Report

Niccolai Lorenzo, Pozzoli Davide

Project report for
Combinatorial Decision Making and Optimization
Module 1
CP

Artificial Intelligence
University of Bologna
Italy

Contents

Introduction

This is the Report for the first module of the Combinatorial Decision Making and Optimization exam for July 2021 for the CP model.

1 Setting

Since the project specification required to take data from a text file - and also to write the results on it - we wrapped Python code around the MiniZinc model. We developed on Python 3.6 and 3.7, so we do not guarantee the correct behaviour on other Python versions. The file `Module1.py` first reads the data from the selected instance txt files (the number of the instance is taken as a keyboard input by the user), then runs the external MiniZinc model (`CP_base.mzn`) on the formatted gathered input. Finally the results of the CP model are written in an output text file as required by the assignment.

The Python code also produces a visual representation of the model's solution with `matplotlib`, this was used mainly for developing purposes and checking the solution validity as advised.

The path for the different input/output operations are set to work for the current folder hierarchy.

The solver in the python code is set to stop automatically after the 300 seconds (5 minutes) time limit. We also used the parameter to set the number of cores to use in parallel during the execution in order to speed up the process, since the machine allowed us to do so.

2 7 points

As advised in the paper describing the project, we are going to tackle the seven points that brought us to the final implementation of the Constraint Programming model.

2.1 Point 1: Variables, Main constraints and objective function

Starting from the variables, we take the values from the input file(one of the instance) and we save them in the corresponding variables. In particular *width* correspond to the first parameter of the txt file, *n_rets* identifies the number of rectangles to place and *sizes* is an array containing the sizes of each rectangle.

```
int: width;

int: n_rets;

set of int: RETS = 1..n_rets;
```

```
array[RETS, 1..2] of int: sizes;
```

The array of pairs *positions* is a variable use to store the coordinates of the left bottom corner of each rectangle inside the solution found.

```
array[RETS, 1..2] of var 0..sum(
   [sizes[i,2]| i in RETS]): positions;
```

The other variable of the model is l, which encodes the height of the board. Here you can also see the bounds of its domain, the minimum is the height of the tallest input circuit, while the maximum is the sum of all the heights of input circuits.

```
int: min_l = max([sizes[i,2] | i in RETS]);

int: max_l = sum([sizes[i,2] | i in RETS]);

var min_l..max_l: l;
```

We then proceed by defining the main constraints.

A predicate has been defined to help us avoid that two rectangle could be one on top of the other.

```
predicate no_overlap(var int:s1,
               int:d1,
               var int:s2,
               int:d2) =
      s1 + d1 <= s2 \/ s2 + d2 <= s1;

constraint forall(i,j in RETS where i != j)
  (no_overlap(positions[i,1], sizes[i,1],
          positions[j,1], sizes[j,1])
      \/
  no_overlap(positions[i,2], sizes[i,2],
          positions[j,2], sizes[j,2]));
```

These two constrains are our first attempt to avoid the overloading of the width.

```
constraint forall(i in RETS)
  (sizes[i,1] + sum([sizes[k,1] | k in RETS where i != k
  /\
  not(no_overlap(positions[i,2],
              sizes[i,2],
              positions[k,2],
              sizes[k,2]))]) <= width);

constraint forall(i in RETS)
   (sizes[i,1]*sizes[i,2] + sum([sizes[k,1] * sizes[k,2] |
```

```
k in RETS where i != k /\
not(no_overlap(positions[i,2],
   sizes[i,2],
   positions[k,2],
   sizes[k,2]))]) <= sizes[i,2] * width);
```

The basic objective function consists in the minimization of the variable that indicates the height of our workspace, in this case l.

```
solve minimize l;
```

2.2 Point 2: Implied constraints

These constraints guarantee that any rectangle is not going to exceed the width in input and the current height.

```
constraint forall(i in RETS)
   (positions[i,2]+sizes[i,2] <= l);

constraint forall(i in RETS)
   (positions[i,1]+sizes[i,1] <= width);
```

Even though the results that we found so far were correct, the solutions were not optimal. With the time limit set and without further optimization, we were not able to solve instances after the 10^{th}.

2.3 Point 3: Global constraints

The first global constraint that we added was the `cumulative` constraint. We first instantiated two one dimensional arrays containing the widths and heights of the input circuits. Then used the `cumulative` to impose that the board dimensions would not be overloaded by the circuits. In fact the first of the two states that at any time the width can not be exceeded by the circuits' width sum (*size1*), and sets the circuits x variables according to their "duration" (*size2*). The second constraint does the same but on the height dimension or the y axis.

```
array[RETS] of int: size1 = [sizes[i,1]| i in RETS];
array[RETS] of int: size2 = [sizes[i,2]| i in RETS];

constraint cumulative([positions[i,2] | i in RETS],
                size2,
                size1,
                width);

constraint cumulative([positions[i,1] | i in RETS],
                size1,
                size2,
```

```
                        l);
```

Another global constraint that we introduced is `diffn`, it replaces the constraint which checked that two rectangles would not overlap in both dimensions.

```
constraint diffn([positions[i,1] | i in RETS],
    [positions[i,2] | i in RETS],
    size1,
    size2);
```

2.4 Point 4: Symmetries

Even if we have seen quite few symmetries in VLSI problem, we found some difficulties in the formalization. For example the found solution could be flipped on the x axis and give an equivalent solution, same by flipping on the y axis and both, this is just 3 more solutions. The most symmetries we detected were not "total", for example a simple symmetry is that given an instance with two identical circuits in every solution they could be swapped. This is how we tried to encode it:

```
constraint forall(i,j in RETS where i<j)(
 if (sizes[j,1] == sizes[i,1] /\
    sizes[j,2] == sizes[i,2])
 then
    let {
          array[RETS,1..2] of var int: s = positions,
          constraint s[j,1] = positions[i,1] /\
                  s[i,1] = positions[j,1] /\
                  s[j,2] = positions[i,2] /\
                  s[i,2] = positions[j,2]
    } in(
          lex_lesseq(array1d(positions), array1d(s))
       )
 else
    1>0
 endif
 );
```

We tried also to avoid that when two circuits are adjacent and have the same size on the adjacent side they are forming a bigger rectangle and therefore they can be swapped.

```
constraint forall(i,j in RETS where i<j)(
  if ((positions[j,1]+sizes[j,1] == positions[i,1]) /\
      (sizes[j,2] == sizes[i,2]))
  then
    let {
        array[RETS,1..2] of var int: s =
```

```
                    positions, constraint s[j,1] = positions[i,1] /\
                                          s[i,1] = positions[j,1]
                } in(
                     lex_lesseq(array1d(positions), array1d(s))
                   )
         else
              1>0
         endif
         );

        constraint forall(i,j in RETS where i<j)(
         if ((positions[j,2]+sizes[j,2] == positions[i,2]) /\
              (sizes[j,1] == sizes[i,1]))
         then
           let {
                array[RETS,1..2] of var int: s =
                          positions, constraint s[j,2] = positions[i,2] /\
                                                s[i,2] = positions[j,2]
                } in(
                     lex_lesseq(array1d(positions), array1d(s))
                   )
         else
              1>0
         endif
         );
```

After introducing these constraint the solving time would not decrease so much so we are not so sure of them working right, in particular the use of the `let` and the `s` local variable may not be set right.

2.5 Point 5: Search

Let we start saying that the first thing we did to improve search was ordering the *sizes* array from the biggest area to the smallest. The first search annotation that we tried to employ was the following:

```
        ann: search_ann = int_search([l] ++ array1d(positions),
                                     input_order,
                                     indomain_min,
                                     complete);

        solve :: search_ann minimize l;
```

The idea is to search on the l first, and then place the circuits giving priority to the y axis, hence the `input_order` annotation on variable order. Then the `indomain_min` annotation for the value choice is due to the fact that for the circuits it will give priority to the space near the low left part and not place them randomly leaving fragmented blank space. With this setting we reached a time of 20 seconds on the 10^{th} instance.

After this we tried to introduce the `first_fail` annotation for variables selection, this led us to resolve 10^{th} instance in 0.7 seconds.

The next step was changing the order of the input array of variables:

```
ann: search_ann = int_search([l] ++
                  [positions[i,2] | i in RETS] ++
                  [positions[i,1] | i in RETS],
                  input_order,
                  indomain_min,
                  complete);

solve :: search_ann minimize l;
```

In this way instead of the `array1d` method we concatenate the array to consider the y position before than the x. Also we returned to `input_order` because l has to be considered before the circuits. With this setting we managed to resolve optimally 28 instances, average resolution time is good (under the minute).

2.6 Point 6: Hypothetical model with rotation allowed

Since CP kind of incorporates propositional logic, the idea would be similar to the approach in SAT. With rotation allowed each circuit can be considered as 2 different instances of such circuit. Only one of these 2 possible circuits will be in the board, this can be done in propositional logic. To do this an array of decision variables for the chosen circuit can be added, of course this means a time overhead.

2.7 Point 7: Hypothetical search with rotation allowed

The decision variables should be considered before the position of the circuit, a variable selection which could do well is `first_fail` since it could detect if a rotation results in a failure. For the value selection `indomain_min` remains one of the best choices.

3 Results

In the zip we put an excel containing all the results for the different search strategies.