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.

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 used Python 3.6 mainly due to compatibility issues, the file `Module1.py` uses the `os` library to read the data from the standard instance txt files, then runs the external MiniZinc model (`CP_base.mzn`) on the gathered input. Finally the results of the CP model are written in an output text file as specified in the specification. The Python code also produces a visual representation of the model's solution with `matplotlib`, this was used mainly for developing purposes as advised.

2 7 points

As advised in the paper describing the project, we are going to described 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 paricular *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);
```

2.3 Results part 1

Even though the results that we found so far were correct, the solutions were not optimal. With the time limit set to 5 minutes, without further optimization, we were not able to solve instances after the 10^{th}.

2.4 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);
```

2.5 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, 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 almost all instances until the 21^{th} in less than 20 seconds.