Final Project of Combinatorial Decision Making and Optimization, Module 1

Constraint Programming Model

Federico Cichetti - federico.cichetti@studio.unibo.it

June 14, 2022

Contents

Introduction

This report describes the choices behind the implementation of different solutions for the Very Large Scale Integration (VLSI) problem, presented as a project work for the course of Combinatorial Decision Making and Optimization, Module 1.

This document in particular focuses on a solution developed using *Constraint Programming.* Section 1 proposes a way to provide an initial solution for the problem, Section 2 describes the CP model, its variables, constraints and objective function in depth, Section 3 contains details about our efforts in implementing an efficient search, Section 4 discusses the way the model can be extended to allow for circuit rotations, Section 5 describes some details about implementing the model and additional extensions and, finally, Section 6 presents the results we have obtained using the implemented model.

1 Initial Solution

We implemented a naive algorithm to obtain an initial solution for the problem that it is far from optimal but still respects all constraints. The algorithm is summarized below:

- We have an array of circuits $c^1, \dots, c^n$, each having a width c^i_w and a height c^i_h. When we say that a circuit *is placed* on the board at coordinates (c^i_x, c^i_y), we mean that its bottom-left corner is at that coordinate and the circuit extends up to (c^i_{x+w}, c^i_{y+h}).
- Then, the input circuits are selected *in order of size* and placed on a single row for as long as it's possible, basically placing circuit c^i in position $(c^{i-1}_{x+w}\ , c^{i-1}_y)$
- When circuit c^{i+1} cannot be placed on the same row as c^i anymore (eg. the width limit would be exceeded by placing a circuit there) c^{i+1}_y is set to the highest row that has been reached yet. For instance, if 3 circuits of height 6, 12 and 2 have been placed on the first row, the following circuits will be placed on row 12.

This algorithm generates a simple solution like the one in Figure 1.

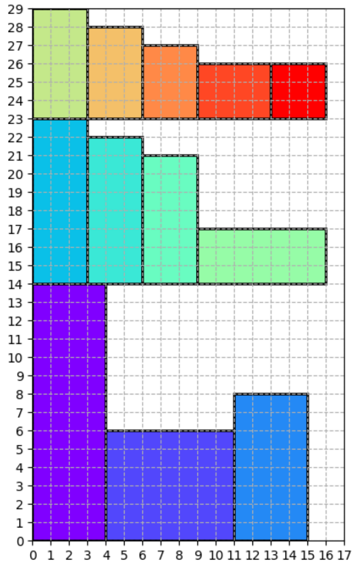


Figure 1: Initial solution for one of the provided instances.

The idea behind using this initial solution is that it could be used as a *warm start* mechanism by the model. This means that rather than searching over the entire search space to create a solution from scratch, we provide an initial good point and ask the solver to make it better.

Furthermore, the search space is explicitly reduced by the fact that this solution provides an *upper bound* to h, the height of the plate. An even more naive upper bound could be obtained by summing together all circuits' heights, as if the initial solution was made of circuits piled up one on top of each other. Our algorithm instead provides a more *compact initial solution* and reduces the initial height, which in turn means automatically discarding all acceptable solutions with higher h that would have slowed down search.

In practice, adding this simple initial solution made us able to solve many more additional problems before the 5 minutes time-out and reduced almost all instances' solving times. Additionally, the overhead for building the initial solution is negligible (2×10^{-3} s for instance 40, which is the largest provided instance).

2 Model

We built a simple but powerful constraint programming model to solve the instances of the problem efficiently.

2.1 Variables

All inputs and variables to the model can be represented as integer numbers (or arrays of integer numbers). The inputs are:

- w: The width of the plate, which is fixed.
- n: The number of circuits to be placed.
- $measures$: A 2D $n \times 2$ array containing width and height of each circuit. We will often use its columns in constraints, so we name them ws (widths) and hs (heights).
- $initx$: A 1D array containing the x-coordinates of the bottom-left corner provided by the initial solution algorithm.
- $inity$: Same, but for y-coordinates.

Additionally, we will use $C = [c_1, \ldots, c_n]$ to represent the set of circuits.
The main variables we are interested in finding values for are:

- h: The height of the plate. Its domain is $lowh <= h <= maxh$, where:
  - $lowh$ is obtained by summing all circuits' areas and dividing by w: $lowh = \lfloor \frac{\sum_{c \in C}(ws_c hs_c)}{w} \rfloor$. It represents the optimal height when there are no empty spaces between circuits.
  - $maxh$ can be obtained as the maximum of the heights reached in the initial solution: $maxh = \max\limits_{c \in C}(inity_c + hs_c)$.
- $xpos$: The 1D array of x-coordinates of the bottom-left corner for each of the $|C|$ circuits. Domain: $\forall c \in C, 0 <= xpos_c <= w$
- $ypos$: The 1D array of y-coordinates of the bottom-left corner for each of the $|C|$ circuits. Domain: $\forall c \in C, 0 <= ypos_c <= maxh$

2.2 Constraints

The main constraints of our model are related to two aspects in the positioning of the blocks:

- The circuits should be placed *entirely* within the w and h boundaries of the board.
- The circuits should not overlap.

The first requirement can be easily expressed with these constraint:

$$\forall c \in C, xpos_c + ws_c <= w;$$

$$\forall c \in C, ypos_c + hs_c <= h$$

The second requirement has a *naive* translation into the following constraint:

$$\begin{gathered}\forall c1, c2 \in C, c1 < c2,\\ xpos_{c1} + ws_{c1} <= xpos_{c2} \lor\\ xpos_{c2} + ws_{c2} <= xpos_{c1} \lor\\ ypos_{c1} + hs_{c1} <= ypos_{c2} \lor\\ ypos_{c2} + hs_{c2} <= ypos_{c1}\end{gathered}$$

We did not use this constraint because we know that $\lor$ does not propagate well (to reach a failure all branches should become false) and furthermore we only check a pair of circuits at a time. Instead, we used global constraints, as described in the next section.

2.2.1 Global Constraints

We enforced the first group of constraints adding two *cumulative* constraints. The cumulative global constraint is used to constrain the usage of a shared resource. Usually, the main agents at play with this constraint are tasks with a starting time, a duration and a resource requirement to be scheduled in an executor with a certain capacity. We use it making the following analogy:

- Tasks are circuits
- The width/height of the plate is the capacity
- The starting time of the task is the assigned *ypos*/*xpos*.
- The duration is the height/width of the circuit
- The resource requirement of the task is the width/height of the circuit

We basically ask the solver to place circuits so that on any row or column circuits never occupy more than w/h spaces:

$$cumulative(ypos, hs, ws, w);$$

$$cumulative(xpos, ws, hs, h)$$

The no-overlap constraint was also implemented using a global constraint that replaces the naive decomposition expressed in the previous section. The *diffn* global constraint is basically a 2D *noOverlap*, where we ask that boxes at a certain position and with certain measures do not intersect:

$$diffn(xpos, ypos, ws, hs)$$

2.2.2 Dual View

We created a dual view for the problem so that we could easily define some additional constraints to help propagation. We *flatten* the board translating the 2D (x, y) coordinates of the circuits into 1D positions. The mapping is done by the following *channeling constraint*: $\forall c \in C, translpos_c = ypos_c \times w + xpos_c$, with the domain of *translpos* being between 0 and $(maxh + 1) \times w$.

With this dual view we can easily pose the following constraints:

$$alldifferent(translpos)$$

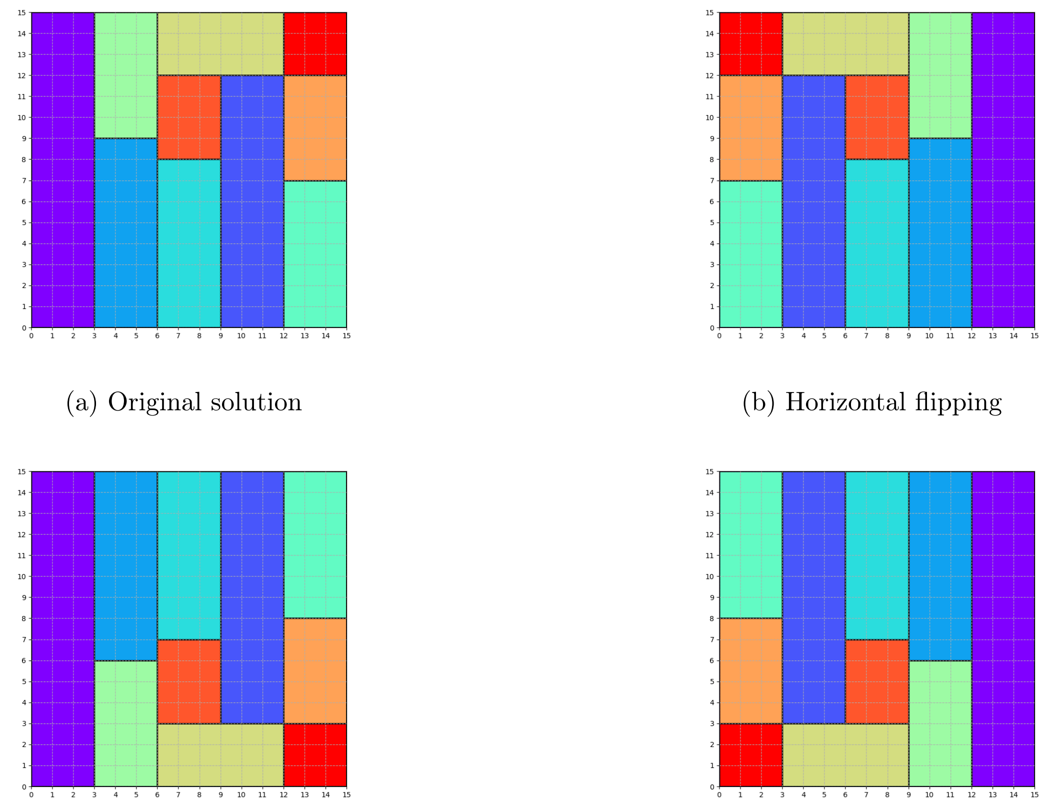
which is an *implied constraint* asking that no circuit is placed in the same position as another, and:

$$\sum_{c \in C} (translpos_c = 0) = 1$$

which is a *meta-constraint* asking that one and only one circuit is placed at position (0,0), as it always makes sense to have a circuit at the origin for any compact solution.

2.2.3 Symmetry Breaking Constraints

Our model has some kinds of *symmetry* that we need to eliminate. We can observe that any solution can be *flipped* vertically, horizontally or on both axes to generate other viable solutions. We cannot say anything about rotations, because, in a general solution, w is not necessarily equal to h. The detected symmetries are shown in Figure 2.



(a) Original solution

(b) Horizontal flipping

(c) Vertical flipping

(d) Flipping on both axes

Figure 2: Solution symmetries. Rotations do not always produce legal solutions, because the board is not necessarily a square

We can observe that in a horizontal flipping, block c at position $(xpos_c, ypos_c)$ moves at position $(w - xpos_c - ws_c, ypos_c)$, while in a vertical flipping, the same block moves at position $(xpos_c, h - ypos_c - hs_c)$. Flipping in both axes maps circuit c to position $(w - xpos_c - ws_c, h - ypos_c - hs_c)$.

We can easily eliminate these symmetries by posing the following *symmetry breaking constraints*, exploiting the dual view we created previously and the *lexical ordering* constraint:

$$lex \leq (translpos, [ypos_c \times w + (w - xpos_c - ws_c) | c \in C])$$
$$lex \leq (translpos, [(h - ypos_c - hs_c) \times w + xpos_c | c \in C])$$
$$lex \leq (translpos, [(h - ypos_c - hs_c) \times w + (w - xpos_c - ws_c) | c \in C])$$

3 Search

The objective of the search algorithm is to *minimize* h, the total height of the board. h is naturally defined as the maximum between the heights that have been reached by the circuits, therefore we add the following constraint and directly set h as the objective function.

$$h = \max_{c \in C}(ypos_c + hs_c)$$

For search, we chose to use the `Gecode` solver, which we tried to help as much as possible with some tricks and a customized search annotation.

First of all, before loading the instance to be solved, we use the same Python script we wrote for creating the initial solution to also *sort circuits by their area*, so that the circuit that occupies the largest amount of space is C_0 and the smallest circuit is C_n. This is done because we can then ask the solver to decide a position for the circuits in that precise order. The underlying expectation is that deciding the placement of the largest circuits as soon as possible, and only when the largest portion of the board has been fixed adding the smallest circuits to fill the gaps is a sensible heuristic.

Furthermore, we ask the solver to assign the *minimum value* to the chosen circuit, meaning that we effectively try to create a *compact* solution and that the largest circuits will probably be close to the bottom-left corner of the board.

Additionally, we try to add randomness to the choice process by *restarting* the search following the *Luby sequence* with a scale of 250. Since we have an initial solution and restarts, we can also use a simple *Large Neighborhood Search* strategy (`relax_and_reconstruct`) that forces the solver to start from the given solution and at each restart fixes 50% of the positions only allowing updates to the other half.

The full search annotations is the following:

```
solve :: seq_search([
    int_search(ypos, input_order, indomain_min),
    int_search(xpos, input_order, indomain_min),
]) :: restart_luby(250) ::
    relax_and_reconstruct(translpos, 50,
        [ inity[c] * w + initx[c] | c in CIRCUITS ])
    minimize h;
```

4 Rotation Extension

Rotating a circuit simply means that we swap its width with its height and vice-versa. By design, circuits cannot be rotated in the model we have described until now, because ws and hs are arrays of constants. We implemented a model that is able to handle rotation of circuits with minimal changes with respect to what we previously described.

Even though we need to maintain the *measures* variable, we are not able to tell a-priori which of the two values per row will be used as width or height of that circuit. Still, we need to be able to assign a width and a height to a circuit for all other constraints to work. The solution is to replace the hs and ws constant arrays with two arrays of integer *variables* that should represent the current choice of width/height for the circuits: *currenths* and *currentws*. The largest value in the *measures* array ($max\_measure = \max(measures)$) is used as an upper bound for *currenths*, while for *currentws* we use $\min(max\_measure, w)$ to avoid having the ability to place circuits that exceed the width limit. The lower bound for the values of both arrays is 1.

We need to constrain *currenths* and *currentws* to pick their values from the corresponding row in the *measures* matrix, as well as constraining the chosen values from that row to be different. This is done with the following constraint:

$$\begin{gathered}\forall c \in C, (currenths_c \in \{measures_{c,1}, measures_{c,2}\}) \wedge \\ (currentws_c \in \{measures_{c,1}, measures_{c,2}\}) \wedge \\ (currenths_c = measures_{c,1} \leftrightarrow currentws_c = measures_{c,2}) \wedge \\ (currenths_c = measures_{c,2} \leftrightarrow currentws_c = measures_{c,1})\end{gathered}$$

Apart from adding the two arrays of variables paired with this constraint and replacing all instances of *hs* and *ws* in the old model's constraints with *currenths* and *currentws*, there are no other changes to the previously described model.

The search annotation was simply updated so that the first choice we force the solver to do is to randomly choose what rotation of the circuit it wants to use.

```
solve :: seq_search([
    int_search(currenths, input_order, indomain_random)
    int_search(ypos, input_order, indomain_min),
    int_search(xpos, input_order, indomain_min),
]) :: ...
```

With this simple addition, we managed to solve 20 of the provided instances, and rotating circuits with respect to their original description was a widely adopted choice by the solver.

5 Implementation

The model has been implemented in the MiniZinc constraint modeling language, but some important additional components have been written in Python. In particular, we implemented a *launcher* and a small library of *utility classes and functions* that are used to simplify the interaction with the model.

The launcher program is able to:

- Load a user-specified model and solver, so that different combinations of constraints/search annotations can be quickly explored.
- Load one or even all of the provided instances (`.txt` text files) and transform them into `ProblemInstance` objects, which are high-level objects containing the particular instances' descriptions and a method to write them as `.dzn` files (the input format used by MiniZinc).
- Solve the instance and provide informative output, both on the terminal and as a well-formatted text file, as well as manage all possible outcomes (optimal solution, time-outs or unfeasible problems).
- Additionally *show* a visual representation of the obtained solution, like the ones we have been using throughout the report.

The initial solution algorithm and the sorting of circuits by their size were also implemented in Python as optional intermediate steps between the loading of the instance and the actual solving procedure.

6 Results

We managed to solve about 31 of the provided instances within the 5 minutes limit (depending on randomness of search and hardware) using our classic model. The model that allows circuit rotations solves up to 20 instances instead. A gap between the two results had to be expected since the rotation model has a larger search space (widths and heights of the circuits are variables rather than constants).

Table 1 shows the results and solution times we obtained for each of the 40 instances using our models.

Instance	Solving Time (s)	Solved	Solving Time (rot., s)	Solved (rot.)
1	0.304	True	0.348999	True
2	0.548999	True	0.580518	True
3	0.549	True	1.148632	True
4	0.627	True	0.540001	True
5	0.656	True	0.632001	True
6	0.560999	True	0.551	True
7	0.563997	True	0.554	True
8	0.587999	True	0.559999	True
9	0.591	True	0.695	True
10	0.682001	True	0.996002	True
11	44.331056	True	16.865054	True
12	0.846	True	12.361999	True
13	0.944998	True	1.704	True
14	0.854	True	14.830001	True
15	0.846001	True	4.856	True
16	>300	False	55.874553	True
17	2.825997	True	43.170001	True
18	0.897001	True	>300	False
19	6.579	True	>300	False
20	8.870002	True	>300	False
21	4.116002	True	>300	False
22	16.018496	True	>300	False
23	1.078998	True	>300	False
24	2.326	True	27.119998	True
25	>300	False	>300	False
26	21.494001	True	>300	False
27	50.65557	True	>300	False
28	0.865003	True	>300	False
29	61.592192	True	>300	False
30	>300	False	>300	False
31	1.894999	True	>300	False
32	>300	False	>300	False
33	17.066003	True	>300	False
34	2.316002	True	>300	False
35	>300	False	13.025189	True
36	2.476996	True	126.130439	True
37	>300	False	>300	False
38	>300	False	>300	False
39	>300	False	>300	False
40	>300	False	>300	False

Table 1: Solving times and optimality status achieved by our models on the 40 instances of the problem. Times don't include the overhead of the Python launcher nor the time for finding an initial solution, but in most cases they are negligible.