Present Wrapping Problem: SAT 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.

In SAT, we can only deal with boolean variables; therefore, we defined our model as a 3D boolean array $\mathbf{B} \in \{0, 1\}^{h \times w \times n}$ where the first and second dimensions (i, j) represent the cells' coordinates in the grid, while the third dimension k is related to the piece of paper.

Such array is implemented in `Z3py` as follows:

```python
B = [[[Bool(f'B_{i}_{j}_{k}') for k in range(n)] for j in range(w)] for i in range(h)]
```

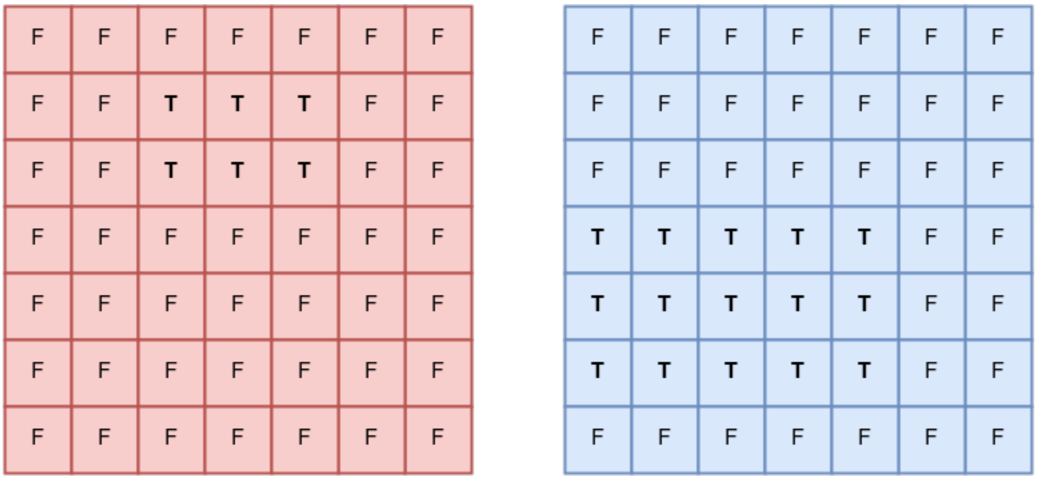

Figure 1: An example of two vertical slices of the 3D boolean array B. On the left, in the red channel, it is visible the position occupied by a 3×2 piece, while in the blue channel there is a 5×3 piece.

Then, we initialized the SAT solver which will contain all the clauses:

```python
solver = Solver()
```

1.2 Main constrains

Firstly, we defined two helper functions to encode the *at least one* and *at most one* constraints that are needed to solve the instances:

```python
def at_least_one(bool_vars: Sequence):
    return Or(bool_vars)

def at_most_one(bool_vars: Sequence):
    return [Not(And(pair[0], pair[1])) for pair in combinations(bool_vars, 2)]
```

where `combinations(bool_vars, 2)` is a utility function from the `itertools` library that generates all the possible pairs from the list `bool_vars`. It is possible to define also an *exactly one* constraint by simply combining the two ones above with the $\wedge$ operator.

Then, we enforced each grid cell to contain at most one piece of paper by means of the following constraint:

Constraint 1: At most one piece of paper

```
for i in range(h):
    for j in range(w):
        solver.add(at_most_one(B[i][j]))
```

Namely, for each column i and row j, $\mathbf{B}_{i,j}$ is an array containing n elements representing the pieces of paper; since only one piece of paper can occupy a cell, we add to the solver an *at most one* constraint:

$$\bigwedge_{i=1}^{h} \bigwedge_{j=1}^{w} \bigwedge_{1 \leq k < k' \leq n} \neg \left(\mathbf{B}_{i,j,k} \wedge \mathbf{B}_{i,j,k'} \right)$$

As a side note, this constraint can work also with instances in which the pieces of paper do not fully cover the whole grid.

Finally, the code in 2 performs the following operations:

1. given the piece of paper `p` with dimensions `dx` and `dy`, we initialize the `package_clauses` variable as an empty list;
2. we iterate over `i` and `j`, representing all the possible coordinates where `p` can fit on the grid;
3. using `f1` and `f2`, we store into `patch_clauses` the cells composing `p`'s patch;
4. then, we impose the constraint "adjacent cells should belong to the same piece of paper" by reducing the `patch_clauses`'s list with the $\wedge$ operator;
5. the clause from the previous step is appended to `package_clauses`;
6. finally, after all these iterations, `package_clauses` will represent all the possible positions of `p` on the grid;
7. since only one position is allowed, we add the *exactly one* constraint to the solver.

Constraint 2: Convolutional-like sliding

```
# Iterate over all the pieces p
for p in range(n):
    dx = DX[p]
    dy = DY[p]

    package_clauses = []
    # Iterate over all the coordinates where p can fit
    for i in range(h - dy + 1):
```

```
        for j in range(w - dx + 1):

            patch_clauses = []
            # Iterate over the cells of p's patch
            for f1 in range(dy):
                for f2 in range(dx):
                    patch_clauses.append(B[i + f1][j + f2][p])

            package_clauses.append(And(patch_clauses))
    # Exactly one
    solver.add(at_least_one(package_clauses))
    solver.add(at_most_one(package_clauses))
```

This implementation is robust even when the pieces of paper do not fully cover the whole grid; however, since the *at most one* constraint is computationally expensive, and since the former situation does not happen in any of the provided instances, it is sufficient to enforce the *at least one* constraint, which is way faster.

We can notice that the above implementation resembles a convolution operation where the presents represent the kernel, sliding on the paper roll.

2 Implied constraints

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

To implement such constraints in SAT, we have reasoned as follows:

- for each row, we have to count how many cells are occupied by a piece of paper;
- then, we constrain such sum to be less than or equal to w;
- whether a cell is occupied or not can be encoded with an *at least one* constraint along the package dimension;
- the constraint about the sum of non-empty cells can be translated to an *at most k* constraint, where $k = w$.

While the implementation of the *at least one* constraint is trivial, an efficient implementation of the *at most k* constraint is challenging: to develop it, we took inspiration from Sinz's paper[1].

Such constraint is based on the concept of Sequential Counter: given a list $x_1, ..., x_n$ of boolean variables on which we want to apply the *at most k* constraint,

[1] cf. "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints"

we create $k \cdot (n-1)$ auxiliary variables which represent the partial sums, such that $s_i = \sum_{l=1}^{i} x_l$. In particular, s_i is encoded as a unary counter with k bits: for instance, with $k = 3$, s_i is composed of 3 bits, the value 1 is encoded as 100, 2 as 110, 3 as 111; if the constraint is not satisfied (i.e. there are more than 3 true variables), the counter will overflow.

As shown in the mentioned paper, the Sequential Counter can be encoded with the following constraints:

$$
\begin{array}{ll}
\text{(SC1)} & x_1 \rightarrow s_{1,1} \\
\text{(SC2)} & \left.\neg s_{1,j}\right\} \text{ for } 1 < j \le k \\
\text{(SC3)} & x_i \rightarrow s_{i,1} \\
\text{(SC4)} & s_{i-1,1} \rightarrow s_{i,1} \\
\text{(SC5)} & (x_i \wedge s_{i-1,j-1}) \rightarrow s_{i,j} \\
\text{(SC6)} & s_{i-1,j} \rightarrow s_{i,j} \\
\text{(SC7)} & x_i \rightarrow \neg s_{i-1,k} \\
\text{(SC8)} & x_n \rightarrow \neg s_{n-1,k}
\end{array}
$$

$$
\left.
\begin{array}{ll}
\text{(SC3)} & x_i \rightarrow s_{i,1} \\
\text{(SC4)} & s_{i-1,1} \rightarrow s_{i,1} \\
\left.\begin{array}{ll}
\text{(SC5)} & (x_i \wedge s_{i-1,j-1}) \rightarrow s_{i,j} \\
\text{(SC6)} & s_{i-1,j} \rightarrow s_{i,j}
\end{array}\right\} & \text{for } 1 < j \le k \\
\text{(SC7)} & x_i \rightarrow \neg s_{i-1,k}
\end{array}
\right\} \text{ for } 1 < i < n
$$

In our case, we have $n = k = w$, since for each row there are in total w cells, and we want at most w cells asserted; therefore, considering only a single row r, there will be w boolean variables $x_1, ..., x_w$, one for each column. In particular, such variables are actually represented by the *at least one* constraint applied on the cell (r, i) along the package dimension (i.e. for $1 \le p \le n$):

$$x_i = \bigvee_{p=1}^{n} \mathbf{B}_{r,i,p}$$

Finally, since there are h rows, we will have h sets of variables $x_1, ..., x_w$ and constraints as the ones defined above.

The implementation of these constraints in `Z3py` is the following:

Constraint 3: Implied (for rows)

```python
for r in range(h):
    solver.add(Or(Not(at_least_one(B[r][0])), Sr[0][0]))  # SC1
    for j in range(1, w):
        solver.add(Not(Sr[0][j]))  # SC2
    for i in range(1, w - 1):
        solver.add(Or(Not(at_least_one(B[r][i])), Sr[i][0]))  # SC3
        solver.add(Or(Not(Sr[i - 1][0]), Sr[i][0]))  # SC4
        for j in range(1, w):
```

```
            solver.add(Or(Not(at_least_one(B[r][i])), Not(Sr[i - 1][j - 1]), Sr[i][j]))  # SC5
            solver.add(Or(Not(Sr[i - 1][j]), Sr[i][j]))  # SC6
        solver.add(Or(Not(at_least_one(B[r][i])), Not(Sr[i - 1][w - 1])))  # SC7
    solver.add(Or(Not(at_least_one(B[r][w - 1])), Not(Sr[w - 2][w - 1])))  # SC8
```

All the previous considerations can be made also for the vertical dimension.

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	62.9	3.4
21×21	61.9	89.3
22×22	20.1	5.5
23×23	—	—
24×24	—	—
25×25	162.1	• 297.1

Table 1: Execution time (in seconds) of the model with and without implied constraints. Results are averaged over 5 runs: the • indicates that one or more runs did not produce a solution.

Table 1 shows that the implied constraints make the search faster only for some instances (20×20 and 22×22). On the other hand, it can be noticed that for the instances 21×21 and 25×25 the version without implied constraints is faster.

For the final experiments on all the instances, we decided to employ the implied constraints with a timeout of 1 hour. The results are shown in table 2: it can be seen that from the instance 26×26 onwards the complexity is so high that the timeout is reached for the majority of the instances.

Final results	
Instances	Execution time
8×8	18.0 ms
9×9	18.0 ms
10×10	30.0 ms
11×11	73.0 ms
12×12	91.0 ms
13×13	190.0 ms
14×14	1.2 s
15×15	238.0 ms
16×16	583.0 ms
17×17	841.0 ms
18×18	627.4 s
19×19	206.1 s
20×20	4.4 s
21×21	91.6 s
22×22	5.4 s
23×23	28 min
24×24	20 min
25×25	298.4 s
26×26	—
27×27	666.7 s
28×28	36 min
29×29	—
30×30	—
31×31	539.2 s
32×32	—
33×33	—
34×34	—
35×35	—
36×36	—
37×37	—
38×38	208.3 s
39×39	—
40×40	—

Table 2: Execution time of the model with implied constraint.

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 $\mathbf{B}$, we defined a support array R with 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_{k}') for k in range(n)]
```

The constraint defined in 2 had to be modified in order to take into account the possible rotation of each piece; therefore, we duplicated it such that the first one deals with not rotated pieces while the second one deals with pieces that are rotated.

Constraint 4: Convolutional-like sliding with rotation

```
# Iterate over all the pieces p
for p in range(n):
    # --- Not rotated pieces ---
    dx = DX[p]
    dy = DY[p]

    package_clauses = []
    # Iterate over all the coordinates where p can fit
    for i in range(h - dy + 1):
        for j in range(w - dx + 1):

            patch_clauses = []
            # Iterate over the cells of p's patch
            for f1 in range(dy):
                for f2 in range(dx):
                    patch_clauses.append(B[i + f1][j + f2][p])

            package_clauses.append(And(patch_clauses))

    # Not(R[p]) -> at_least_one(package_clauses)
    solver.add(Or(R[p], at_least_one(package_clauses)))


```

```
24      # --- Rotated pieces ---
25      dx, dy = dy, dx  # swap dimensions
26
27      package_clauses = []
28      # Iterate over all the coordinates where p can fit
29      for i in range(h - dy + 1):
30          for j in range(w - dx + 1):
31
32              patch_clauses = []
33              # Iterate over the cells of p's patch
34              for f1 in range(dy):
35                  for f2 in range(dx):
36                      patch_clauses.append(B[i + f1][j + f2][p])
37
38              package_clauses.append(And(patch_clauses))
39
40      # R[p] -> at_least_one(package_clauses)
41      solver.add(Or(Not(R[p]), at_least_one(package_clauses)))
```

The main difference with respect to the constraint 2 is that a piece can have two states, "rotated" or "not rotated": if it is not rotated (R_p is false), then the model must assert the *at least one* constraint between all the possible positions in which p has its original orientation; on the other hand, if it is rotated (R_p is true), then the model must assert the *at least one* constraint between all the possible positions in which p has its dimensions swapped (cf. line 25 of constraint 4).

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 p_1, p_2 have the same dimensions, then p_2 must be positioned further to the right and upper with respect to p_1.

To do so, we first have to group together the identical pieces in the `indexes` dictionary, which associates to each pair of dimensions the list of pieces' indexes sharing such dimensions:

```
indexes = {}
for idx, dim in enumerate(zip(DX, DY)):
    if dim in indexes:
        indexes[dim] += [idx]
```

```
    else:
        indexes[dim] = [idx]
```

Then, we define a utility function to filter the valid positions of p_2, given p_1:

```
def is_valid(i1, j1, i2, j2, dx, dy):
    right = (j2 >= j1 + dx) and (i2 <= i1)
    up = (i2 <= i1 - dy) and (j2 >=j1)
    return right or up
```

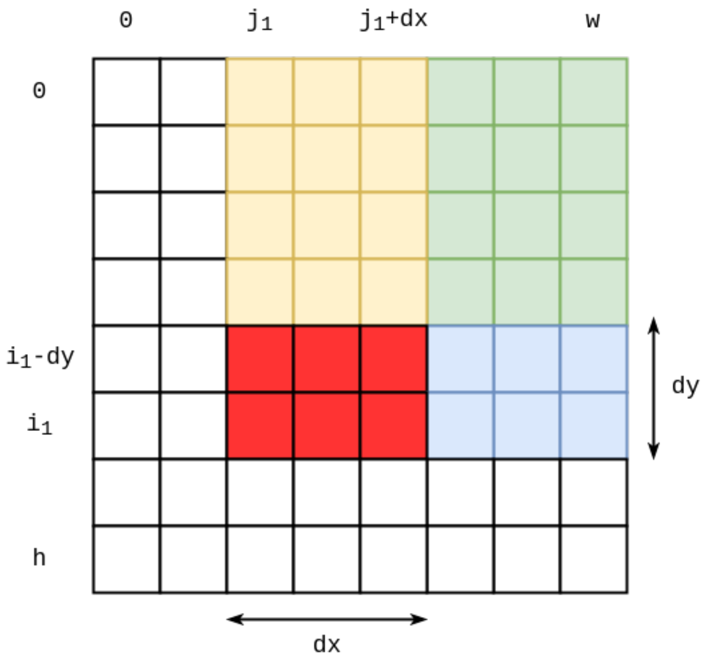

Figure 2: An example of the position filtering performed with identical pieces p_1 (in red) and a hypothetical, not shown p_2. p_1 piece position determines that p_2 must be on the right (blue region) or above (yellow region). If both conditions hold then p_2 will end up in the green region.

Finally, in 5 we modify the convolutional-like sliding (defined originally in 2) in order to take into account identical pieces:

- we no longer iterate over the single pieces but on the pieces' (unique) dimensions, thanks to the `indexes` dictionary defined above;
- if the list of pieces associated to the given dimensions contains only one piece (i.e. meaning that such piece is unique), then we apply the standard algorithm defined in 2;

- otherwise, we iterate over each piece p_i, comparing it with its successor p_{i+1};
- given $p_1 = p_i$ and $p_2 = p_{i+1}$, we obtain the lists representing all the possible positions of p_1 and all the possible positions of p_2;
- such lists are independent from each other, thus, for each p_1 position, we filter the p_2 positions in order to keep only the valid ones (using the `is_valid()` function defined above);
- then, we construct a clause as a $\wedge$ between the current p_1 position and *at least one* of the current valid p_2 positions, appending the result to `package_clauses_joint`;
- finally, we add to the solver the *at least one* constraint between all the clauses in `package_clauses_joint`.

Constraint 5: Convolutional-like sliding with identical pieces

```python
# Iterate over all the dimensions
for (dx, dy), p_list in indexes.items():
    if len(p_list) > 1:  # case 1: multiple identical pieces
        # Iterate over identical pieces in pairs
        for i in range(len(p_list) - 1):
            p1 = p_list[i]
            p2 = p_list[i + 1]

            package_clauses_p1 = {}
            package_clauses_p2 = {}
            package_clauses_joint = []

            # Iterate over all the coordinates where p can fit
            for i in range(h - dy + 1):
                for j in range(w - dx + 1):

                    patch_clauses_p1 = []
                    patch_clauses_p2 = []
                    # Iterate over the cells of p's patch
                    for f1 in range(dy):
                        for f2 in range(dx):
                            patch_clauses_p1.append(B[i + f1][j +
                            ↪  f2][p1])
                            patch_clauses_p2.append(B[i + f1][j +
                            ↪  f2][p2])

                    # (i + dy - 1, j) bottom-left corner
                    package_clauses_p1[(i + dy - 1, j)] =
                    ↪  And(patch_clauses_p1)
```

```python
                    package_clauses_p2[(i + dy - 1, j)] = And(patch_clauses_p2)

            # Filter valid p2 clauses
            for (i1, j1), patch_p1 in package_clauses_p1.items():
                # Condition for validity: i2 <= i1 and j2 >= j1
                valid_patches_p2 = [patch_p2 for (i2, j2), patch_p2 in package_clauses_p2.items() if is_valid(i1, j1, i2, j2, dx, dy)]

                package_clauses_joint.append(And(patch_p1, at_least_one(valid_patches_p2)))

            solver.add(at_least_one(package_clauses_joint))
    else:  # case 2: unique piece
        p = p_list[0]
        package_clauses = []

        # Iterate over all the coordinates where p can fit
        for i in range(h - dy + 1):
            for j in range(w - dx + 1):

                patch_clauses = []
                # Iterate over the cells of p's patch
                for f1 in range(dy):
                    for f2 in range(dx):
                        patch_clauses.append(B[i + f1][j + f2][p])

                package_clauses.append(And(patch_clauses))

        solver.add(at_least_one(package_clauses))
```

In our tests, we have found that using the above constraint on an instance with multiple identical pieces leads to a variable speed-up: in some instances we obtained up to a $\sim$ 10x speed-up, whereas in others the performance was comparable with respect to the standard model.