Present Wrapping Problem

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

September 16, 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` ⟵ number of presents to be wrapped
- `w_paper` or `w` ⟵ width of the paper roll
- `h_paper` or `h` ⟵ height of the paper roll
- `presents` or `p` ⟵ 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`: $0 \dots h - \min(py)$

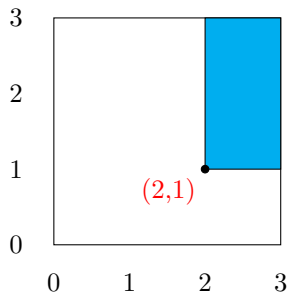

Figure 1: Bottom-left corner example

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:

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

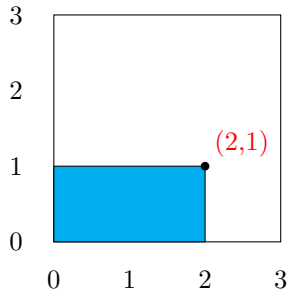

Figure 2: Top-right corner example

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:

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

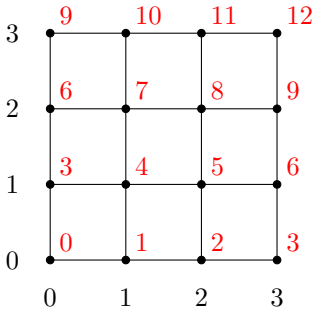

Figure 3: Example of 2D-coordinates linearization in a 3 by 3 box

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]` → `B[y]`: Constraint `A` was removed in favor of `B`, in model `y`
- `A[x]` → ✗: 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] → 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 [4] 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.

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

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] → ✗
- Anchor points [5] → Anchor points [6]→ ✗

Global `count_eq` [2]

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

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

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.

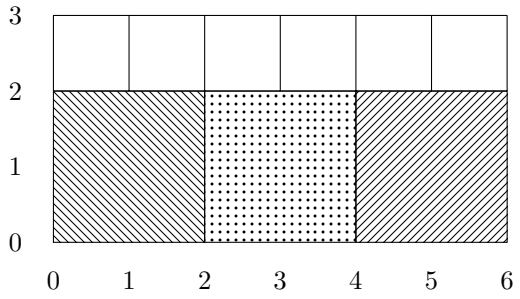

Figure 4: Intervals example: assigning a 4×2 to $[0, 2]$

As shown in figure 4, a 4×2 rectangle assigned an x-interval of $[0, 2]$ consumes 2 units of area at each x-cordinate in $[2, 3]$, represented by the dotted area.

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.

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, and by forcing each bounding box corner to coincide with exactly one corner, over this new list of variables.

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

Anchor points [6]

This constraint 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-left corner of another rectangle or a top-left corner of another rectangle, thus reducing the amount of available presents positions combinations inside the wrapping paper.

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

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 a global constraint.

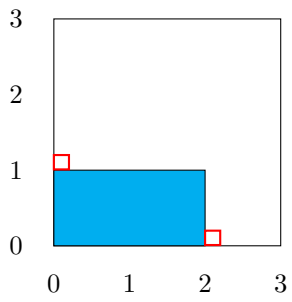

Figure 5: Anchor points example

3.2.4 Stacking

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

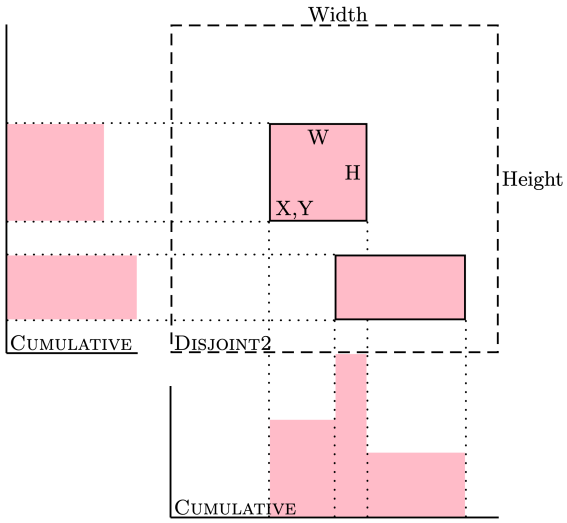

Figure 6: Cumulative global constraint (image taken from [3])

Global `cumulative` [3]

Column stacking by two [4]

General column stacking [5]

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]

Ordering by areas [6]

In-column ordering by width [4]

3.3 Models

Search strategy

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