ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CDMO Module - 01

(Combinatorial Decision Making and Optimization)

Project Report

Group Members

Davide Mercanti (davide.mercanti@studio.unibo.it)

Faisal Ramzan (Faisal.ramzan@studio.unibo.it)

Index

Introduction to the Problem and basic structure of the project

Purpose: VLSI (**very large-scale integration**) - the integration of circuits into a silicon chip following an efficient layout.

The idea is the one of developing a silicon chip composed by the greatest number of rectangular circuits that can possibly fit in that area (to make the device that use the chip more powerful and efficient).

Problem design: This is formalized by a minimization problem whose objective function is the HEIGHT of the chip (given that the width is assigned).

The chip design will be such that each circuit is placed somewhere in the space with fixed orientation with respect to the other (circuits can't be rotated). Then a version of the problem that allows rotations is investigated.
From now on we will refer to the basic problem with VLSI.

Format of instance: The instance representing a specific VLSI problem is a textual file containing, in order: the predefined width (integer), the number of circuits to be placed and the dimension of each circuit (x, y) as integers. For example:

```
8                   # This is the width of silicon plate.
4                   # Number of circuits
# x y
3 3
3 5
5 3
5 5
```

Solutions: In order to place the circuits on the silicon plate in their x and y position (that refers to bottom-left), two circuits can't overlap with each other, and they must fit in the chip boundaries.

The problem has been solved with both constraint programming techniques (CP) and satisfiability techniques (SAT).

For the implementation of VLSI, we developed some Python code (*main_cp.py* for CP, *main_sat.py* for SAT) that is responsible for fetching the instance files automatically from the desired folder, managing the time duration of the solution process (not necessarly the sole time needed by the solver) and for displaying the output textually and graphically.
The actual solution is found thanks to:

- for the CP part, a model in Minizinc that is loaded using Python APIs and solved by Minizinc solvers (Gecode and Chuffed have been tried);
- for the SAT part, a model for the z3 solver, expressed with z3 Python APIs.

Common Python Files: some utility functions are available for both the CP and SAT modules in the folder **util**. Everything that is used by both the *main* files is placed here. Some global constants are available in the **commons.py** file in the root directory.

Output: The output of the execution of the instances can be given into two ways: first it is generated in textual form in the desired output folder (in a format similar to the input file with the addition of the positions of the circuits on the plate). After that, the same solution can be displayed graphically, thanks to the *DrawSvg* Python library (see the file: util/**disp_sol.py**)

Project Work - CP

The Minizinc model is responsible for satisfying the problem constraints.
The use of global constraints (for efficient propagation) and symmetry breaking will be discussed in the following sections.

Variables, problem constraints and objective function (1)

For the implementation of the VLSI problem, we encoded the rectangular silicon plate with these properties: a given width (W), the height (H) to be determined and a certain number N of circuits. We know we needed to place circuits at a certain width and height (respectively X and Y in our coordinate system), so we used two arrays (one for each dimension).
Note that the flattening of two-dimensional array such as
`array[1..N,1..N] of int: dim;`
into two separate arrays allows for a cleaner encoding and a slightly more efficient solution process.

```
int: W;  % width
int: N;  % n. of circuits to be placed

% for each i-th module, its x and y dimensions:
array[1..N] of int: x_dim;
array[1..N] of int: y_dim;

% for each i-th module, its x and y positions:
array[1..N] of var 0..W-1: x_pos;
array[1..N] of var 0..sum(y_dim): y_pos;
```

Our goal and objective function of the problem is to minimize the height of the plate. The domain of H can be bounded (for efficiency) to the height obtained by piling the circuits, given that the best solution will be always better (or equal) than this one.

```
var 1..sum(y_dim): H = max([y_pos[i]+y_dim[i] | i in 1..N]);
solve minimize H;
```

Constraints that restrict the horizontal width of plate (2)

In this step we tried to use constraints that can bound the circuits within the available width (and the chosen height) of the plate.
Here we show two constraints that can be used for such bounding problem; the first bounds the rows to W, the second bounds the columns to H.

```
constraint forall (k in 1..H) (
  sum ([x_dim[i] | i in 1..N where y_pos[i]<k /\ k<=y_dim[i]+y_pos[i] ])
<= W
);
constraint forall (k in 1..W) (
  sum ([y_dim[i] | i in 1..N where x_pos[i]<k /\ k<=x_dim[i]+x_pos[i] ])
<= H
);
```

We have found that the first constraint increase the search time, while the second decreases it, therefore only the latter has been included in the model. This behaviour can be probabily explained by the fact that H is unknow during the constraints check until the very end and so the solver has to check a big portion of the domain of H (`0..sum(y_dim)` or even `int`, in the first version).
This does not happen for W, that is bounded to a low value from the very beginning.

The first model

After this we will show some of the statistic of the very first model with regards to efficiency and performance.

Since in this first model we defined constraints that apply restriction on the domains and an explicit formulation of the *diffn* constraint, there is no overlap between the circuits and the width limitation is also satisfied. Note that, initially, the height was trivially fixed to reduce the search space.

```
int: H = 30;
% /// OMISSIS ///

% domain restrictions:
constraint forall (i in 1..N) (x_pos[i]>=0);
constraint forall (i in 1..N) (y_pos[i]>=0);
constraint forall (i in 1..N) (x_pos[i]+dims[i,x]<=W);
constraint forall (i in 1..N) (y_pos[i]+dims[i,y]<=H);

% non-overlapping (both axis) [C1]
constraint forall (i in 1..N, j in 1..N where i!=j) (
    (x_pos[i]<x_pos[j] /\ (x_pos[i] + dims[i,x] <= x_pos[j]))
    \/
    (y_pos[i]<y_pos[j] /\ (y_pos[i] + dims[i,y] <= y_pos[j]))
    \/
    (x_pos[i]>=x_pos[j] /\ (x_pos[j] + dims[j,x] <= x_pos[i]))
    \/
    (y_pos[i]>=y_pos[j] /\ (y_pos[j] + dims[j,y] <= y_pos[i]))
);
```

```
% no overwidth (except for holes)
constraint forall (k in 1..H) (
    sum ( [dims[i,x] | i in 1..N where y_pos[i]<k /\ k<=dims[i,y]
+y_pos[i] ] ) <= W
);

solve minimize max([y_pos[i]+dims[i,y] | i in 1..N]);
```

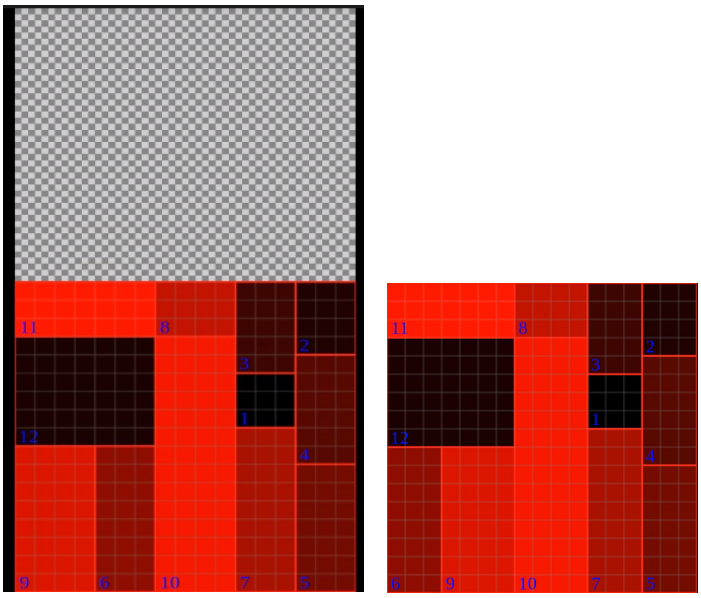
This model already works well on the first instances:

```
(5224) $ python3 main.py
Solution found for instance  1 in   0.3 secs.
  Generating image... done
Solution found for instance  2 in   0.4 secs.
  Generating image... done
Solution found for instance  3 in   0.4 secs.
  Generating image... done
Solution found for instance  4 in   0.4 secs.
  Generating image... done
Solution found for instance  5 in   0.4 secs.
  Generating image... done
Solution found for instance  6 in   0.5 secs.
  Generating image... done
Solution found for instance  7 in   0.6 secs.
  Generating image... done
Solution found for instance  8 in   0.5 secs.
  Generating image... done
Solution found for instance  9 in   0.5 secs.
  Generating image... done
Solution found for instance 10 in   1.9 secs.
```

The output of instance 10 is reported below.

The subsequent step was the definition of a (properly bounded) objective function:
`var 1..sum(y_dim): H = max([y_pos[i]+y_dim[i] | i in 1..N]);`
so that the solver can minimize the height of the plat in this regard:
`solve minimize H;`

The outputs for instance 10 of the two versions of the model were:



Use of Global Constraints (3)

Since in the previous model we defined explicit constraints that are already defined in Minizinc library, for a better (possibly more efficient) result we need to use these global constraints.

```
include "globals.mzn";
% /// OMISSIS ///
% The constraint prev. marked with C1 is now replaced with:
constraint diffn_nonstrict(x_pos, y_pos, x_dim, y_dim);
```

Surprisingly, this replacement did not increase significantly the execution time. Nonetheless, readability is better.

Conversely, an efficiency gain was obtained using the following (implied) global constraint:

```
constraint cumulative(x_pos, x_dim, y_dim, H);
```

This scheduling constraint, in its form `cumulative(s, d, r, b)`, enforces that "a set of tasks given by start times `s` , durations `d` , and resource requirements `r` , never require more than a global resource bound `b` at any one time." The height is interpreted as a resource here.

Symmetry breaking (4)

Given that the chip is rectangular, it is evident that if we geometrically reflect a solution on its x or y median axis we get another valid solution. This can be encoded as:

```
constraint lex_less(x_pos, [W-x_pos[i] | i in 1..N]);
constraint lex_less(y_pos, [H-y_pos[i] | i in 1..N]);
```

Again, the introduction of these constraints did not decrease significantly the time to find the solutions.

Another easy symmetry exists. Let's consider:
x := array of x-positions
y := array of y-positions
xd := array of x-dimensions
yd := array of y-dimensions
Any permutation over k of list of tuples *<x_k, y_k, xd_k, yd_k>* is an equivalent solution.

Although our model only returns the two arrays of positions as a result, for the positions to be meaningful these arrays containing the dimensions of each block must be paired with the ones containing the dimensions. These are considered fixed from

the beginning. Therefore the symmetry that exists among the said permutations of the variables is already enforced by the model structure.

Search (5)

To improve search efficiency we have used **int_search** for further specify how the search had to be carried out.

```
int_search(<variables>, <varchoice>, <constrainchoice>)
```

We have tried, in a very empirical way, different combinations of variable choice strategies and value choice strategies to check the efficiency of the combination. We have also decided to vary the solver, using both **Gecode and Chuffed.**
The results are shown in APPENDIX 1. At the end we have selected Gecode, which allows a better customization (even if Chuffed is generally faster in non-parallel mode).

The best combination appeard to be Gecode with **dom_w_deg** for variable selection and **indomain_random** for value selection.
Gecode was always used in a "parallel mode", by setting the parameter "processes" to 4, after a small experiment that we have done had determined that the best results are obtained with a number of processes close to the one of cores in the processor (see below).

Comparison for Gecode (on a 4-cores machine) with 90 s timeout, using *int_search(x_pos, input_order, indomain_min)*

[TIME in SEC.]		
Instance n. --->	11	15 *
n. of processes		
1	42.8	TIMEOUT
2	26.9	0.7
3	12.8	0.8
4	15.7	1.5
5	18.0	1.3
10	35.3 - 3.1 **	1.9
15	53.8	3.7
20	70.1	4.3

(*) this weird behaviour may suggest that every process simultaneously explores different paths, not just portions of the same path. How this is implemented both in randomized and non-randomized searches is unclear. It seems that with a high number of processes we partially emulate the advantages of a random search + restart.
(**) 3.1 s was obtained in another run, supporting the hypotesis in *.

Two last ways to increase search efficiency were the ones of adding a proper restart policy (that makes sense because we are using a randomized search strategy) and

integrating the *large neighborhood* stratgy. For the restart policy, the Luby-sequence-based seemed very effective.

The final search parameters are the following:

```
solve
  :: int_search(x_pos, dom_w_deg, indomain_random)
  :: restart_luby(400)
  :: relax_and_reconstruct(x_dim, 70)
minimize H;
```

Allow rotations: n*m -> m*n (6, 7)

The easiest way for allowing rotation is the one of considering the dimensions as another output component of the model by creating two arrays of variables that represent the dimensions of each circuit, thus imposing that each <x_dim, y_dim> dimension can be one among: <x_orig_dim, y_orig_dim> and <y_orig_dim, x_orig_dim>. Note that some domains must be changed (such as the one of x_dim and y_dim).

```
array[1..N] of int: x_dim_orig;
array[1..N] of int: y_dim_orig;

% for the domain:
int: LIMIT_dim = max(x_dim_orig ++ y_dim_orig);

% for each i-th module, its x and y dimensions:
array[1..N] of var 0..LIMIT_dim: x_dim;
array[1..N] of var 0..LIMIT_dim: y_dim;

% NEW for this model:
constraint forall (i in 1..N)
  (x_dim[i]==x_dim_orig[i] \/ x_dim[i]==y_dim_orig[i]);
constraint forall (i in 1..N)
  (y_dim[i]==x_dim_orig[i]+y_dim_orig[i]-x_dim[i]);

int: LIMIT_H = sum([max(x_dim_orig[i],y_dim_orig[i]) | i in 1..N]);

% for each i-th module, its x and y positions:
array[1..N] of var 0..W-1: x_pos;
array[1..N] of var 0..LIMIT_H: y_pos;
```

This model has proved itself very inefficient.
Another model has been tried, whose key idea was to insert as input two versions for each block so that the model had to "mask" alternatively one of the two. The performance of this model was similar to the previous one.

With respect to the search model discussed at point (5), the only difference in the search strategy that we have found important is that *relax_and_reconstruct* doesn't improve the search here. On the contrary, the solver is dramatically slackened.

APPENDIX 1

Gecode

- **First combination: dom_w_deg and indomain_random**

Instances	Solver: gecode
1	0.8
2	0.8
3	0.8
4	0.7
5	0.8
6	0.8
7	0.8
8	0.8
9	1.0
10	0.9
11	1.0
12	1.8
13	1.2
14	1.0
15	2.0

- **Second combination: dom_w_deg and indomain_min**

Instances	Solver: gecode
1	0.6
2	0.7
3	0.8
4	0.9
5	0.8
6	0.8
7	0.8
8	0.8
9	0.8
10	0.8
11	Time out (satisfied)
12	2.9
13	22.7
14	1.4
15	1.0

- **Third combination: dom_w_deg and indomain_median**

Instances	Solver: gecode
1	0.7
2	0.7
3	0.7
4	0.7
5	0.8
6	0.9
7	0.9

8	1.0
9	1.6
10	1.9
11	1.2
12	182.4
13	6.2
14	15.6
15	Time out (satisfied)

- **Fourth combination: dom_w_deg and indomain_split**

Instances	**Solver: gecode**
1	0.7
2	0.7
3	0.7
4	0.9
5	0.9
6	0.8
7	0.8
8	0.9
9	0.8
10	0.8
11	1.3
12	1.0
13	0.9
14	1.0
15	1.7

- **Fifth combination: input_order and indomain_min**

Instances	**Solver: gecode**
1	0.7
2	0.7
3	0.7
4	0.7
5	0.8
6	0.7
7	0.7
8	1.0
9	1.5
10	4.5
11	54.9
12	1.7
13	Time out (satisfied)
14	Time out (satisfied)
15	Time out (satisfied)

- **Sixth combination: first_fail and indomain_min**

Instances	**Solver: gecode**
1	0.7
2	0.7
3	0.7

4	0.7
5	0.7
6	0.7
7	0.7
8	0.7
9	0.8
10	0.8
11	Time out (satisfied)
12	1.1
13	12.8
14	1.1
15	1.1

- **Seventh combination: smallest and indomain_min**

Instances	**Solver: gecode**
1	0.7
2	0.7
3	0.8
4	1.0
5	0.7
6	0.8
7	0.8
8	0.8
9	0.7
10	0.9
11	1.0
12	0.9
13	21.5
14	239.7
15	1.0

Chuffed

- **First combination: input_order and indomain_min**

Instances	**Solver: chuffed**
1	1.2
2	0.7
3	0.7
4	0.7
5	2.1
6	11.8
7	22.6
8	timeout
9	Timeout
10	Timeout
11	Timeout
12	Timeout
13	Timeout

14	Timeout
15	timeout

➢ Second combination: Chuffed (default, without any annotation)

Instances	**Solver: chuffed**
1	0.7
2	0.7
3	0.7
4	0.7
5	0.7
6	0.8
7	0.9
8	0.8
9	0.7
10	0.8
11	88.4
12	1.2
13	1.1
14	1.6
15	1.1