CP Report

Muhammad Zohaib Shafiq and Fabio Merizzi

Muhammad.shafiq6@studio.unibo.it

Fabio.merizzi@studio.unibo.it

August 2021

Contents

1 Introduction

In this report we will discuss the optimization of the VLSI (Very large scale integration) problem. VLSI is a famous problem of electronics, and it refers to fitting multiple integrating circuits into a silicon chip. The main objective of the problem is to fit rectangles of arbitrary dimension in a fixed width rectangle, minimizing the height.

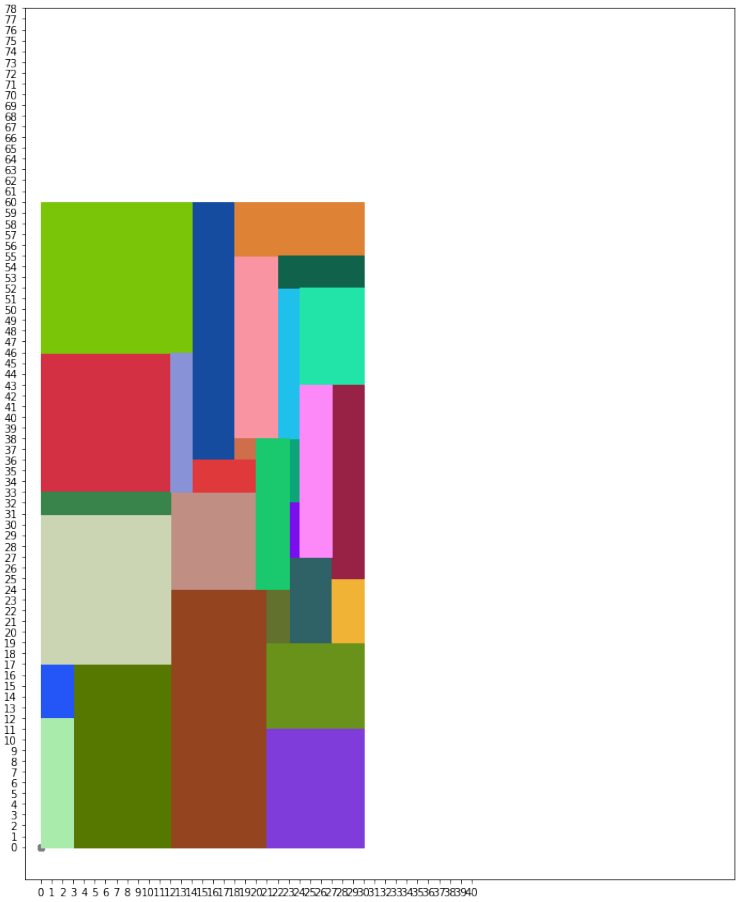

Figure 1: example of a solved instance of the VLSI problem

In this paper we address the problem with both Minizinc and SMT (z3). In the following pages we will discuss the main implementation of both solutions, describing the progressive improvements applied to the model.

1.1 Python plotter

Visualizing the plotted instances is useful for the early noticing of errors, and to have a better general understanding of the solver's work. We developed a simple python program that plots the rectangles using the python matplotlib library.

```
1
2  import matplotlib.pyplot as plt
3  import random
4  import numpy as np
5
6  fig, ax = plt.subplots(figsize=[15,15])
7
8  def plot_rect(ax,x,y,w,h):
9      r = random.random()
10     b = random.random()
11     g = random.random()
12     color = (r, g, b)
13     rect = plt.Rectangle((x,y), w, h,color=color)
14     ax.add_patch(rect)
15     ax.scatter(0,0)
16
17 for i in range(len(xCoordinate)):
18     plot_rect(ax,xCoordinate[i],yCoordinate[i],xDimension[i],
       yDimension[i])
19
20 plt.axis('square')
21 plt.xticks(np.arange(0, max(xCoordinate) + max(xDimension), 1))
22 plt.yticks(np.arange(0, max(yCoordinate) + max(yDimension), 1))
23 plt.show()
```

Listing 1: plot.py

2 Minizinc

2.1 Variables

The model discussed in this paper will use two sets of values, one representing the rectangle's leftmost corner with X and Y coordinates, and the other containing the values for the respective height and width.

Below, the first section of the Minizinc problem is reported. The main encoding is made of 4 arrays, called respectively circuitX, circuitY, X, Y.

- circuitX, contains the width of each rectangle and is given a data .dzn file
- circuitY, contains the height of each rectangle and is given a data .dzn file
- X, its one of the objective of the optimization and contains the x coordinate of the leftmost corner for each rectangle
- Y, its one of the objective of the optimization and contains the y coordinate of the leftmost corner for each rectangle
- width and num_circuits are two other values defined by the data instance

Other possible encoding were proposed and discussed, the subject is briefly addressed in the conclusion of the paper.

```
include "globals.mzn";
include "diffn.mzn";
include "data.dzn";

int: width;
int: num_circuits;

set of int: CIRCUIT_DIM = 1..num_circuits;
array[CIRCUIT_DIM] of int: circuitX;
array[CIRCUIT_DIM] of int: circuitY;

int: min_height = max(circuitY);
int: max_height   = sum(circuitY);

array [ CIRCUIT_DIM ] of var 0..width : X;
array [ CIRCUIT_DIM ] of var 0..max_height: Y;
```

```
var min_height..max_height: Height;
```

We define the variable Height, which is the variable the objective function is trying to minimize.

2.1.1 min height and max height

The two crucial values min-height and max-height set the boundaries (lower and upper) of the search.

- min-height, the minimum height of the solution is fixed to the height of the highest circuit. No solution can be lower than that.
- max-height, the maximum height of the solution is fixed to the sum of all the heights of all the circuits (y-axis), this represent the worst case scenario in which each rectangle is placed directly on top of each other.

2.2 Main problem constraints

The problem requires multiple constraints to be solved. Most of the constraints were improved during the developement of the final CP model.

2.2.1 set the boundaries of the space

```
constraint
        forall( i in CIRCUIT_DIM ) (
            X[i] + circuitX[i]  <= width
        );

constraint
        forall( i in CIRCUIT_DIM  ) (
            Y[i] + circuitY[i] <= Height
        );
```

The first constraint set the boundaries of the space where the recangles have to fit. It simply requires every rectangle, in both height and width, to be contained in the fitting space. This constraint was kept identical in all the improved models. We think that no global constraint can be applied here, and a forall implementation was used.

2.2.2 the non overlap constraints

the next constraint is the non-overlap, the one that prevents each rectangle from being put on top of another. This is the most challenging and complex constraint to satisfy.

```
constraint
    forall( i, j in CIRCUIT_DIM where i < j ) (
        X[i] + circuitX[i] <= X[j]
    \/  X[j] + circuitX[j] <= X[i]
    \/  Y[i] + circuitY[i] <= Y[j]
    \/  Y[j] + circuitY[j] <= Y[i]
    );
```

Above is reported the first non-overlap constraint we implemented. It worked consistently but we later discovered the global diffn constraint, which proved to be more efficient. In the basic non-overlap constraint we have 4 conditions, and at least one has to be true, for every couple of rectangles, the first constraint imply that the first rectangle is one the left of the second, the second constraint imply that the first rectangle is on the right of the second, and the same is applied for top and bottom.

```
constraint diffn(X,Y,circuitX,circuitY);
```

Below, a brief comparison between two searches on the same instance is reported, using the diffn global constraint and the simple the non-overlap constraint. The report is obtained in the Minizinc IDE, with the statistics tool of the solver. The comparison is correctly scaled, on the y axis the search depth is displayed and on the x axis the number of tries.

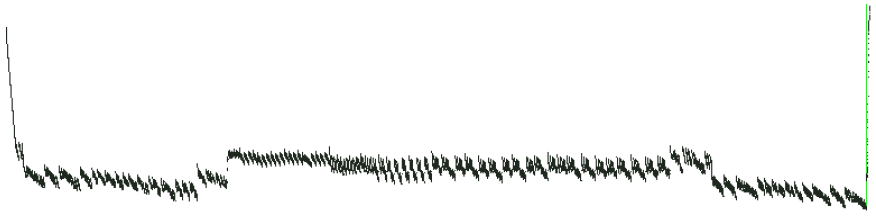

Figure 2: solving an instance using forall constraint

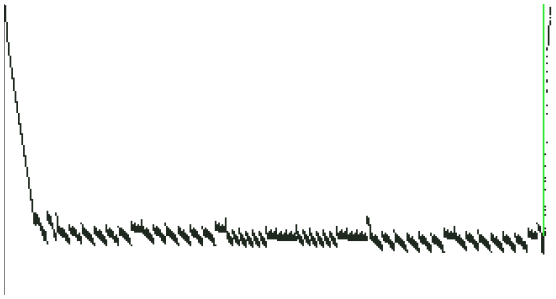

Figure 3: solving an instance using dffn constraint

2.2.3 the cumulative constraints

the cumulative constraints are very powerful global constraints. They are similar to packing constraints and can enforce the amount of space we add up in each dimension.

```
constraint cumulative(
            [ X[i] | i in CIRCUIT_DIM  ],
            [ circuitX[i]  | i in CIRCUIT_DIM ],
            [ circuitY[i]  | i in CIRCUIT_DIM ],
            Height
        );

constraint cumulative(
            [ Y[i] | i in CIRCUIT_DIM],
            [ circuitY[i]  | i in CIRCUIT_DIM ],
            [ circuitX[i]  | i in CIRCUIT_DIM  ],
            width
```

```
);
```

We use a cumulative constraint for each dimension, width and height. They are redundant but they do not enforce packing, they just check that the sum is correct. Intuitively, if there are empty spaces between the rectangles they will not be counted in the cumulative sum.

The cumulative constraint are redundant, but during search they could be triggered before the dffn constraint, therefore being more efficient. The use of cumulative constraint proved to have the biggest improvement from the baseline model.

2.3 Symmetry breaking

2.3.1 breaking the "same dimension" constraint

If two retangles have the exact same dimension, they are interchangable, therfore we can exploit the symmetry breaking.

```
constraint symmetry_breaking_constraint(
    forall( i,j in CIRCUIT_DIM where i<j ) (
        not (circuitX[i] == circuitX[j] /\
        circuitY[i] == circuitY[j]) \/
        (if X[i] == X[j] then Y[i] >= Y[j] else X[i] > X[j] endif)
    ));
```

The constraint proved to be very effective in all instances with a lot of rectangles with the same dimension. Of course, in all other instances is ineffective. The constraint check that for each rectangle with same dimension, one must be placed in a upper right position with respect to the other, avoiding the possibilty of interchange between them.

2.3.2 improving the "same dimension" constraint

The same dimension constraint can be implemented using the global lex_greater constraint, improving the general performance. It is also worth mentioning that the same dimension constraint does not only apply to single rectangles, but also to more complex shapes. For example if we have two identical shapes made up of different rectangles, those two shapes can be interchanged and the symmetry exploited. A viable implementation of this solution can be found in the Coursera course in discrete optimization by Prof. Peter Stuckey and Prof. Jimmy Lee [2]

2.3.3 other symmetry constraints

Many other symmetries can be found in the VLSI problem. The following will be briefly mentioned:

- flip symmetry, if we flip the fitting space, we have an equivalently correct solution. Therefore there is a symmetry that can be exploited, possibly using the the lex constraint.
- rotation symmetry, In a similiar way as the flip symmetry, its clear that rotating the space produces equivalently corrent solutions, therefore a symmetry can be exploited.

2.4 Search

The objective of the optimization is to place all the rectangles in the fitting space, filling the arrays of coordinates X and Y while minimizing the Height value. The first models used a simple "solve minimize height" search. In the following versions we decided to switch to a int_search. After ample testing we opted for the following search annotations:

```
solve::
int_search([ Height ]
            ++ [ X[num_circuits] | i in CIRCUIT_DIM ]
            ++ [ Y[num_circuits] | i in CIRCUIT_DIM ],
            dom_w_deg, indomain_split,
            )satisfy;
```

- dom_w_deg, the next variable is chosen with the smallest domain size divided by the number of times it has been in a constraint that caused failure earlier in the search. Its an improoved version of the first_fail search.
- indomain_split, It is a variable constraint in the assignment of the domain, in this case the variable's domain is bisected and the upper half excluded.

It is worth mentioning that the domain constraint indomain_random proved to be very effective in solving a few specific instances, But generally speaking the global performance was reduced. [1]

2.5 Rotation

In this section we discuss the task of implementing a model that can place the rectangles in both possible orientations. This task is widely discussed because orientation is often a critical feature on integrated circuits, and on most of the other implementations as well. This model is, of course, slower; it has to take into account many more possible configurations. On the other hand, the optimal solution that is found is usually better than the non-rotational model.

2.5.1 variables definition

```
int: num_circuits;
set of int: CIRCUIT_DIM = 1..num_circuits;

% the number of rotations is just 2 times the number of circuits
set of int: rotations = 1..(num_circuits*2);
array[rotations,1..2] of int: circuits;
array[CIRCUIT_DIM,1..2] of set of rotations: shape;
int: max_height;
int: width;

array[CIRCUIT_DIM] of var 0..width: x;
array[CIRCUIT_DIM] of var 0..max_height: y;
array[CIRCUIT_DIM] of var 1..2: rotation_array;

var 0..max_height: h;
```

For achieving this goal we introduce significative changes in the variables description.

- We aggregate the width and height dimensions into a single two dimensional array, called circuits. In this array are stored, for all the rectangles, the dimensions for both the orientations.
- We then use a new array of sets, called shape, which keeps into account which coordinates belongs to a given rectangle. This solution can be also configured so that a rectangle could have only one shape, if its a square. If our model was adapted to more complex shapes, we could also add more than 2 configurations.

We implement this solution mainly as a workaround the fact that no array-inside-array is possible in Minizinc. [3] For clarity, below is reported a data file describing a simple instance for use in the rotational model.

```
num_circuits = 5;
max_height = 15;
width = 5;

circuits = [| 2,1
            | 1,2
            | 3,2
            | 2,3
            | 1,4
            | 4,1
            | 2,2
            | 2,2
            | 2,4
            | 4,2
        |];

shape = [| {1}, {2}
         | {3}, {4}
         | {5}, {6}
         | {7}, {8}
         | {9}, {10}
         |];
```

The main workflow of the solution is preserved, we add in the boundaries constraint and in the non-overlap constraint the possibility for the solver to choose one of the two possible orientation for a given rectangle, introducing a new array of rotations.

2.5.2 set the boundaries of the space

```
constraint forall(i in CIRCUIT_DIM)(forall(r in rotations)
    (r in shape[i,rotation_array[i]] ->
    (x[i] + circuits[r,1]  <= width /\
    y[i] + circuits[r,2]  <= h)));
```

2.5.3 non overlap constraint

```
constraint forall(i,j in CIRCUIT_DIM where i < j)
  (forall(r1,r2 in rotations)
  (r1 in shape[i,rotation_array[i]] /\
  r2 in shape[j,rotation_array[j]] ->
  (x[i] +  circuits[r1,1] <= x[j] \/
  x[j] +  circuits[r2,1] <= x[i]  \/
  y[i] +  circuits[r1,2] <= y[j]  \/
  y[j] +  circuits[r2,2] <= y[i] )));
```

In both constraints the change is implemented using a forall for testing the different rotations.

2.5.4 example instance

Below is reported an example instance, solved with and without rotation. The rotation task is, of course, a larger search but it ends up finding a better solution.

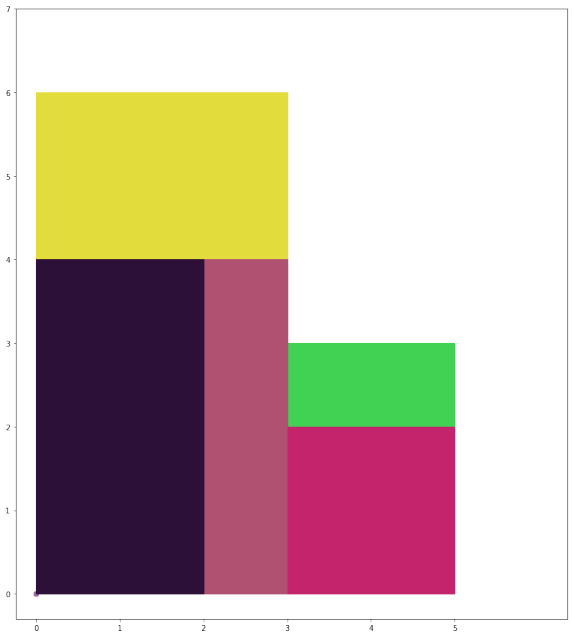

Figure 4: instance solved without the use of rotation (max height 6)

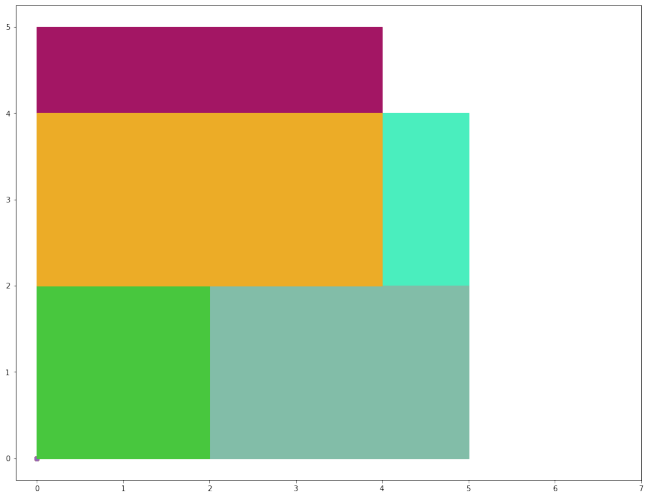

Figure 5: instance solved with rotation enabled (max height 5)

2.5.5 improved rotational model

We approached the task of improving the rotational model.

- diffn_k, One possible improvement of the model can be obtained by using the diffn_k constraint as the non overlap constraint. This global constraint can implement non overlap to multi-dimensional objects, much like our rectangles with multiple orientations.

- geost, This being a well known problem, there is a global constraint that is developed exactly for this task. geost ensures non-overlap for k dimensional objects with different shapes. Possibly, the most challenging thing about using this constraint is to adapt to the data structures required by the geost signature.

2.6 Brief performance report

In the following pages a brief performance report is displayed, showing the solver statistics of instance 15. The first model, called baseline_model.mzn is the simplest possible model, it only has the non overlap constrain (without diffn) and the boundries of the space constraint. The second model, called faster_model.mzn its the latest implementation, it has the two redundant cumulative constraint, the diffn global constraint, optimized search and symmetry breaking.

```
Running baseline_model.mzn on instance 15
```

```
--max height = 22
--width =23
--X coordinates [3, 13, 3, 13, 0, 19, 16, 3, 0, 13, 16, 19, 10, 6, 6, 0]
--Y coordinates [19, 18, 14, 12, 15, 14, 13, 4, 4, 0, 0, 0, 0, 0, 3, 0]
--X dimensions [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 6]
--Y dimensions [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 22, 3, 19, 4]
% time elapsed: 0.56 s
----------
%%%mzn-stat: initTime=0.004159
%%%mzn-stat: solveTime=0.404543
%%%mzn-stat: solutions=1
%%%mzn-stat: variables=513
%%%mzn-stat: propagators=616
%%%mzn-stat: propagations=4572868
%%%mzn-stat: nodes=93013
%%%mzn-stat: failures=46487
%%%mzn-stat: restarts=0
%%%mzn-stat: peakDepth=81
%%%mzn-stat-end
Finished in 574msec

Running faster_model.mzn on instance 15

--max height = 22
--width =23
--X coordinates [13, 19, 13, 19, 0, 3, 16, 3, 0, 19, 16, 13, 10, 6, 6, 0]
--Y coordinates [19, 18, 14, 12, 15, 14, 13, 4, 4, 0, 0, 0, 0, 19, 0, 0]
--X dimensions [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 6]
--Y dimensions [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 22, 3, 19, 4]
% time elapsed: 0.14 s
----------
%%%mzn-stat: initTime=0.000431
%%%mzn-stat: solveTime=0.004266
%%%mzn-stat: solutions=1
%%%mzn-stat: variables=33
%%%mzn-stat: propagators=21
%%%mzn-stat: propagations=2161
%%%mzn-stat: nodes=793
```

```
%%%mzn-stat: failures=378
%%%mzn-stat: restarts=0
%%%mzn-stat: peakDepth=45
%%%mzn-stat-end
Finished in 155msec
```

As we can see from the report, the amount of propagations dramatically change, from 4572868 in the baseline model to 2161 in the faster model. Similarly, the number of failures, peak dept and execution time are greatly improved in the final model.

After exhaustive testing, it appears that the biggest improvement to the model is the use of the redundant cumulative constraints.

3 Conclusions

3.1 The discarded matrix model

When we initially approached the problem, we started working on a model that described the fitting space using a matrix, in which every element was assigned to a specific rectangle. This model is sometimes used for modeling tetris instances. This proved being an inefficient solution, computationally more expensive than the widely used coordinate based model. We soon switched to the other model, but we went as far as having a basic model able to solve simple instances. We thought that it could be of interest for the reader to describe here in the conclusion some of the key concept of this alternative representation.

o	o	2
o	3	2
1	1	2
1	1	2

Figure 6: example instance of the matrix model

The model assigned a number to each element of the matrix, with 0 being the empty space and the following numbers assigned progressively to each rectangle. A set of constraint was used to ensure that the number of elements was consistent with the rectangle's definition, and that the shape was correct. Ultimately the model was discarded because of general inefficiency and because most of the Minizinc constraints were difficult to adapt to this variable definition.

```
set of int: w = 1..4;
var int: n;
set of int: h = 1..4;
int: num_obj = 8;
```

```
array[1..num_obj,1..2] of int: objs;

array[w,h] of var 0..num_obj: table;
array[w] of var int: height;

height = [ sum([table[i,j] | j in h]) | i in w ];

constraint alldifferent_except(forall [table[i,0] | i in w], 0..1);
constraint among(num_obj,table,1..num_obj);
constraint alldifferent_except(table,0..0);

solve minimize max(height)
```

the constraint alldifferent_except was used to ensure that the correct number of 1x1 squares were present in the matrix, and the among constraint, among others, was used to ensure that the correct shape of each rectangle was preserved.

References

[1] Richard E. Korf Eric Huang. Optimal rectangle packing: An absolute placement approach. *arxiv*, 2014.

[2] Prof. Peter James Stuckey Prof. Jimmy Ho Man Lee. Advanced modeling for discrete optimization, coursera. the university of melbourne & the chinese university of hong kong.

[3] Peter J. Stuckey. *Principles and Practice of Constraint Programming*. 2008.