Project Report

Combinatorial Decision Making and Optimization (CP)

by Hanying Zhang, Nour Bou-Nasr

March, 2021

1 Proposed mainlines

1.1 The variables and the main problem constraints

We wrote a python script file named *txt2dzn.py* whose function is converting the instance files from *text* file format to *dzn* file format. We then read in *W, H, N, Ws* and *Hs* respectivaly as the width of the whole paper, the height of the whole paper, the number of the small pieces, the widths of the small pieces and the heights of the small pieces. We created two decision variables Xs and Ys, which represent the coordinates of the left-bottom corner of all the small pieces.

1.1.1 The small pieces should not exceed the border of the whole paper

We used *forall* in Minizinc to make sure that the right borders of all the small pieces don't exceed the right border of the whole paper, also the upper borders of all the small pieces don't exceed the upper border of the whole paper.

1.1.2 There should be no overlap between all the small pieces

We used *forall* function to make sure that all the small pieces don't overlap. The hint here is that if two small pieces don't overlap horizontally **OR** vertically, then they don't overlap.

We also used *assert* to make sure that the total size of all the small pieces are not bigger than the size of the whole paper.

1.2 Implied constraints

Take a vertical line for instance, the total heights of all the traversed pieces should not be bigger than H, and we should check all the vertical lines. For the CP problem we use *sum* and *forall* for these two requirements respectively.

1.3 Global constraints

1.3.1 Main constraints

We used *diffn* in Minizinc to make sure all small pieces don't overlap.

1.3.2 Implied constraints

For the global implied constrants, we use *cumulative* constraint in Minizinc. This contraint is originally used for scheduling problems, but it's suitable here. Specifically, we take the left coner coordinate of a small piece as the starting time of a task, the width or height as the duration of the task, the height or width as the resource requirement, and H or W as the capacity.

1.4 The best way of searching

At first we searched Xs and Ys together (Xs++Ys), then we switched to use *seq_search*, which means that we would try to fix Xs first, then Ys. The results showed that the second method is better than the first one.

At the beginning we tried several combinations of searching techniques. For variable selection, we tried *dom_ w_ deg* and *first_ fail*, for value selection we tried *indomain_ min* and *indomin_ random*. We also tried *restart* when using *indomin_ random*. The most robust combination we found is *dom_ w_ deg* and *indomain_ random* with *restart*. But there were still about 5 instances which we could not find a solution within 5 minutes.

We then tried another method which was placing the bigger pieces before smaller ones. When converting *.txt* input files into *.dzn* input files, we sorted all the dimension lines by their sizes. We then did the search using *input_ order* and *indomain_ min*, which means that we tried to place the bigger pieces first, and to place them close to the left bottom corner of the whole sheet. In this way we also solved the symmetry problem. The result was better, there's only 1 instance which couldn't be solved within 5 minutes(37x37).

For figuring out whether 37x37 has a solution, we manually solved this instance. The method was placing the widest rectangle first, if rectangles had the same widths, place the highest first. We modified our model according to this method. We inverted the dimension lines in the original converted input file and used *input_ order* and *indomain_ min* for the searching. All instances could be solved immediatly except for 4 instances, namely 23x23, 26x26, 33x33 and 38x38. We then introduced *dom_ w_ deg* and *indomain_ random* in the searching of *Ys*, together with *luby restart*. Finally all the instances could be solved within 5 minitues (193s for the worst case on our machine).

1.5 Rotation

We need to set a new bool variable O_i to define if piece i is rotated. When O_i is true, the width and height of piece i is exchanged.

TODO: which one of CP and SMT are easier to implement rotation?

1.6 Multiple pieces of the same dimension

If there are multiple pieces with the same dimension, they should be placed together with each other to form a bigger piece. The problem would be easier as the number of small pieces are smaller and we are just forming new rectangles. But we should also consider that they couldn't exceed the borders of the whole sheet of paper, also if they can't perfectly fit the width or length of the whole paper, we should consider leaving enough space for other small rectangels. We should also consider the direction they grow, for example, two pieces of size 3x4, they could form a bigger piece with size 6x4 or 3x8.

2 Other remarks

2.1 Auxilliary constraints

2.1.1 The small pieces whose heights *Hs[i] > 0.5 * H* could only be placed horizontally

If the total heights of 2 rectangles are both bigger than half the height of the whole sheet, then they could not be placed vertically, which means that the x coordinates are different. Thus we could use global constrint *alldifferent* upon them. More strictly, suppose their width are w_i and w_j respectively, and $w_i < w_j$, then the minimum distance between them should be w_i.

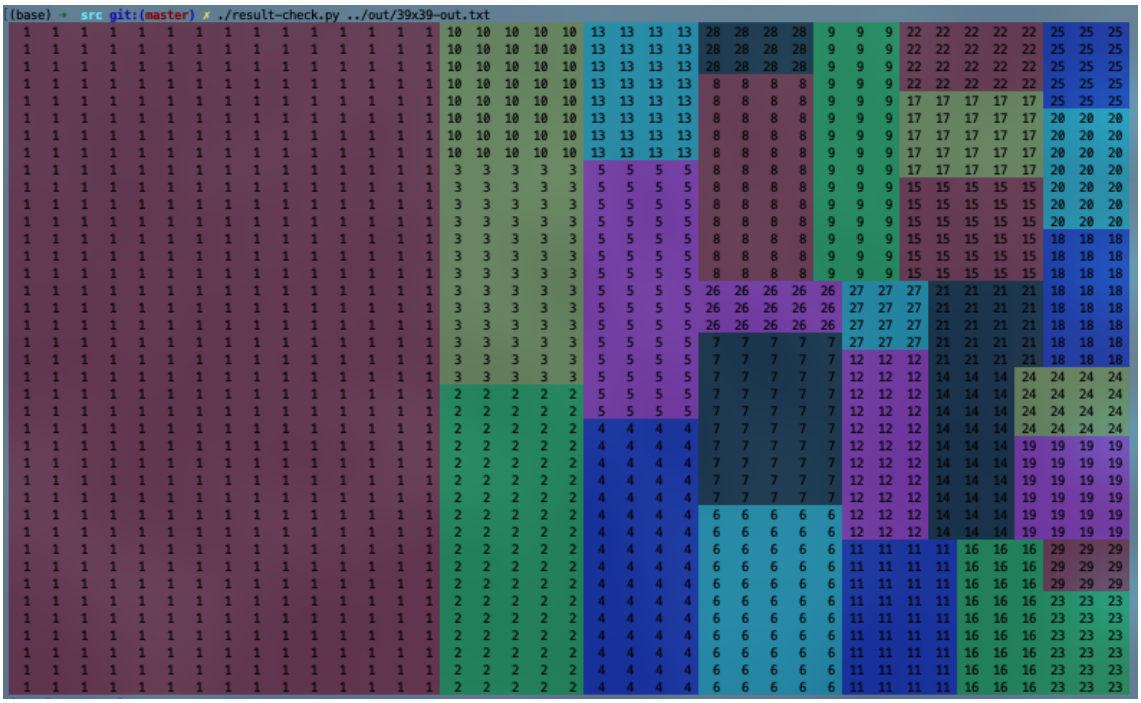
2.1.2 Encouraging two suitable rectangles to form a whole column

The main idea behind this is that by forming whole columns, to some sense we are actually decresing the size and complexity of the problem. Specifically, we are transforming a problem with size $w_1 \times h$ into a problem $w_2 \times h$, where $w_2 < w_1$. Suppose we have two rectangles with the same width, the height of one rectangle is bigger than 0.5 * H, and the sum of their heights is H, then we try to make them into a column.

The problem with this constraint is the new and less complex problem also has a smaller solution space. It may result in a UNSATISFIABLE situation. In our model we loosed this constraint by also allowing the two rectangles to be placed side by size horizontally. For this project it didn't cause unsatisfication but strictly speaking, this is not a robust constraint.

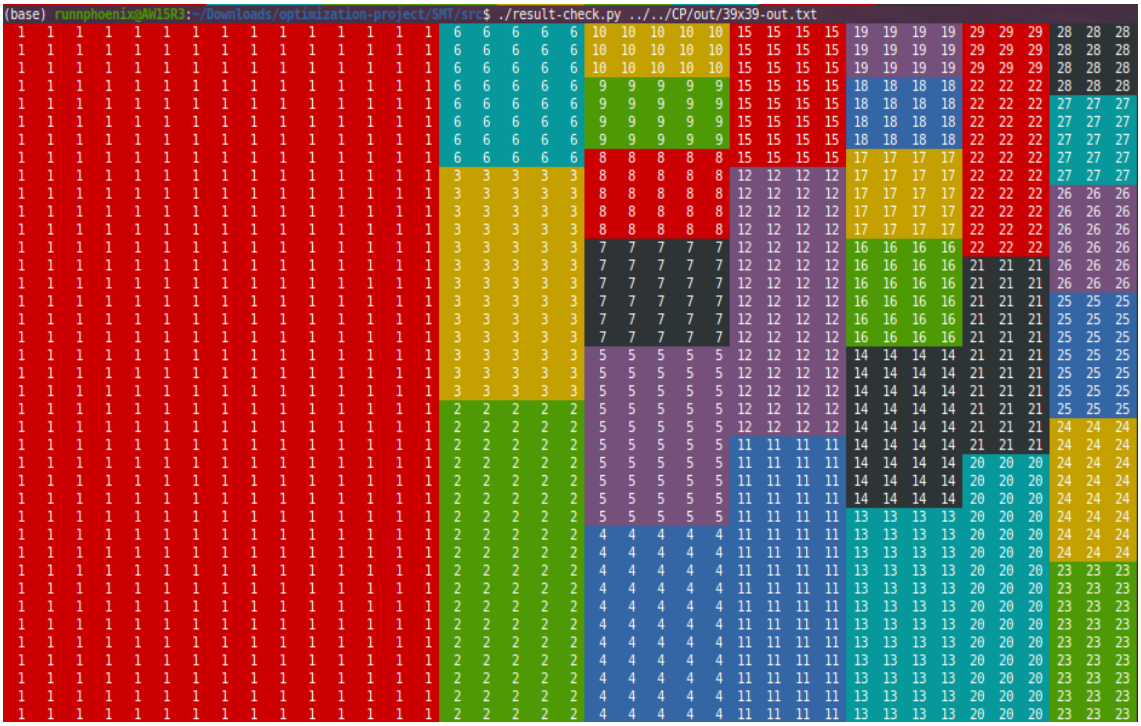
2.2 Result verification

We wrote a python script file named *result_ check.py* for checkig whether the result we got is right. This file is run in terminal and would give a colored representation of the whole sheet. One of the result picture is as following:



We can see from this picture that our model (the one placing rectangles by their sizes) is actually working. Basiclly speaking, the bigger the rectangles, the closer they are to the left bottom corner of the whole sheet.

The result of our experiment which sorted the rectangles by their widths is as following:



The problem with this script file is that some adjacent rectangles have the same color, as we use only 7 different colors(maybe we should use CP again to solve this problem :p).

3 References

[1] Helmut Simonis and Barry O'Sullivan. Using Global Constraint for Rectangle Packing.