2D Packing - Constraint Programming

Ciccarello Andrea, Simonazzi Gian Marco

February 2025

1 Introduction

Minimizing wasted space is a critical challenge in various industries, including logistics, manufacturing, and warehousing. One fundamental problem in this context is the 2D Packing Problem, which involves arranging a set of rectangular items within a confined two-dimensional area without overlaps and minimizing wasted space.

The problem addressed in this project is the following: A square pallet of size $k \times k$ has to be filled with boxes of size $x \times y$. All boxes have the same size. The boxes can be rotated by 90 degrees and their edges must be parallel to the pallet's edges. Boxes are placed at integer coordinates and must not overflow the pallet and must not overlap other boxes. Given k, x, y, the problem is to find the 2D arrangement that maximizes the number of boxes on the pallet.

To test the model, a series of benchmarks are also established. The 36 benchmarks will use $x = \{5, 10, 15, 20\}$ and $y = x + 5$ boxes and a $k = \{20 - 1, 30 - 1, \dots, 100 - 1\}$ pallet. A timeout of 5 minutes (300 seconds) is also established.

2 Modelling

From the definition of the problem we can assume that the values k,x and y are provided by the user, and thus will be defined as parameters of the model. Since the goal is to maximize the number of boxes, this value has to be a variable for the model, which will be called `boxes`. We require at least one box to be placed: without this change, when the boxes are bigger than the pallet, the model sometimes is unsatisfiable and other times results `boxes = 0`.

Next, we need to define the positions of the boxes and their rotation as variables. MiniZinc offers the global constraint `diffn_k`, which ensures that the boxes are non-overlapping in d-dimensions. This constraint takes two integer matrices, one for positions and the other for the sizes of the boxes, of dimension $n \times d$, with n the number of boxes. In our model we consequently define both matrices of variables to be used with the constraint. The domain for the `positions` matrix is $[1, k]$ and the domain for the `sizes` matrix is $\{x, y\}$. They each define the position of the corner closest to the origin and the sizes along the two axes for every box. We have explored the possibility of redifining `sizes` to use boolean values to indicate whether a box is horizontal or vertical in order to reduce the number of variables. This approach was, however, quickly dropped since it only increased the complexity of the model (we would need to compute the sizes of the boxes for many constraints) and did not improve performance.

Since the number of boxes to be placed is a variable (`boxes`), this value cannot be used as a dimension for the aforementioned variables. To work around this limitation, we define an upper limit for the number of boxes using the area of both the pallet and the boxes:

$$n = \left\lfloor \frac{k^2}{x \cdot y} \right\rfloor$$

With this, we can define the matrices with dimension $n \times 2$.

The variables and parameters used in the model are then the following:

```
int: k;          % Size of the pallet
int: x; int: y; % Dimensions of the boxes
int: n = (k*k) div (x*y); % Upper bound for the boxes

array [1..n,1..2] of var 1..k: positions; % Position of the
                                          % upper-left corner
                                          % of every box
array [1..n,1..2] of var {x,y}: sizes; % Sizes for every box
var 1..n: boxes; % Number of boxes
```

Next we define the model's constraints. First, we need to ensure that every box is either of size $x \times y$ or $y \times x$. This is equivalent to asking that for every box the sizes on the two dimensions must be different if $x \neq y$.

Another constraint ensures that boxes do not overflow the pallet. For this, we ensure that, for both dimensions, the sum of the position of each box with its size is within the pallet's limits.

As mentioned before, the non-overlapping property is ensured by the global constraint `diffn_k` on the arrays `positions` and `sizes`. We have wrote and used a custom version of this constraint when we tried to change the structure of the variables (for example, the boolean values for `sizes` mentioned earlier). However, we decided to use `diffn_k` in order to fully take advantage of the custom propagator and to improve the readability of the model.

The constraints used by the model are the following:

```
% Respect box size in rotation
constraint forall(i in 1..boxes)(
  sizes[i,1] != sizes[i,2] \/ x = y
);

% The boxes must not overflow the pallet
constraint forall(i in 1..boxes)(
  positions[i,1] + sizes[i,1] <= k + 1 /\
  positions[i,2] + sizes[i,2] <= k + 1
);

% The boxes must not overlap
constraint diffn_k(positions, sizes);
```

We also included a formatted output for the model variables. This output can be visualized using a Python web app, described in Additional Tools (Chapter 6).

These constraints and variables can describe a first iteration of the model. The results are discussed in the following section.

2.1 Performance

2.1.1 Solver

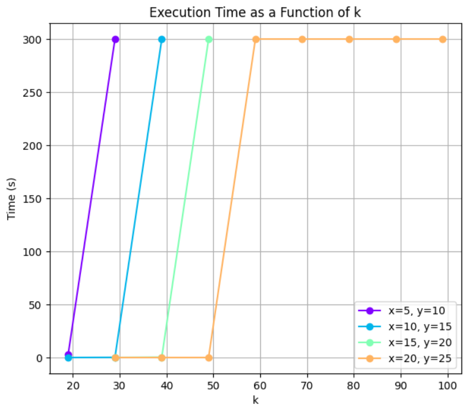
We initially used Gecode, the most common solver for Minizinc, since it was the easiest to use and provided a useful tool to visualize the search tree (Gist). We quickly realized, however, that it would be difficult to scale the problem up using this solver, given the lack of propagation for this model.

In order to address these performance limitations, we switched to `Chuffed`. Chuffed is a high-performance lazy clause solver that uses a hybrid approach called lazy clause generation, which blends finite domain propagation with Boolean satisfiability techniques. Thanks to its hybrid architecture and its efficient pruning of nogoods, `Chuffed` often delivers faster solve times on challenging combinatorial benchmarks than classical finite-domain solvers.

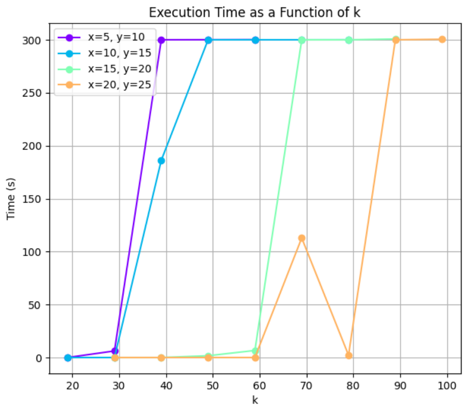
Switching to Chuffed significantly improved the solve times and the scalability of the model (as seen in Figure 1).

2.1.2 Results

We now show the first results for the model, using the benchmarks described in the Introduction. The results are shown in Figure 1b and a table of the raw results (Table 1) can be found in Appendix A. We also included a plot for the results using Gecode (Figure 1a) as reference.



(a) Execution time Gecode



(b) Execution time Chuffed

Figure 1: Gecode vs Chuffed.

We can see from Figure 1b that there are lots of tests that are very fast (less than a second), but the time quickly rises to minutes and over the timeout of 5 minutes. In particular, the cases in which there are few boxes to place (from Table 1, `boxes <= 4`) are very fast, and, for an increasing number of boxes to be placed, the time rises.

It is also worth noting, looking at some visualizations of the results (Figure 2), that the boxes seem to be placed a bit randomly on the pallet. There are definitely many translation symmetries that have to be addressed in order to optimize the solving.

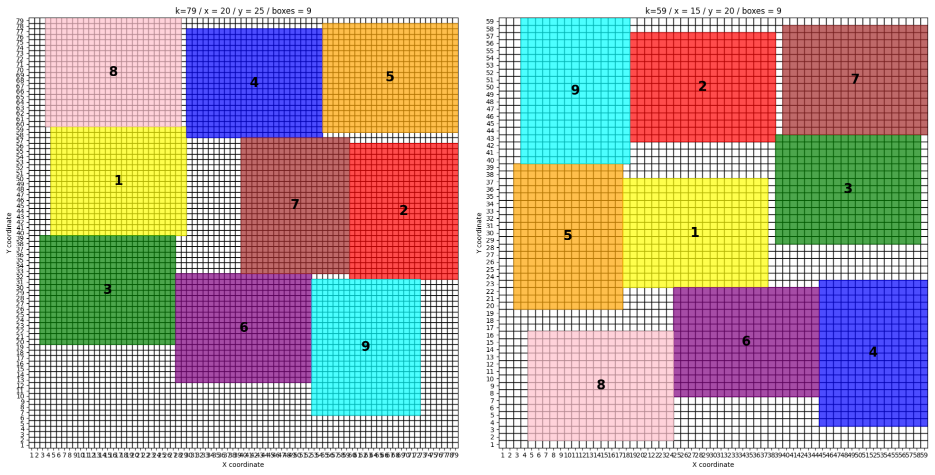


Figure 2: Some visualizations of the benchmark results

Looking at the visualizations we also find the first instances of a particular pattern that frequently emerges. We call this pattern *donut* (given the hole in the middle) and it is the optimal arrangement for 4 boxes when all of the following conditions are true:

- $3x > k \wedge 3y > k$
- $x + y \leq k$

It is also valid for $4k$ boxes, with $k \geq 1$, since we can treat groups of k boxes as a single box. Note that the donut is square and is of size $x + y$, meanwhile the hole inside is also square and is of size $x - y$.

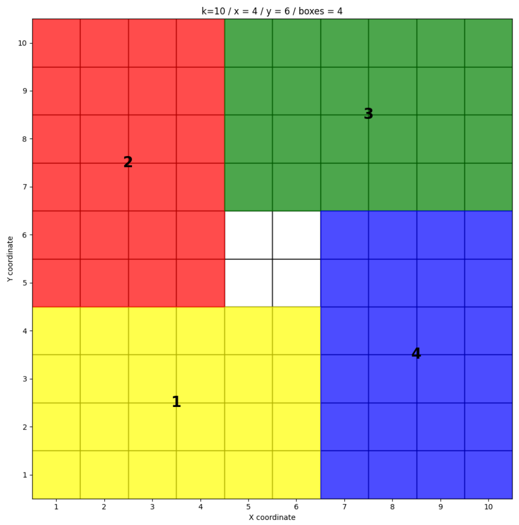


Figure 3: Example of a donut with k = 10, x = 4 and y = 6

3 Symmetry breaking

We have observed that symmetries arise quite naturally, both for individual boxes and within larger groups.

- **Translation symmetries**: The boxes can be shifted by some amount in any direction if there is empty space available. This can affect single boxes or groups of them.
- **Rotation symmetries**: Given an optimal solution, the boxes can be rotated without greatly changing the structure of the result. This, as well, affects single boxes or groups of them.
- **Mirror symmetries**: A solution (or parts of it) can be flipped along the x or y axis.
- **Box ordering**: Given an optimal result, there are multiple solutions that have the same positions for the boxes but multiple permutations of the positions inside the array.

In general, our goal to remove symmetries is to only accept solutions that are as dense as possible and as close to the origin as possible. This means that we do not want any space left in between boxes, with exception for donuts and similar structures.

The global constraint `diffn_k` does not enforce an ordering on the boxes that it places on the plane. To prevent this, we force the ordering of the array using the global constraint `lex2` that ensures a lexicographic ordering for a matrix. Note that this constraint can be used only on the array `positions`.

We will talk about the symmetry breaking for box translation later. Assume for now that we have already constrained the boxes to stay together and to not waste any space in between them. Even if this constraint makes it possible to create a single group of boxes that acts like a single shape, this group can still translate if there is available space around it. To prevent this symmetry, we force the first box to be placed at the origin. Furthermore we ensure that the first box is horizontal, in order to (partially) prevent rotations.

These symmetry breaking constraints are the following:

```
% Symmetry breaking 1: The first box must be at the origin (1,1) and is horizontal
constraint (
  positions[1,1] = 1 /\ positions[1,2] = 1 /\
  sizes[1,1] = max(x,y) /\ sizes[1,2] = min(x,y)
);

% Symmetry breaking 2: Order the positions
constraint lex2(positions);
```

3.1 Restricting the positions using GCD

As mentioned before, we still need to address the translation symmetries. We first tried forcing a dense placement of boxes using an "adjacency" constraint. We determined that two boxes were adjacent when they shared an extreme on one dimension (eg. for boxes i, j,
`positions[i,1] = positions[j,1] + sizes[j,1]`) and they (partially) overlap edges in the other dimension. This made it possible to create an *adjacency graph*, where we constrain each box to have at least two adjacent boxes. This constraint worked but did not improve performance enough, given the computation complexity of the graph's creation.

However, looking at the dense solutions for the benchmarks, we noticed that the boxes were always placed at multiples of 5 in both dimensions (+1 given that the positions array starts at 1). This hinted at the fact that boxes can be placed at multiples of $gcd(x, y)$. This works thanks to Bézout's identity. One of its generalizations states that, given $d = gcd(x, y)$, it is possible to express multiples of d as a linear combination of x, y:

$$ax + by = kd$$

with a, b non-zero integers. This is actually what we were looking for: given a specific column or row of the grid (so fixing a specific value of x or y), we find a succession of vertical or horizontal boxes without gaps in between.

Most importantly, this also works with *donuts*. Given that the square hole in the donut has edges of size $x - y$, we know that $gcd(x, y)$ divides $x - y$, and thus the size of the entire donut is a multiple of $gcd(x, y)$.

This enabled us to restrict the domain of positions only to multiples of $gcd(x, y)$ without losing solutions. This change was the most impactful to the performance of the model, slashing the search time by orders of magnitude and making it possible to solve some of the bigger pallets in the benchmarks.

We decided to implement this directly as a set to use as domain for `positions`, in order to bypass domain propagations that the solver may not perform. To do this we used a recursive function that implements Euclid's algorithm in order to compute the GCD and to statically define the domain of the matrix.

```
function int: gcd(int: a, int: b) =
  let {
    int: abs_a = abs(a);
    int: abs_b = abs(b);
  } in
  if abs_b == 0 then abs_a
  else gcd(abs_b, abs_a mod abs_b)
  endif;

int: modulo = gcd(x, y);

% Position of the upper-left corner of every box
array [1..n,1..2] of var {1 + modulo * t | t in 0..(k div modulo)}: positions;
```

3.2 Search strategy

We also analyzed the default search strategy of the solvers used. Using Gist, we saw that, using a default optimization search, the solvers tried to first assign a value to `boxes` and then tried to find a valid box placement for that value. This was very inefficient since it meant that the solver tried to create the same arrangement multiple times (even with clause learning in Chuffed).

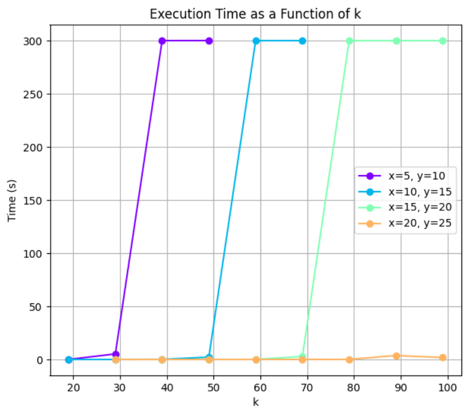
We then tried to improve the search strategy using the annotations provided by MiniZinc. Ideally, we want the solver to construct the optimal solution by arranging the boxes on the pallet (so working on `positions` and `sizes`) and shrink the domain of `boxes` accordingly. To do this we used a sequential search on `positions` and `sizes` with `input_order` as variable choice annotation. This means that the solver is supposed to place one box at a time in ascending order by index. For `positions` we also place a value choice annotation `indomain_min` in order to place the box as close to the origin as possible.

The search strategy we used is the following:

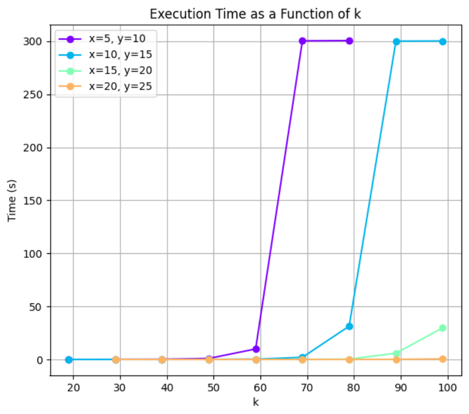
```
    solve :: seq_search([
              int_search(positions, input_order, indomain_min),
              int_search(sizes, input_order, indomain_max)
              ])
      maximize boxes;
```

3.3 Results

In this section we test both the model with only the basic symmetry breaking constraints and the model with also the GCD domain reduction. As before, Figure 4 shows the time requirements for each benchmark and Tables 2,**??** show the complete results for the benchmarks. Also, as mentioned before, all the tests are now run with Chuffed. The model without GCD (Figure 4a) shows some



(a) Execution time (without GCD)



(b) Execution time (with GCD)

Figure 4: Symmetry breaking results.

improvements, making it now possible to compute the solution for all pallets with the biggest box size (20×25) and to compute all benchmarks for pallets of size up to 39.

However, the model with GCD (Figure 4b) shows impressive improvements. Not only is it able to compute all benchmarks for pallets of size up to 69, it also computes all benchmarks for box size 15×20 and, in general, is much faster than the one without GCD.

Note that, given our search strategy, both models have the same optimal arrangements as final results, since they both find the same first optimal solution. Looking at some results (Figure 5) we can see that we have now restricted the randomness in the arrangement of boxes and it is much more dense.

4 Conclusions

As we have seen, this problem can be tricky to solve using constraint programming. It is fairly trivial to model, thanks to the global constraint `diffn_k`, but it is difficult to solve efficiently given the

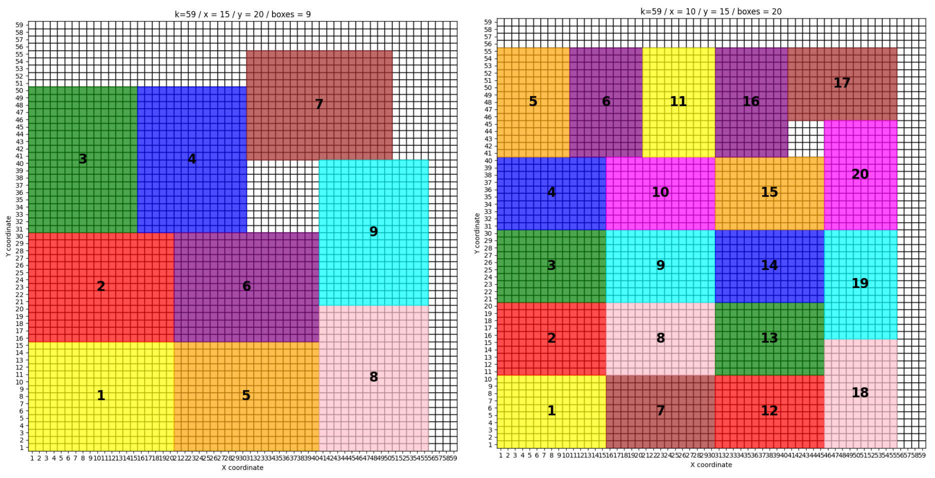


Figure 5: Some visualizations of the benchmark results

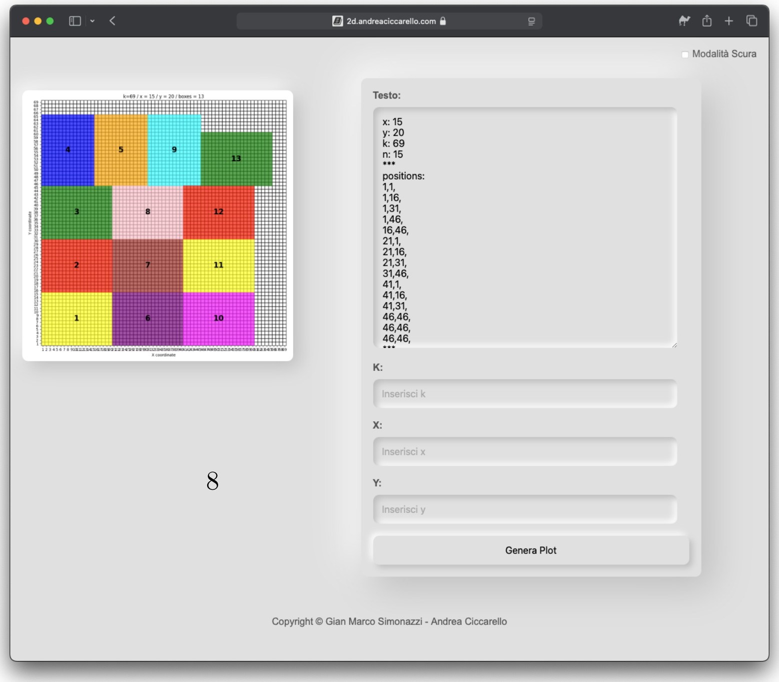
amount of symmetries involved. We managed to exploit an interesting property of the problem to cut translation symmetries, but there is still work to be done.

There are still symmetries involving rotations of the individual boxes that exponentially increase the number of possible solutions as the number of boxes grows. Similar to the adjacency constraint, we tried multiple different constraints in order to break these symmetries, without, however, improving the efficiency of the search. Unfortunately we could not manage to break all of the symmetries that we have found and we think that breaking them could make it possible to pass all benchmarks with ease.

5 Additional tools

5.1 Visualization Tool

This tool displays our 2D packing model's output, showing how items are arranged in a two-dimensional space without overlapping. It offers an intuitive interface to load input data (item sizes, grid size) and then provides a graphical representation of the solution. It has been developed with a Python script that generates the visualization, a Flask server, and everything is containerized



```
x: 15
y: 20
k: 69
n: 15
***
positions:
1,1,
1,16,
1,31,
1,46,
16,46,
21,1,
21,16,
21,31,
31,46,
41,1,
41,16,
41,31,
46,46,
46,46,
46,46,
***
```

Copyright © Gian Marco Simonazzi - Andrea Ciccarello

Figure 6: Visualization Tool

thanks to Docker.

Try it here: `https://2d.andreaciccarello.com`

A Result tables

Since there are many timeouts, we show only the cases that have results and the first timeout for a given pallet size (since, given a pallet size, the problem gets harder as the boxes shrink in size).

k	**x**	**y**	**boxes**	**time (s)**
19	20	25	Unsatisfiable	-
19	15	20	Unsatisfiable	-
19	10	15	1	0.08
19	5	10	4	0.10
29	20	25	1	0.08
29	15	20	1	0.09
29	10	15	4	0.09
29	5	10	Time out	-
39	20	25	1	0.08
39	15	20	4	0.09
39	10	15	8	137.87
39	5	10	Time out	-
49	20	25	4	0.09
49	15	20	6	1.59
49	10	15	Time out	
59	20	25	4	0.10
59	15	20	9	5.17
59	10	15	Time out	-
69	20	25	8	86.98
69	15	20	Time out	-
79	20	25	9	2.15
79	15	20	Time out	-
89	20	25	Time out	-
99	20	25	Time out	-

Table 1: First model (Chuffed)

k	**x**	**y**	**boxes**	**time (s)**
19	20	25	Unsatisfiable	-
19	15	20	Unsatisfiable	-
19	10	15	1	0.09
19	5	10	4	0.09
29	20	25	1	0.09
29	15	20	1	0.09
29	10	15	4	0.09
29	5	10	12	5.30
39	20	25	1	0.09
39	15	20	4	0.09
39	10	15	8	0.14
39	5	10	Time out	-
49	20	25	4	0.09
49	15	20	6	0.10
49	10	15	13	2.29
49	5	10	Time out	-
59	20	25	4	0.09
59	15	20	9	0.12
59	10	15	Time out	-
69	20	25	8	0.15
69	15	20	13	2.94
69	10	15	Time out	-
79	20	25	9	0.11
79	15	20	Time out	-
89	20	25	13	3.80
89	15	20	Time out	-
99	20	25	16	1.89
99	15	20	Time out	-

Table 2: Symmetry breaking (no GCD)

k	**x**	**y**	**boxes**	**time (s)**
19	20	25	Unsatisfiable	-
19	15	20	Unsatisfiable	-
19	10	15	1	0.09
19	5	10	4	0.10
29	20	25	1	0.09
29	15	20	1	0.09
29	10	15	4	0.09
29	5	10	12	0.12
39	20	25	1	0.09
39	15	20	4	0.09
39	10	15	8	0.10
39	5	10	24	0.24
49	20	25	4	0.09
49	15	20	6	0.10
49	10	15	13	0.13
49	5	10	40	0.90
59	20	25	4	0.09
59	15	20	9	0.11
59	10	15	20	0.32
59	5	10	60	8.37
69	20	25	8	0.10
69	15	20	13	0.16
69	10	15	28	2.08
69	5	10	84	250.44
79	20	25	9	0.11
79	15	20	18	0.31
79	10	15	37	26.42
79	5	10	Time out	-
89	20	25	13	0.19
89	15	20	24	5.37
89	10	15	Time out	-
99	20	25	16	0.65
99	15	20	30	27.93
99	10	15	Time out	-

Table 3: Symmetry breaking (with GCD)