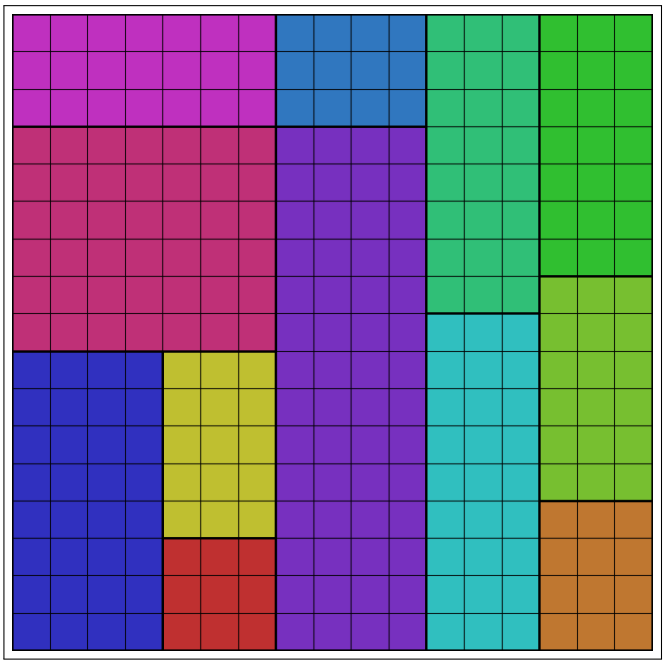
Flatland Challenge

G. Berselli, M. M. L. Pulici

August 29, 2021



Contents

List of Figures

1 Introduction

2 Modelling

2.1 Variables

Each instance comes with 4 parameters describing the chip configuration:

- `chip_w`: the width of the chip;
- `n`: the number of blocks to fit on the chip;
- `inst_x`: a list containing the ordered widths of the blocks;
- `inst_y`: a list containing the ordered heights of the blocks.

In addition, 3 auxiliary variables are created:

- `min_index`: the index of the smallest block (see Section 2.3);
- `min_h`: the minimum possible height of the chip (see Section 2.5);
- `max_h`: the maximum possible height of the chip (see Sections 2.5).

The results of the various solving procedures consists in 3 variables:

- `chip_h`: the height of the chip;
- `bl_x`: a list containing the ordered horizontal positions of the bottom left corners of the blocks;
- `bl_y`: a list containing the ordered vertical positions of the bottom left corners of the blocks.

2.2 Constraints

Given the problem to fit all blocks into the chip, the first obvious constraint is that of not making the blocks fall out of the chip. This is quite easy: it is sufficient to impose that, for every block `k`:

- `bl_x[k] > 0`;
- `bl_x[k] + inst_x[k] <= chip_w`;

- `bl_y[k] > 0`.

The next step is to impose a non-overlapping constraint between each block pair. In order to do that, the basic idea is to cycle through all the blocks and to check that the distance between every bottom left corner pair is at least equal ti the dimension of the block in one direction. More practically, the following four constraints are inserted disjunctively for every pair of blocks `k` and `l`:

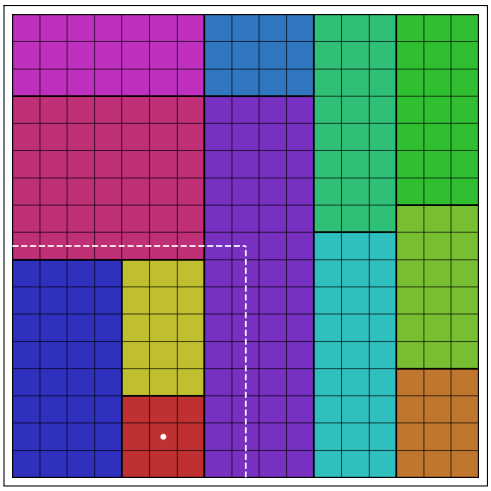
- `bl_x[k] + inst_x[k] <= bl_x[l]`;
- `bl_y[k] + inst_y[k] <= bl_y[l]`;
- `bl_x[l] + inst_x[l] <= bl_x[k]`;
- `bl_y[l] + inst_y[l] <= bl_y[l]`.

2.3 Symmetry breaking

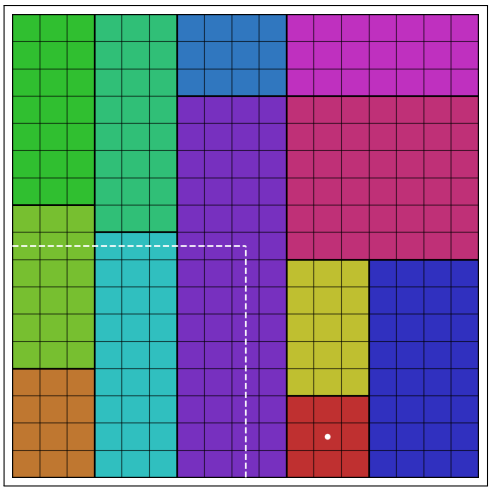
In order to eliminate symmetrical solutions, a simple symmetry breaking constraint is introduced. As shown in Fig. 1, each solution has four symmetrical configurations, where the blocks are located in symmetrical positions with respect to an horizontal and/or a vertical central symmetry ax.

In each of the configurations, the center of the smallest block is highlighted with a white dot and the bottom left quadrant is indicated by a dashed white line. It is can be noted that, with the exception of very rare cases, the white dot lies in the highlighted quadrant in only one of the 4 symmetrical configurations. Therefore, the chosen symmetry breaking constraint consists in forcing the white dot to lie in the bottom left quadrant, i.e. adding the following conjunctive constraints, where `min_index` is the index of the smallest block:

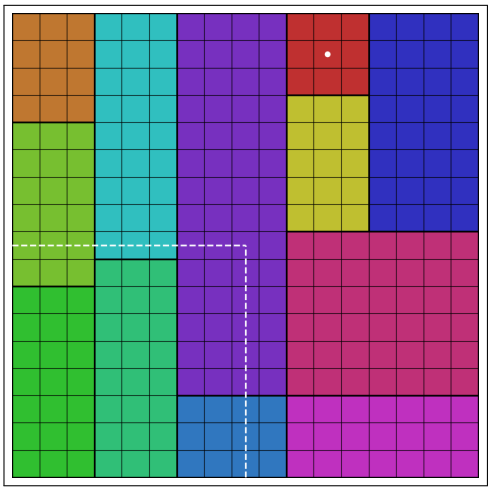
- `(2 * bl_x[min_index] + inst_x[min_index]) <= chip_w`;
- `(2 * bl_y[min_index] + inst_y[min_index]) <= chip_h`.



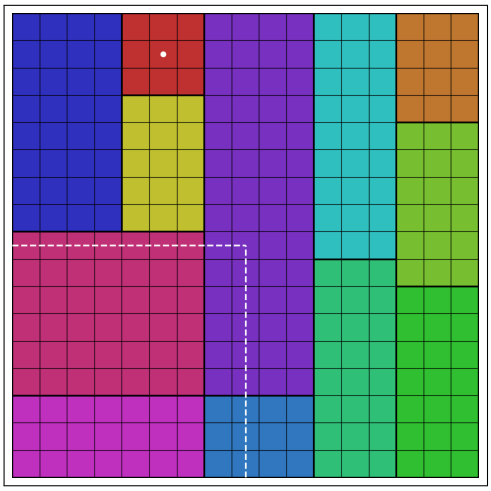
(a) Configuration 1.



(b) Configuration 2.

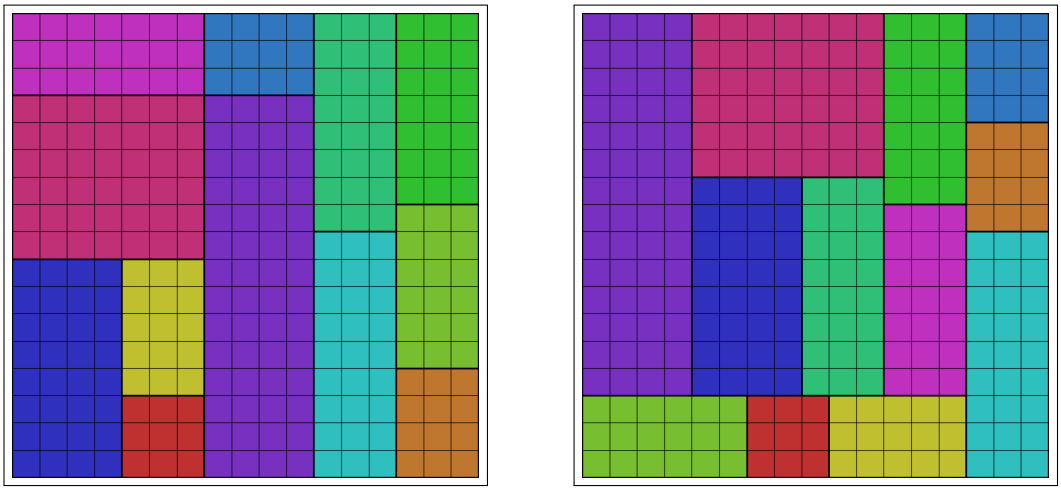


(c) Configuration 3.



(d) Configuration 4.

Figure 1: Symmetrical configurations of Instance 10, with a white dot in the center of the smallest block and a dashed white line indicating the bottom left quadrant.



(a) Without rotations. (b) With rotations.

Figure 2: Solutions of Instance 10 with and without enabled rotations.

2.4 Rotation

The next step is to implement rotations. Since the blocks are rectangular and nothing is known *a priori* about whether they would wedge perfectly without gaps, it may be possible that the minimal height of the chip is obtained by rotating some of the blocks. For example, the difference between the two solutions for Instance 10 is shown in Fig. 2. Even if in the given instances rotating does not actually reduce the chip height, it can still be observed that some blocks get rotated by the solver.

These rotations are handled differently in each implementation, but the basic idea is to offer both the normal and the rotated block to the solver and to impose an exclusive disjunction between the two orientations. In particular, it is useful to create a boolean vector of flags signalling whether each block is rotated (`rotated`) and 2 auxiliary lists containing the actual dimensions of blocks (`new_inst_x` and `new_inst_y`). This way, since boolean values are by nature exclusive, the dimensions of each block `k` can be linked to the respective boolean value with something similar to:

```
new_inst_x[k] = inst_y[k] if rotated[k] == True else inst_x[k]
new_inst_y[k] = inst_x[k] if rotated[k] == True else inst_y[k]
```

2.5 Improvements

The main improvement of the model consists in adding boundaries to `chip_h`. The minimum height is fairly easy to evaluate, since in the best case scenario (i.e. if the blocks leave no gaps) `min_h = total_area / chip_w`, where `total_area` is calculated by simply adding up all the products of the blocks' dimensions. Computing the maximum height, on the other hand, is more subtle. A naive approach consists in simply adding all heights together, considering the worst case as that where all blocks are stacked vertically. This is of course a reasonable upper boundary, but, as the number of blocks increases, `max_h` greatly overestimates the actual height.

A more clever solution can be reached with minimal preprocessing reasoning as follows. If blocks are ordered by height and width, the following code can be run:

```
chip_cumulative = chip_w
k = 0
heights = []
while k < len(inst):
    if inst[k][0] <= chip_cumulative:
        chip_cumulative -= inst[k][0]
        heights.append(inst[k][1])
        del inst[k]
    else:
        k += 1
max_h += max(heights)
```

The reasoning is the following. The blocks' widths are compared with the free horizontal space (`chip_cumulative`) and, if each block fits, the free space is decreased. When no more blocks fit on the line, the maximum height is taken among the located blocks and it is added to the total `max_h`. The process is then repeated for the next lines as long as there are blocks left. This way, a much more optimistic, but strictly valid, `max_h` is computed, as shown in Fig. 3.

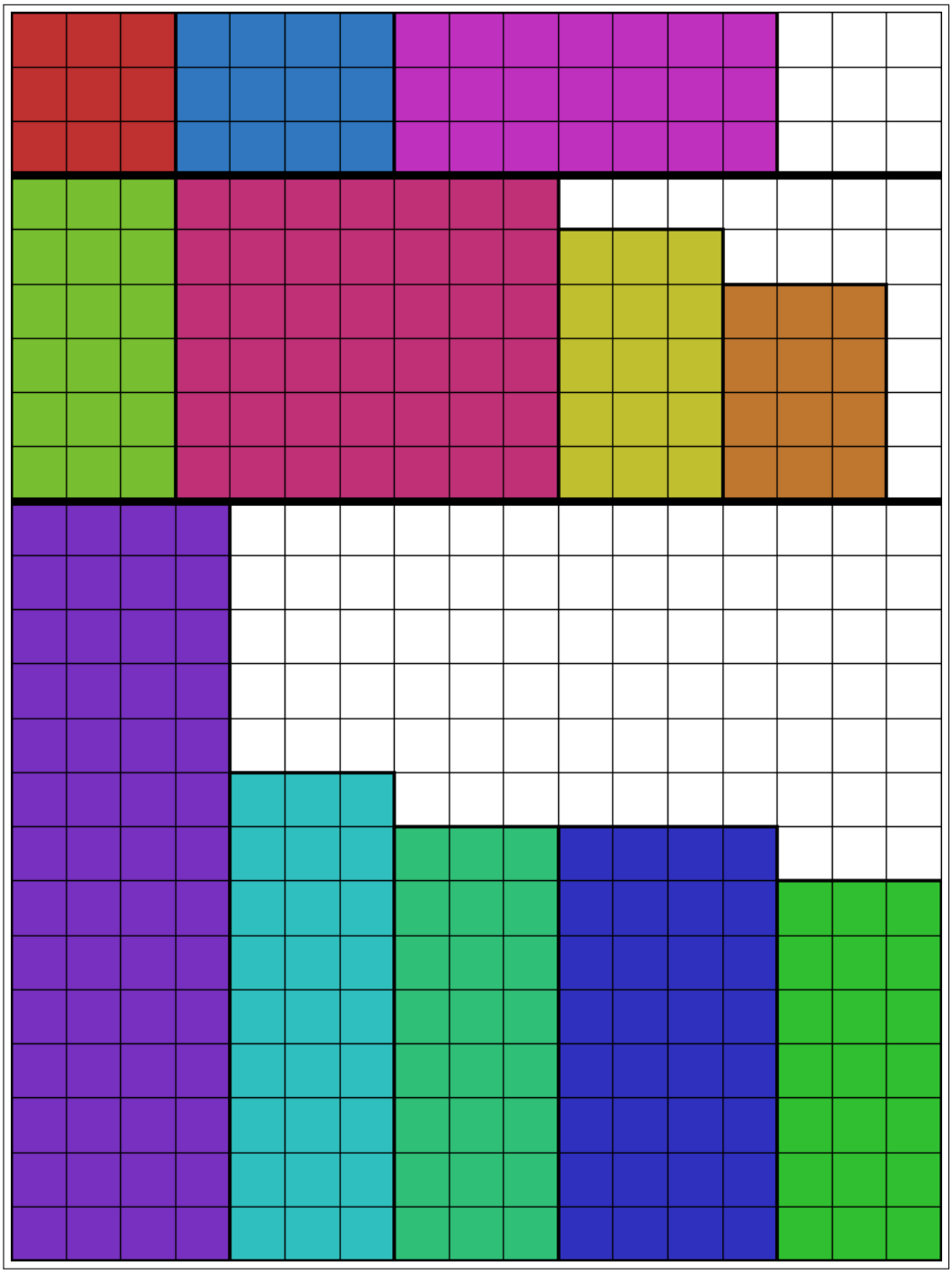


Figure 3: A graphical representation of the `max_h` algorithm for Instance 10.