Present Wrapping Problem: SMT implementation

Lorenzo Mario Amorosa
Mattia Orlandi
Giacomo Pinardi

Master in Artificial Intelligence - University of Bologna

A.Y. 2020-2021

1 Model definition

1.1 Data representation

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

```
9   12
5
3   3
2   4
2   8
3   9
4   12
```

where:

- the first line contains w and h, which are the wrapping paper width and height, respectively;
- the second line contains n, the number of pieces of paper to cut off;
- the following lines contains the horizontal and vertical sizes of each piece of paper.

The SMT model is encoded using a matrix $\mathbf{XY} \in \mathbb{N}^{n \times 2}$: it contains, for each present, the (x, y) coordinates of its bottom-left corner; namely, given a present k, its coordinates will be $x = \mathbf{XY}_{k,1}$ and $y = \mathbf{XY}_{k,2}$.

Such array is implemented in `Z3py` as follows:

```
XY = [(Int(f'XY_{i}_0'), Int(f'XY_{i}_1')) for i in range(n)]
```

1.2 Main constraints

First of all, 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.

Such constraint is implemented in 1:

Constraint 1: No overlap

```
for (i, j) in combinations(range(n), 2):
    solver.add(Or(XY[i][0] + DX[i] <= XY[j][0],
                  XY[j][0] + DX[j] <= XY[i][0],
                  XY[i][1] + DY[i] <= XY[j][1],
                  XY[j][1] + DY[j] <= XY[i][1]))
```

where `combinations(range(n), 2)` is a utility function from the `itertools` library that generates all the possible pairs from the iterable `range(n)`.

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.

In addition, 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

```
# Boundaries consistency constraint
for i in range(n):
    solver.add(XY[i][0] >= 0)
    solver.add(XY[i][1] >= 0)
    solver.add(XY[i][0] + DX[i] <= w)
    solver.add(XY[i][1] + DY[i] <= h)
```

2 Implied constraints

As requested, we added the implied constraints to our model: if we draw a vertical line, and sum the vertical size of the traversed pieces, the sum can be at most h; an equivalent constraint must hold for the horizontal dimension.

As seen in the CP implementation, we can achieve this using the *cumulative* constraint. Given a list of k tasks with start time S_i, duration D_i and resource requirement R_i, and a total resource capacity C, then the *cumulative* constraint holds iff

$$\left(\sum_{i|S_i \leq u < S_i + D_i}^{k} R_i \right) \leq C$$

for each $u \in D$.

Such constraint is implemented in the following function:

```
def cumulative(solver, S: Sequence, D: Sequence, R: Sequence, C: int):
    # Iterate over the durations
    for u in D:
        solver.add(
            Sum(
                [If(And(S[i] <= u, u < S[i] + D[i]), R[i], 0)
                    for i in range(n)]
            ) <= C)
```

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 impose such constraint on our model as follows:

```
cumulative(solver,
           S=list(map(lambda t: t[0], XY)),  # take x coordinates
           D=DX,
           R=DY,
           C=h)
cumulative(solver,
           S=list(map(lambda t: t[1], XY)),  # take y coordinates
           D=DY,
           R=DX,
           C=w)
```

We performed a benchmark using an Intel i7-10750H CPU with 12 cores (6 physical and 6 logical) between the model without implied constraints and the model with implied constraints on all instances from 20×20 to 25×25, with a timeout of 5 minutes. The results are shown in table 1.

Implied constraints benchmark		
Instances	No implied constraints	Implied constraints
20×20	0.2	0.9
21×21	0.1	0.7
22×22	0.4	1.9
23×23	10.7	145.2
24×24	4.3	12.8
25×25	13.9	1.7

Table 1: Execution time (in seconds) of the model with and without implied constraints. Results are averaged over 5 runs.

As it can be seen, there are some instances, such as 25×25, where the implied constraints reduce the execution time, but in general we noticed that the version without implied constraints is faster.

For the final experiments on all the instances, we decided not to employ the implied constraints, and we set a timeout of 1 hour. The results are shown in table 2.

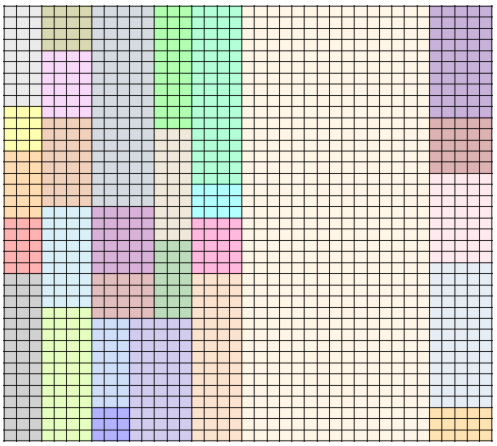

Figure 1: Solution to the 39×39 instance.

We notice that the program is able to solve all the instances within 2 minutes, except for the 39×39, which requires about 34 minutes.

Final results	
Instances	Execution time
8×8	8.0 ms
9×9	12.0 ms
10×10	11.0 ms
11×11	12.0 ms
12×12	29.0 ms
13×13	28.0 ms
14×14	28.0 ms
15×15	17.0 ms
16×16	45.0 ms
17×17	41.0 ms
18×18	951.0 ms
19×19	352.0 ms
20×20	290.0 ms
21×21	173.0 ms
22×22	490.0 ms
23×23	11.4 s
24×24	7.0 s
25×25	18.3 s
26×26	27.0 s
27×27	32.5 s
28×28	26.4 s
29×29	25.1 s
30×30	16.8 s
31×31	1.1 s
32×32	33.4 s
33×33	25.8 s
34×34	26.9 s
35×35	20.6 s
36×36	13.4 s
37×37	112.8 s
38×38	955.0 ms
39×39	34 min
40×40	801.0 ms

Table 2: Execution time of the model without implied constraints.

3 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 model explained so far.

3.1 Rotation

Together with the matrix $\mathbf{XY}$, we defined a support array R of length n, which encodes the information about the pieces' rotation: namely, each element R_k is asserted iff the k-th piece is rotated.

```
R = [Bool(f'R_{i}') for i in range(n)]
```

Moreover, we defined 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).

```
TRUE_DX = [If(And(DX[i] != DY[i], R[i]), DY[i], DX[i]) for i in range(n)]
TRUE_DY = [If(And(DX[i] != DY[i], R[i]), DX[i], DY[i]) for i in range(n)]
```

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

3.2 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; such check is performed by the following helper function:

```
def is_valid(x_i, y_i, x_j, y_j, dx, dy):
    right = And(x_j >= x_i + dx, y_j >= y_i)
    up = And(y_j >= y_i + dy, x_j >= x_i)
    return Or(right, up)
```

To satisfy the above constraint, the following must hold:

$$(DX_i = DX_j \wedge DY_i = DY_j) \rightarrow \text{is\_valid}(X_i, Y_i, X_j, Y_j, DX_i, DY_i) \;\; \forall i, j \leq n, i \neq j$$

In Z3py:

```
for (i, j) in combinations(range(n), 2):
    solver.add(Or(Not(And(DX[i] == DX[j], DY[i] == DY[j])),
    ↪  is_valid(XY[i][0], XY[i][1], XY[j][0], XY[j][1], DX[i], DY[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.

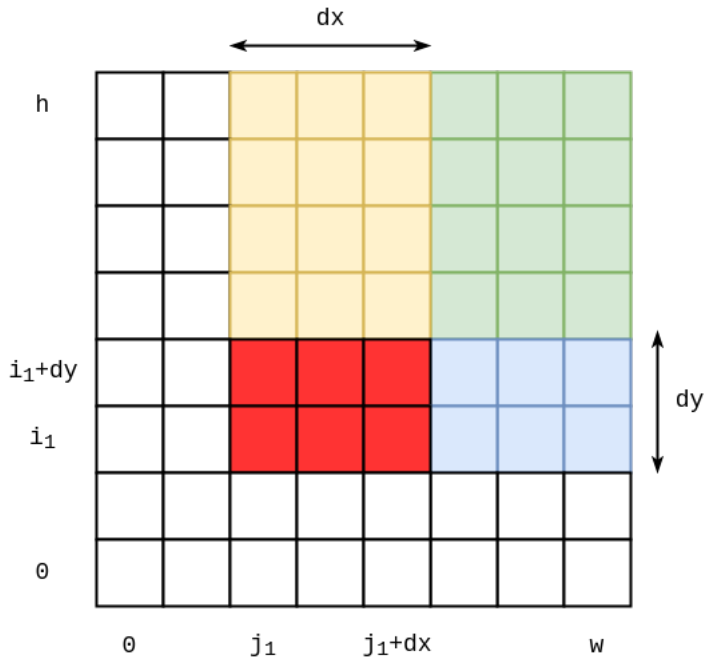

Figure 2: An example of the position filtering performed with identical pieces p_1 (in red) and a hypothetical, not shown p_2. p_1 piece position determines that p_2 must be on the right (blue region) or above (yellow region). If both conditions hold then p_2 will end up in the green region.