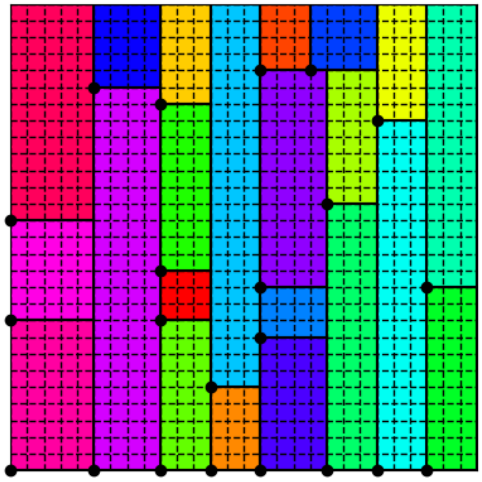
Alma Mater Studiorum - Università di Bologna

Constraint Programming approach to VLSI



Project for the course Combinatorial Decision Making and Optimization (module 1)

Martina Rossini
martina.rossini3@studio.unibo.it

July 2021

Contents

1 Introduction **3**

2 Simple initial model **3**

3 Refining the model **5**
3.1 Implied constraints 5
3.2 first_row constraint 6
3.3 Changes in the objective function 7
3.4 Cumulative constraints 7
3.5 Adding rectangle rotation 8

4 Symmetry Breaking **8**
4.1 Breaking symmetries in rotation 11

5 Search Strategies **11**

6 Experimental results **12**
6.1 Allowing rotation 13

7 Final models and results **18**

References **25**

1 Introduction

VLSI stands for Very Large Scale Integration and refers to the problem of placing a set of rectangular circuits of fixed height and width onto a silicon plate of width W in such a way that the plate's final height H is minimized. This problem has become central in latter years, allowing modern smartphones to be small enough to carry comfortably around but to still contain a touchscreen, more than one camera and many other useful sensors (accelerometer, gyroscope, pedometer, temperature and lighting sensors, etc..). Note that in the precise application we are considering, rotation is not allowed. This is due to the fact that - in order for a device to work properly - each circuit should be placed with a fixed orientation with respect to the others. However, we will still briefly explore how to modify our model in order to support rectangle rotation, abstracting away the details of the practical application we started from.

2 Simple initial model

To build an initial simple model, we first had to decide how to encode the input: let `num_circuits` be the number of rectangles we need to pack and `width` be the width of the plate, while `circuits_w[i]` and `circuits_h[i]` are respectively the width and the height of circuit `i`, with `i = 1` $\cdots$ `num_circuits`. Then we defined the decision variables `circuits_x[i]` and `circuits_y[i]` for every `i = 1` $\cdots$ `num_circuits`, where the x and y coordinates of the bottom-left corner of each rectangle `i` will be stored at the end of the computation. Note that `circuits_x[i]` and `circuits_y[i]`, in conjunction with `circuits_w[i]` and `circuits_h[i]`, are enough to fully qualify the position of circuit `i` inside a 2D plane. The only other decision variables we introduced was `height`, which encodes the height of the silicon plate and is also the objective function we want to minimize.

Seeing as MiniZinc mainly targets finite domain propagation engines, which work better when the variables involved have tight bounds, we focused on trying to determine effective lower and upper bounds for our decision variables [1]. First of all we determined a lower bound for `height` as:

$$\texttt{min\_height} = \frac{\sum_{\texttt{i=1}}^{\texttt{num\_circuits}} \texttt{circuits\_h[i]*circuits\_w[i]}}{\texttt{width}} \tag{1}$$

Then we determined the upper bound for `height` as:

$$\texttt{max\_height} = \sum_{\texttt{i=1}}^{\texttt{num\_circuits}} \texttt{circuits\_h[i]} \tag{2}$$

Finally, regarding the x and y coordinates of each circuit `i` we set these bounds:

$$0 \leq \texttt{circuits\_x[i]} \leq \texttt{width} \quad \text{for} \quad \texttt{i} \quad \text{in} \quad 1 \cdots \texttt{num\_circuits} \tag{3}$$

$$0 \leq \texttt{circuits\_y[i]} \leq \texttt{max\_height} \quad \text{for} \quad \texttt{i} \quad \text{in} \quad 1 \cdots \texttt{num\_circuits} \tag{4}$$

that were further refined by adding the constraint:

$$\begin{aligned} &\texttt{circuits\_x[i]} + \texttt{circuits\_w[i]} \leq \texttt{width} \quad \wedge \\ &\texttt{circuits\_y[i]} + \texttt{circuits\_h[i]} \leq \texttt{height} \\ &\quad \text{for} \quad \texttt{i} \quad \text{in} \quad 1 \cdots \texttt{num\_circuits} \end{aligned} \tag{5}$$

The only other constraint we introduced in this initial phase aimed at making sure that our rectangles would not overlap. Indeed, we did not have to write the constraint by hand as MiniZinc offers a global 2D non-overlapping constraint called `diffn`, which perfectly fit our needs. The final CP model produced in this phase can be found in Listing 1.

```
include "globals.mzn";

int: num_circuits;
int: width;

array[1..num_circuits] of int: circuits_w;
array[1..num_circuits] of int: circuits_h;

int: area_min = sum([circuits_h[i] * circuits_w[i] | i in
   1..num_circuits]);
int: min_height = max(max(circuits_h),
   floor(area_min/width));

var min_height..sum(circuits_h): height;
array[1..num_circuits] of var 0..width: circuits_x;
array[1..num_circuits] of var 0..sum(circuits_h): circuits_y;
```

```
constraint diffn(circuits_x, circuits_y, circuits_w,
    circuits_h);
constraint forall(i in 1..num_circuits) (circuits_x[i] +
    circuits_w[i] <= width /\ circuits_y[i] + circuits_h[i]
    <= height);

solve minimize height;

output ["\(width) \(height)\n" ++ "\(num_circuits)\n" ++
    concat([
     "\(circuits_w[j]) \(circuits_h[j]) \(circuits_x[j])
    \(circuits_y[j])\n" | j in 1..num_circuits])];
```

Listing 1: First simple model

3 Refining the model

Once we had an initial rough model, we started refining it, aiming to obtain a more efficient solution. In particular, we added the implied constraint suggested by the second point in the problem specification, a constraint to make it so that there are no empty spaces in the first row of the silicon board and two redundant cumulative constraints that were somewhat borrowed from scheduling problems. Finally, as already anticipated, we briefly explored how to modify the obtained model in order to allow rectangle rotation by 90 degrees. The following sections will go into deeper detail regarding each of these changes.

3.1 Implied constraints

As suggested in the project specification, for every possible width `x_thr` if we draw a vertical line starting from it and consider all the traversed circuits, the sum of their heights `circuits_h[i]` must not be greater than the height of the silicon board. Considering that a vertical line in `x_thr` traverses rectangle `i` if and only if `circuits_x[i]` $\leq$ `x_thr` $\wedge$ (`circuits_x[i]` + `circuits_w[i]`) $>$ `x_thr`, the corresponding implied constraint can be written like this:

```
constraint implied_constraint(forall(i in
    0..width)(sum([circuits_h[j] | j in 1..num_circuits where
```

```
    circuits_x[j] <= i /\ (circuits_x[j] + circuits_w[j]) >
    i]) <= height));
```

Since similar considerations can be done for an horizontal line cut at every possible height `y_thr`, we can write the corresponding constraint as:

```
constraint implied_constraint(forall(i in
    0..height)(sum([circuits_w[j] | j in 1..num_circuits
    where circuits_y[j] <= i /\ (circuits_y[j] +
    circuits_h[j]) > i]) <= width));
```

3.2 first_row constraint

While testing the model we built so far on the provided instances and printing the intermediate solutions, we noticed that – particularly the circuit placements proposed early on – tended to stack many rectangles upon each other, thus not occupying in an optimal way the available horizontal space. We wanted to try limiting this phenomenon in order to get to an optimal solution faster, thus we introduced an array of decision variables called `first_row`. The domain of these new variables was restricted to $\{0, 1\}$ and a constraint was added to make sure that, for every `i=0` $\cdots$ `width-1`, `first_row[i]` $= 1$ if and only if a circuit occupies the `i`-th cell of the first row of the silicon plate.

Since a circuit `j` occupies cell `i` of the board if `circuits_x[j]` $\leq$ `i` $\wedge$ `circuits_x[j]` $+$ `circuits_w[j]` $>$ `i` $\wedge$ `circuits_y[j]` $==$ 0, we wrote the above constraints as:

```
array[1..width] of var 0..1: first_row;

constraint forall(i in 0..width-1)(
  first_row[i+1] = 1 <-> card({j | j in 1..num_circuits
    where circuits_x[j] <= i /\ circuits_x[j] + circuits_w[j]
    > i /\ circuits_y[j] == 0}) >= 1);
constraint all_equal(first_row) /\ first_row[1] == 1;
```

Also note that, in case the given circuits can not perfectly fit within a board, thus leaving some free space, this constraint can also break some symmetries regarding the position of the white cells – which can never be located in the bottom row of the plate.

3.3 Changes in the objective function

Thinking about the possibility that an optimal solution has some wasted white space, we also decided that it could be beneficial to minimize at the same time both the board height and the amount of free cells left in it. Our idea was to optimize for two different objectives, with a clear ordering between them: the most important one is the height, thus if we can minimize it we could potentially also allow the white cells number to grow. This technique is called lexicographic optimization and it is not currently natively supported in MiniZinc. However, a simple workaround scales the first objective by a value that is at least as much as the maximum value of the second objective and then sum the two values together. The amount of white cells is computed as `width*height − area_min`, where `area_min` is computed as shown in Listing 1, thus – for any reasonable solution – a simple rough approximation of the maximum value `white_space` can have is given by `area_min`. Given the above considerations, the modified objective function for our model is:

```
int: area_min = sum([circuits_h[i] * circuits_w[i] | i in
   1..num_circuits]);
var 0..area_min: white_space = width*height - area_min;

...

solve minimize area_min*height+white_space;
```

3.4 Cumulative constraints

The last improvement we made to our model was to add the following constraints:

```
constraint cumulative(circuits_x, circuits_w, circuits_h,
   height);
constraint cumulative(circuits_y, circuits_h, circuits_w,
   width);
```

The global `cumulative` constraint is provided by MiniZinc and it is typically used for scheduling and resource allocation problems. Indeed, usually, the first parameter passed to `cumulative` is an array containing each task's start time, the second one is an array of tasks' durations, the third one is an array containing the resource requirements for each task and the last parameter is an upper bound for the resources available.

We were able to use this constraint in our model because we can think of each rectangle `i` as a task with start time `circuits_x[i]`, duration `circuits_w[i]` and resource requirement `circuits_h[i]` and, at the same time, also as a task with start time `circuits_y[i]`, duration `circuits_h[i]` and resource requirement `circuits_w[i]`. Note that these two constraints are absolutely redundant but still improve the propagation of our model and, thus, its performance.

3.5 Adding rectangle rotation

Finally, to add support for rectangle rotation we introduced an array `rot` of decision variables with domain $\{0, 1\}$ such that if rectangle `i` is rotated, `rot[i]` $= 1$, otherwise `rot[i]` $= 0$. Thanks to these new variables we were able to define the arrays `actual_w` and `actual_h`, which represent the actual height and width of every rectangle to use during the computation:

```
array[1..num_circuits] of var 0..1: rot;

...

array[1..num_circuits] of var 0..max(width,
    sum(circuits_h)): actual_w =
    [circuits_w[i]*(1-rot[i])+circuits_h[i]*rot[i] | i in
    1..num_circuits];
array[1..num_circuits] of var 0..max(width,
    sum(circuits_h)): actual_h =
    [circuits_h[i]*(1-rot[i])+circuits_w[i]*rot[i] | i in
    1..num_circuits];
```

Note that `rot` helped us to avoid using disjunctions – which sometimes do not propagate well in CP – while building our model.

Once we had `actual_h` and `actual_w`, we simply substituted them respectively to `circuits_h` and `circuits_w` everywhere in the model.

4 Symmetry Breaking

This problem has many intrinsic symmetries that we tried to break. The first constraint we imposed to break some symmetries was to select the bigger rectangle (according to a condition that will be better explained in Section 5) and place it in the bottom-left corner of the silicon plate. This ensured

that the bigger object would be placed first and also that it would not be swapped with multiple small rectangles that – stacked together – reach the same size. The constraint we just described is shown in Listing 2.

```
array[1..num_circuits] of int: order =
   reverse(arg_sort([max(circuits_h[i], circuits_w[i]) | i
   in 1..num_circuits]));

...

constraint symmetry_breaking_constraint(circuits_x[order[1]]
   = 0 /\ circuits_y[order[1]] = 0);
```

Listing 2: Symmetry breaking by placing first the bigger rectangle

Other obvious symmetries that are present in this problem have to do with horizontal and vertical flipping of the axis: they can easily be broken by imposing a lexicographic order between the actual rectangles (x, y) positions and their flipped (x, y) positions – which, for a given rectangle `i` can be computed as `width` − `circuits_x[i]` − `circuits_w[i]` and `height` − `circuits_y[i]` − `circuits_h[i]`. Thus these constraints can be written as

```
constraint symmetry_breaking_constraint(
  lex_lesseq(circuits_x, [width - circuits_w[i] -
    circuits_x[i] | i in 1..num_circuits]) /\
  lex_lesseq(circuits_y, [height - circuits_h[i] -
    circuits_y[i] | i in 1..num_circuits])
);
```

Listing 3: lex_lesseq symmetry breaking constraints

Unfortunately, this constraint conflicts with the one that fixes the position of the bigger circuit. Since restricting the constraint in Listing 3 to only the portion of the plane not occupied by the fixed rectangle proved to be too computationally expensive, we had to choose a single alternative between the two symmetry breaking constraints introduced so far. Comparing them on some instances we found that the constraint placing the bigger rectangle in position $(0, 0)$ appeared to eliminate a much bigger number of symmetric solutions, thus proving more effective. This is why we decided to discard these lexicographic constraints.

Finally, we also noticed other symmetries we tried to break: first of all, two circuits with same w (respectively, same h) and placed in the same x coordinate (respectively, y coordinate) could easily be swapped, so we introduced an ordering between them. Similarly, two blocks with same width and

same y coordinate could also be swapped. These symmetries were initially broken using the following constraints:

```
constraint symmetry_breaking_constraint(
% breaks swapping of same-width and same-x blocks
forall(i, j in order) (if i>j /\ circuits_w[i] ==
    circuits_w[j] /\ circuits_x[i] == circuits_x[j] then
    circuits_y[i] < circuits_y[j] endif) /\
% breaks swapping of same-height and same-y blocks
forall(i,j in order) (if i>j /\ circuits_h[i] ==
    circuits_h[j] /\ circuits_y[i] == circuits_y[j] then
    circuits_x[i] < circuits_x[j] endif) /\
% makes it so a block placed on x=3,y=0 and large 4 and
    another block placed on, say, x=8,y=0 and large 4 cannot
    be swapped
forall(i,j in order) (if i>j /\ circuits_w[i] ==
    circuits_w[j] /\ circuits_y[i] == circuits_y[j] then
    circuits_x[i] < circuits_x[j] endif)
);
```

However, this required to iterate four different times over the circuits, which could prove to be computationally expensive as the number of circuits increased. Thus, we re-formulated them like this:

```
constraint symmetry_breaking_constraint(
forall(i, j in order) (
  if i>j then
    (if circuits_x[i] == circuits_x[j] then
        if circuits_w[i] == circuits_w[j] then
       % breaks swapping of same-width and same-x blocks
           circuits_y[i] < circuits_y[j]
         elseif circuits_h[i] == circuits_h[j] then
         % makes it so a block placed on x=5,y=0 and high 4
   and another block placed on, say, x=5,y=9 and high 4
         % cannot be swapped
           circuits_y[i] < circuits_y[j]
         else true endif
     elseif circuits_y[i] == circuits_y[j] /\ circuits_h[i]
   == circuits_h[j] then
          % breaks swapping of same-height and same-y blocks
           circuits_x[i] < circuits_x[j]
     else true endif)
  endif)
);
```

Listing 4: Improved symmetry breaking for swapping of same-size objects

4.1 Breaking symmetries in rotation

When the rectangles are free to rotate, a huge amount of other symmetries appear. Thus, we tried to break them by making use of the `rot` array like this:

```
constraint symmetry_breaking_constraint(
lex_lesseq(rot, [(1-rot[i]) | i in 1..num_circuits]) /\
lex_greatereq(rot, [rot[num_circuits+1-i] | i in
   1..num_circuits])
);
```

Listing 5: Symmetry breaking in rotation

Note that all the other symmetries we found in the model without rotation are still valid for this variation.

5 Search Strategies

Sometimes, we may want to specify how the search should be undertaken, instead of leaving it completely up to the underlying solver. This requires us to communicate with said solver using an apposite MiniZinc construct called "search annotations".

Aiming to solve as many instances of this problem as possible, we tried to specify different search strategies, all based on the concept of assigning values to the variables `circutis_x[i]` and `circuits_y[i]` for every `i=1` $\cdots$ `num_circuits` in a precise order. We tried different orderings: assign first the position to the rectangle – among the remaining ones – with bigger height, with bigger width, with bigger area or with bigger maximum value between height and width. The reasoning for this was that by placing the harder rectangles first, we should be able to fail sooner in the search and thus get to the optimal solution with less backtracking.

Finally, we first tried to assign the x and y coordinates in no precise order, but soon realized that assigning first all the xs and only then placing the ys let to a greatly improved performance, as the number of values from which we can choose when placing a y coordinate, with all the x coordinates already fixed, is much smaller than before (i.e.: we greatly reduce the search space). Note, however, that this comes with the cost of a potentially huge amount of backtracking if we run in a failure when placing the y coordinates, as we may need to also modify all the assigned x coordinates. Nevertheless,

this approach worked really well in practice, as we will show in Section 6. Overall, the search strategy we used looks like this:

```
solve :: seq_search([int_search([circuits_x[i] | i in
    order], input_order, indomain_min, complete),
    int_search([circuits_y[i] | i in order], input_order,
    indomain_min, complete)]) minimize area_min*height +
    white_space;
```

Finally, note that we did not add any restart annotation as our search strategy is, fundamentally, deterministic and restarting search does not make much sense if the underlying search strategy does not do something different the next time search starts [1].

6 Experimental results

We will now show the results we obtained with different search strategies, different rectangle orderings and different symmetry breaking constraints on a subset of the provided instances (in particular ins-5, ins-11, ins-14, ins-18, ins-22, ins-25, ins-30, ins-34, ins-38 and ins-40). We took care to select instances at varying levels of complexity to get a complete idea of the model's performance.

Note that all our experiments were conducted using the Chuffed MiniZinc solver with the option "free search", which allows it to switch between the search strategy we defined and its activity-based search; according to the MiniZinc Docs [1] this is the best way to take complete advantage of Chuffed's performance.

We first tried to solve our selected instances using the two symmetry breaking constraints of Listings 2 and 4 and searching the variables in descending order of the maximum between rectangles' height and width. Then, using the same variable ordering, we slightly modified the constraint in Listing 4 so as not to break the symmetry related to the swapping of same-y and same-h objects. Indeed, this constraint imposes the x coordinate of circuit `i` to be less than the x coordinate of the circuits `j` if `i` and `j` have same y coordinate and also same height, but, since we assign all the x values before the y ones, we were concerned with the possibility that imposing such a constraint may lead to useless backtracking. The results we obtained for both these experiments are shown in Tables 1 and 2, respectively. When analyzing the tables note that a cell containing "—" means that, for that specific instance,

Table 1: Results for model without rotation; order = arg_max(max(w,h)) and strict swapping constraint used

Instance name	**First solution**	**Optimal solution**
out-5.txt	0.95	0.95
out-11.txt	2.28	2.42
out-14.txt	2.54	2.73
out-18.txt	4.69	5.39
out-22.txt	6.19	176.96
out-25.txt	8.34	124.13
out-30.txt	8.57	111.13
out-34.txt	4.58	8.40
out-38.txt	8.79	—
out-40.txt	62.66	—

our model could not find the optima solution in the provided time limit of 5 minutes. As we can clearly see, the modified version of the symmetry breaking constraint for same-size rectangle swapping gets better results in some of the harder instances of our sample. Thus, this is the symmetry breaking constraint we are going to adopt for all the following experiments.

All other experiments we performed were related to the variable ordering. Tables 3, 4 and 5 respectively show our results when ordering the variables based on descending rectangle height, descending rectangle width and descending rectangle area. We can clearly see that, with all these three variable orderings we are able to also solve instance 38; however, among them, we preferred the one based on rectangle width as it proved to be much faster for some intermediate instances (ins-22, ins-25, ins-30) with respect to the others.

6.1 Allowing rotation

For this model we investigated the performance with and without the specific symmetry breaking constraints introduced for rotation in Section 4.1. We also investigated which version of the constraint in Listing 4 works best, the one that also breaks swapping of same-y and same-height objects by imposing an ordering on their x coordinates or the one that does not. Finally, regarding variable ordering, we choose to only investigate what happens in two cases: when we follow a descending order based on the maximum value between

Table 2: Results for model without rotation; order = arg_max(max(w,h)) and less-strict swapping constraint used

Instance name	**First solution**	**Optimal solution**
out-5.txt	0.90	0.90
out-11.txt	2.48	2.66
out-14.txt	2.83	2.99
out-18.txt	4.22	4.95
out-22.txt	6.31	111.27
out-25.txt	8.72	15.56
out-30.txt	8.91	67.61
out-34.txt	4.66	6.88
out-38.txt	8.58	—
out-40.txt	61.21	—

Table 3: Results for model without rotation; order = arg_max(h) and less-strict swapping constraint used

Instance name	**First solution**	**Optimal solution**
out-5.txt	0.97	0.97
out-11.txt	2.45	2.60
out-14.txt	2.69	2.87
out-18.txt	4.19	4.48
out-22.txt	6.14	102.80
out-25.txt	9.09	92.58
out-30.txt	8.57	73.22
out-34.txt	4.58	13.01
out-38.txt	8.69	172.59
out-40.txt	88.72	—

Table 4: Results for model without rotation; order = arg_max(w) and less-strict swapping constraint used

Instance name	**First solution**	**Optimal solution**
out-5.txt	0.86	0.87
out-11.txt	2.35	2.47
out-14.txt	2.60	2.92
out-18.txt	4.24	5.88
out-22.txt	6.32	7.98
out-25.txt	8.83	11.19
out-30.txt	8.62	10.74
out-34.txt	4.65	17.24
out-38.txt	8.83	186.51
out-40.txt	61.52	—

Table 5: Results for model without rotation; order = arg_max(w*h) and less-strict swapping constraint used

Instance name	**First solution**	**Optimal solution**
out-5.txt	0.91	0.92
out-11.txt	2.34	2.63
out-14.txt	2.50	2.57
out-18.txt	4.16	4.65
out-22.txt	6.15	42.54
out-25.txt	8.27	14.36
out-30.txt	8.58	152.91
out-34.txt	4.40	8.76
out-38.txt	8.43	259.82
out-40.txt	61.69	—

Table 6: Results for model with rotation; order = arg_max(max(h,w)) and strict swapping constraint used

Instance name	**First solution**	**Optimal solution**
out-5.txt	0.93	0.94
out-11.txt	2.33	3.96
out-14.txt	2.70	70.89
out-18.txt	4.00	5.12
out-22.txt	6.30	—
out-25.txt	9.50	—
out-30.txt	9.19	—
out-34.txt	4.71	56.41
out-38.txt	9.43	—
out-40.txt	78.80	—

`circuits_w[i]` and `circuits_h[i]` and when we follow a descending order based on the rectangles' area. The main reason why we choose not to try also orderings based on just rectangles' height and width is that those two quantities are not fixed in this new model; indeed, they depend on the values of the `rot` array. This means that the variable order would not be anymore fixed at the start of the computation but would become itself a decision variable, further complicating the model. In Tables 6, 7 and 8 we can find the results obtained with variable ordered descending by the maximum value between a rectangle width and its height for the three different combination of symmetry breaking constraints described above. Note that, in this case, not breaking rotation symmetries actually produces better solutions – indeed, in Table 8, we have an optimal rectangle placement also for instance 22). Thus, we tried only this third combination of constraints for the new variable ordering based on rectangles' area. The results of this trial are shown in Table 9. As we can clearly see, we obtain better results by not breaking rotation symmetries, using the less-strict version of our constraints to break swappings between same sizes objects and by using the maximum value between a rectangle's height and width to order the variables.

Table 7: Results for model with rotation; order = arg_max(max(h,w)) and less-strict swapping constraint used

Instance name	**First solution**	**Optimal solution**
out-5.txt	0.94	0.95
out-11.txt	2.19	9.10
out-14.txt	2.39	27.18
out-18.txt	4.09	45.84
out-22.txt	6.79	—
out-25.txt	9.33	—
out-30.txt	8.90	—
out-34.txt	4.23	15.62
out-38.txt	8.98	—
out-40.txt	83.55	—

Table 8: Results for model with rotation; order = arg_max(max(h,w)), no rotation sym breaking and less-strict swapping constraint used

Instance name	**First solution**	**Optimal solution**
out-5.txt	0.84	0.84
out-11.txt	1.93	5.75
out-14.txt	2.24	2.66
out-18.txt	3.87	159.14
out-22.txt	5.67	27.33
out-25.txt	8.01	—
out-30.txt	8.39	—
out-34.txt	4.36	115.74
out-38.txt	8.32	—
out-40.txt	98.37	—

Table 9: Results for model with rotation; order = arg_max(h*w), no rotation sym breaking and less-strict swapping constraint used

Instance name	First solution	Optimal solution
.out-5.txt	0.83	0.83
out-11.txt	1.93	6.22
out-14.txt	2.17	2.33
out-18.txt	3.65	32.06
out-22.txt	5.76	—
out-25.txt	8.10	—
out-30.txt	8.54	—
out-34.txt	4.27	86.31
out-38.txt	8.78	—
out-40.txt	86.47	—

7 Final models and results

As it is easy to see from the results in Section 6, the best combination of symmetry breaking constraints and variable ordering for the model without rotation is obtained ordering variables according to rectangle's width and by using the less-strict version of the swapping symmetry-breaking constraint. For completeness sake, such a model is shown in Listing 6. The results obtained with this CP model for all the proposed instances are shown in Table 10; we are able to solve – in the given time limit – 36 out of the 40 instances proposed. Note that actually the number could be higher as - for example - in the preliminary results showed Section 6 we were also able to solve instance 38. Indeed this is due to the fact that there's some inherent randomness in the search done by MiniZinc, which could be controlled by setting a random seed at execution time.

Similarly, the best combination of symmetry breaking constraints and variable ordering for the model with rotation is reached ordering variables according the the maximum between a rectangle's height and its width and using exactly the same symmetry breaking constraints used in the standard model. Again, for completeness sake, the full MiniZinc code used in the experiment is shown in Listing 7 and the obtained results are listed in Table 11. Note that, even if the model with rotation is more complex, we were still able to solve 30 instances out of the 40 provided in the given time limit of 5 minutes.

Table 10: Final results for model without rotation

Instance name	**First solution**	**Optimal solution**
out-1.txt	0.57	0.57
out-10.txt	1.66	1.69
out-11.txt	2.49	2.61
out-12.txt	2.26	2.28
out-13.txt	2.41	2.44
out-14.txt	2.65	2.97
out-15.txt	2.98	3.64
out-16.txt	3.78	4.69
out-17.txt	3.65	4.04
out-18.txt	4.27	5.93
out-19.txt	5.20	9.06
out-2.txt	0.67	0.67
out-20.txt	5.18	6.54
out-21.txt	5.72	7.00
out-22.txt	6.54	8.16
out-23.txt	5.29	7.04
out-24.txt	4.73	5.49
out-25.txt	8.78	12.54
out-26.txt	6.91	12.86
out-27.txt	6.26	7.52
out-28.txt	6.25	7.77
out-29.txt	7.25	8.56
out-3.txt	0.65	0.66
out-30.txt	8.70	13.36
out-31.txt	4.98	5.72
out-32.txt	10.00	14.20
out-33.txt	5.87	6.76
out-34.txt	4.51	10.20
out-35.txt	4.34	7.12
out-36.txt	4.58	6.50
out-37.txt	8.46	—
out-38.txt	8.66	—
out-39.txt	8.29	—
out-4.txt	0.84	0.84
out-40.txt	62.01	—
out-5.txt	0.91	0.92
out-6.txt	0.97	0.97
out-7.txt	1.06	1.06
out-8.txt	1.22	1.23
out-9.txt	1.27	1.27

Table 11: Final results for model with rotation

Instance name	**First solution**	**Optimal solution**
out-1.txt	0.60	0.60
out-10.txt	1.74	1.88
out-11.txt	2.30	6.04
out-12.txt	2.32	4.40
out-13.txt	2.43	3.23
out-14.txt	2.49	2.89
out-15.txt	2.87	3.27
out-16.txt	3.65	26.14
out-17.txt	3.61	23.99
out-18.txt	4.10	176.81
out-19.txt	5.07	—
out-2.txt	0.63	0.63
out-20.txt	5.06	—
out-21.txt	5.73	—
out-22.txt	6.33	29.43
out-23.txt	4.76	12.58
out-24.txt	4.71	9.27
out-25.txt	8.70	—
out-26.txt	6.59	11.28
out-27.txt	5.62	7.89
out-28.txt	5.74	7.14
out-29.txt	6.75	29.42
out-3.txt	0.65	0.65
out-30.txt	8.37	—
out-31.txt	4.51	7.31
out-32.txt	9.06	—
out-33.txt	5.40	8.92
out-34.txt	4.25	114.29
out-35.txt	4.09	28.11
out-36.txt	4.10	12.77
out-37.txt	8.09	—
out-38.txt	8.15	—
out-39.txt	7.84	—
out-4.txt	0.72	0.72
out-40.txt	102.53	—
out-5.txt	0.87	0.87
out-6.txt	0.99	1.00
out-7.txt	1.01	1.02
out-8.txt	1.17	1.23
out-9.txt	1.17	1.19

```
include "globals.mzn";

int: num_circuits;
int: width;

array[1..num_circuits] of int: circuits_w;
array[1..num_circuits] of int: circuits_h;


int: area_min = sum([circuits_h[i] * circuits_w[i] | i in
   1..num_circuits]);
var 0..area_min: white_space = width*height - area_min;
int: min_height = max(max(circuits_h),
   floor(area_min/width));

var min_height..sum(circuits_h): height;
array[1..num_circuits] of var 0..width: circuits_x;
array[1..num_circuits] of var 0..sum(circuits_h): circuits_y;


array[1..num_circuits] of int: order =
   reverse(arg_sort([circuits_w[i] | i in 1..num_circuits]));

array[1..width] of var 0..1: first_row;

% Always set the rectangles so that there are no blank
   spaces in the bottom row of the board; also breaks
   symmetries related to blank spaces
constraint forall(i in 0..width-1)(
  first_row[i+1] = 1 <-> card({j | j in 1..num_circuits
   where circuits_x[j] <= i /\ circuits_x[j] + circuits_w[j]
   > i /\ circuits_y[j] == 0}) >= 1);
constraint all_equal(first_row) /\ first_row[1] == 1;

constraint cumulative(circuits_x, circuits_w, circuits_h,
   height);
constraint cumulative(circuits_y, circuits_h, circuits_w,
   width);

constraint diffn(circuits_x, circuits_y, circuits_w,
   circuits_h);
constraint forall(i in 1..num_circuits) (circuits_x[i] +
   circuits_w[i] <= width /\ circuits_y[i] + circuits_h[i]
   <= height);
```

```
% Implied constraints suggested by prof. (point 2)
constraint implied_constraint(forall(i in
    0..width)(sum([circuits_h[j] | j in 1..num_circuits where
    circuits_x[j] <= i /\ (circuits_x[j] + circuits_w[j]) >
    i]) <= height));
constraint implied_constraint(forall(i in
    0..height)(sum([circuits_w[j] | j in 1..num_circuits
    where circuits_y[j] <= i /\ (circuits_y[j] +
    circuits_h[j]) > i]) <= width));

constraint symmetry_breaking_constraint(circuits_x[order[1]]
    = 0 /\ circuits_y[order[1]] = 0);

constraint symmetry_breaking_constraint(
forall(i, j in order) (
  if i>j then
    (if circuits_x[i] == circuits_x[j] then
        if circuits_w[i] == circuits_w[j] then
       % breaks swapping of same-width and same-x blocks
           circuits_y[i] < circuits_y[j]
         elseif circuits_h[i] == circuits_h[j] then
         % makes it so a block placed on x=5,y=0 and high 4
    and another block placed on, say, x=5,y=9 and high 4
         % cannot be swapped
           circuits_y[i] < circuits_y[j]
         else true endif
     endif)
  endif)
);


solve :: seq_search([int_search([circuits_x[i] | i in
    order], input_order, indomain_min, complete),
    int_search([circuits_y[i] | i in order], input_order,
    indomain_min, complete)]) minimize
    area_min*height+white_space;

output ["\(width) \(height)\n" ++ "\(num_circuits)\n" ++
    concat([
     "\(circuits_w[j]) \(circuits_h[j]) \(circuits_x[j])
    \(circuits_y[j])\n" | j in 1..num_circuits])];
```

Listing 6: Final MiniZinc code for model without rotation.

```
include "globals.mzn";
```

```
int: num_circuits;
int: width;

array[1..num_circuits] of int: circuits_w;
array[1..num_circuits] of int: circuits_h;
array[1..num_circuits] of var 0..1: rot;

int: area_min = sum([circuits_h[i] * circuits_w[i] | i in
   1..num_circuits]);
var 0..area_min: white_space = width*height - area_min;
int: min_height = max(max(circuits_h),
   floor(area_min/width));

var min_height..sum(circuits_h): height;
array[1..num_circuits] of var 0..width: circuits_x;
array[1..num_circuits] of var 0..sum(circuits_h): circuits_y;

array[1..num_circuits] of int: order =
   reverse(arg_sort([max(circuits_h[i], circuits_w[i]) | i
   in 1..num_circuits]));

array[1..width] of var 0..1: first_row;

array[1..num_circuits] of var 0..max(width,
   sum(circuits_h)): actual_w =
   [circuits_w[i]*(1-rot[i])+circuits_h[i]*rot[i] | i in
   1..num_circuits];
array[1..num_circuits] of var 0..max(width,
   sum(circuits_h)): actual_h =
   [circuits_h[i]*(1-rot[i])+circuits_w[i]*rot[i] | i in
   1..num_circuits];

constraint forall(i in 0..width-1)(
  first_row[i+1] = 1 <-> card({j | j in 1..num_circuits
   where circuits_x[j] <= i /\ circuits_x[j] + actual_w[j] >
   i /\ circuits_y[j] == 0}) >= 1);
constraint all_equal(first_row) /\ first_row[1] == 1;

constraint cumulative(circuits_x, actual_w, actual_h,
   height);
constraint cumulative(circuits_y, actual_h, actual_w, width);

constraint diffn(circuits_x, circuits_y, actual_w, actual_h);

constraint forall(i in 1..num_circuits) (circuits_x[i] +
```

```
    actual_w[i] <= width /\ circuits_y[i] + actual_h[i] <=
    height);

% Implied constraints suggested by prof. (point 2)
constraint implied_constraint(forall(i in
    0..width)(sum([actual_h[j] | j in 1..num_circuits where
    circuits_x[j] <= i /\ (circuits_x[j] + actual_w[j]) > i])
    <= height));
constraint implied_constraint(forall(i in
    0..height)(sum([actual_w[j] | j in 1..num_circuits where
    circuits_y[j] <= i /\ (circuits_y[j] + actual_h[j]) > i])
    <= width));

constraint symmetry_breaking_constraint(circuits_x[order[1]]
    = 0 /\ circuits_y[order[1]] = 0);

constraint symmetry_breaking_constraint(
forall(i, j in order) (
  if i>j then
    (if circuits_x[i] == circuits_x[j] then
        if circuits_w[i] == circuits_w[j] then
       % breaks swapping of same-width and same-x blocks
           circuits_y[i] < circuits_y[j]
         elseif circuits_h[i] == circuits_h[j] then
         % makes it so a block placed on x=5,y=0 and high 4
    and another block placed on, say, x=5,y=9 and high 4
         % cannot be swapped
           circuits_y[i] < circuits_y[j]
         else true endif
     endif)
  endif)
);

solve :: seq_search([int_search([circuits_x[i] | i in
    order], input_order, indomain_min, complete),
    int_search([circuits_y[i] | i in order], input_order,
    indomain_min, complete)]) minimize
    area_min*height+white_space;

output ["\(width) \(height)\n" ++ "\(num_circuits)\n"  ++
    concat([
     "\(actual_w[j]) \(actual_h[j]) \(circuits_x[j])
    \(circuits_y[j])\n" | j in 1..num_circuits])];
```

Listing 7: Final MiniZinc code for model with rotation

References

[1] Peter J. Stuckey, Kim Marrioh, and Guido Tack. *The MiniZinc Handbook 2.5.5*. URL: `https://ww.minizinc.org/doc-2.5.5/en/index.html`.