Box Wrapping Problem

Course project

Constraint Programming

Lluís Alemany

May 11, 2018

Contents

1 Introduction

The Box Wrapping Problem (BWP) is easy to formulate: given a list of $N \in \mathbb{N}$ boxes b_i each of dimensions (w_i, l_i), where $w_i \in \mathbb{N}$ and $l_i \in \mathbb{N}$ are the width and length of box b_i respectively, and a roll width $W \in \mathbb{N}$, find the coordinates of the top-left corner of each box $(x_i^{(tl)}, y_i^{(tl)})$ so that the length of roll $L \in \mathbb{N}$ is minimised. We call the set of top-left corners, i.e., the solution, the placement of all the boxes. In this project we consider only the 2-dimensional case, that is, boxes actually have eight sides that have to be wrapped but, to make it simple, we consider that a box is just a rectangle. In addition to this, boxes can be rotated.

An example of an instance for this problem is the following: given the list of boxes each of dimensions $(1, 1)$, $(1, 1)$, $(2, 1)$, $(1, 3)$, and $W = 3$, find the placement of these boxes that minimises the roll length. The optimal solution is clearly the one shown in figure 1.

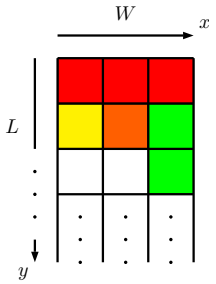


Figure 1: Optimal placement for $W = 3$ and boxes $(1, 1)$, $(1, 1)$, $(2, 1)$, $(1, 3)$.

There are many ways to solve this problem, but in this project we will solve it from three different points of view: constraint programming (see section 3), linear programming (see section**??**), and satisfiability (see section **??**). Since all these approaches require a mathematical modelling of the problem (variables related by constraints), we give, in section 2, a generic model that should describe the problem well enough to solve it with each technology. However, each technology may require some extra variables or constraints.

2 Modelling the problem

In this section we describe the generic mathematical model used to solve the problem. First, the variables (see section 2.1) and then the constraints (see section 2.2).

2.1 Variables

Before formulating each of the constraints, we define first the Boolean variables used:

1. $C_{b,x,y} = 1 \iff$ cell (x, y) is occupied by box b. $C_{b,x,y} \in \mathbb{B}$, $\forall b, x, y$ $1 \leq x \leq W$, $1 \leq y \leq L$, $1 \leq b \leq N$.

2. $X_{b,x,y} = 1 \iff$ the top-left corner of box b is placed at cell (x, y). $X_{b,x,y} \in \mathbb{B}$, $\forall b, x, y$, $1 \leq x \leq W$, $1 \leq y \leq L$, $1 \leq b \leq N$.

3. $R_b \in \mathbb{B}$ indicates the rotation of box b. If $R_b = 0$ then the box is not rotated, that is, the dimensions of the box (width and length) are considered to be those given in the input's order: the bottom-right corner of box b is:

$$(x_b^{(br)}, y_b^{(br)}) = (x_b^{(tl)} + w_b, y_b^{(tl)} + l_b) \tag{1}$$

If $R_b = 1$ then the box is rotated, that is, the bottom-right corner of box b is:

$$(x_b^{(br)}, y_b^{(br)}) = (x_b^{(tl)} + l_b, y_b^{(tl)} + w_b) \tag{2}$$

(the width and length specified in the input are used in "reverse" order), $\forall 1 \leq b \leq N$.

Therefore, our model consists of two 3-dimensional matrices, each containing $N \times W \times L$ Boolean variables, and one array of N Boolean variables, giving us a total of $N(2WL + 1)$ Boolean variables.

2.1.1 Determining the sizes of the matrices

Notice that this model has a slight drawback: using Boolean variables forces us to define two fixed-size matrices for which we need to know the value of L. This is only a minor issue since we can easily determine an upper bound on the length of the roll used: in the worst case we might need to put all boxes "one next to the other" in the roll. Therefore, in order to build the matrices we use the value

$$L = \sum_{b=1}^{N} \max\{w_b, l_b\} \tag{3}$$

Notice that the maximum ensures a solution when the boxes are allowed to rotate: a simpler model can be obtained by not implementing variables R_b (and changing the subsequent constraints so that boxes are not rotated at all) so that an upper bound on L can be obtained by just adding up the lengths of the input boxes, but this does not guarantee a solution to all instances (imagine an instance where at least one box had width $w_b > W$: without rotations this instance does not have a solution).

2.2 Constraints

The general modelling of this problem requires 4 types of constraints appropriate for the variable selection made.

1. Each box must have its top-left corner placed in the roll. This constraint is there to ensure that the placement contains all boxes.

$$\sum_{x=1}^{W} \sum_{y=1}^{L} X_{b,x,y} = 1, \qquad \forall b,\ 1 \leq b \leq N \tag{4}$$

2. No overlapping, that is, no cell of the roll can have more than one box in it.

$$\sum_{b=1}^{N} C_{b,x,y} \leq 1, \qquad \forall x, y,\ 1 \leq x \leq W,\ 1 \leq y \leq L \tag{5}$$

3. Depending on the rotation of the box and its top-left corner's position, a box occupies certain cells of the roll. In order to speed up the solvers, the constraints for square boxes are simpler than those for rectangular boxes (because there is no need to consider a rotation for square boxes). Furthermore, since a box should not be placed where there is no space for it in the roll (see figure 2) these constraints will not be added for these positions. In case it is not rotated, box b cannot be placed in cell (x, y) if $x + w_b > W$ or if $y + l_b > L$. In case it is rotated, if $x + l_b > W$ or if $y + w_b > L$.

- For square boxes:

$$(R_b = 0) \wedge (X_{b,x,y} = 1 \implies C_{b,x+j,y+i} = 1)\,, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N,\ \text{s.t.}\ (w_b = l_b \wedge \\ \quad \forall x, y,\ 1 \leq x \leq W, 1 \leq y \leq L \\ \quad\quad \text{s.t.}\ (x + w_b \leq W \wedge y + l_b \leq L) \\ \quad \forall i, j,\ 0 \leq i < l_b, 0 \leq j < w_b) \end{array} \tag{6}$$

- For rectangular boxes:

$$(R_b = 0 \wedge X_{b,x,y} = 1) \implies C_{b,x+j,y+i} = 1, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N,\ \text{s.t.}\ (w_b \neq l_b \wedge \\ \quad \forall x, y,\ 1 \leq x \leq W, 1 \leq y \leq L \\ \quad\quad \text{s.t.}\ (x + w_b \leq W \wedge y + l_b \leq L) \\ \quad \forall i, j,\ 0 \leq i < l_b, 0 \leq j < w_b) \end{array} \tag{7}$$

$$(R_b = 1 \wedge X_{b,x,y} = 1) \implies C_{b,x+i,y+j} = 1, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N,\ \text{s.t.}\ (w_b \neq l_b \wedge \\ \quad \forall x, y,\ 1 \leq x \leq W, 1 \leq y \leq L \\ \quad\quad \text{s.t.}\ (x + l_b \leq W \wedge y + w_b \leq L) \\ \quad \forall i, j,\ 0 \leq i < l_b, 0 \leq j < w_b) \end{array} \tag{8}$$

The constraints for the square boxes should be read as: for all boxes b that are squares ($w_b = l_b$), take all cells of the roll (x, y) within the corresponding limits $1 \leq x \leq W, 1 \leq y \leq L$ that are candidates for its top-left corner "s.t. $(x + w_b \leq W \wedge y + l_b \leq L)$" and, for each of these cells, impose that the corresponding cells occupied by b according to its dimensions are occupied by it: $\forall i, j,\ 0 \leq i < w_b, 0 \leq j < l_b : X_{b,x,y} = 1 \implies C_{b,x+j,y+i} = 1$. The constraints for rectangular boxes are read in a similar fashion but adding "if the box is not/is rotated" after the "for all rectangular boxes".

(a) A non-rotated box that would be out of bounds if placed in the light-red cell.

(b) A rotated box that would be out of bounds if placed in the light-red cell.

Figure 2: Two situations in which a box would be out of bounds

4. Forbid the placement of a box in certain cells: it is known that, for a given rotation, a box cannot possibly be placed in certain cells of the roll because it would end up out of its bounds. This will stop the solvers from assigning positive values to these cells, that is, assign a box, in the hope that they detect the unfeasibility of an instance as soon as possible. Again, the constraints for square boxes are different from those for rectangular boxes. Moreover, it suffices to forbid the placement of a box's top-left corner in these cells.

- For square boxes:

$$X_{b,x+j,y+i} = 0, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N,\ \text{s.t. } (w_b = l_b \wedge \\ \quad \forall x, y,\ 1 \leq x \leq W, 1 \leq y \leq L \\ \quad\quad \text{s.t. } (x + w_b > W \vee y + l_b > L) \\ \quad \forall i, j,\ 0 \leq i \leq L - y, 0 \leq j \leq W - x) \end{array} \tag{9}$$

- For rectangular boxes:

$$R_b = 0 \implies X_{b,x+j,y+i} = 0, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N,\ \text{s.t. } (w_b \neq l_b \wedge \\ \quad \forall x, y,\ 1 \leq x \leq W, 1 \leq y \leq L \\ \quad\quad \text{s.t. } (x + w_b > W \vee y + l_b > L) \\ \quad \forall i, j,\ 0 \leq i \leq L - y, 0 \leq j \leq W - x) \end{array} \tag{10}$$

$$R_b = 1 \implies X_{b,x+i,y+j} = 0, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N,\ \text{s.t. } (w_b \neq l_b \wedge \\ \quad \forall x, y,\ 1 \leq x \leq W, 1 \leq y \leq L \\ \quad\quad \text{s.t. } (x + l_b > W \vee y + w_b > L) \\ \quad \forall i, j,\ 0 \leq i \leq L - y, 0 \leq j \leq W - x) \end{array} \tag{11}$$

The constraints that forbid placing a box at a certain cell are read similarly to those that allow a box to be placed at a certain cell. In this case, we want those cells (x, y) that are not valid candidates for the top-left corner for a box.

3 Constraint Programming

In this part of the project the problem is solved using the Constraint Programming paradigm. In particular, we used the library Gecode for C++ (version 6.0.0) (see [3]) in order to implement the mathematical model that models the problem and helps us solve it without having to implement our own algorithm.

For this, we used 3 arrays of Boolean variables, each of them containing the variables described in items 1, 2 and 3 in section 2.1. Taking L as the upper bound on the roll's length calculated with equation (3), the arrays are initialised as follows:

1. Array for variables in 1:

```
box_cell = BoolVarArray(*this, N*W*L, 0, 1);
```

2. Array for variables in 2:

```
box_corner = BoolVarArray(*this, N*W*L, 0, 1);
```

3. Array for variables in 3:

```
box_rotated = BoolVarArray(*this, N, 0, 1);
```

Also, the implementation of the constraints is as follows:

1. All boxes must be in the solution (constraint formalised in 1, equation 4), also read as "assign a top-left corner to each box".

```
for (int b = 0; b < N; ++b) {
    rel(*this, sum(box_corner.slice(b*W*L, 1, W*L)) == 1);
}
```

2. Boxes cannot overlap (constraint formalised in 2, constraint 5).

```
for (int b = 0; b < N; ++b) {l
    for (length i = 0; i < L; ++i) {
        for (width j = 0; j < W; ++j) {
            rel(*this, sum(box_cell.slice(b*W*L + i*W + j, W*L, N)) <= 1);
        }
    }
}
```

3. Depending on their rotation, boxes occupy certain cells of the roll (constraint formalised in 3). For those cells (x, y) within bounds for b-th box:

   - For square boxes (constraint 6):

```
for (length i = y; i <= y + b_length - 1; ++i) {
    for (width j = x; j <= x + b_width - 1; ++j) {
        rel(*this,
            (box_corner[b*W*L + y*W + x] == 1)
            >>
            (box_cell[b*W*L + i*W + j] == 1)
        );
    }
}
rel(*this, box_rotated[b] == 0);
```

   - For rectangular boxes (constraints 7 and 8):

```
for (length i = y; i <= y + b_length - 1; ++i) {
    for (width j = x; j <= x + b_width - 1; ++j) {
        rel(*this,
            ((box_rotated[b] == 0) &&
            (box_corner[b*W*L + y*W + x] == 1))
            >>
            (box_cell[b*W*L + i*W + j] == 1)
        );
    }
}

for (length i = y; i <= y + b_width - 1; ++i) {
    for (width j = x; j <= x + b_length - 1; ++j) {
        rel(*this,
            ((box_rotated[b] == 1) &&
            (box_corner[b*W*L + y*W + x] == 1))
            >>
            (box_cell[b*W*L + i*W + j] == 1)
        );
    }
}
```

4. Depending on their rotation, boxes cannot occupy certain cells of the roll (constraint formalised in 4). For those cells (x, y) out of bounds for b-th box:

   - For square boxes (constraint 9):

```
rel(*this, box_corner[b*W*L + y*W + x] == 0);
```

   - For rectangular boxes (constraints 10 and 11):

```
rel(*this,
    (box_rotated[b] == 0)
    >>
    (box_corner[b*W*L + y*W + x] == 0)
);

rel(*this,
    (box_rotated[b] == 1)
    >>
    (box_corner[b*W*L + y*W + x] == 0)
);
```

3.1 Finding the optimum solution

The constraints presented in the previous section only guarantee that if a feasible solution exists then, eventually, it will be found. However, they do not, and can not possibly guarantee, by definition, the optimality of the solution. For this reason, in order to make Gecode look for an optimum solution, we use the *Branch & Bound* solver that needs the implementation of a function, *constrain*, in which we can impose new constraints that can help the solver find better solutions.

```
virtual void constrain(const Space& _b);
```

Figure 3: The header of the function *constrain*

The function *constrain* (see figure 3) is simple to understand: it receives a single parameter of type *Space*, which can be cast to the same type as our solver, and contains the current best solution. Using this object, we can access its arrays of variables and their value(s) and impose new constraints to force Gecode find better solutions.

An easy approach to make Gecode find solutions that use shorter roll length is to forbid placing any box on all those cells with length $\mathcal{L}$, where $\mathcal{L}$ is the shortest length found so far. That is, for all those cells (x, y) such that $y \geq \mathcal{L}$, impose that they cannot contain a box. This way the next solution found, if there is any, will have stricter shorter length. In general, given a roll length $\mathcal{L}$, the new constraints imposed are:

$$C_{b,x,y} = 0, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N \\ \forall x, y,\ 1 \leq x \leq W, \mathcal{L} \leq y \leq L \end{array} \tag{12}$$

The implementation of the constraints in 12 in Gecode is done with the following code:

```
for (int b = 0; b < N; ++b) {
    for (length i = RL - 1; i < L; ++i) {
        for (width j = 0; j < W; ++j) {
            rel(*this, box_cell[b*W*L + i*W + j] == 0);
        }
    }
}
```

where $RL = \mathcal{L}$ can be calculated using the values assigned to the variables in the *Space* object passed as parameter to the function *constrain*.

3.1.1 Heuristics

When using the method described in section 3.1 and when given enough time, Gecode is ensured to eventually be able to find an optimal solution: each solution found using that method will have strictly shorter roll length than the previous. Therefore, eventually, Gecode will find a solution with a series of constraints (limiting the roll length used) that will make the instance infeasible. The solution found before that point is the optimal. However, that will only happen "given enough time", that is, even though Gecode is rather efficient this might mean several hours. In order to tackle this issue, we implemented a number of heuristics that should make Gecode stop as soon as possible.

All the heuristics implemented are very simple. The variable and value selection strategies are kept as simple as possible: for the variables, choose the first unassigned, and for the values, the maximum in the domain of the variable chosen. Since all the variables are Boolean, the value chosen is 1, interpreted as "always assign". In Gecode, this is implemented as follows:

```
branch(*this, box_rotated + box_corner, BOOL_VAR_NONE(), BOOL_VAL_MAX());
```

Now, notice that in the implementation of the constraints of the generic model in Gecode needs an array with the list of boxes from the input. The only thing done as an attempt to speed up the solver's execution time is to rearrange the boxes in this array so that the constraints are ordered in a particular way. For example, we may want the solver to assign larger boxes first, and smaller boxes last. Or simply select each boxes at random[1].

The order is not the only attempt at speeding up the code. Several strategies are "concatenated", that is, executed one after the other, in the following way: after the solver has finished its execution with a particular ordering of the boxes with a solution of roll length $\mathcal{L}_1$, execute the solver again with a different ordering and, instead of using the upper bound on the maximum length L defined in equation 3, use the minimum of the two (see equation 13) which means that the matrices for variables 1 and 2 will become smaller with every step.

$$L' = \min\{L, \mathcal{L}_1\} \tag{13}$$

Algorithmically, this can be formalised in algorithm 1.

Algorithm 1: Generic heuristic

```
1  L := UpperBoundRollLength(B), using equation 3
2  B' := sort boxes in B increasingly by area
3  S_1 := Solve(B', L − 1)
4
5  L := min{length(S_1), L}
6  B' := sort boxes in B decreasingly by area
7  S_2 := Solve(B', L − 1)
8
9  ...
10
11 L := min{length(S_{n−1}), L}
12 B' := sort boxes in B using some other criteria
13 S_n := Solve(B', L − 1)
14 return S_n
```

where length(S_i) is the roll's length used in solution S_i, and n is the limit on the amount of heuristics concatenated. Needless to say that, if S_i is an optimal solution, the solver should not take much time in concluding that it can not possible find a solution at step $i + 1$, basically due to not being enough variables. Each call to Solve executes the solver once.

Several heuristics were studied:

1. Sort increasingly by area + increasingly by width.
2. Sort decreasingly by area + decreasingly by width.
3. Random permutation: scramble randomly the contents of $\mathcal{B}$ and find a solution for it. This step is concatenated several times.
4. Mixed heuristic: concatenate all the previous heuristics in the order they are listed.

The mixed heuristic proved to be the best in practice.

[1] This could have been implemented in the variable selection strategy. However, this way we avoid the solver computing a maximum or minimum each time it has to select a variable.

3.2 Compilation and execution

In order to compile this part of the project, while assuming that we are at the root directory of the project, one should issue the following commands:

```
~/box-wrapping$ cd CP/build-rules
~/box-wrapping/CP/build-rules$ make -f Makefile release
~/box-wrapping/CP/build-rules$ cd ..
~/box-wrapping/CP$ cd build-release
```

Once the compilation has finished we can execute the binary file generated[2]. The complete usage can be seen by issuing the command:

```
$ ./wrapping-boxes --help
```

Here are a few of the options explained:

- If one wants to find n solutions to a certain input, and store the best found in a certain file, issue the following command:

  ```
  $ ./wrapping-boxes -i $INPUT_FILE -o $OUTPUT_FILE --stop-at n --enumerate --rotate
  ```

  Storing the best solution is indicated by specifying an output file with "-o". The option "enumerate" will print on standard output all n solutions found. The solver used is "rotate", which will not try to find an optimal solution. The "stop-at" option indicates the solver to stop when n solutions were found.

- If one wants to find the best solution to an instance, use the "optim" solver instead of the solver "rotate".

  ```
  $ ./wrapping-boxes -i $INPUT_FILE -o $OUTPUT_FILE --stop-when 10.0 --enumerate --optim
  ```

  The option "enumerate" will make the program list all solutions found from the first (and probably worst) solution to the optimal (or quasi-optimal) one. With the option "stop-when" the solver will stop finding solutions after 10 seconds.

There are more options one could use: indicate the number of threads that the solver can use, scramble (permute randomly) the input data and seed the random number generator so that the permutation is different at every execution. It is important to say that the option "stop-at" should always be specified, otherwise the solver will try to find all feasible solutions within the matrix defined for variables 1 (what cells does a box occupy) and 2 (where is a box's corner placed at).

In order to use a heuristic, use the option "heuris-*", where '*' is either: "incr", "decr", "rand", or "mix", each of these being an alias for the heuristics described in section 3.1.1, in the same order of appearance. These also have options of their own, like indicating how many times the solver should be executed with the input data randomly permuted (the other strategies are used only once).

4 Benchmarking

A quick shortcut for the execution of any solver on all files, describing an input each, inside a directory can be found in the script "benchmark.sh" in the "scripts/" directory. Its usage is very simple: each solver is called with the parameters that should make it behave at its best. Therefore, we only have to indicate the solver we want to execute, the directory with all the inputs and where to store the output found for each input. An example can be found in figure 4.

[2] There is no need to create the directory *build-release/* prior to compiling.

```
$ ./benchmark.sh --solver=CP -i=../inputs/material -o=../outputs/CP
```

Figure 4: How to use the benchmarking script to execute the solver for Constraint Programming on the inputs in the *inputs/material* directory.

This script will, for every input, invoke the specified solver and check the quality of the solution found with a "hand-made" solution of that same instance, that can be found in the directory *outputs/hand-made/*. If the solution is as good as the hand-made then it will output a message similar to the one in figure 5.

```
Executing solver with input file: ../inputs/material/bwp_11_10_1.in
    Optimal solution reached
    In 34.622 seconds
    Current progress:
        Solved optimally    : 90 / 105 ( 85.71% )
        Solved sub-optimally: 15 / 105 ( 14.28% )
        Total time elapsed  : 2095.598 seconds
```

Figure 5: Example of the output of the benchmark script for the Constraint Programming solver when the solution found is optimal.

The format of the information shown in figure 5 is as follows: it tells whether the optimal solution was reached or not (assuming that the corresponding output in the *outputs/hand-made/* directory is indeed optimal), the time needed to reach that solution, and finally the progress of the script. In a Pentium IV CPU, 1.8 GHz, with a single core, it took around 23 minutes to solve a total of 105 inputs, 85 of them optimally and 15 of them suboptimally. The inputs used to get the message in figure 5 were all the inputs sorted lexicographically from the "bwp_3_3_1.in" to the "bwp_11_10_1.in". The total output for the same CPU can be found in the file "scripts/CP-p4-benchmark-log"[3]. An example of the message output by the script when the solution is not optimal is shown in figure 6.

```
Executing solver with input file: ../inputs/material/bwp_11_8_1.in
    Suboptimal solution:
        Optimal: 12
        CP:      15
    In 45.022 seconds
    Current progress:
        Solved optimally    : 88 / 103 ( 85.43% )
        Solved sub-optimally: 15 / 103 ( 14.56% )
        Total time elapsed  : 2050.956 seconds
```

Figure 6: Example of the output of the benchmark script for the Constraint Programming solver when the solution found is not optimal.

One can design their own inputs and allow the benchmark script to compare the solution found by any solver to the optimal solution found by hand by making a file containing the output following the format specified in the report. Also, one could use the Box Wrapper user interface used to make the optimal outputs for all the inputs in directory *inputs/material/* (see section A).

[3] In a Unix-like environment use the *cat*, not the *less*, software to display it on the command shell.

4.1 Constraint Programming

The parameters used to execute the Constraint Programming solver described in section 3 are the following:

- Use the mixed heuristic (described in section 3.1.1).
- The solver will stop as soon as it found one solution.
- The solver will stop after 5 seconds of its execution.
- The solver will be executed on randomly permuted data 5 times (needless to say that the data will also be permuted 5 times).

With these parameters, the solver has a "natural" timeout of **45** seconds. That is, each execution of the solver with the data sorted increasingly by area, by width, decreasingly sorted by area, by width, and each of the 5 random permutations have a time limit of 5 seconds. With a total of 9 executions that makes $5 \cdot 9 = 45$ seconds of time limit.

4.1.1 Results

Needless to say that depending on the machine this software is executed on, in spite of having the same time limits, a more powerful computer is more likely to find better solutions since it can explore the search tree more exhaustively in the same amount of time. Indeed, this is confirmed in table 1.

	Pentium IV 1.8 GHz	i7-6700 HQ 3.5 GHz
Optimal solutions	93 / 108	97 / 108
Sub-optimal solutions	15 / 108	11 / 108
Total time	37 mins	33 mins

Table 1: Comparison between two different CPUs over all the 108 instances in the *inputs/material/* directory.

The outputs can be found in the *outputs/* directory, inside folders *CP-p4/* and *CP-i7/* respectively. Table 2 gives the list of instances solved suboptimally for each CPU.

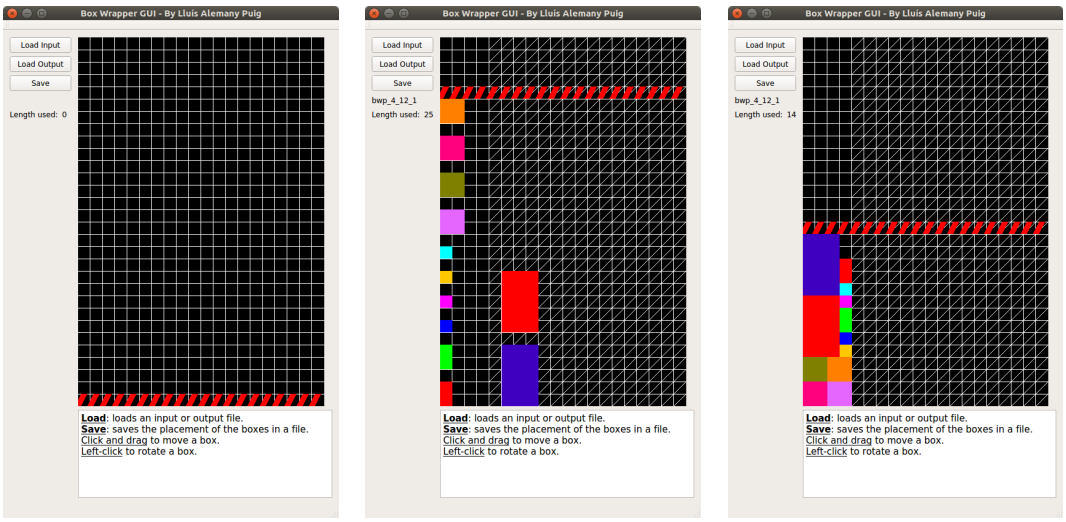
Instance name			Pentium IV 1.8 GHz	i7-6700 HQ 3.5 GHz	Optimal value
3	7	1	13		12
3	11	1	31	31	29
3	12	1	16	16	15
3	13	1	23	23	22
4	11	1	12	12	11
5	12	1	27	27	24
6	11	1	15	15	13
6	13	1	36	36	35
7	13	1	15	15	14
8	13	1	20	20	17
9	11	1	7		6
9	13	1	16		15
10	11	1	6		5
10	13	1	15	15	14
11	8	1	15	15	12

Table 2: Instances in the *inputs/material/* directory solved suboptimally. An empty cell in the table indicates that the instance was solved optimally.

A Box wrapper GUI

The Box Wrapper is a very simple user interface implemented with Qt 5.5 (see [2]), and OpenGL (see [1]), used to easily make by hand the output of any instance hence saving us from all the work that requires to do it using pen and paper. The software can load any input and it will display all boxes on the screen, allowing the user to place them, by clicking and dragging, wherever they think best. Obviously the boxes can also be rotated. Furthermore, outputs can also be loaded and modified for further improvement.

The program also helps the user in evaluating the quality of the solution by displaying a red line indicating the roll's length used. The numeric value for this length is also displayed on the window. Besides, the software will indicate the user what are the valid cells for the boxes to be placed at (according to the width of the roll) by crossing the "out-of-bounds" cells. See figure 7 for three screenshots of the software.



(a) Initial state of the software. (b) After loading an instance. (c) After solving an instance.

Figure 7: The different states of the software. We can load an instance (7b) and edit the positions of the boxes and their rotation to obtain a solution (7c).

References

[1] Khronos Group. *OpenGL*. URL: `https://www.opengl.org/` (visited on 05/01/2018).

[2] *Qt 5.5*. URL: `https://www1.qt.io/qt5-5/` (visited on 05/01/2018).

[3] Christian Schulte, Mikael Lagerkvist, and Guido Tack. *Gecode*. URL: `http://www.gecode.org/` (visited on 04/22/2018).