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 *positions* is a variable use to store the starting position of each rectangle inside the optimal solution found.

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

We then proceed by defining the main constraints.

These are a guarantee that any rectangle is not going to be wider than the width in input and it cannot be placed outside the boundary imposed (totally or partially).

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

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

A predicate has been defined to help us avoid that two rectangle could be place 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]));
```

We tried also to define some constraints in order to improve the solution. Even though the results were correct, they were not optimal and we had to find another way.

```
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 objective functions consist in the minimization of the variable that indicates the height of our workspace, in this case l.

```
var 0..sum([sizes[i,2]| i in RETS]): l =
   max([positions[i,2] + sizes[i,2] | i in RETS]);

solve minimize l;
```