Present Wrapping Problem

Satisfiability Modulo Theories (SMT)

Pierpasquale Colagrande

October, 2020

1 Introduction

Given a wrapping paper roll of a certain width and height and a list of pieces to cut, with their relative widths and heights, the Present Wrapping Problem (PWP), also know as Two-dimensional Orthogonal Packing Problem (2OPP), consists in finding a way to cut all the pieces from the paper roll, without excluding none.

The aim of this model is to solve the Present Wrapping Problem using Satisfiability Modulo Theories with Z3 Python API. The problem was solved by building a Z3 encoding, using Python language, that was then executed by a master Python program.

1.1 Specifications

The problem was modeled using input instances, two models and output files. All of these componenst are managed by a CLI-based Python program that is shared between all the solving methods, which allows the user to choose the instance(s) to solve, the solver (CP or SMT) and the model (general or standard) to use. The program saves the solution on a specific output file and then it also allows to plot the solution visually.

1.1.1 Input

An input instance is a .txt file where:

- the first line contains the **width** and **height** of the paper roll
- the second line contains the number **n** of the pieces to cut
- the following n lines contain the **widths** and **heights** of the n pieces to cut

1.1.2 Encodings

There are two encodings:

- **standard**: basic encoding that allows pieces to be cutted from the paper roll without the possibility to rotate them and without any optimization for pieces of the same size
- **general**: improved encoding that allows pieces to be cutted from the paper roll also by rotating them and also optimizing pieces of the same size

1.1.3 Output

The solution is saved to an output .txt file where:

- the first line contains the **width** and **height** of the paper roll
- the second line contains the number **n** of the pieces to cut
- the following n lines contain the **widths** and **heights**, along with the **x and y coordinates** of the bottom-left corners of the n pieces to cut; for the general model (the model with rotation), these n lines also contain a boolean string representing the rotation of each piece

The solution is, in fact, represented by the coordinate on the paper roll of the bottom-left corner of the piece to cut.

2 Satisfiability Modulo Theories

This section will cover and explain the parameters, variables and expressions used in the two Z3 encodings, along with the modifications done to also implement the features of the general encoding.

2.1 Parameters

Several useful parameters have been defined:

- two integers x and y representing **the x and y axes** of the paper roll
- an integer value *roll_width* representing the **paper roll width**
- an integer value *roll_height* representing the **paper roll height**
- a set of integer values *X_COORDINATES* representing the coordinates along the x axis (the columns of the paper roll)
- a set of integer values *Y_COORDINATES* representing the coordinates along the y axis (the rows of the paper roll)
- an integer value *n_pieces* representing the **number of pieces** to cut

- a set of integer values *PIECES* representing the *n_pieces* **pieces of paper** to cut
- a matrix of integer values *pieces_dimensions* with as many rows as *PIECES* and two columns representing the **dimensions of each piece of paper** over the two axes x and y defined above
- an array of integer values *lower_bounds* with two elements representing the **lower bounds** of the possible coordinates over the two axes x and y defined above
- an array of integer values *upper_bounds* with two elements representing the **upper bounds** of the possible coordinates over the two axes x and y defined above

It is important to specify that some of these parameters were only defined to make the mode more readable and understandable.

2.2 Variables

The problem has been encoded using a matrix of integer decision variables *pieces_corners* which represents the coordinates of the bottom-left corner of all the pieces of paper over the two axes x and y. **Figure 1** represents the encoding of the bottom-left coordinates of the pieces and their relative visualization on the paper roll.

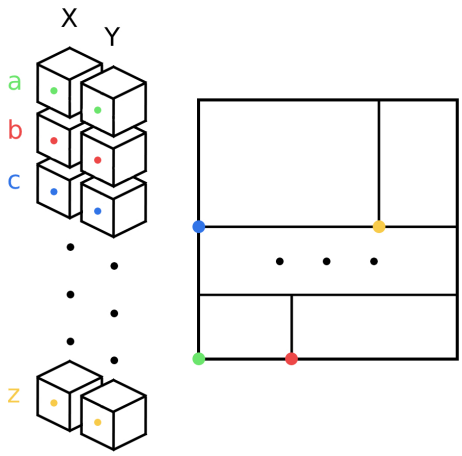

Figure 1: Decision variables modeling and their visual representation on the roll

The rows of the matrix are the *PIECES* while the columns are the x and y axes. The first column represents all the **x** coordinates of the bottom-left corner of the pieces of paper, while the second column represents the **y** coordinates. The matrix has as many rows as the number of pieces of paper to cut.

2.3 Main constraints

Before defining the main constraint expressions, it is needed to do some observations:

1. the pieces must be cutted **inside** the limits of the paper roll, so the position of the bottom-left corner must stay inside the coordinates described by the paper roll limits, which have been defined as *lower_bounds* and *upper_bounds* parameters; this property is defined by the following constraint:

$$\forall i \in PIECES$$

$$pieces\_corners[i,\ x] \geq lower\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ x] \leq upper\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \geq lower\_bounds[y]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \leq upper\_bounds[y]$$

2. it is possible to calculate the coordinates occupied by a piece of paper by adding to the y coordinate of the bottom-left corner its height and by adding to the x coordinate its width; these coordinates **must not exceed** the paper roll limits; this property is defined by the following constraint:

$$\forall i \in PIECES$$

$$pieces\_corners[i,\ x] + pieces\_dimensions[i,\ x] \leq upper\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ y] + pieces\_dimensions[i,\ y] \leq upper\_bounds[y]$$

3. the pieces of paper must also **not overlap**, which means that the coordinates occupied by a piece of paper must not be occupied by other pieces of paper

There is, however, a better way to define the property 1. and 2. using only one constraint:

$$\forall i \in PIECES$$

$$pieces\_corners[i,\ x] \geq lower\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ x] \leq upper\_bounds[x] - pieces\_dimensions[i,\ x]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \geq lower\_bounds[y]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \leq upper\_bounds[y] - pieces\_dimensions[i,\ y]$$

This constraint represents the fact that each piece of paper must be cutted inside the boundaries defined by the paper roll dimensions. In this constraint, it is also needed to consider the piece of paper dimensions in order to consider also the fact that the coordinates occupied by the piece of paper must not exceed the limits of the paper roll. This constraint is written in Z3 using the expression:

```
[And(
  And(pieces_corners[i][x] >= lower_bounds[x],
  pieces_corners[i][x] <= upper_bounds[x] - pieces_dimensions[i][x]),
  And(pieces_corners[i][y] >= lower_bounds[y],
  pieces_corners[i][y] <= upper_bounds[y] - pieces_dimensions[i][y]))
for i in PIECES]
```

The third property can instead be defined as the following constraint:

$$\forall i, j \in PIECES,\ i < j$$

$$pieces\_corners[i,\ x] + pieces\_dimensions[i,\ x] \leq pieces\_corners[j,\ x]$$

$$\vee$$

$$pieces\_corners[i,\ y] + pieces\_dimensions[i,\ y] \leq pieces\_corners[j,\ y]$$

$$\vee$$

$$pieces\_corners[j,\ x] + pieces\_dimensions[j,\ x] \leq pieces\_corners[i,\ x]$$

$$\vee$$

$$pieces\_corners[j,\ y] + pieces\_dimensions[j,\ y] \leq pieces\_corners[i,\ y]$$

This constraint represent the fact that, for each pair of pieces of paper, the first must not overlap the coordinates of the second and viceversa.

For this constraint, it is encoded into the following Z3 expression:

```
[Or(
  pieces_corners[i][x] + pieces_dimensions[i][x] <=
                                                 pieces_corners[j][x],
  pieces_corners[i][y] + pieces_dimensions[i][y] <=
                                                 pieces_corners[j][y],
  pieces_corners[j][x] + pieces_dimensions[j][x] <=
                                                 pieces_corners[i][x],
  pieces_corners[j][y] + pieces_dimensions[j][y] <=
                                                 pieces_corners[i][y])
for i in PIECES for j in PIECES if i < j]
```

2.4 Implied cumulative constraints

If we draw a vertical line and sum the vertical sides of the traversed pieces, the sum can be **at most equal to the height** of the paper roll. The same property must hold if we draw an horizontal line, which means that the sum of traversed pieces can be **at most equal to the width** of the paper roll.

This property can be expressed with the following constraint:

$$\forall y\_coord \in Y\_COORDINATES$$

$$\sum_{i \in PIECES} width(i,\ y\_coord) \leq roll\_width$$

where

$$width(i,\ y\_coord) = \begin{cases} pieces\_dimensions[i,\ x] & \text{if } y\_coord \geq pieces\_corners[i,\ y] \wedge \\ & y\_coord < pieces\_corners[i,\ y] \\ & + pieces\_dimensions[i,\ y] \\ 0 & \text{otherwise} \end{cases}$$

$$\bigwedge$$

$$\forall x\_coord \in X\_COORDINATES$$

$$\sum_{i \in PIECES} height(i,\ x\_coord) \leq roll\_height$$

where

$$height(i,\ x\_coord) = \begin{cases} pieces\_dimensions[i,\ y] & \text{if } x\_coord \geq pieces\_corners[i,\ x] \wedge \\ & x\_coord < pieces\_corners[i,\ x] \\ & + pieces\_dimensions[i,\ x]) \\ 0 & \text{otherwise} \end{cases}$$

What this constraint is doing is checking, for each row of the paper roll, if the sum of the pieces widths on that row is at most the paper roll width. So, basically, for each row of the paper roll, it checks if each piece is placed on that row (by checking the coordinates occupied by each single piece). If a piece is on that row, the width of that piece is used for the sum, otherwise, its width is not used in the sum. This sum is calculated for each row of the paper roll and it must be at most the paper roll width for each row of the paper roll.

This operation is done also for the columns of the paper roll by checking if each piece is placed on that column and by summing their heights if they are, as described before. The sum over each column must be at most the paper roll height. These two conditions must be both satisfied.

This constraint is encoded into the following Z3 expression:

```
[Sum(
  [If(And(y_coord >= pieces_corners[i][y], y_coord < pieces_corners[i
                                    ][y] + pieces_dimensions[i][y])
                                    ,
    pieces_dimensions[i][x], 0) for i in PIECES]) <= roll_width
      for y_coord in Y_COORDINATES] + [Sum(
  [If(And(x_coord >= pieces_corners[i][x], x_coord < pieces_corners[i
                                    ][x] + pieces_dimensions[i][x])
                                    ,
    pieces_dimensions[i][y], 0) for i in PIECES]) <= roll_height
      for x_coord in X_COORDINATES]
```

However, after some testing, we noticed that it was necessary a modification. We replaced these constraint defined with MiniZinc's *cumulative* global constraint with the implied constraints described above with a little modification: the sum of the widths/heights of the pieces in each column/row of the paper roll must be **exactly equal** to the paper roll height/width, not at most equal. By doing this, we are removing the possibility to have a white space at the edges of the paper roll or between the pieces. This is a problem in case we have input instances where the solution can be found without using the full paper roll, meaning that we can have space between pieces or at the edges of the roll, but in this case none of our input instances presented this situation.

The two modified constraints are

$$\forall y\_coord \in Y\_COORDINATES$$

$$\sum_{i \in PIECES} width(i,\ y\_coord) = roll\_width$$

where

$$width(i,\ y\_coord) = \begin{cases} pieces\_dimensions[i,\ x] & \text{if } y\_coord \geq pieces\_corners[i,\ y] \wedge \\ & y\_coord < pieces\_corners[i,\ y] \\ & + pieces\_dimensions[i,\ y] \\ 0 & \text{otherwise} \end{cases}$$

$$\bigwedge$$

$$\forall x\_coord \in X\_COORDINATES$$

$$\sum_{i \in PIECES} height(i,\ x\_coord) = roll\_height$$

where

$$height(i,\ x\_coord) = \begin{cases} pieces\_dimensions[i,\ y] & \text{if } x\_coord \geq pieces\_corners[i,\ x] \wedge \\ & x\_coord < pieces\_corners[i,\ x] \\ & + pieces\_dimensions[i,\ x]) \\ 0 & \text{otherwise} \end{cases}$$

which in Z3 were described as

```
[Sum(
      [If(And(y_coord >= pieces_corners[i][y], y_coord <
                                          pieces_corners[i][y] +
                                          pieces_dimensions[i][y]),
          pieces_dimensions[i][x], 0) for i in PIECES]) == roll_width
                                              for y_coord in
                                            Y_COORDINATES] + [Sum(
      [If(And(x_coord >= pieces_corners[i][x], x_coord <
                                          pieces_corners[i][x] +
                                          pieces_dimensions[i][x]),
          pieces_dimensions[i][y], 0) for i in PIECES]) ==
                                              roll_height for x_coord
                                               in X_COORDINATES]
```

2.5 General model

This section will cover the necessary modifications to add the possibility to rotate pieces and to manage pieces of the same size.

2.5.1 Rotation

It must be considered that a piece could be rotated in 4 different ways:

- 0° rotation, where width and height are not swapped
- 90° rotation, where width and height are swapped
- 180° rotation, where width and height are not swapped
- 270° rotation, where width and height are swapped

Since the pieces of paper have regular shapes (they are rectangles or squares), the 0° and 180° rotations will result in the same shape with the same width and height, as well ass 90° and 270° rotations. It is so possible to just encode two cases:

- the piece is **not rotated**, so width and height are **not swapped**
- the piece is **rotated**, so width and height are **swapped**

This rotation property has been modeled using another array of boolean decision variables called *pieces_rotation*. The array has as many elements as *PIECES*. This array of decision variables describes if a present is rotated or not.

In order to encode this property, some modifications have been made to the standard model. In fact, if a piece is rotated, its width and height are swapped, so the previous constraints must be modified in order to use the correct widths and heights for all the pieces, whether they are rotated or not.

In particular, a function which returns the correct dimension of the specified piece along the specified axis, based on the piece rotation, has been defined:

$$get\_dimension(i,\ axis) = \begin{cases} pieces\_dimensions[i,\ y] & \text{if } axis = x \land pieces\_rotation[i] \\ pieces\_dimensions[i,\ x] & \text{if } axis = x \land \neg pieces\_rotation[i] \\ pieces\_dimensions[i,\ x] & \text{if } axis = y \land pieces\_rotation[i] \\ pieces\_dimensions[i,\ y] & \text{if } axis = y \land \neg pieces\_rotation[i] \end{cases}$$

This function, in Z3, is defined as a the following python function:

```
def get_dimension(i, axis):
  if axis == x:
    return If(pieces_rotation[i], pieces_dimensions[i][y],
                                  pieces_dimensions[i][x])
  else:
    return If(pieces_rotation[i], pieces_dimensions[i][x],
                                  pieces_dimensions[i][y])
```

Also, it is necessary to modify the previously defined constraints: in fact, it is necessary to use the previously defined function to get the correct dimensions for the piece when specified inside the constraints. The constraint defining the paper limits is modified to:

$$\forall i \in PIECES$$

$$pieces\_corners[i,\ x] \geq lower\_bounds[x]$$

$$\wedge$$

$$pieces\_corners[i,\ x] \leq upper\_bounds[x] - get\_dimension(i,\ x)$$

$$\wedge$$

$$pieces\_corners[i,\ y] \geq lower\_bounds[y]$$

$$\wedge$$

$$pieces\_corners[i,\ y] \leq upper\_bounds[y] - get\_dimension(i,\ y))$$

In Z3, this constraint is encoded as the following expression:

```
[And(And(pieces_corners[i][x] >= lower_bounds[x],
  pieces_corners[i][x] <= upper_bounds[x] - get_dimension(i, x)),
  And(pieces_corners[i][y] >= lower_bounds[y],
  pieces_corners[i][y] <= upper_bounds[y] - get_dimension(i, y)))
for i in PIECES]
```

Then it is also necessary to modify the non-overlap constraint:

$$\forall i, j \in PIECES,\ i < j$$

$$pieces\_corners[i,\ x] + get\_dimension(i,\ x) \leq pieces\_corners[j,\ x]$$

$$\vee$$

$$pieces\_corners[i,\ y] + get\_dimension(i,\ y) \leq pieces\_corners[j,\ y]$$

$$\vee$$

$$pieces\_corners[j,\ x] + get\_dimension(j,\ x) \leq pieces\_corners[i,\ x]$$

$$\vee$$

$$pieces\_corners[j,\ y] + get\_dimension(j,\ y) \leq pieces\_corners[i,\ y]$$

In Z3, this is encoded as:

```
[Or(
  pieces_corners[i][x] + get_dimension(i, x) <=
```

```
                                          pieces_corners[j][x],
    pieces_corners[i][y] + get_dimension(i, y) <=
                                          pieces_corners[j][y],
    pieces_corners[j][x] + get_dimension(j, x) <=
                                          pieces_corners[i][x],
    pieces_corners[j][y] + get_dimension(j, y) <=
                                          pieces_corners[i][y])
  for i in PIECES for j in PIECES if i < j]
```

The implied cumulative constraints are modified:

$$\forall y\_coord \in Y\_COORDINATES$$

$$\sum_{i \in PIECES} width(i,\ y\_coord) = roll\_width$$

where

$$width(i,\ y\_coord) = \begin{cases} get\_dimension(i,\ x) & \text{if } y\_coord \geq pieces\_corners[i,\ y] \wedge \\ & y\_coord < pieces\_corners[i,\ y] \\ & + get\_dimension(i,\ y) \\ 0 & \text{otherwise} \end{cases}$$

$$\bigwedge$$

$$\forall x\_coord \in X\_COORDINATES$$

$$\sum_{i \in PIECES} height(i,\ x\_coord) = roll\_height$$

where

$$height(i,\ x\_coord) = \begin{cases} get\_dimension(i,\ y) & \text{if } x\_coord \geq pieces\_corners[i,\ x] \wedge \\ & x\_coord < pieces\_corners[i,\ x] \\ & + get\_dimension(i,\ x)) \\ 0 & \text{otherwise} \end{cases}$$

In Z3, the constraint is encoded as:

```
[Sum(
  [If(And(y_coord >= pieces_corners[i][y], y_coord < pieces_corners[i][
                                      y] + get_dimension(i, y)),
```

```
    get_dimension(i, x), 0) for i in PIECES]) == roll_width for
                                          y_coord in Y_COORDINATES]
    + [Sum(
 [If(And(x_coord >= pieces_corners[i][x], x_coord < pieces_corners[i][
                                      x] + get_dimension(i, x)),
    get_dimension(i, y), 0) for i in PIECES]) == roll_height for
                                          x_coord in X_COORDINATES]
```

2.5.2 Pieces with same dimension

If an input instance has multiple pieces of the same dimension, there is no need to consider the solutions where the positions of these pieces are swapped. In fact, these solutions will be all equivalent, the only difference will be that pieces with the same dimension will be interchanged.

To do this, it is sufficient to lexicographically order pieces of the same dimension. The constraint is then

$$\forall_{\substack{i,j \in PIECES,\\ i<j \ \wedge \\ pieces\_dimensions[i,\ x]=pieces\_dimensions[j,\ x] \ \wedge \\ pieces\_dimensions[i,\ y]=pieces\_dimensions[j,\ y]}}$$

$$[pieces\_corners[i,\ axis] \mid \text{for } axis \in AXES]$$

$$<$$

$$[pieces\_corners[j,\ axis] \mid \text{for } axis \in AXES]$$

where $<$ is a lexicographical comparison.

The lexicographical comparison has been implemented as a recursive Python function which compares lexicographically two arrays

```
def lex_less(a, b):
    def lex_less_auxiliary(a, b, i, accumulator):
        if i == len(a) - 1:
            return lex_less_auxiliary(a, b, i - 1, a[i] < b[i])
        elif i == 0:
            return Or(a[i] < b[i], And(a[i] == b[i], accumulator))
        else:
            return lex_less_auxiliary(a, b, i - 1, Or(a[i] < b[i], And(
                                                  a[i] == b[i],
                                                  accumulator)))
    return lex_less_auxiliary(a, b, len(a) - 1, True)
```

The constraint is then encoded in Z3 as the following expression:

```
[lex_less(pieces_corners[i], pieces_corners[j])
    for i in PIECES for j in PIECES if i < j and pieces_dimensions[i] =
                                      = pieces_dimensions[j]]
```

For the model including the possibility to rotate pieces, it is needed to modify this constraint by considering two pieces being of the same dimension also if one

of the two pieces has the same dimension of the other, but width and height are inverted. This is needed because if a piece having the same dimensions of another, but with width and height swapped, is rotated, then it will have exactly the same dimensions of the other piece, and for this reason these two pieces need to be ordered as well.

The constraint described before can be modified to

$$\forall_{\substack{i,j \in PIECES,\\ i<j \ \wedge \\ (pieces\_dimensions[i,\ x]=pieces\_dimensions[j,\ x] \wedge \\ pieces\_dimensions[i,\ y]=pieces\_dimensions[j,\ y]) \\ \vee \\ (pieces\_dimensions[i,\ x]=pieces\_dimensions[j,\ y] \wedge \\ pieces\_dimensions[i,\ y]=pieces\_dimensions[j,\ x])}}$$

$$[pieces\_corners[i,\ axis] \mid \text{for } axis \in AXES]$$

$$<$$

$$[pieces\_corners[j,\ axis] \mid \text{for } axis \in AXES]$$

where $<$ is a lexicographical comparison. In Z3, the corresponding expression is:

```
[lex_less(pieces_corners[i], pieces_corners[j])
    for i in PIECES for j in PIECES if i < j and
    ((pieces_dimensions[i][x] == pieces_dimensions[j][y] and
    pieces_dimensions[i][y] == pieces_dimensions[j][x]) or
    (pieces_dimensions[i][x] == pieces_dimensions[j][x] and
    pieces_dimensions[i][y] == pieces_dimensions[j][y]))]
```

3 Optimization

In this section, the constraints and techniques used to optimize the search of the solution will be described. Also, a comparison of the performances before and after applying these modifications will be showed.

3.1 Optimization constraints

Optimizing a model means reducing the number of possible solutions by the use of constraints, which also means pruning the search tree and reducing the search space. This allows to solve more instances because, with a reduced search space, hardest instances will require less time to be solved. In order to do so, it is possible to add constraint which are specifically designed for this purpose.

For the standard model, there's no optimization to do.

3.1.1 General model

For the general model, an optimization constraint can be added in order to optimize it, which deals with managing the rotation of square pieces.

If a piece is square (width and height are the same) there is no need to consider also solutions where this square pieces are rotated. It is so possible to add a constraint which excludes these solutions by forcing square pieces to not be rotated:

$$\forall i \in PIECES,\ pieces\_dimensions[i,\ x] = pieces\_dimensions[i,\ y]$$

$$\neg pieces\_rotation[i]$$

In Z3, this becomes:

```
[Not(pieces_rotation[i]) for i in PIECES if pieces_dimensions[i][x] =
                                    = pieces_dimensions[i][y]]
```

The constraint specified in **Section 2.5.2** is also an optimization constraint since it manages pieces with the same dimension by ordering them, as explained in that section. In particular, it is a symmetry breaking constraint.

4 Conclusions

To solve hardest instances, as said in **Section 2.4**, a modified cumulative implied constraint was used. Choosing the right implied cumulative constraint was a difficult challenge: in fact, as said in section **Section 2.4**, these modified constraint will cause the model to fail in case an input instance has pieces which can be cutted from the paper roll without consume it all. This is not a problem, however, since all our input instances make use of all the paper roll, so this won't be a problem in our case.