Project Work in Combinatorial Decision Making and Optimization, Module 1

Constraint Programming

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

February 12, 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:

- The input circuits are selected in order and placed on a single row as long as it's possible, basically placing circuit c^i in position $(c_x^{i-1} + c_w^{i-1}, c_y^{i-1})$
- 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_y^{i+1} 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.

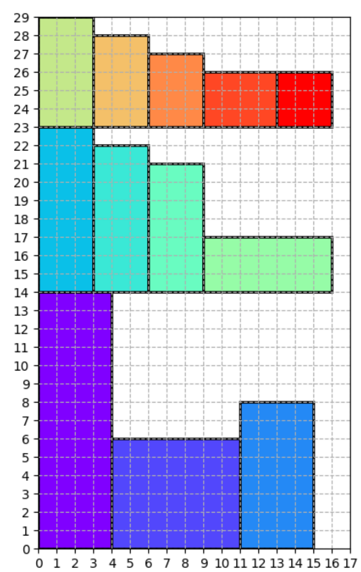


Figure 1: Initial solution for one of the provided instances.

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 slightly modify it until it gets 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, the largest).

2 Model

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

2.1 Variables

The inputs to the model are:

- w: The width of the plate, which is fixed.
- n: The number of circuits to be placed.
- $measures$: A 2D $c \times 2$ 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 $0 <= h <= maxh$, where $maxh$ is an intermediate variable that can be obtained as $maxh = \max\limits_{c \in C}(inity_c + hs_c)$
- $xpos$: 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:

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

We did not use this constraint because we know that $\vee$ 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 a global constraint 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 placed in a resource 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:

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

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

This pair of cumulative constraints also has the benefit of implicitly producing solutions where circuits don't overlap. Nevertheless, we make the no-overlap requirement explicit by adding another global constraints that replaces the naive constraint 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 $(maxh + 1) \times w$.

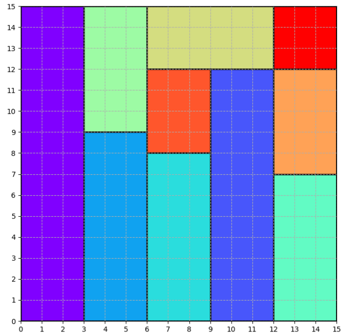
With this dual view we can easily pose the following constraints:

$$alldifferent(translpos)$$

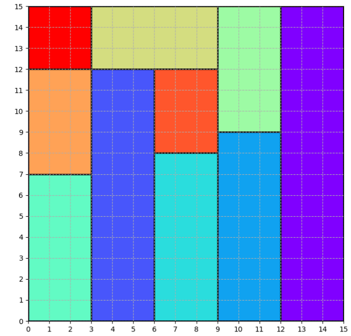
which is a redundant constraint asking that no circuit is placed in the same position as another, and:

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

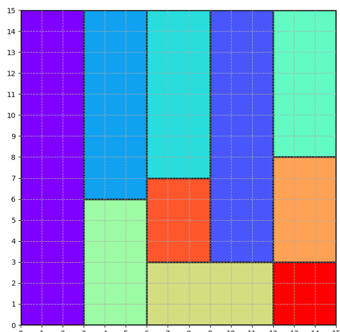
which is a *meta-constraint* asking that one and only one circuit is placed at position (0,0), as it always makes sense to have a circuit at the origin for any compact solution.



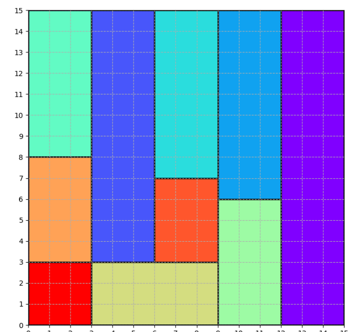
(a) Original solution



(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

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. This is not true for rotations, because, in any solution, w is not necessarily equal to h. The symmetries are shown in Figure 2.

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 $(xpos_c, h - ypos_c - hs_x)$. 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:

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

2.3 Objective Function

The goal of the CP solver is to *minimize* h, so a naive solution would be to directly use h as objective function. The problem with this formulation is that there are a lot of sub-optimal dispositions in which h remains the same, so constraints on h might not propagate much during search.

We can instead use the notion of *compactness* and *minimize the number of empty spaces* in the plate. The optimal solution does not change, because the solution with lower h is also the most *compact*, but the number of empty spaces changes more frequently during search than simply h.

We define an additional integer variable *blanks* and constrain its value to be:

$$blanks = w \times h - \sum_{c \in C} ws_c \times hs_c$$

Then, *blanks* becomes the variable to be minimized. We additionally require that $blanks >= 0$ to fix an explicit lower bound.

3 Search

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 value, meaning that we effectively try to create a *compact* solution and that the largest circuits will probably be at the bottom left 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 blanks;
```

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 constants. Therefore, 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:

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

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 17 of the provided instances, and rotating circuits with respect to their original description was a widely adopted choice by the model.

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 between 27 and 30 of the provided instances (depending on the hardware and randomness of search) within the 5 minutes limit using our classic model. The model that allows circuit rotations solves between 14 and 17 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	0.314999	True	0.860949	True
2	0.543002	True	0.554001	True
3	0.554	True	0.552001	True
4	0.585001	True	0.550001	True
5	0.565	True	0.586	True
6	0.572999	True	0.548002	True
7	0.560998	True	0.550001	True
8	0.574999	True	0.645999	True
9	0.567522	True	0.989469	True
10	0.637085	True	5.923334	True
11	4.731	True	4.483126	True
12	0.565001	True	36.857086	True
13	0.711002	True	10.185666	True
14	0.564	True	23.717034	True
15	0.646998	True	5.206	True
16	300.360417	False	300.414715	False
17	0.620003	True	300.383291	False
18	3.114	True	300.39198	False
19	6.149	True	300.76958	False
20	300.37129	False	300.374177	False
21	33.893002	True	300.388324	False
22	15.969997	True	300.395068	False
23	151.479982	True	300.385231	False
24	10.944999	True	300.387171	False
25	300.371578	False	300.795229	False
26	300.37106	False	300.398054	False
27	9.151057	True	300.393979	False
28	4.866002	True	300.39154	False
29	300.371459	False	300.38684	False
30	300.382932	False	300.40579	False
31	1.705999	True	126.084613	True
32	300.38466	False	300.407867	False
33	2.581002	True	300.394137	False
34	300.372767	False	300.409911	False
35	300.378607	False	252.30406	True
36	2.454001	True	300.385843	False
37	300.393666	False	300.394756	False
38	300.391297	False	300.403336	False
39	300.39428	False	300.409642	False
40	300.533569	False	300.560193	False

Table 1: Results obtained on the 40 instances of the problem. All times don't include the overhead of the Python launcher nor the time for finding an initial solution, but they are negligible.