Present Wrapping Problem

Leonardo Calbi (leonardo.calbi@studio.unibo.it)
Alessio Falai (alessio.falai@studio.unibo.it)

September 18, 2020

Contents

Foreword

The problem is presented as: given a wrapping paper roll of a certain dimension and a list of presents, decide how to cut off pieces of paper so that all the presents can be wrapped.

Consider that each present is described by the dimensions of the piece of paper needed to wrap it. Moreover, each necessary piece of paper cannot be rotated when cutting off, to respect the direction of the patterns in the paper.

A more general case also requires the following conditions:

- Rotation of the pieces of paper is allowed
- There can be multiple presents of the same dimensions

1 Introduction

The non-overlapment requirement of *PWP* links it to a specialization of the more general rectangle packing problem, in which we have a set of rectangles (our presents) of given dimensions that have to fit into a pre-determined square (the wrapping paper) of a given size.

Observing the assigned problem instances, we assume that the items will perfectly fit into the given container, without any kind of wasted space. This assumption greatly simplifies the problem, by reducing it from a minimization problem to a satisfiability one.

The following sections describe our implementation of different *PWP* solutions using both Constraint Programming and Satisfiability Modulo Theory approaches.

2 Input

Each instance of the problem is defined by:

- `n` $\longleftarrow$ number of presents to be wrapped
- `w_paper` or `w` $\longleftarrow$ width of the paper roll
- `h_paper` or `h` $\longleftarrow$ height of the paper roll
- `presents` or `p` $\longleftarrow$ list of presents dimensions, in the form $[width, height]$

To better represent equations in the following sections, `presents` is divided in two additional lists, i.e. `presents_xs` or `px` and `presents_ys` or `py`.

3 Constraint Programming

CP models are implemented with the MiniZinc language and models execution is managed by the official MiniZinc Jupyter extension, called iMiniZinc.

Following standard CP model guidelines we proceded by searching for global constraints, since they enable stronger propagation w.r.t user-defined ones, implied constraints, to allow a reduction of the search tree by pruning, channeling constraints, which can be used to gain a different point of view over the problem, symmetry-breaking constraints, that remove symmetric non-solutions from being analyzed.

In our case-study we tried different approaches, by developing different models. Some of them tend to be faster in a specific subset of instances, w.r.t. the others. In the final model, we tried to put together the different key-points of each model.

In the following subsections each and every tested constraint, along with associated decision variables, will be carefully explained.

Inserire qui roba riguardo variabili/vincoli scartati.

3.1 Decision variables

Bottom-left corners

This is a two-dimensional list of decision variables (`bl_corners` or `b`), where each entry represents the bottom-left corner of a rectangle in the bounding box. Finding a satisfying assignement for this list is the main goal of this project. Moreover, the list is also used to graphically represent every instance solution.

To ease its usage two additional lists were defined (`bl_corners_xs` or `bx` and `bl_corners_ys` or `by`), by channeling over each dimension of the original list.

To reduce the search space, bottom-left corners variables domains are defined as follows:

- `bl_corners`: $0 \dots \max(h, w) - \min(\min(px), \min(py))$
- `bl_corners_xs`: $0 \dots w - \min(px)$
- `bl_corners_ys`: $0 \dots h - \min(py)$

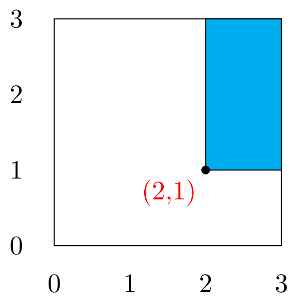


Figure 1: Bottom-left corner example

Top-right corners

As but representing the top-right corner of each rectangle (`tr_corners`). It is used to reduce the number of positions in which a rectangle can fall in, because it must be inside the bounding box.

To reduce the search space, `tr_corners` variables domain is defined as follows:

$$\min(\min(px), \min(py)) \dots \max(h, w)$$

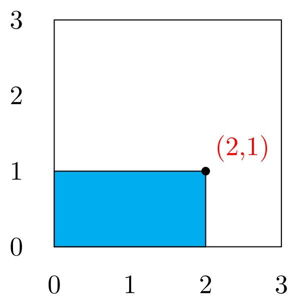


Figure 2: Top-right corner example

Bottom-left corners values

This is a list of decision variables representing a linearization of bottom-left corners (`bl_corners_values`), which uses a one-to-one mapping from each two-dimensional coordinate in the bounding box to an integer value.

The mapping operates as follows:

$$c : (x, y) \mapsto x + (y \cdot m),$$

where $m = \max(h, w)$.

To reduce the search space, `bl_corners_values` variables domain is defined as follows:

$$0 \ldots c(w, h)$$

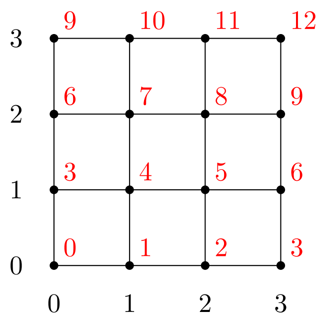


Figure 3: Example of 2D-coordinates linearization in a 3 by 3 box

3.2 Constraints

The constraints described below, divided by scope, are presented at the top of each section with a simple schema depicting their evolution throughout different models. The following is a legend explaining how constraints advancement is achieved:

- `A[x]`: Constraint `A` has been introduced in model number `x`
- `A[x]` $\rightarrow$ `B[y]`: Constraint `A` was removed in favor of `B`, in model `y`

- `A[x]` $\rightarrow$ ✗: Constraint `A` has not been carried over to models `x + 1, ...`

Model numbers are related to their organization inside the attached Jupyter notebook.

3.2.1 Non-overlapment

- Presents cannot overlap [1] $\rightarrow$ Global `diffn_k` [3]
- Global `all_different` [2]

Presents cannot overlap [1]

The idea behind this simple constraint is, given a rectangle, to avoid the existance of areas of overlap with every other rectangle.

$$\max(bx_i, bx_j) \geq \min(bx_i + px_i, bx_j + px_j)$$
$$\vee$$
$$\max(by_i, by_j) \geq \min(by_i + py_i, by_j + py_j)$$
$$\forall i, j = 1 \dots n \mid j > i$$

The described constraint has been observed to be efficient enough for relatively small instances of the problem, while already suffering to position rectangles in a 17×17 bounding box. Results are justified by the disjunctive nature of the constraint, which implies an higher burden in the propagation phase.

Global `diffn_k` [3]

The `diffn_k` global constraint is defined by the official MiniZinc documentation [5] as follows:
Constrains k-dimensional boxes to be non-overlapping. For each box i and dimension j, box_posn[i, j] is the base position of the box in dimension j, and box_size[i, j] is the size in that dimension. Boxes whose size is 0 in any dimension still cannot overlap with any other box.

```
constraint diffn_k(bl_corners, presents);
```

Being a global constraint, it gives a stronger propagation and a more efficient search w.r.t to Presents cannot overlap [1], allowing us to solve bigger instances, up to a 23×23 bounding box.

It's also notable, as described by [2], that `diffn_k` is an onerous constraint. In [2] it accounts for 30 to 80% of the total running time, in an implementation of the *PSP (Perfect Square Packing)* problem, which is very much related to *PWP*.

Global `all_different` [2]

The `all_different` global constraint asserts that every variable has a different value assigned to it.

In our models it is used w.r.t `bl_corners_values` to ensure that every present has different `bl_corners`. The choice of the constrained variables is related to their one-dimensional nature, which guarantees compatibility with MiniZinc's implementation of `all_different`.

```
constraint alldifferent(bl_corners_values);
```

3.2.2 Containment

- Reduce presents domains [1]
- Areas summation [4]

Reduce presents domains [1]

The original description is the following one, where the lenght l corresponds to the height `h_paper` of the bounding box.:
In any solution, if we draw a vertical line and sum the vertical sides of the traversed pieces, the sum can be at most l. A similar property holds if we draw a horizontal line.

As suggested by the assignment, we implemented this simple implied constraint which avoids pieces overflow in both directions.

$$bx_i \leq w + px_i \wedge by_i \leq h + py_i, \forall i = 1 \ldots n$$

Areas summation [4]

This implied constraint is used to enforce presents to occupy the entire bounding box, without any kind of wasted space. In particular, presents areas computed by using top-right and bottom-left corners are linked to the areas calculated using input pieces dimensions.

$$\sum_{i=1}^{n}(tx_i - bx_i) \cdot (ty_i - by_i) \leq w \cdot h$$

$$\sum_{i=1}^{n}(tx_i - bx_i) \cdot (ty_i - by_i) = \sum_{i=1}^{n}(px_i \cdot py_i)$$

3.2.3 Positioning

- Global `count_eq` [2]
- Intervals approach [5] $\rightarrow$ ✗
- Anchor points [5] $\rightarrow$ Anchor points [6]$\rightarrow$ ✗

Global `count_eq` [2]

This implied constraint exploits again the usage of bottom-left corners linearization, i.e. `bl_corners_values`, by stating that one and only one present should be placed with its bottom-left corner at the origin.

$$|\{\, i \mid bx_i = 0 \wedge by_i = 0 \,\}| = 1$$

As mentioned in the section title, this idea has been implemented using the `count_eq` global constraint.

```
constraint count_eq(bl_corners_values, 0, 1);
```

Intervals approach [5]

It represents an idea taken from [1], where domains associated with the x-coordinate of bottom-left corners are reduced on the basis of a variable-sized interval:
[...] a rectangle is assigned an interval of x-coordinates. Interval sizes are hand-picked for each rectangle prior to search, and they induce a smaller rectangle representing the common intersecting area of placing the rectangle in any location in the interval. [...] we assign all x-coordinates prior to any y-coordinates, and use interval variables for the x-coordinates. We set a rectangle's interval size to 0.35 times its width, which gave us the best performance. Finally, we do not use interval variables for the y-coordinates.

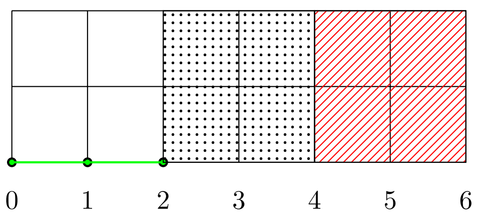


Figure 4: Intervals example: assigning $[0, 2]$ to a 4×2 rectangle

As shown in figure 4, a 4×2 rectangle assigned an x-interval of $[0, 2]$ in a 6×2 box always consumes the units of area represented by the dotted rectangle, while the rectangle containing red lines heading south west only has the possibility, and not the certainty, of being partially or totally consumed. Moreover, the points on the green line represent the feasible assignments to the x-coordinate of the 4×2 rectangle's bottom-left corner.

In a more general situation, where the rectangle's height is less than the bounding box height, the x-interval reasoning remains the same, while the number of feasbile assignments to bottom-left corner's y-coordinate increases.

As a side note, the chosen interval in figure 4 has a size of 2 because

$$\lceil 4 \cdot 0.35 \rceil = \lceil 1.4 \rceil = 2,$$

where 4 is the rectangle's width and 0.35 is the selected parameter to compute interval sizes.

If there were no feasible set of interval assignments, then the constraint would save us from having to try individual x values. However, if we do find a set of interval assignments, then we must search for a set of single x-coordinate values.

In the end, the implementation of this constraint didn't provide significant improvements. Hence, it is not present in the final CP model.

Anchor points [5]

It represents a reduction on each present's domain, such that bottom-left corners reside on corners of other rectangles or on the wrapping paper borders.

The main implementation-wise problem was our inability of correctly expressing the constraint in an efficient way: the only thing we were able to describe is an upper bound on the number of the overall distinct corners that can be found inside the bounding box at the same time, while also satisfying every other constraint. To do that, a new list of decision variables had to be created (`corners_values`), containing the linearization (as in 3.1) of every corner for each present. The constraint was then posted by limiting unique values, i.e. unique corners, over this new list of variables. Then, each bounding box corner was forced to coincide with exactly one value inside `corners_values`.

```
constraint nvalue(corners_values) <= 2 * n + 2
```

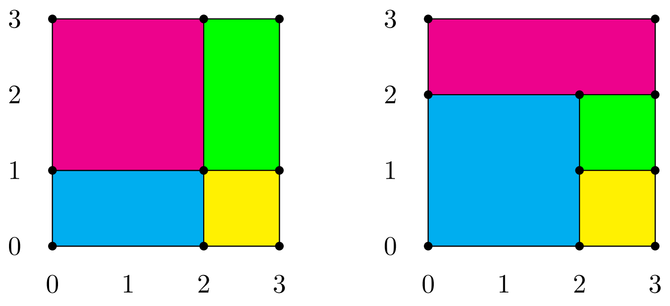
The reported upper bound has been computed as follows:

Observation 1. *Let n be the number of rectangles to be placed inside a given squared bounding box. Let's assume that rectangles will completely fit inside the container, without free space. Then, we have that the number of distinct rectangles corners k should be less than or equal to $2 \cdot n + 2$.*

Reasoning. Since we have n rectangles, the total number of corners is exactly $4 \cdot n$. Moreover, we know that 4 of these corners are reserved by the bounding box ones. For at least half of the other corners ($\frac{4 \cdot n - 4}{2}$), each one must be shared with at least two rectangles, because of our no-free-space assumption. By following this reasoning, we obtain these results:

$$k \leq \frac{4 \cdot n - 4}{2} + 4 = 2 \cdot n + 2 = u$$

□



(a) $k = 9 \leq 10 = u$ (b) $k = 10 \leq 10 = u$

Figure 5: Distinct corners upper bound example

Anchor points [6]

This constraint, taken from [2] is an evolution of Anchor points [5], which introduces a more coincise and efficient anchor points approach. This time the main idea is about causing the bottom-left corner of a single rectangle to fit a bottom-right or top-left corner of another already-placed rectangle, thus reducing the amount of available positions.

$$bx_i \in \{\,0\,\} \cup \{\, bx_j + px_j \mid j = 1 \dots i-1, i+1, \dots n \,\}$$
$$\wedge$$
$$by_i \in \{\,0\,\} \cup \{\, by_j + py_j \mid j = 1 \dots i-1, i+1, \dots n \,\}$$
$$\forall i = 1 \dots n$$

The described formula was implemented using the global constraint `member`, so as to achieve better propagation.

The constraint was later removed from the CP model, since it didn't seem to make any major difference, at least w.r.t. running times over the test instances, even though it was implemented using the mentioned global constraint.

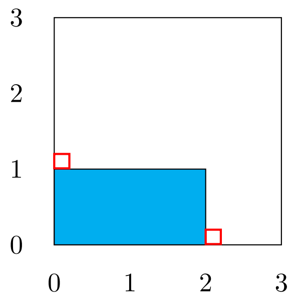


Figure 6: Anchor points example

Figure 6 shows the available placements of a rectangle, different than the one filled in cyan, which could be positioned in such a way that its bottom-left corner would overlap with one of the two small red rectangles.

3.2.4 Stacking

- Global `cumulative` [3]
- Column stacking by two [4] → General column stacking [5]

Global `cumulative` [3]

The `cumulative` global constraint is defined by the official MiniZinc documentation [5] as follows:
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.

```
constraint cumulative(
   bl_corners_xs, presents_xs, presents_ys, h_paper
);
constraint cumulative(
   bl_corners_ys, presents_ys, presents_xs, w_paper
);
```

In the context of rectangle packing, the cumulative global constraint can be used by selecting x-coordinates of bottom-left corners as start times, by assigning durations to presents widths, resource requirements to presents heights and the global resource bound to the paper roll height. The same reasoning can be applied for the other dimension, in order to obtain overflow avoidance and stacking maximization over both axis.

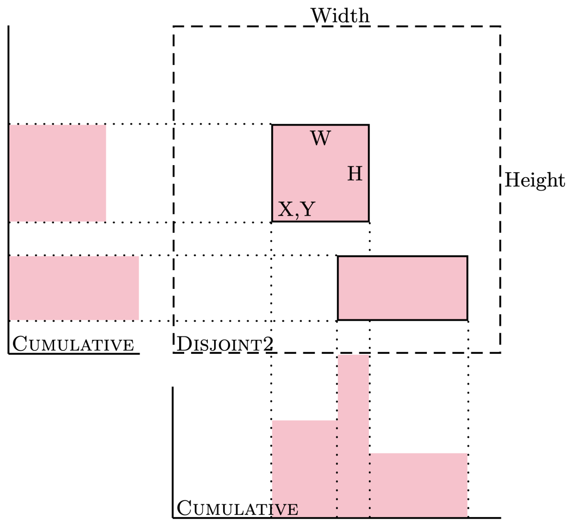


Figure 7: Cumulative global constraint (image taken from [3])

Figure 7 shows an example of the `cumulative` global constraint, along with the usage of `disjoint2`, which is the twin constraint of the already mentioned `diffn_k`.

The combination of `cumulative` and `diffn_k` is actually the core of the CP model, since together they enable a very strong propagation, which can be directly observed by looking at solving times.

Column stacking by two [4]

The main idea behind this simple constraint is to stack presents in a single row or column such that their widths or heights would sum to the total width or height, respectively, of the entire paper roll.

In figure 8, we can observe two different groups, the cyan and the magenta one. In this example, we are stacking by columns, since each color group contains rectangles with the same width and such that their heights sum to the total bounding box height.

When the constraint is applied, both rectangles must have the same x-coordinate for bottom-left corners, while the y-coordinate is assigned to zero for the "first" rectangle and to the height of the first present for the "second" rectangle.

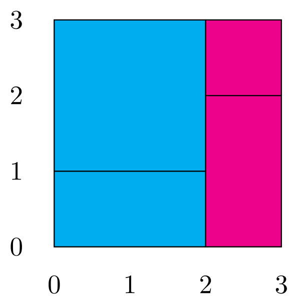


Figure 8: Column stacking by two example

General column stacking [5]

This constraint is a generalization of the simpler Column stacking by two [4], which introduces packing multiple rectangles, i.e. groups of size greater than two, into multiple columns.

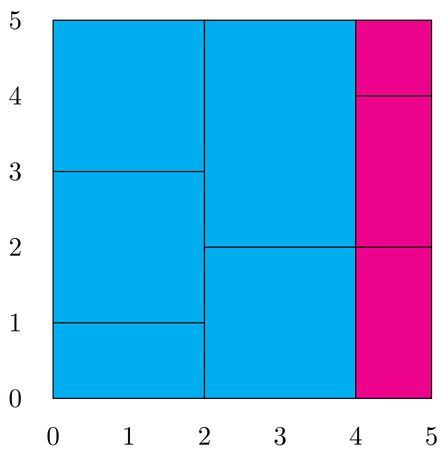


Figure 9: General column stacking example

This predicate is subdivided into different steps:

- Groups identification: finds and groups together presents with the same width
- Feasibility check: ensures that every group can occupy at least one entire column
- Columns computation: calculates the number of columns (and the corresponding widths) occupied by each group (e.g. a group of width 2 with heights $[1, 2, 3]$ in a 12×3 box will occupy exactly two columns of width 2)
- Columns/coordinates assignment: using the `bin_packing` global constraint to identify the column for each rectangle in the same group, it incrementally fixes x and y bottom-left corners coordinates

Groups are ordered by decreasing widths, while rectangles in the same group are sorted by decreasing heights.

In figure 9 we can see how the entire group of cyan-filled rectangles, which has a common width of 2, gets splitted into subsequent columns, while the magenta-colored group can only occupy one column, the last one in this example.

3.2.5 Symmetry breaking

- Biggest rectangle in lower left quadrant [4] → Ordering by areas [6]
- In-column ordering by width [4] → ✗

Biggest rectangle in lower left quadrant [4]

Inspired by the standard n-queens problem, we tried to analyze PWP and its solutions, obtained by reflecting axes.

To cut the search space and avoid exploring paths leading to symmetric non-solutions, we decided to force the biggest rectangle, i.e. the one with greatest area, to have its bottom-left corner inside the lower-left quadrant of the bounding box.

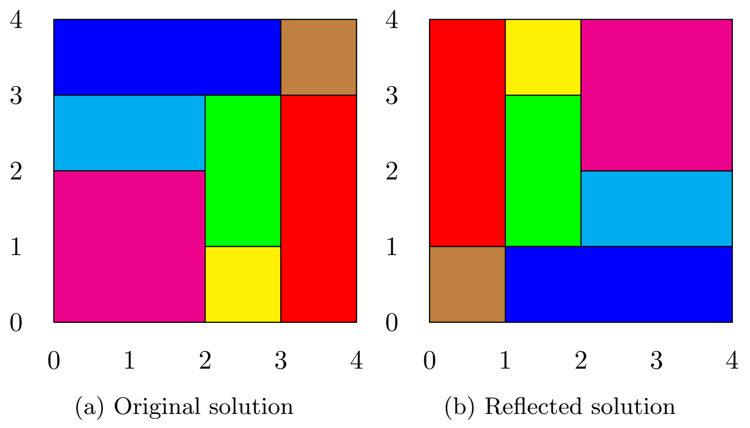


Figure 10: Symmetric solutions example

Figure 10 depicts an example of specular solutions, s.t. subfigure 10a satisfies the constraint, while 10b does not.

In the end, we realized that this approach was too limiting, thus leading to an increase in the number of failures. Because of this, we decided to remove it in favor of Ordering by areas [6].

Ordering by areas [6]

This constraint is based on [2] and it is used to create a lexicographical ordering between the bottom-left corners coordinates of the biggest rectangle r_1, i.e. the one with the greatest area, and the coordinates of the second biggest present r_2. In this way, r_1 must be placed below and/or to the left of r_2.

```
lex_less(
      [bl_corners_ys[r1], bl_corners_xs[r1]],
      [bl_corners_ys[r2], bl_corners_xs[r2]]
   );
```

Implementation-wise, we decided to use the global constraint `lex_less`, as shown above.

In-column ordering by width [4]

This was our first try to enable some kind of ordering between presents with the same bottom-left corners x-coordinates. In particular, those presents which happen to have the same x-coordinate will be constrained to be ordered by their heights s.t. smaller pieces would lie below the taller ones.

Results were not good at all, since this approach transformed a subset of solvable instances into unfeasible ones, because it was highly dependent on input presents order. Hence, the constraint is not present in the final model.

3.3 Models

In this section are presented our final models and their search strategy. The optional model has the ability to handle presents rotation and rectangles with the same shapes.

Search strategy

We decided to prioritize positioning "bigger" presents first, since their placement greatly reduce the domain of the other ones. After trying every meaningful ordering, i.e. by height, width or area, the one that gave us the best outcomes was sorting presents by decreasing values of area.

```
array[1..n] of int: areas =
   [presents_xs[i] * presents_ys[i] | i in 1..n];
array[1..n] of 1..n: order =
   sort_by(1..n, [-areas[i] | i in 1..n]);
```

The actual search strategy starts by checking if General column stacking [5] is feasible. In case of a positive answer we do not need to apply complex approaches, since everything is already handled by column stacking positioning. On the other hand, when column stacking is not feasible, we rely on the previously defined ordering by assigning firstly y-coordinates and then x-coordinates. These assignments are done by selecting values starting from the minimum one in each domain.

In this kind of search strategy applying restarts was not an option, since we didn't have any random component. Anyway we tried to use `luby` and `constant` restarts in a simpler search strategy, but it resulted into worse running times.

```
solve::
 if not(col_stacking_feasible) then
     seq_search([
         int_search(
            [bl_corners_ys[i] | i in order],
            input_order,
            indomain_min,
            complete
         ),
         int_search(
            [bl_corners_xs[i] | i in order],
            input_order,
            indomain_min,
            complete
         )
     ])
 else
   int_search(
      bl_corners_values,
      dom_w_deg,
      indomain_min,
      complete
   )
 endif
satisfy;
```

Final model

To present a global undestanding of our final model, below we report a list linking to each and every included constraint.

- Global diffn_k [3]
- Global cumulative [3]
- Reduce presents domains [1]
- Global all_different [2]
- Global count_eq [2]
- Areas summation [4]
- Ordering by areas [6]
- General column stacking [5]

Using this model we were able to solve every test instance, with the exception of the 23×23 one, in under one second. This was mostly possible thanks to the General column stacking [5] constraint, which enabled us to reduce running times, in particular w.r.t. bigger instances.

Optional model

This model is dedicated to the implementation of the optional parts of the project, namely rotation and same dimensions handling.

About rotation, we removed the column stacking approach, since presents dimensions are now considered as variables, and we introduced the global constraint geost_bb which can be used to find appropriate positions and rotations

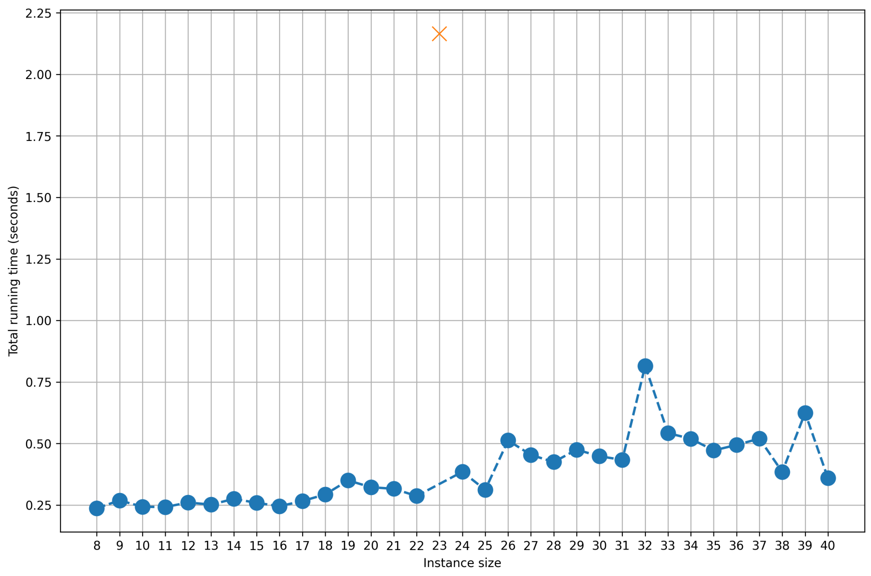


Figure 11: Final CP model running times

of the given rectangles without overlaps, by constraining them to be contained in a given bounding box. This approach obviously turned out to be much slower w.r.t. our final model.

The `geost_bb` global constraint is defined by the official MiniZinc documentation [5] as follows:
A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap, and that all objects fit within a global k dimensional bounding box.

```
constraint geost_bb(
   k, rect_size, rect_offset, shape, bl_corners, kind, l, u
);
```

About `geost_bb` parameters, as described in [4]:

- `k`: The number of dimensions, i.e. 2
- `rect_size`: The size of each rectangle in `k` dimensions, i.e. a list of $2 \cdot n$ elements given by the dimensions of each present and its $90°$ rotation
- `rect_offset`: The offset of each rectangle from the base position in `k` dimensions, i.e. a list of $2 \cdot n$ elements fixed to $[0, 0]$, representing the offset of each rectangle from the top-right corner of its minimum enclosing bounding-box
- `shape`: The set of rectangles defining the i-th shape, i.e.a list of $2 \cdot n$ indexes going from 1 to $2 \cdot n$, representing that each figure is directly associated with exactly one rectangle
- `bl_corners`: The base position of each object

- `kind`: A list of n elements, representing the shape used by each object, i.e. either the original rectangle or its 90° rotation
- `l`: An array of lower bounds, i.e. the bottom-left corner of our paper roll
- `u`: An array of upper bounds, i.e. the top-right corner of our paper roll.

In order to achieve better results, it could be possible to find the minimum number of rotations needed to make column stacking feasible and proceed like before.

About rectangles with same dimensions, we simply included a symmetry breaking constraint accounting for pairwise positioning of equally-sized presents, because these presents can be swapped without any kind of difference in the results.

```
 constraint forall(i in 1..n)(
    let {
        array[int] of int: same_dim_ind = [
             j | j in 1..n where
             presents_xs[j] = presents_xs[i] /\
             presents_ys[j] = presents_ys[i]
          ]
    } in
        if
          length(same_dim_ind) > 1 /\
          min(same_dim_ind) = i
        then
          forall(j in index_set(same_dim_ind) where j > 1) (
             lex_less(
                [
                   bl_corners_ys[same_dim_ind[j - 1]],
                   bl_corners_xs[same_dim_ind[j - 1]]
                ],
                [
                   bl_corners_ys[same_dim_ind[j]],
                   bl_corners_xs[same_dim_ind[j]]
                ]
             )
          )
        else true
        endif
);
```

The above code groups rectangles with the same dimensions and constrains the one with the minimum index to be placed below and/or to the left of the second one; the same happens between the second and the third presents in the same group; and so on and so forth. This approach creates a chain of constraints s.t. every symmetric solution and non-solution is eliminated from the search path.

4 Satisfiability Modulo Theory

References

[1] Eric Huang and Richard Korf. “New Improvements in Optimal Rectangle Packing”. In: Jan. 2009, pp. 511–516.

[2] Mikael Östlund. “Implementation and Evaluation of a Sweep-Based Propagator for Diffn in Gecode”. In: 2017.

[3] Helmut Simonis and Barry O’Sullivan. “Using Global Constraints for Rectangle Packing”. In: (Jan. 2008).

[4] Patrick Trentin. *Understanding the input format of Minizincs geost constraint.* URL: `https://tinyurl.com/geost-bb`.

[5] Monash University. *MiniZinc.* URL: `https://www.minizinc.org/`.