SMT Model - Present Wrapping Problem

Filippo Lo Bue

September 2020

1 Description of the Problem

Given a wrapping paper roll of a certain dimension and a list of presents, decide how to cut off pieces of paper so that all the presents can be wrapped. Consider that each present is described by the dimensions of the piece of paper needed to wrap it. Moreover, each necessary piece of paper cannot be rotated when cutting off, to respect the direction of the patterns in the paper.

The purpose of this project is to solve PWP proceeding as follows:

1. Start with the variables and the main problem constraints.
2. In any solution, if we draw a vertical line and sum the vertical sides of the traversed pieces, the sum can be at most l.
3. Use global constraints to impose the main problem constraints and the implied constraints in your SMT model.
4. Investigate the best way to search for solutions in SMT.
5. If rotation was enabled? which of the SMT model encoding is easier to modify to take this into account? How would you modify that model/encoding?
6. there can be multiple pieces of the same dimension: how would you improve the SMT model encoding?

Points **1 - 4** are part of the Z3py program.
Point **5** is expressed using mathematical notation and it was included in a second Z3py program.
Point **6** is expressed using mathematical notation and it was included in a third Z3py program.

2 1^{st}model - create_model (Points 1-4)

2.1 Parameters

- **pr_w**: width of the wrapping paper roll.
- **pr_h**: height of the wrapping paper roll.
- **n_pieces**: number of the presents/pieces.
- **L**: dimensions of the piece of paper needed to wrap each presents.
- **index_largest_p**: index of the biggest present/piece by area.
- **indep**: array of set pieces' indexes grouped by widths if it is possible to speed up the process of finding 1 solution(independent solving on w), False otherwise.

2.2 Variables

Bottom left corner of each pieces

$$
\begin{aligned}
\mathbf{q} \quad &= [[piece\_1\_x, piece\_1\_y] \\
&= \ [piece\_2\_x, piece\_2\_y] \\
&\ \ ... \\
&= \ [piece\_n\_x, piece\_n\_y]]
\end{aligned}
$$

$\forall\ i,j,\ i<j \quad lr_{i,j} \in [false, true]$	$true$ if r_i are placed at the left to the r_j, otherwise $false$
$\forall\ i,j,\ i<j \quad ud_{i,j} \in [false, true]$	$true$ if r_i are placed at the downward to the r_j, otherwise $false$
$\forall\ i,e,\ 0 \leq e \leq pr\_w - w_i \quad px_{i,e} \in [false, true]$	$true$ if r_i are placed at less than or equal to e, otherwise $false$
$\forall\ i,f,\ 0 \leq f \leq pr\_h - h_i \quad py_{i,f} \in [false, true]$	$true$ if r_i are placed at less than or equal to f, otherwise $false$
$\forall\ i,h,\ h :: [0..pr\_w] \quad ax_{i,h} \in [false, true]$	$true$ if r_i is "active"/located at h as x coordinate, otherwise $false$
$\forall\ i,h,\ h :: [0..pr\_h] \quad ay_{i,h} \in [false, true]$	$true$ if r_i is "active"/located at h as y coordinate, otherwise $false$

The value of the boolean variables are constrain to the value of the q variable.

2.3 Constraints

2.3.1 Domain

For each gift/piece I can restrict the domain of the bottom left corner.

$$\forall\ i,\ \ 0 \leq q[i]\_x \leq pr\_w - i\_width\ \ \&\ \ 0 \leq q[i]\_y \leq pr\_h - i\_height$$

2.3.2 Symmetry breaking rules[2]

The Problem so far contains a number of symmetries, which we need to remove as we want to speed up the finding process of a solution. The symmetries that we want to remove are the horizontal reflection, vertical reflection and a combination of both. Considering a generic solution:

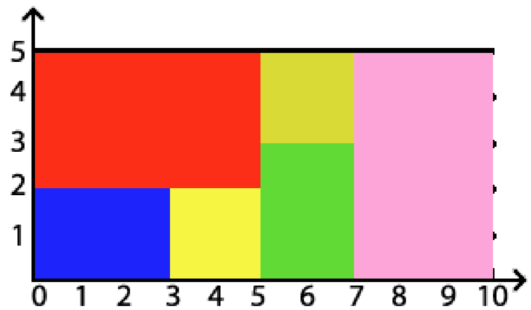
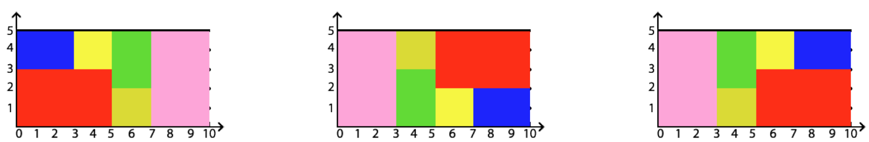


Figure 1: generic solution

We have the following 3 “reflected” solutions:



(a) horizontal symmetry (b) vertical symmetry (c) double symmetry

Figure 2: Reflected solutions

To remove those we can restrict the domain of one present in an enclosing rectangle and since we can choose any present, it is better to restrict the domain of the **largest** one of size $w\_max \times h\_max$ for ensuring a much bigger domain reduction propagation for the others presents:

$$0 \leq q[index\_largest\_p]\_x \leq \lfloor \frac{pr\_w - w\_max}{2} \rfloor,$$

$$0 \leq q[index\_largest\_p]\_y \leq \lfloor \frac{pr\_h - h\_max}{2} \rfloor$$

2.3.3 Order Encoding

There have been several studies on translation methods which encode a CSP into a SAT problem, among them, *order encoding* aims to make a more natural explanation of the order relation of integers. Let x be an integer variable and c be an integer value, the following constraint $x \leq c$ is encoded into a Boolean variable $p_{x,c}$[1].

$\forall$ rectangle r_i,we have the following 2-literal axiom clauses[1]:

$$\forall i, 0 \le e \le pr\_w - w_i, 0 \le f \le pr\_h - h_i :$$

$$\neg px_{i,e} \vee px_{i,e+1}$$

$$\neg py_{i,f} \vee py_{i,f+1}$$

2.3.4 Double Cumulative Constraint (Point 2)

The cumulative constraint is not only used for scheduling tasks but it is used also for spatial positioning of objects(the presents) inside a container(paper roll). Essentially We can see every presents/pieces as an action which need to be scheduled, the x coordinate as the starting time of that action, the width as the duration and the height as the resources consuming.

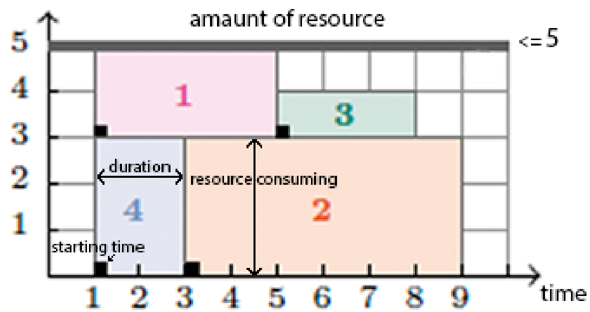


Figure 3: "Horizontal" cumulative constraint

The second cumulative constraint has the same application as the first inverting the axes and the meaning of the actions'(pieces') duration(the heights) and the resource consuming(the widths).

In a SMT program it is possible to replicate the application of the cumulative constraint as follow[5]:

- at all times used resources do not exceed the total capacity.
- starting times respect feasible window.

A pure SMT approach has been used in the z3 program using the variables $ax_{i,h}$ and $ay_{i,h}$:

$$\forall i, h :: [0..pr\_w] \;\; ax_{i,h} \textbf{ iff } \widehat{Sx_{i,h-Lx_i}} \wedge Sx_{i,h} \text{ where } Sx_{i,h} \text{ means } q[i][0] \le h$$
$$\forall i, h :: [0..pr\_h] \;\; ay_{i,h} \textbf{ iff } \widehat{Sy_{i,h-Ly_i}} \wedge Sy_{i,h} \text{ where } Sy_{i,h} \text{ means } q[i][1] \le h$$

Once we have constrained the $ax_{i,h}$ and $ay_{i,h}$ we can apply the following two "cumulative" constraints:

```
%cumulative on the x axis
sum([ L[i][1]*ax_i_h for i in [0..n_pieces] ]) <= pr_h
    for h in [0..pr_w-1]

%cumulative on the y axis
sum([ L[i][0]*ay_i_h for i in [0..n_pieces] ]) <= pr_w
    for h in [0..pr_h-1]
```

2.3.5 Non-overlapping Constraints 1

$\forall$ rectangle r_i, r_j (i < j), we have the following 4-literal clauses:

$$lr_{i,j} \vee lr_{j,i} \vee ud_{i,j} \vee ud_{j,i}$$

2.3.6 Independent Solving based on widths

Constraint applied only if *independent_solving_on_w* is *true*(only on the x axis). *independent_solving_on_w* is *true* if $\forall$ pieces group by width the sum of the pieces' heights are greater or equal then *pr_h* so $\forall$ group of pieces based on their width in *indep* we can constraint their relative position to speed up the finding process of one feasible solution.

All the pieces with the lower width must be placed on the left side of the container(paper roll), next we will place those with the slightly larger width and so on. This constraint allows us to speed up the resolution of many otherwise unsolvable instances.

E.g. Instance 18 × 18

pr_w=18, **pr_h**=18, **n_pieces**=16

$$\begin{aligned} \mathbf{L} &= [[3,3],[3,4],[3,5],\ [3,6] \\ &= \ [3,7],[3,8],[3,10],[3,11] \\ &= \ [4,3],[4,4],[4,5],\ [4,6] \\ &= \ [5,3],[5,4],[5,5],\ [5,6]] \end{aligned}$$

$$\begin{array}{lll} \mathbf{indep} & = [[1,2,3,4,5,6,7,8], & \text{- width of 3} \\ & \quad [9,10,11,12], & \text{- width of 4} \\ & \quad [13,14,15,16]] & \text{- width of 5} \end{array}$$

So:

$\forall i \in [1 \,..\, 8], j \in [9..16]$ $\ lr_{i,j}$ *true*
$\forall i \in [9..12], j \in [13..16]$ $\ lr_{i,j}$ *true*

3 2^{nd}model -create_model_with_rotation (Point 5)

The same previous model with handling the possible rotation of each presents/-pieces.

3.1 New Variables

rot: Array of 0|1 with dimension equal to the number of presents/pieces. If the i^{th} element of the array is *'0'* means no rotation otherwise if *'1'* that piece is rotated by 90°

$$\textbf{rot} \quad = [rot\_1, rot\_3, \ \ldots\ , rot\_n]$$

3.2 New Function

def **get_dim**(i, rot, d) : According to the rotation(rot), return the correct value of the width(*d=0*)/height(*d=1*) of a given piece index(i). This function is used every time I need to know the width/height of a piece, the pieces's dimension is not constant anymore like in the previous model.

3.3 New Constraints

In all constraints I no longer use the initial dimensions of the pieces but I use the new function *get_dim*.

3.3.1 square = no rotation

It is unnecessary to rotate any square piece (width = height) because I will get the same solution (same coordinates for each piece).

4 3^{rd}model - create_model_same_dim (Point 6)

The same model as *create_model* but taking into consideration the fact that there can be multiple pieces with the same dimension. Also the previous models are able to solve instances in which there are several pieces with the same dimensions but in those cases it is possible to fix the positional relation with each other reducing the domains for some pieces and speeding up the instance resolution itself[1].

4.1 New Constraints

4.1.1 same dimension

If we have rectangles/pieces r_i, r_j and r_k which have the same dimensions we can fix/constrain the positional relation of those rectangles, between r_i-r_j and r_j-r_k. For example, if we have 2 pieces with the same dimension(2×2), the second piece(the pink one) will never placed to the left or under the first one(the yellow one). it will be placed always on the right or upper the first one.

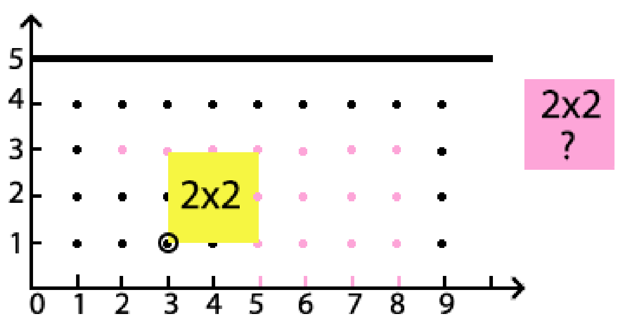


Figure 4: Positional relation constraint

Formally:

$\forall$ rectangle r_i, r_j (i < j, r_i_w=r_j_w, r_i_h=r_j_h), we can assign:

$$lr_{j,i} = false$$

$$lr_{i,j} \vee \neg ud_{j,i}$$

5 Experiments Results

Time to find the first solution for each instance.
Independent solving on width activated.

n	Time[s]	n	Time[s]	n	Time[s]
8x8	.035	19x19	.169	30x30	.461
9x9	.017	20x20	.209	31x31	.423
10x10	.028	21x21	.105	32x32	1.718
11x11	.030	22x22	.363	33x33	.908
12x12	.047	23x23	**22.414**	34x34	.608
13x13	.057	24x24	.294	35x35	.438
14x14	.043	25x25	**6.767**	36x36	.681
15x15	.092	26x26	.693	37x37	.650
16x16	.089	27x27	.327	38x38	.349
17x17	.170	28x28	.491	39x39	.757
18x18	.150	29x29	.579	40x40	.267
Tot:	**40.429s**				

All the instances are solved in 40.429 seconds and the 23x23 instance is the mostdifficult one to solve.

6 References

[1] Takehide, S., Katsumi, I., Naoyuki, T., Mutsunori, B., Hidetomo, N. *A SAT-based Method for Solving the Two-dimensional Strip Packing Problem.* http://ceur-ws.org/Vol-451/paper16soh.pdf

[2] Simonis, H. & O'Sullivan, B. (2008) *Search Strategies for Rectangle Packing.* International Conference on Principles and Practice of Constraint Programming: pp 52-66.

[3] Clautiaux, F., Jouglet, A., Carlier, J., Moukrim, A. (2006) *A new constraint programming approach for the orthogonal packing problem.* http://vmk.ugatu.ac.ru/c%26p/article/new_2009/2D_OPP_clautiaux_constraint_progr.pdf

[4] Huang, E., Korf, R.E. (2012) *Optimal Rectangle Packing: An Absolute Placement Approach.* https://arxiv.org/ftp/arxiv/papers/1402/1402.0557.pdf

[5] Nieuwenhuis, R., Oliveras, A., Rodrìguez-Carbonell, E. (2011) *Introduction to SMT. Solving CSP's with SMT.* https://www.cs.upc.edu/ erodri/webpage/papers/bergen2.pdf