric variants of a solution.

- Dominance [3]: This constraint type allow us to rule out unpromising configurations which appear much worse than others. In particular any solution in which circuits are adjacent without leaving blank spaces between them, when

possible, is clearly better than the ones in which they are farther apart. We can thus say that any of these alternative solutions is dominated by the more densely populated one.

```
constraint
  forall(c in circuits)(
    member([0] ++ [x[o] + widths[o] | o in circuits where o != c], x[c]) /\
    member([0] ++ [y[o] + heights[o] | o in circuits where o != c], y[c])
  );
```

For each solution containing a circuit which is not partially adjacent on the bottom and left side either to the plate border or another circuit there exists a better or equal solution not respecting that property.

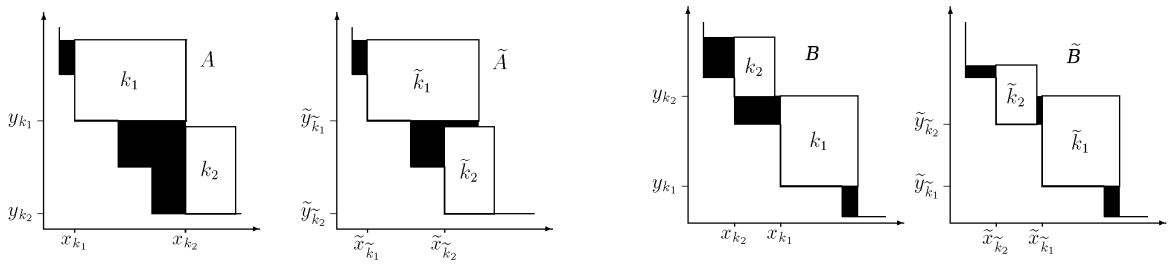


Figure 1: Dominance constraint application effect

- Ordering: To further remove symmetries, we only considered solutions where the relative position of the two largest rectangles [4] was fixed. Any other solution can be mapped to such solution by flipping the board along one or both axes. In particular we decided to relegate the largest rectangle on the lower left with respect to the second largest one. We achieved this by imposing a lexicographical ordering constraint on the two pairs of coordinates referring to the bottom-left corner of such rectangles.

```
 constraint symmetry_breaking_constraint(
  let {
    int: c1 = ordered_c[1], int: c2 = ordered_c[2]
  } in lex_less([y[c1],x[c1]], [y[c2],x[c2]])
);
```

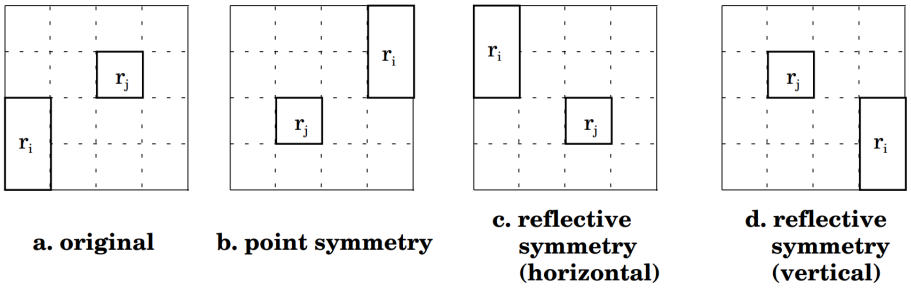


Figure 2: Symmetries between two given rectangles r_1 and r_2

- Circuits equality: When two rectangle have both the same width and the same height, their position is completely interchangeable; thus we can a priori fix the relative position of all the same size rectangles in any of the specific instances.

```
constraint symmetry_breaking_constraint(
  forall (c in circuits) (
    let {
      array[int] of int: equals = [i | i in c+1..num_circuits
      where widths[i] = widths[c] /\ heights[i] = heights[c]]
    } in
      forall (i in 1..length(equals)-1)
      (lex_less([y[equals[i-1]], x[equals[i-1]]], [y[equals[i]],x[equals[i]]]))
  )
);
```

A list comprehension for each set of equal circuits is generated, then, the `x` and `y` axis indexed by the given list are constrained to be in lexicographic order pairwise, such that no permutation of the coordinates is allowed.

2.5 Search

Search annotations in MiniZinc specify how to search in order to find a solution to the problem. 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 solve, it is though possible to specify how the search should be undertaken. The computational cost of solving combinatorial optimisation problems depends strongly on the search tree. The shape of this tree is impacted by propagation, variable ordering and value ordering. The propagation behaviour is a property of the constraint solver.

Solve annotations request that the solver use a search method over finite integer variables: this is the meaning of `int_search`. The second argument specifies the order in which the values should appear in the search tree. In this example `input_order`

imposes that the three variables are selected in the order in which they are given. The final argument `indomain_min` specifies which choice of value for each variable will be explored next—specifically the minimum of the remaining values in the variable's domain.

- No annotations

  ```
  solve minimize plate_height;
  ```

- Sequential search: The idea behind this search annotations relies on the fact that placing bigger circuits should be harder than placing smaller ones, therefore an input ordering heuristic is used to select the variable in a decreasing order by area.

  ```
  solve
    :: seq_search([
        int_search([y[c] | c in ordered_c], input_order, indomain_min,complete),
        int_search([x[c] | c in ordered_c], input_order, indomain_min,complete),
        int_search([plate_height], input_order, indomain_min, complete),
      ])
    minimize plate_height;
  ```

- Height annotation: The idea behind this search annotations relies on the fact that by starting from the lower bound of the plate height, the first satisfiable solution found would be already the best solution.

  ```
  solve
  :: int_search([plate_height], input_order, indomain_min, complete)
  minimize plate_height;
  ```

After many experiments, we obtained our best results with the last search annotation.

2.6 Rotation

The original problem requirements do not allow the rotation of the circuits. If instead the rotation has to be taken into account, some changes to the model need to be put in place. To handle properly the possibility for each rectangle to have height and width swapped, we extended our model:

- We decided to add three more arrays of variables:
  - `r`: A boolean array that stores the information whether each rectangle is rotated or not
  - `actual_widths`: An array of variables that stores the actual width, after the eventual rotation, for each rectangle.

  - `actual heights`: An array of variables that stores the actual heights, after the eventual rotation, for each rectangle.

```
array[circuits] of var bool : r;
array[circuits] of var int: actual_widths =
    [(widths[c] * (1-r[c])) + (heights[c] * r[c]) | c in circuits];
array[circuits] of var int: actual_heights =
    [(heights[c] * (1-r[c])) + (widths[c] * r[c]) | c in circuits];
```

- We implemented one more constraint: for any rectangle whose dimensions assume the same value (namely they are of square shape), we force the corresponding element in the array `r` to be set to 0, thus forbidding its rotation that would not introduce any difference in the final solution.

```
constraint symmetry_breaking_constraint(
  forall (c in circuits) (heights[c] = widths[c] -> r[c] = 0)
);
```

- We investigated a different way to search for solutions. We can construct more complex search strategies using search constructor annotations. The sequential search constructor first undertakes the search given by the first annotation in its list, when all variables in this annotation are fixed it undertakes the second search annotation in particular we added to the sequential search a `bool_search` over the variables of the array right after the `int_search` relative to the `plate_height` since the decision of rotating a circuit should be performed before searching for its coordinates.

2.7 Results and performances

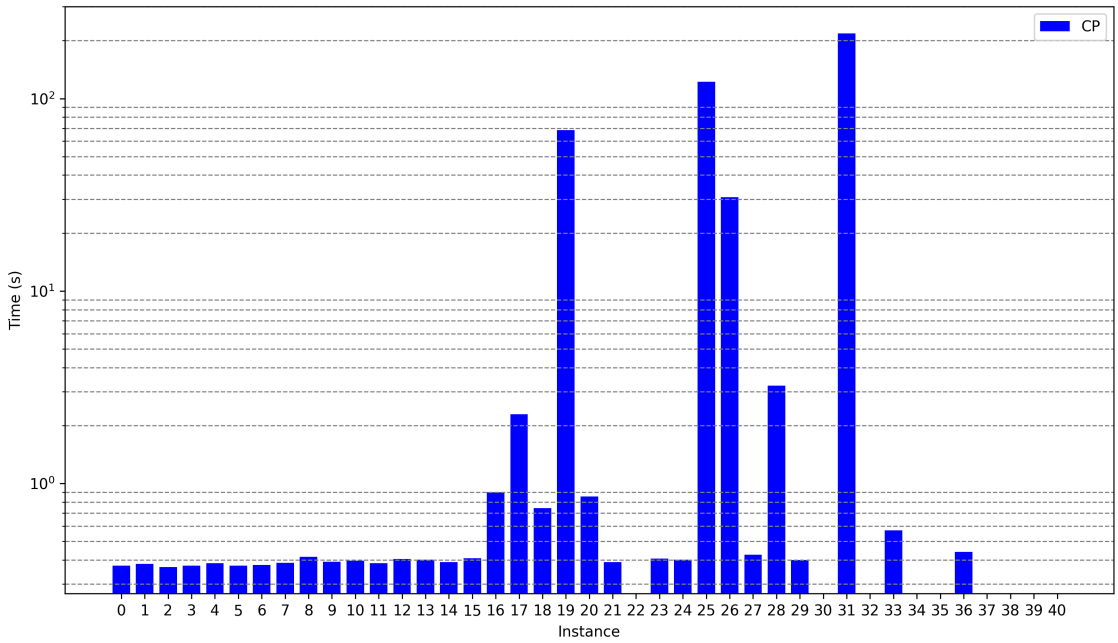


Figure 3: Constraint Programming results

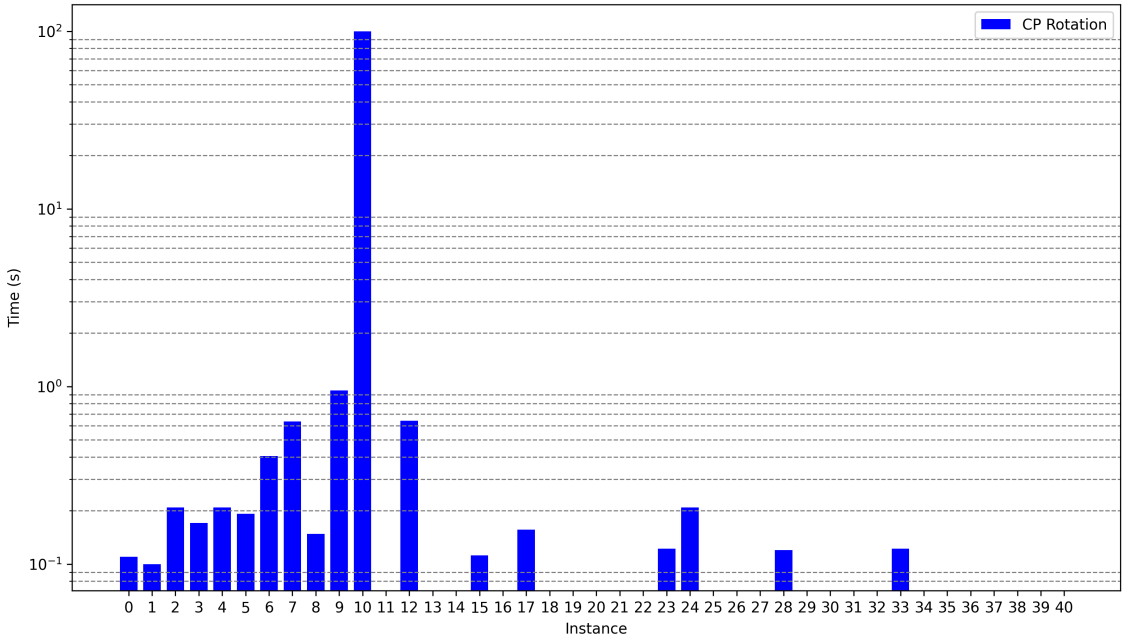


Figure 4: Constraint Programming results with rotation

3 SAT

The Boolean satisfiability problem asks whether it is possible to find a combination of assignments to the boolean input variables for which a given propositional logic formula holds; when this is the case, the formula is considered satisfiable.

A SAT solver is an algorithm that establishes satisfiability of formulas by taking them as input and returning either SAT if it finds a combination of values that can satisfy them or UNSAT otherwise.

3.1 Model

The problem has been modelled by means of a 3-Dimensional boolean array $B_{x,y,k}$, where each variable indicates whether the k circuit occupies the coordinates x, y. In this notation, H and W represents respectively the height and width of the plate, H_k and W_k the height and width of the circuit k and C the set of circuits.

A subset of the board B, which we will call **section**, determines the potential positions that a circuit k can take. If we consider the bottom left corner to be at the coordinates (x, y), the **section** would contain each variable whose indexes are in the ranges of $[x, x + W_k]$ and $[y, y + H_k]$.

In this implementation, H is iterated starting from the lowest value acceptable, given by the minimum height necessary to fit the area of all the circuits put together, and a satisfiability check is performed over each H value.

3.2 Constraints

3.2.1 Circuits placement

For each circuit, exactly one section should be active at any time. For a **section** $S_{x,y,k}$ to be active the conjunction of the variables that it contains should evaluate to true.

$$S_{x,y,k} = \bigwedge_{w_k=0,h_k=0}^{W_k,H_k} B_{x+w_k,y+h_k,k}$$

The set of all the possible sections for a circuit is defined as $S_k = \{ \bigcup_{x=0,y=0}^{W,H} S_{x,y,k} \}$.

To force only one sections to be active, the conjunction of each pair of possible sections is negated, and at the same time, all the sections are put in a disjunction to ensure that at least one of them will be active.

$$\bigwedge_{k \in C} (\bigwedge_{S_1,S_2 \in S_k} \neg(S_1 \wedge S_2) \wedge \bigvee_{S_i \in S_k} S_i)$$

This formula guarantees that every circuit is placed only once inside the board respecting its dimensions.

3.2.2 Non-Overlapping

To achieve a valid solution, a further constraint of non-overlapping is needed. This is done by imposing that at most one circuit exist at each coordinate by negating the conjunction of all the pairs of circuits for every given (x, y).

$$\bigwedge_{x=0,y=0}^{W,H} \bigwedge_{i \in C, j \in C, i \neq j} \neg B_{x,y,i} \wedge B_{x,y,j}$$

3.3 Symmetry Breaking Constraints

Symmetry breaking is achieved by setting up a channeling towards a different model consisting in two projections of the board over the x and y axis, resulting in two 2-Dimensional arrays $X_{x,k}$ and $Y_{y,k}$, which express the presence of the k circuit, respectively in the coordinate x and in the coordinate y.

$$\bigwedge_{k \in C} Y_{y,k} \leftrightarrow (\bigvee_{x=0}^{W} B_{x,y,k})$$

$$\bigwedge_{k \in C} X_{x,k} \leftrightarrow (\bigvee_{y=0}^{H} B_{x,y,k})$$

The symmetry breaking consists in adding a lexicographic ordering constraint between the arrays (X, Y) and their flipped version along the horizontal axis ($\overline{X}$, $\overline{Y}$). $\overline{X}$ and $\overline{Y}$ represent a flip of the solution either on the x or on the y axis. This is achieved by using AND Decomposition.

$$\bigwedge_{i=0}^{W*H} (\bigwedge_{j=1}^{i} (X_{i-j} \leftrightarrow \overline{X}_{i-j})) \rightarrow (X_i \rightarrow \overline{X}_i)))$$

$$\bigwedge_{i=0}^{W*H} (\bigwedge_{j=1}^{i} (Y_{i-j} \leftrightarrow \overline{Y}_{i-j})) \rightarrow (Y_i \rightarrow \overline{Y}_i)))$$

3.4 Rotation

To allow the circuits to rotate, a boolean array r_k is added to the model, stating whether the circuit k is rotated or not.

In order to keep the formalization consistent, it is necessary to update the **Circuits placement** constraint such that for a certain circuit k, a section can be valid only if the piece isn't rotated, resulting in r_k being false, otherwise, another section $\overline{S}_{x,y,k}$

representing the rotated circuit should be considered. In this formalization, the set $R_k = \{ \bigcup_{x=0,y=0}^{W,H} (S_{x,y,k} \wedge \neg r_k) \vee (\overline{S}_{x,y,k} \wedge r_k)\}$ represents all the possible sections including their rotation.

$$\bigwedge_{k \in C} (\bigwedge_{R_1, R_2 \in R_k} \neg (R_1 \wedge R_2) \wedge \bigvee_{R_i \in R_k} R_i)$$

3.5 Results and performances

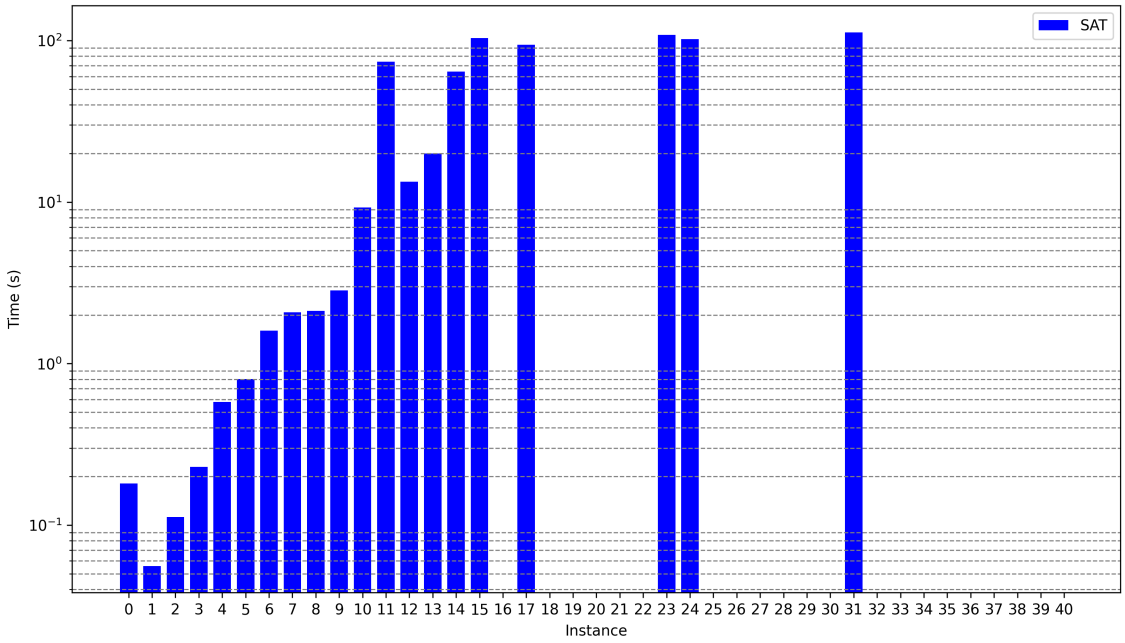


Figure 5: SAT results

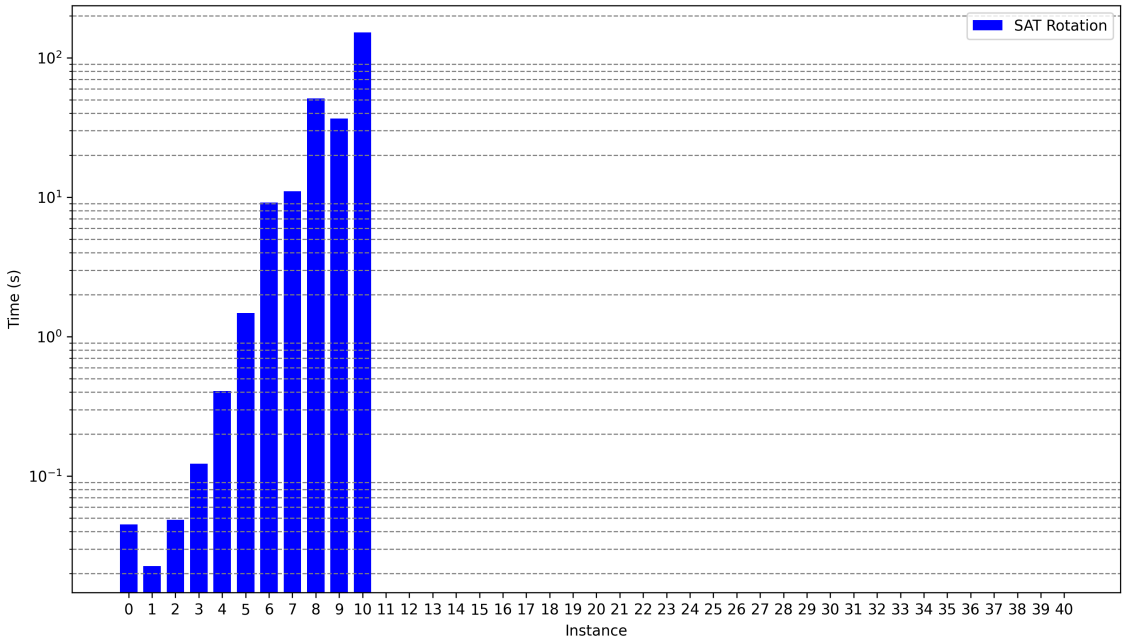


Figure 6: SAT results with rotation

4 SMT

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

SMT solvers allows first order formalization, together with boolean and domain-specific reasoning. This comes with an inevitable but acceptable loss of efficiency but also with a much improved expressivity and scalability.

First order logic allows the use of a formal notation for mathematics, of function, predicates and quantifiers whereas boolean logic only involved propositional symbols and operators.

4.1 Model and variables

We decided to encode the model in a very similar way to what we did in CP. Two `IntVectors` x and y that will contain the x position and y position respectively of the bottom-left corner of each rectangle. We naturally also have two arrays containing width and height of each rectangles together with an `IntValue plate_height` which will be our solution, that is the minimum height which allows for every rectangle to be fit into the board.

The objective function is given by a combination of the class `Optimize` and `sol.minimize()` that Z3py provides; the latter with `plate_height` as parameter.

4.2 Reducing variables domains

To actively speed up the search we managed to reduce the domains of different variables:

- The domain of the xs was reduced in a way that each variable would lay in the range between 0 and a a number given by the expression: `plate_width-min(w)`. This is because no x coordinate can assume a value beyond the right most position where the bottom left corner of the slimmest rectangle can appear without exceeding the board perimeter

$$\bigwedge_{i \in C} x_i \geq 0 \wedge x_i \leq W - min(w)$$

- The domain of the ys was reduced in a way that each variable would lay in the range between 0 and a number resulting from the expression: `Sum(h)-min(h)`. The sum and min operation are here exploited for the same reasons as above.

$$\bigwedge_{i \in C} y_i \geq 0 \wedge y_i \leq \sum_{k \in C} h_k - min(h)$$

- The domain of `plate_height` was reduced in a way that would span the interval between to defined numbers :
  - Lower bound : $(\sum_{i \in C} h_i * w_i)/W$
    As the height of the board cannot assume a value for which the corresponding area fails to be able to contain all the rectangles to be packed inside it
  - Upper bound : $\sum_{i \in C} h_i$
    Since the trivial solution would be to stack all the rectangles one on top of each other

4.3 General constraints

We deployed two sets of constraints:

- Constraints limiting the x and y coordinates of each rectangle in a way that to its position cannot exceed the board boundaries both vertically and horizontally

$$\bigwedge_{i \in C} x_i \geq 0 \wedge x_i + w_i \leq W$$

$$\bigwedge_{i \in C} y_i \geq 0 \wedge y_i + h_i \leq H$$

- Constraints that ensure that no pair of rectangles overlap

$$\bigwedge_{i \in C, j \in C, i \neq j} x_i + w_i \leq x_j \vee x_j + w_j \leq x_i \vee y_i + h_i \leq y_j \vee y_j + h_j \leq y_i$$

4.4 Symmetry Breaking Constraints

For solving this rectangle packing problem efficiently we took advantage of some custom constraints whose aim was that of reducing the search space of specific involved variables:

- Reducing the possibilities for placing large rectangles:
  - If the sum of the widths of two rectangles exceeds the width of the board, the two rectangles cannot be packed side by side along the x axis. Hence we can certainly say that they won't overlap on the y axis but instead either one of the two will be completely above the other.

$$\bigwedge_{i \in C, j \in C, i \neq j} (w_i + w_j > W) \rightarrow (y_i + h_i \leq y_j \vee y_j + h_j \leq y_i)$$

  - if the sum of the heights of two rectangles exceeds the height under current consideration we can state that they cannot be packed side by side along the y axis and therefore they won't overlap on the x axis. All the above considerations holds here in a similar way.

$$\bigwedge_{i \in C, j \in C, i \neq j} (h_i + h_j > H) \rightarrow (x_i + w_i \leq x_j \vee x_j + w_j \leq x_i)$$

- Breaking symmetries for same-sized rectangles : for each pair of rectangles where $(w_i, h_i) = (w_j, h_j)$, meaning that they have both the same height and width, we can fix the positional relation of there rectangles. In order to do this we imposed some sort of lexicographical order between the two pair of (x, y) coordinates of the two rectangles

$$\bigwedge_{i \in C, j \in C, i \neq j} (w_i = w_j \wedge h_i = h_j) \rightarrow (x_j > x_i \vee (x_j = x_i \wedge y_j \geq y_i))$$

- Breaking symmetries for the largest pair [5] of rectangles: for one pair of rectangles, in particular the two largest ones according to their area, we can restrict their relative position in order to remove solutions that would be symmetrical. Again this is done by imposing a lexicographical order between the two.

$$x_{max_2} > x_{max_1} \vee (x_{max_2} == x_{max_1} \wedge y_{max_2} \geq y_{max_1})$$

4.5 Rotation

In order to handle properly the possibility for each of the rectangle to be rotated, we slightly changed the model. We decided to introduce two new `IntVectors` of variables, namely w and h, which will contain the actual widths and heights considered. That means that either w and h values stay the same as per the data input file, or hold the two values swapped.

$$\bigwedge_{i \in C, j \in C, i \neq j} (w_i = widths_i \wedge h_i = heights_i) \vee (w_i = heights_i \wedge h_i = widths_i)$$

Allowing the rectangles to rotate also meant that we could make use of another symmetry breaking constraint. In fact for each rectangle whose h and w are the same (it is a square), the solution in which it is considered rotated is identical to the solution in which it is not. For this reason we decided to coerce its w and h values to the original ones, in this way considering it not rotated and forbidding a possible rotation.

$$\bigwedge_{i \in C} w_i = h_i \rightarrow w_i = widths_i \wedge h_i = heights_i$$

4.6 Results and performances

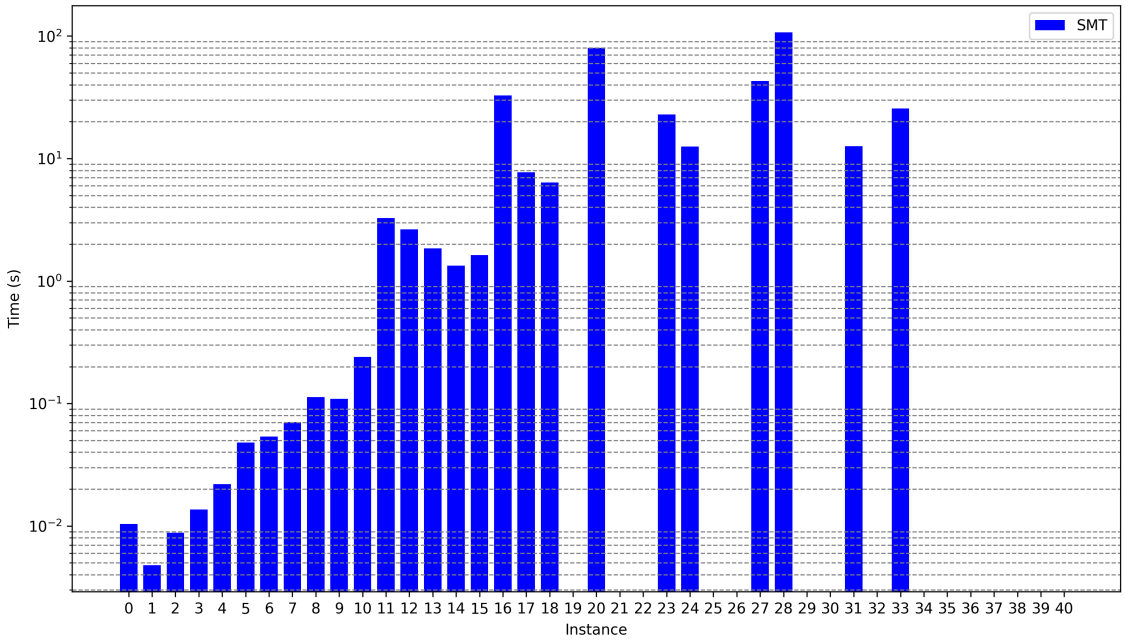


Figure 7: SMT results

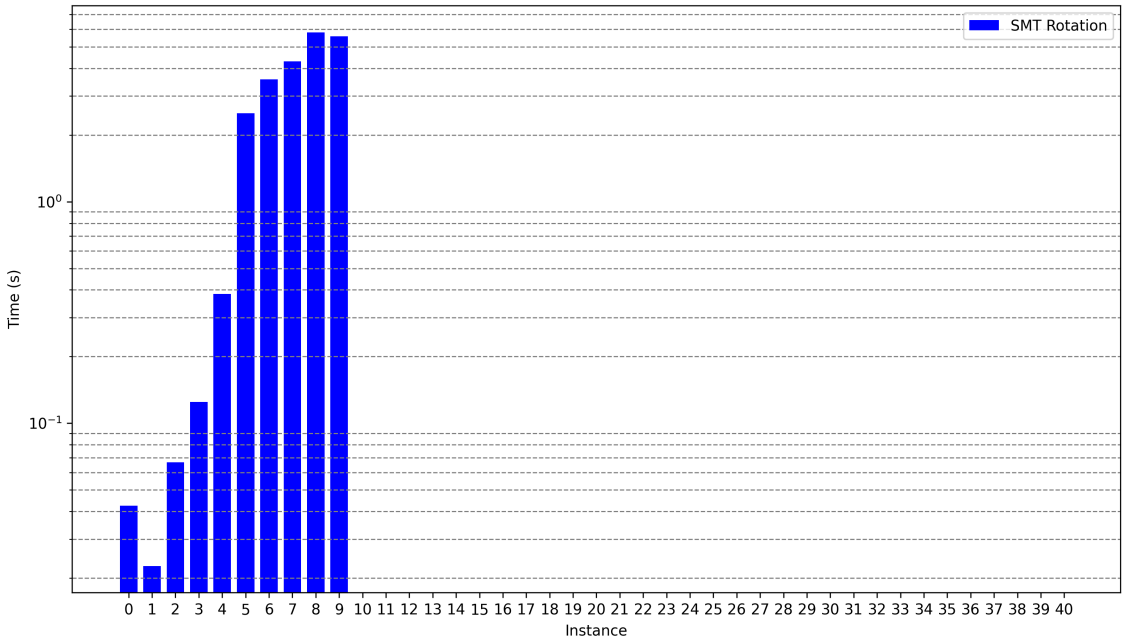


Figure 8: SMT results with rotation

5 Conclusion

With CP and SMT we managed to solve many instances, despite that fact that both solvers failed to give a solution of some others. On the other hand, SAT solver didn't perform as well as the other two, is is probably due to the low expressivity of it's language, resulting in an inefficient model.

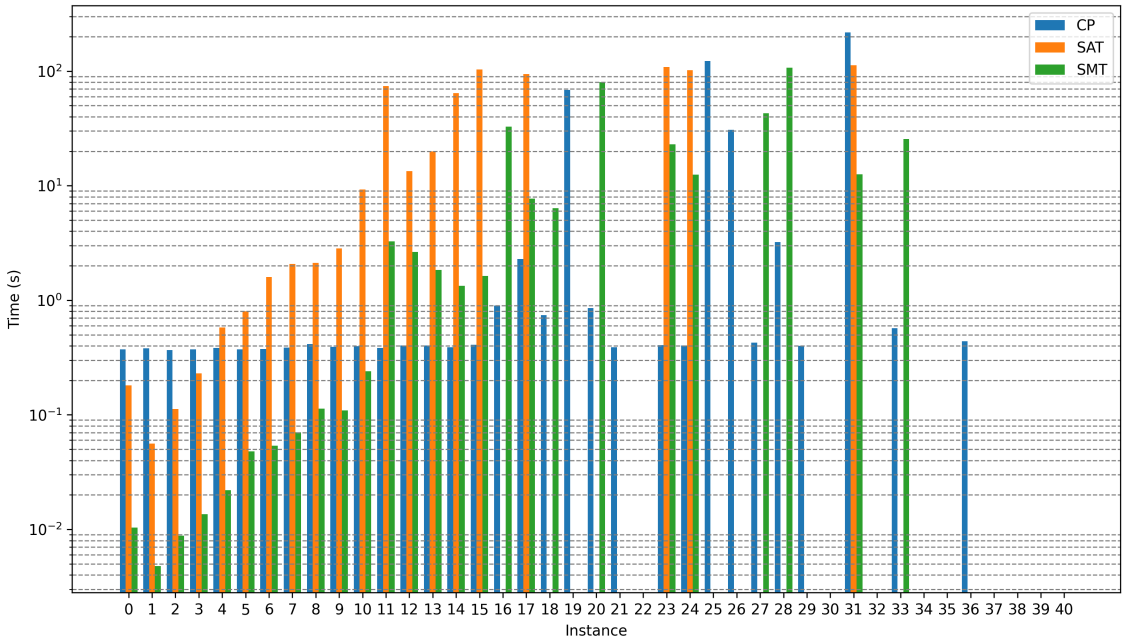


Figure 9: Mixed results

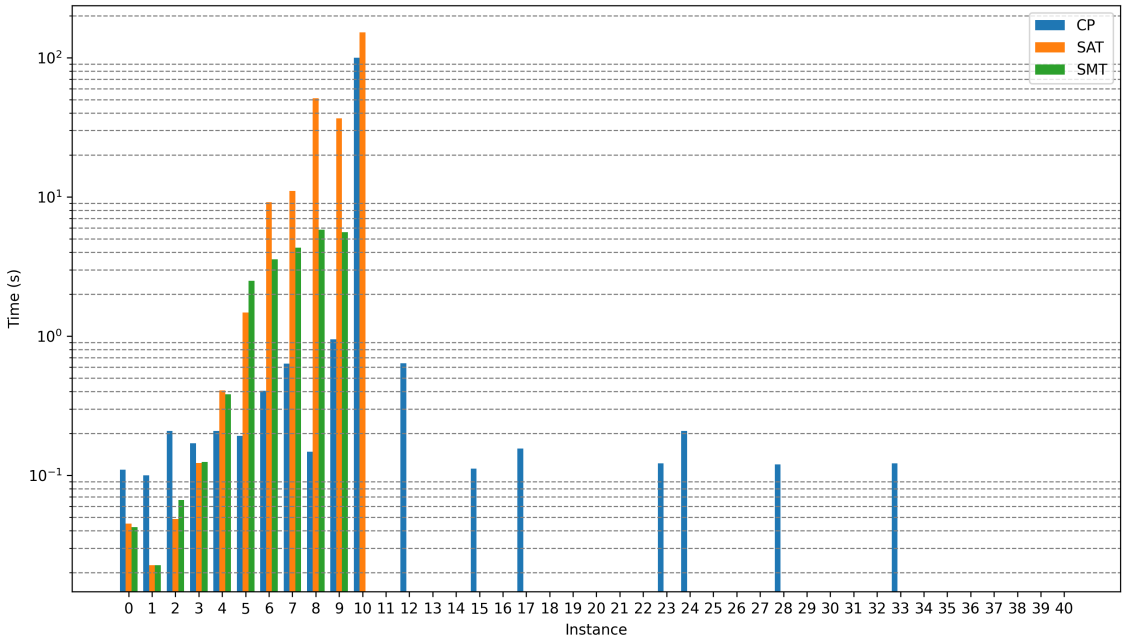


Figure 10: Mixed results with rotation

References

[1] *Cumulative scheduling constraint*. URL: https://sofdem.github.io/gccat/gccat/Ccumulative.html.

[2] *Generalised multi-dimensional non-overlapping constraint*. URL: https://sofdem.github.io/gccat/gccat/Cdiffn.html.

[3] Guntram Scheithauer. *Equivalence and Dominance for Problems of Optimal Packing of Rectangles*. 1995.

[4] Takehide Soh et al. *A SAT-based Method for Solving theTwo-dimensional Strip Packing Problem*. 2008.

[5] Richard E. Korf, Michael D. Moffitt, and Martha E. Pollack. *Optimal rectangle packing*. 2008.