Present Wrapping Problem: CP implementation

Lorenzo Mario Amorosa
Mattia Orlandi
Giacomo Pinardi

Master in Artificial Intelligence - University of Bologna

A.Y. 2020-2021

1 Naive baseline model

1.1 Data representation

Each instance of the problem is encoded in a `.dzn` file as follows:

```
w = 9;
h = 12;
n = 5;
DX = [3, 2, 2, 3, 4];
DY = [3, 4, 8, 9, 12];
```

where:

- w and h represent, respectively, the wrapping paper width and height;
- n is the number of pieces of paper to cut off;
- DX and DY are two arrays, indexed from 1 to n, containing the horizontal and vertical sizes of each piece of paper (for example, in this case the fifth piece of paper has a size equal to 4×12).

The solution will be encoded using two arrays of decision variables X and Y, indexed from 1 to n. Each element i of the X (resp. Y) array will contain the x (resp. y) bottom-left corner coordinate of the i-th piece of paper.

For example, a possible solution of the previous instance is the following:

```
X = [6, 4, 4, 6, 0];
Y = [0, 8, 0, 3, 0];
```

A visualization of such solution is shown in figure 1: as it can be seen, the bottom-left corner of the yellow rectangle (which corresponds to index 2) has coordinates (4, 8), thus $X_2 = 4$ and $Y_2 = 8$.

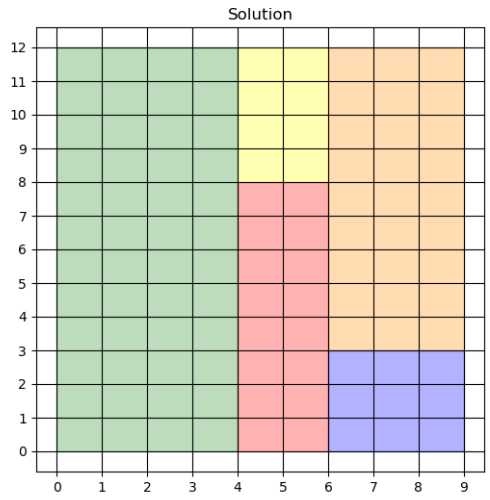

Figure 1: Possible solution to the 9×12 instance.

1.2 Main constraints

First of all, the domain of each decision variable in X (resp. Y) is restricted to $[0, w - \min(DX)]$ (resp. $[0, h - \min(DY)]$): in fact, considering the piece of paper i with the smallest width, the bottom-left corner will be placed at most at a distance DX_i from the vertical border. An equivalent reasoning can be applied to take into account also the height.

In MiniZinc we can express the concepts above by defining the decision variables' domains as follows:

```
array[1..n] of var 0..(w - min(DX)): X;
array[1..n] of var 0..(h - min(DY)): Y;
```

Secondly, we impose a non-overlapping constraint:

- given a piece of paper i, $X_i + DX_i$ represents the x coordinate of its right-side border;

- given a second piece of paper j, X_j represents the x coordinate of its left-side border;
- by enforcing the $X_i + DX_i \leq X_j$ constraint, we avoid the two borders overlapping;
- an equivalent reasoning is behind the other three expressions in the constraint below.

Constraint 1: No overlap

```
constraint forall(i, j in 1..n where i < j) (
     X[i] + DX[i] <= X[j]
  \/ X[j] + DX[j] <= X[i]
  \/ Y[i] + DY[i] <= Y[j]
  \/ Y[j] + DY[j] <= Y[i]
);
```

In particular, the expressions are connected using the $\vee$ operator since it is sufficient that at least one of the following conditions is satisfied:

- piece i is at the left hand of piece j;
- piece i is at the right hand of piece j;
- piece i is below piece j;
- piece i is above piece j.

Finally, we enforce the pieces not to overflow the wrapping paper's horizontal and vertical borders by means of the following constraint:

Constraint 2: No overflow

```
constraint forall(i in 1..n) (
  X[i] + DX[i] <= w /\ Y[i] + DY[i] <= h
);
```

2 Dual model

To enhance constraint propagation and facilitate the expression of implied constraints, we developed a dual model: it is implemented with a 2-dimensional matrix $\mathbf{B} \in \mathbb{N}^{h \times w}$ which represents the whole wrapping paper. Each cell $\mathbf{B}_{i,j}$ contains a number $v \in 0..n$ representing either the "empty cell" (when $v = 0$) or the piece of paper to wrap-up the v-th present (when $v = 1..n$).

For example, the dual model of the previous solution (cf. figure 1) results in the $\mathbf{B}$ matrix shown in figure 2. In this case, since there are no empty cells, the matrix does not contain zeroes.

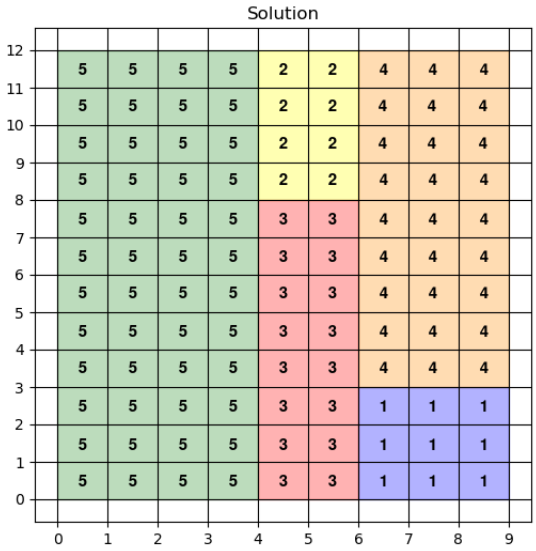

Figure 2: Dual model of the solution to the 9×12 instance.

To maintain consistency between the primal model (i.e., the X and Y arrays) and the dual model (i.e., the $\mathbf{B}$ matrix), we added the following channelling constraint:

Constraint 3: Channelling

```
constraint forall(i in 1..h, j in 1..w, v in 1..n) (
  B[i, j] = v <-> i > Y[v] /\ i <= Y[v] + DY[v] /\ j > X[v] /\ j <= X[v]
  ↪  + DX[v]
);
```

Such constraint states that each cell $\mathbf{B}_{i,j}$ must contain the value v iff the coordinates (i, j) are between the v-th piece of paper's boundaries.

With the dual formulation, the constraint related to the overflow (cf. constraint 2) becomes semantically redundant: however, it is still computationally relevant because it greatly reduces the number of failures, according to our tests. Therefore, we decided to keep it in our model.

2.1 Implied constraints

To implement the implied constraints, we relied on the helper function defined below:

```
function var set of 0..n: get_set(int: k, bool: along_x) =
if along_x then
  array2set([B[k, j] | j in 1..w])
else
  array2set([B[i, k] | i in 1..h])
endif;
```

Such function receives in input an index k and a boolean flag `along_x` indicating whether the index is a row (`along_x` = `true`) or a column (`along_x` = `false`): then, it scans the k-th row/column of $\mathbf{B}$ and outputs the set of unique encountered values v.

Then, this values are used in combination with the primal model in order to gather the lengths of the corresponding pieces of paper and constrain their sum (according to the horizontal/vertical dimension). Empty cells are not taken into account.

Constraint 4: Implied (version 1)

```
constraint forall(i in 1..h) (
  sum([DX[v] | v in get_set(i, true) where v != 0]) <= w
);
constraint forall(j in 1..w) (
  sum([DY[v] | v in get_set(j, false) where v != 0]) <= h
);
```

Alternatively, such constraints can be simplified using the `among` global constraint instead of the `get_set` function:

Constraint 5: Implied (version 2)

```
constraint forall(i in 1..h) (
  among([B[i, j] | j in 1..w], 1..n) <= w
);
constraint forall(j in 1..w) (
  among([B[i, j] | i in 1..h], 1..n) <= h
);
```

2.2 Symmetry breaking

To further reduce the number of solutions and unfeasible assignments, we applied some symmetry breaking constraints:

Constraint 6: Symmetry breaking (version 1)

```
constraint symmetry_breaking_constraint(
     lex_lesseq(array1d(B), [B[i, j] | i in reverse(1..h), j in 1..w])
  /\ lex_lesseq(array1d(B), [B[i, j] | i in 1..n, j in reverse(1..n)])
  /\ lex_lesseq(array1d(B), [B[i, j] | i, j in reverse(1..n)])
);
```

In particular:

- the first constraint avoids vertical symmetries;
- the second constraint avoids horizontal symmetries;
- the third constraint avoids 180° rotations.

These three constraints are sufficient because, in the base case, rotations are not allowed. However, the third constraint is still relevant since, due to the fact that all pieces are rectangular, a 180° rotation is equivalent to a translation (as it can be seen in figure 3).

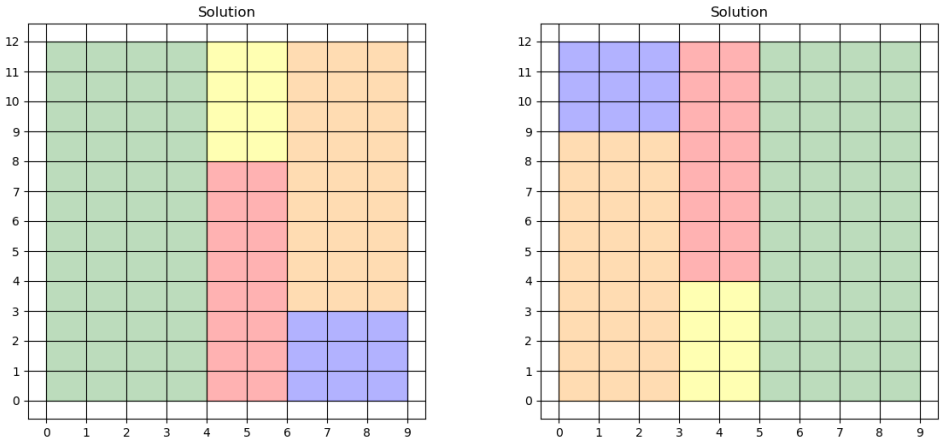

Figure 3: The original solution (on the left) and its 180° rotation.

3 Final model

Empirically, we noticed some limits of the dual model: although the symmetry breaking constraints greatly reduced the number of unfeasible assignment, the channelling constraints were too onerous from a computational point of view, and hence contributed in increasing the execution time.

Therefore, to improve the model's performance we discarded the dual model and re-implemented the two implied constraints using the global `cumulative` constraint:

Constraint 7: Global implied

```
constraint cumulative(X, DX, DY, h);
constraint cumulative(Y, DY, DX, w);
```

Although such constraint is related to scheduling, we can reduce our problem to a scheduling one: in fact, if we take into account the horizontal dimension, we can think of each piece of paper i as a task with start time X_i and duration DX_i, while its vertical size DY_i can be thought of as the resource requirement of the i-th piece of paper. In this interpretation, the `cumulative` constraint ensures that, at any given time (i.e. at any position along the x axis), the required shared resource (i.e. the vertical size of each piece of paper in such position) never exceeds the capacity h. An equivalent reasoning can be applied to the vertical dimension.

We also replaced the implementation of the `no-overlap` constraint (cf. constraint 1) with the more efficient global constraint `diffn`:

Constraint 8: Global no-overlap

```
constraint diffn(X, Y, DX, DY);
```

Such constraint enforces each rectangle i, given by its origins X_i and Y_i and by its sizes DX_i and DY_i, not to overlap with each other.

Finally, we kept the `no-overflow` constraint (cf. constraint 2) since, without the dual model, it is no longer semantically redundant.

Concerning the symmetry breaking constraints, they are difficult to express without the dual model; however, we can at least rule out some unfeasible solution by enforcing the first block to be placed in the bottom-left quadrant:

Constraint 9: Symmetry breaking (version 2)

```
constraint symmetry_breaking_constraint(
  X[1] <= w div 2 /\ Y[1] <= h div 2
);
```

4 Benchmark

In this section we analyse the results of our benchmarks between the models provided so far and the different search strategies. All the tests were performed on an Intel i7-10750H CPU with 12 cores (6 physical and 6 logical).

4.1 Models

We tested the following configurations:

- naive model;
- dual model;
- final model without symmetry breaking;
- final model with symmetry breaking;

on all instances from 10×10 to 15×15. In all the experiments, we employed the default search heuristic, and we set the number of threads to 12 and the time limit to 1 minute.

The results are shown in table 1.

Model benchmark				
	Naive	Dual	Final (no sym)	**Final (sym)**
10×10	T: 25.2 ms S: 64 F: 171	T: 26.7 ms S: 22 F: 156	T: 22.9 ms S: 64 F: 92	T: **27.0 ms** S: **20** F: **38**
11×11	T: 25.4 ms S: 128 F: 3.9K	T: 62.8 ms S: 48 F: 2.8K	T: 31.0 ms S: 128 F: 500	T: **19.7 ms** S: **66** F: **395**
12×12	T: 128.6 ms S: 192 F: 86.6K	T: 854.6 ms S: 58 F: 48.3K	T: 32.7 ms S: 192 F: 1.9K	T: **28.4 ms** S: **120** F: **1.7K**
13×13	T: 445.0 ms S: 1568 F: 333.6K	T: 2.6 s S: 448 F: 207.5K	T: 108.6 ms S: 1568 F: 5.3K	T: **68.0 ms** S: **644** F: **6.0K**
14×14	T: 451.3 ms S: 1344 F: 325.7K	T: 5.0 s S: 370 F: 304.7K	T: 89.6 ms S: 1344 F: 4.7K	T: **47.6 ms** S: **544** F: **3.8K**
15×15	T: 12.5 s S: 10752 F: 8.8M	T: 57.6 s S: 2732 F: 2.4M	T: 955.1 ms S: 10752 F: 22.5K	T: **356.9 ms** S: **3576** F: **34.1K**

Table 1: Execution time (T) and number of solutions (S) and failures (F) for each model and for each instance. T and F were averaged over 5 runs.

As it can be seen, the dual model is the slowest, but it produces fewer solutions and failures compared to the naive model, meaning that the original symmetry breaking constraints (cf. constraint 6) work well.

However, the advantage of using global constraints is clear: their specialized propagation mechanisms lead to a faster execution time and to a dramatic reduction in the number of failures. Moreover, the second version of the symmetry breaking constraints (cf. constraint 9), despite being weaker than the original one (which exploits the dual formulation of the problem), is still effective in improving the model's performance overall.

4.2 Search heuristics

Considered the results obtained in the previous experiments, we decided to test only the final model with symmetry breaking. In particular, we tested the following search heuristics:

- `input_order/indomain_min` (A);
- `first_fail/indomain_min` (B);
- `dom_w_deg/indomain_min` (C);
- `input_order/indomain_random` (D);
- `dom_w_deg/indomain_random` (E);

on all instances from 15×15 to 20×20. We set the number of threads to 12 and the time limit to 2 minutes.

The results are shown in figure 4.

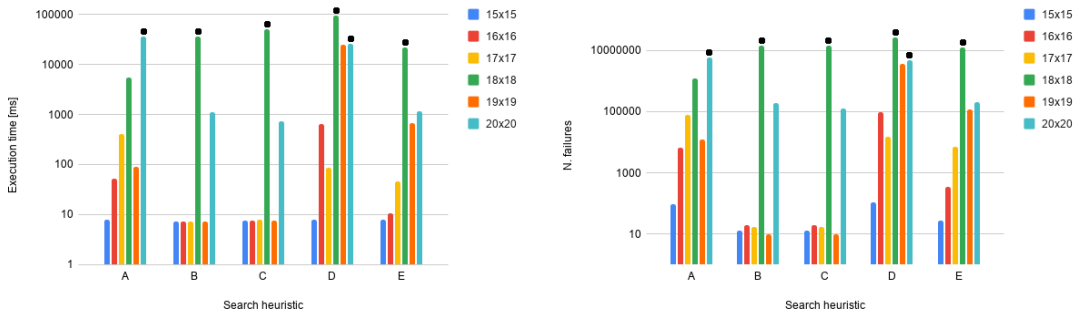

Figure 4: Execution time and number of failures, in logarithmic scale, for different search heuristics. The results were averaged over 5 runs: the • sign on the top of the bar indicates that one or more runs produced no solution.

As it can be seen, the best strategies are `first_fail/indomain_min` (B) and `dom_w_deg/indomain_min` (C), which have a quite similar performance.

Another observation that can be made is that the 18×18 and 20×20 instances are, on average, more difficult than the others.

4.3 Restart

Considered the results obtained in the previous experiments, we decided to choose the heuristic `dom_w_deg/indomain_min` because of its non-deterministic behaviour: this, in combination with a restart strategy, allows the model to change the explored solution space at every restart. We tested the following restart strategies:

- constant ($L = 100$);
- geometric ($\alpha = 1.5$, $L = 100$);
- Luby ($L = 100$).

on all instances from 20×20 to 25×25. We set the number of threads to 12 and the time limit to 3 minutes.

The results are shown in figure 5.

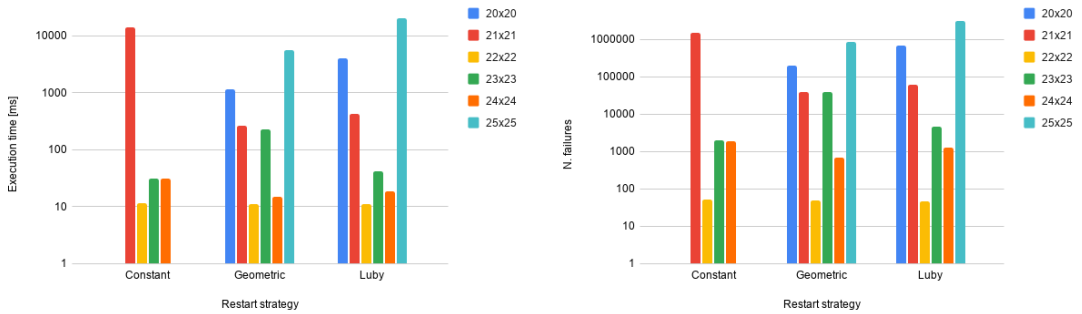

Figure 5: Execution time and number of failures, in logarithmic scale, for different restart strategy. The results were averaged over 5 runs: the absence of a bar indicates that the model was not able to provide a solution in the given time.

As it can be observed, the constant strategy failed both in the 20×20 and in the 25×25 instances. The geometric and Luby strategies showed comparable performances, with the former being slightly better on the 20×20 and 25×25 instances. Since such instances are, on average, the most difficult ones, we chose to use the geometric restart strategy in the final experiment.

4.4 Final result

In this final experiment, we tested the final model with the symmetry breaking constraint, together with the `dom_w_deg/indomain_min` search heuristic and the geometric restart strategy on all the instances with 12 threads and a time limit of 5 minutes.

Final results		
Instances	Execution time	Failures
8×8	4.9 ms	0
9×9	32.7 ms	11
10×10	10.6 ms	15
11×11	11.3 ms	129
12×12	11.9 ms	28
13×13	14.2 ms	323
14×14	4.9 ms	0
15×15	9.6 ms	51
16×16	12.4 ms	107
17×17	10.7 ms	111
18×18	272.3 s	53.8M
19×19	11.1 ms	32
20×20	694.2 ms	116.0K
21×21	674.3 ms	102.0K
22×22	10.6 ms	48
23×23	17.3 ms	1075
24×24	28.7 ms	1842
25×25	5.2 s	817.0K
26×26	—	50.7M
27×27	54.4 s	8.1M
28×28	51.7 s	8.9M
29×29	—	46.4M
30×30	12.5 ms	208
31×31	14.2 ms	706
32×32	—	48.3M
33×33	155.0 ms	19.9K
34×34	21.2 ms	255
35×35	12.3 ms	130
36×36	10.8 ms	64
37×37	65.3 s	1.1M
38×38	15.4 ms	633
39×39	—	5.8M
40×40	12.8 ms	51

Table 2: Execution time and number of failures of the final model with symmetry breaking, `dom_w_deg/indomain_min` heuristic and geometric restart strategy.

As it can be seen in the table 2, the model managed to solve all instances except four (26×26, 29×29, 32×32 and 39×39, in all of which the number of failures was around 50 millions). As hinted before, he most difficult instance it was able to solve was the 18×18, which required almost 5 minutes and slightly more than 50 million failures. On the other hand, the easiest instances were the 8×8 and the 14×14, which were solved in ~ 5 ms and with 0 failures.

5 A more general case

In this section we address a more general formulation of the present wrapping problem, which allows rotation and identical pieces. Both the extensions are developed starting from the final model with symmetry breaking.

5.1 Rotation

To take into account the rotation of some pieces, we had to include an array of boolean variables R. Each element of R specifies the rotation of the corresponding piece of paper: in particular, R_i is true iff the i-th piece of paper is rotated with respect to the original shape in the input file.

Moreover, we define two additional arrays $TRUE\_DX$ and $TRUE\_DY$ that contains the horizontal and vertical dimensions according to the actual rotation specified in array R (the rotation is relevant only if the piece is not a square).

```
array[1..n] of var bool: R;
array[1..n] of var int: TRUE_DX = [if (DX[i] != DY[i] /\ R[i]) then DY[i]
↪  else DX[i] endif | i in 1..n];
array[1..n] of var int: TRUE_DY = [if (DX[i] != DY[i] /\ R[i]) then DX[i]
↪  else DY[i] endif | i in 1..n];
```

For example, if the i-th piece has an original shape of $(DX_i = n, DY_i = m)$ with $DX_i \neq DY_i$, and R_i is true, then the actual shape will be $(TRUE\_DX = m, TRUE\_DY = n)$.

This formulation allows us to keep unchanged all the original constraints, only replacing DX with $TRUE\_DX$ and DY with $TRUE\_DY$.

5.2 Identical pieces

Having multiple pieces with the same dimensions leads to an unnecessary waste of resources: in fact, the solver will try to evaluate equivalent unfeasible configurations multiple times, by simply swapping the identical pieces.

To solve this problem, we imposed an ordering between the identical pieces, namely if two pieces i, j have the same dimensions, then j must be positioned further to the right and upper with respect to i:

```
constraint forall(i, j in 1..n where i < j) (
  (DX[i] = DX[j] /\ DY[i] = DY[j]) ->
  (X[j] >= X[i] + DX[i] /\ Y[j] >= Y[i]) \/ (Y[j] >= Y[i] + DY[i] /\ X[j]
  ↪  >= X[i])
);
```

In our tests, finding all the solutions of an instance with multiple identical pieces while using the above constraint resulted in a $\sim$ 3x speed-up with respect to the standard model.

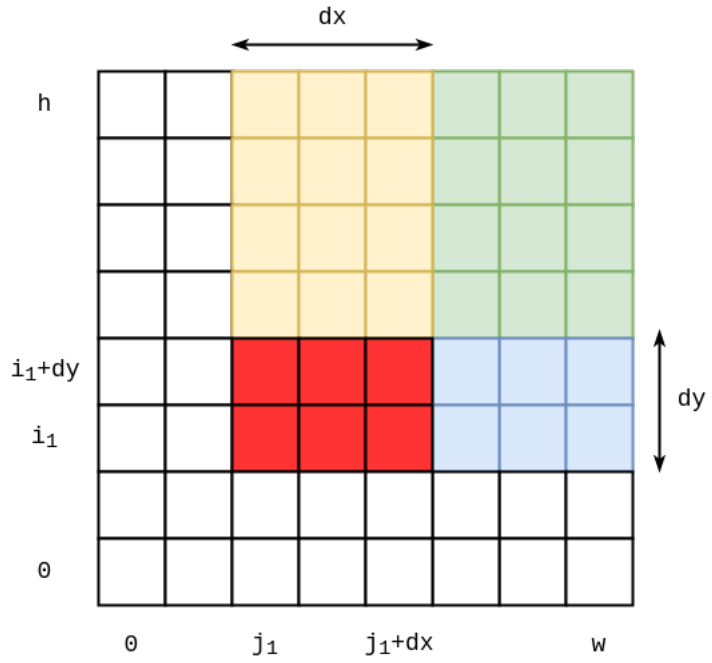

Figure 6: An example of the ordering between two identical pieces p_1 (in red) and a hypothetical, not shown p_2. p_1 piece position determines that p_2 must be on the right (blue region) or above (yellow region). If both conditions hold then p_2 will end up in the green region.