Box Wrapping Problem

Combinatorial Problem Solving
Course project

Academic year 2017-2018 Q2

Lluís Alemany Puig

June 4, 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.

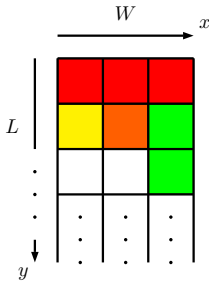


Figure 1: Optimal placement for $W = 3$ and boxes $(1, 1)$, $(1, 1)$, $(2, 1)$, $(1, 3)$.

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 section4), and Satisfiability (see section 5). 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. Finally, results regarding quality and efficiency of each solver are presented in the Benchmarking section (see section 6).

2 Modelling the problem

In this section we describe the generic mathematical model used to solve the problem. Although the variables needed may depend on the paradigm used to tackle the problem, in this section are described both the variable selection made (see section 2.1) and the constraints used to model the problem (see section 2.2) in a general enough way. Therefore, their use in the implementation of the algorithms may make them to change in some way so that they can be adapted properly to the paradigm.

2.1 Variables

The variables chosen to formulate the problem are all Boolean variables. They are defined as follows:

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). $X_{b,x,y} \in \mathbb{B}$, $\forall b, x, y$, $1 \leq b \leq N$, $1 \leq x \leq W$, $1 \leq y \leq L$.

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, the model consists of, at least, 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 at least $N(2WL + 1)$ Boolean variables. The actual implementation of the solvers may require more (or maybe less) variables than the ones presented.

2.1.1 Determining the sizes of the matrices

Notice that this choice of variables 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. Each constraint is defined with plain words, and also has some mathematical formulation.

1. Each box must have its top-left corner placed in the roll. This constraint is used to ensure that the placement contains all boxes. A general way of formulating this constraint is by saying that "among all values $X_{b,x,y}$ for a fixed box b, there must be exactly one of them that is set to true". A very common formulation using the Linear Programming paradigm is the one presented in equation 4.

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

2. No overlapping, that is, no cell of the roll can have more than one box in it. Likewise, this constraint can be formulated generally by saying that "among all values of $C_{b,x,y}$ for a fixed pair of values for x and y there can be at most one of them set to true". A very common formulation using the Linear Programming paradigm is the one presented in equation 5.

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

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:

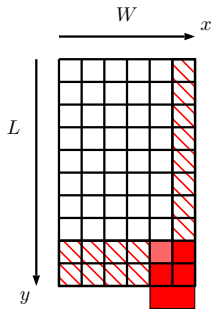
$$(R_b = 0) \wedge (X_{b,x,y} = 1 \implies C_{b,x+j,y+i} = 1)\,, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N,\ \text{s.t.}\ (w_b = l_b \wedge \\ \quad \forall x, y,\ 1 \leq x \leq W, 1 \leq y \leq L \\ \quad\quad \text{s.t.}\ (x + w_b \leq W \wedge y + l_b \leq L) \\ \quad \forall i, j,\ 0 \leq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

   - For rectangular boxes:

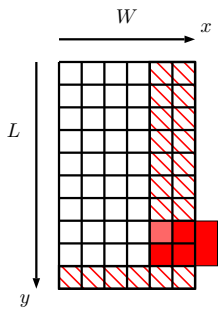
$$(R_b = 0 \wedge X_{b,x,y} = 1) \implies C_{b,x+j,y+i} = 1, \qquad \begin{array}{l} \forall b,\ 1 \leq b \leq N,\ \text{s.t.}\ (w_b \neq l_b \wedge \\ \quad \forall x, y,\ 1 \leq x \leq W, 1 \leq y \leq L \\ \quad\quad \text{s.t.}\ (x + w_b \leq W \wedge y + l_b \leq L) \\ \quad \forall i, j,\ 0 \leq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

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

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". Since no square box needs to be rotated then the rotation is fixed to a null value ($R_b = 0$).



(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 (see figure 2). 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:

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

   - For rectangular boxes:

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

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

   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: "s.t. $(x + w_b > W \vee y + l_b > L)$".

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 [5]) in order to implement the mathematical model that models the problem and helps us solve it without having to implement our own algorithm.

3.1 Implementation

In order to implement the variables detailed in section 2.1 three arrays of Boolean variables were used, each of them containing the variables described in items 1, 2 and 3. 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 ($C_{b,x,y}$):

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

2. Array for variables in 2 ($X_{b,x,y}$):

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

3. Array for variables in 3 (R_b):

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

Assuming the implementation of the following functions to facilitate access to the arrays of variables:

```
// 0 <= b < N, 0 <= x < W, 0 <= y < L
inline BoolVar X(size_t b, size_t x, size_t y) const { return box_corner[b*W*L + y*W + x]; }
inline BoolVar C(size_t b, size_t x, size_t y) const { return box_cell[b*W*L + y*W + x]; }
inline BoolVar R(size_t b) const { return box_rotated[b]; }

inline BoolVar X(size_t b, size_t x, size_t y) { return box_corner[b*W*L + y*W + x]; }
inline BoolVar C(size_t b, size_t x, size_t y) { return box_cell[b*W*L + y*W + x]; }
inline BoolVar R(size_t b) { return box_rotated[b]; }
```

the implementation of the constraints detailed in section 2.2 using the Gecode library is as follows:

1. All boxes must be in the solution (constraint formalised in 1), also read as "all boxes must have assigned a top-left corner".

```
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).

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

for (length i = y; i <= y + b_width - 1; ++i) {
    for (width j = x; j <= x + b_length - 1; ++j) {
        rel(*this, ((R(b) == 1) && (X(b,x,y) == 1)) >> (C(b,j,i) == 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, X(b,x,y) == 0);
```

   - For rectangular boxes (constraints 10 and 11):

```
rel(*this, (R(b) == 0) >> (X(b,x,y) == 0) );

rel(*this, (R(b) == 1) >> (X(b,x,y) == 0) );
```

3.2 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. 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, C(b,j,i) == 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*.

3.2.1 Heuristics

When using the method described in section 3.2 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, this only happens when "given enough time", that is, however efficient Gecode might be it might take 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

$L :=$ UpperBoundRollLength($\mathcal{B}$), using equation 3
$\mathcal{B}' :=$ sort boxes in $\mathcal{B}$ increasingly by area
$S_1 :=$ Solve($\mathcal{B}', L-1$)

$L := \min\{\text{length}(S_1), L\}$
$\mathcal{B}' :=$ sort boxes in $\mathcal{B}$ decreasingly by area
$S_2 :=$ Solve($\mathcal{B}', L-1$)

$\cdots$

$L := \min\{\text{length}(S_{n-1}), L\}$
$\mathcal{B}' :=$ sort boxes in $\mathcal{B}$ using some other criteria
$S_n :=$ Solve($\mathcal{B}', L-1$)
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 then the solver should not take much time in concluding that it can not possibly find a solution at step $i+1$. 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.

3.3 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 are 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 3.2.1, 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 Linear Programming

In this part of the project the problem is solved using the Linear Programming paradigm. In particular, we used the library CPLEX for C++ (version 12.7) (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.

The use of this paradigm has its advantages and disadvantages. One of the the advantages is that we can naturally model optimisation problems with their own objective function (whose value has to be maximised or minimised). One of the disadvantages is that it is not as expressive as the constraint programming paradigm. For example, it does not easily allow logical expressions, since they are not

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

linear. For example, in order to implement the logical or ($a \vee b$, where a and b are linear expressions) we have to introduce some new variables to help the solver satisfy one of the two conditions. Now follows an explanation on how to transform several logical expressions into linear expressions, assuming them to be integer-valued for all feasible solution, using simple transformations that relate each expression to 0/1 variables. Let δ_1, δ_2 be such variables:

- $(\delta_1 = 1) \vee (\delta_2 = 1)$. Impose $\delta_1 + \delta_2 \geq 1$.
- $(\delta_1 = 0) \vee (\delta_2 = 1)$. Impose $\delta_1 \leq \delta_2$.
- $(\delta_1 = 0) \vee (\delta_2 = 0)$. Impose $\delta_1 + \delta_2 \leq 1$.
- $(\delta_1 = 1) \wedge (\delta_2 = 1)$. Impose $\delta_1 + \delta_2 = 2$.
- $(\delta_1 = 0) \wedge (\delta_2 = 1)$. Impose $1 - \delta_1 + \delta_2 = 2$.
- $(\delta_1 = 0) \wedge (\delta_2 = 0)$. Impose $\delta_1 + \delta_2 = 0$.
- $(\delta_1 = b_1) \Longrightarrow (\delta_2 = b_2) \equiv (\delta_1 = 1 - b_1) \vee (\delta_2 = b_2)$, for any two values $b_1, b_2 \in \mathbb{B}$.
- $(\delta_1 = b_1) \Longleftrightarrow (\delta_2 = b_2)$. Impose the two logical constraints $(\delta_1 = b_1) \Longrightarrow (\delta_2 = b_2)$ and $(\delta_2 = b_2) \Longrightarrow (\delta_1 = b_1)$, for any two values $b_1, b_2 \in \mathbb{B}$.

Relating linear expressions through logical operators is slightly more complicated. In general, given two integer-valued linear expressions $a_1^T x \leq b_1$ and $a_2^T x \leq b_2$ are for all feasible solution x, to model the logical expression

$$(a_1^T x \leq b_1) \square (a_2^T x \leq b_2)$$

where $\square$ is any binary logical operator $\vee, \wedge, \Longrightarrow, \Longleftrightarrow$, we have to apply the following procedure: for each of the two linear expressions introduce a 0/1 variable. Let δ_1 and δ_2 be these variables defined so that:

$$(\delta_1 = 1) \Longleftrightarrow (a_1^T x \leq b_1), \qquad (\delta_2 = 1) \Longleftrightarrow (a_2^T x \leq b_2)$$

It only remains to impose the logical constraint $(\delta_1 = 1) \square (\delta_2 = 1)$, and use linear expressions for the two double implications:

$$a_i^T x - b_i \leq \mathcal{U}_i(1 - \delta_i), \qquad a_i^T x - b_i \geq (\mathcal{L}_i - 1)\delta_i + 1$$

where $\mathcal{U}_i$ and $\mathcal{L}_i$ are upper and lower bounds, respectively, for the expression $a_i^T x - b_i$, for $i \in \{1, 2\}$.

In spite of logical constraints not being linear expressions, we do not need to reformulate the constraints explained in section 2 because CPLEX does the necessary transformations automatically.

4.1 Implementation

In order to implement the variables detailed in section 2.1, three arrays of integer variables were used, each of them containing the variables described in items 1, 2 and 3. 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 ($C_{b,x,y}$):

```
box_cell = IloNumVarArray(env, N*W*L, 0, 1, ILOINT);
```

2. Array for variables in 2 ($X_{b,x,y}$):

```
box_corner = IloNumVarArray(env, N*W*L, 0, 1, ILOINT);
```

3. Array for variables in 3 (R_b):

```
box_rotated = IloNumVarArray(env, N, 0, 1, ILOINT);
```

Following the CPLEX's requirements for a proper use of its solvers, we required the use of these three objects:

```
IloEnv env;
IloModel model(env);
IloCplex cplex(model);
```

Assuming the implementation of the following functions to facilitate access to the arrays of variables:

```
// 0 <= b < N, 0 <= x < W, 0 <= y < L
inline IloNumVar X(size_t b, size_t x, size_t y) const { return box_corner[b*W*L + y*W + x]; }
inline IloNumVar C(size_t b, size_t x, size_t y) const { return box_cell[b*W*L + y*W + x]; }
inline IloNumVar R(size_t b) const { return box_rotated[b]; }

inline IloNumVar X(size_t b, size_t x, size_t y) { return box_corner[b*W*L + y*W + x]; }
inline IloNumVar C(size_t b, size_t x, size_t y) { return box_cell[b*W*L + y*W + x]; }
inline IloNumVar R(size_t b) { return box_rotated[b]; }
```

the implementation of the constraints detailed in section 2.2 using the CPLEX library 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) {
    IloExpr exa_X(env);
    for (length y = 0; y < L; ++y) {
        for (width x = 0; x < W; ++x) {
            exa_X += X(b,y,x);
        }
    }
    model.add(exa_X == 1);
    exa_X.end();
}
```

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

```
for (length y = 0; y < L; ++y) {
    for (width x = 0; x < W; ++x) {
        IloExpr amo_C(env);
        for (int b = 0; b < N; ++b) {
            amo_C += C(b,x,y);
        }
        model.add(amo_C <= 1);
        amo_C.end();
    }
}
```

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) {
        model.add(
            IloIfThen(env, (X(b,x,y) == 1), (C(b,j,i) == 1))
        );
    }
}
model.add(R(b) == 0);
```

   - For rectangular boxes (constraints 7 and 8):

```
for (length i = y; ii <= y + b_length - 1; ++i) {
    for (width j = x; j <= x + b_width - 1; ++j) {
        model.add(
            IloIfThen(env, ((R(b) == 0) && (X(b,x,y) == 1)), (C(b,j,i) == 1))
        );
    }
}
```

```
for (length i = y; i <= y + b_width - 1; ++i) {
    for (width j = x; j <= x + b_length - 1; ++j) {
        model.add(
            IloIfThen(env, ((R(b) == 1) && (X(b,x,y) == 1)), (C(b,j,i) == 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):

   ```
   model.add(X(b,x,y) == 0);
   ```

   - For rectangular boxes (constraints 10 and 11):

   ```
   model.add(
       IloIfThen(env, (R(b) == 0), (X(b,x,y) == 0))
   );

   model.add(
       IloIfThen(env, (R(b) == 1), (X(b,x,y) == 0))
   );
   ```

The use of non-linear expressions in the Linear Programming model is not allowed. For example, constraint 7

$$(R_b = 0) \wedge (X_{b,x,y} = 1) \Longrightarrow (C_{b,j,i} = 1)$$

is clearly not a linear expression. Although CPLEX does the necessary transformations of non-linear expressions to linear ones, it can be transformed manually into one by noticing that the logical constraint is equivalent to

$$(R_b = 1) \vee (X_{b,x,y} = 0) \vee (C_{b,j,i} = 1)$$

From here we can obtain the linear expressions that model this logical constraint using the transformations explained at the beginning of this section. The transformation for this particular case is detailed in section B.1.

4.1.1 Completing the model

All the "linear" expressions above are the only required according to the general modelling of the problem described in section 2. However, the linear programming paradigm requires an extra constraint: each box can not occupy more cells than its area. This constraint is required for the following reason: since is logically true that "false implies true" we can have many cells c of the roll assigned to some box b even if its top-left corner is not assigned to any cell t nearby[3]. This way we may encounter an assignation that makes a box have, not only more cells than it should, but also cells assigned throughout the roll completely scattered. See figure 4 for an illustration of this phenomenon.

[3] By "nearby" we mean "placed at a cell so that according to the box's rotation and its dimensions cell c falls within the area that should be occupied by b when its top-left corner is assigned to t."

L	W 0	1	2
0	13	13	12
1	6	6	12
2	6	6	12
3	6	6	12
4	6	6	12
5	8	8	13
6	8	8	13
7	8	8	11
8	8	8	13
9	8	8	13

L	W 0	1	2
10	12	13	9
11	11	13	8
12	13	13	13
13	13	5	5
14	13	5	5
15	12	5	5
16	13	5	5
17	13	12	13
18	7	12	12
19	13	13	11

L	W 0	1	2
20	13	13	13
21	4	13	13
22	13	13	13
23	1	12	13
24	4	8	13
25	11	4	13
26	13	4	13
27	13	11	13
28	13	12	4
29	4	4	13

L	W 0	1	2
30	4	4	4
31	4	4	4
32	4	4	1
33	12	2	1
34	13	2	1
35	11	2	13
36	12	13	11
37	12	13	11
38	3	13	11
39	3	11	11

L	W 0	1	2
40	3	13	11
41	7	7	7
42	7	7	12
43	7	7	13
44	7	7	12
45	7	13	10

Figure 4: Invalid contents of the roll without rotations and optimisation for instance *instances/material/bwp_3_13_1.in*

In spite of this, this only affects the extraction of the solution. Moreover, it is just a minor problem: the construction of the solution only requires knowing the rotation and the position of the top-left corner of each box. Knowing these two, and having access to the input data, we can easily construct the valid roll[4] (see figure 5). Therefore, the constraint is not only superfluous, but also its implementation and addition to the model only slows down its performance.

L	W 0	1	2
0	.	.	12
1	6	6	12
2	6	6	12
3	6	6	12
4	6	6	12
5	.	.	.
6	8	8	.
7	8	8	.
8	8	8	.
9	8	8	.

L	W 0	1	2
10	.	.	9
11	.	.	.
12	.	.	.
13	.	5	5
14	.	5	5
15	.	5	5
16	.	5	5
17	.	.	.
18	.	.	.
19	.	.	.

L	W 0	1	2
20	.	.	.
21	.	.	.
22	.	.	13
23	.	.	13
24	.	.	13
25	.	.	13
26	.	.	13
27	.	.	.
28	.	.	.
29	4	4	.

L	W 0	1	2
30	4	4	.
31	4	4	.
32	4	4	1
33	.	2	1
34	.	2	1
35	.	2	.
36	.	.	11
37	.	.	11
38	3	.	11
39	3	.	11

L	W 0	1	2
40	3	.	11
41	7	7	.
42	7	7	.
43	7	7	.
44	7	7	.
45	.	.	10

Figure 5: Valid contents of the roll without rotations and optimisation for instance *instances/material/bwp_3_13_1.in*

4.2 Finding the optimium solution

In the Linear Programming paradigm, the solver uses an objective function to find an optimal solution. This objective function is simple: it suffices to minimise the maximum length coordinate of roll used over all boxes' coordinate length S_b. A box's length coordinate is simply the coordinate of its bottom-left (or bottom-right) corner. Therefore, the objective function can be formalised as

$$\min \max_{1 \leq b \leq N} S_b \tag{14}$$

Although the $\max_b$ is not a linear function, CPLEX supports it perfectly. However, it has been implemented using linear expressions due to its simplicity. For this, a new variable M was introduced, defined so that $M \geq S_b$, for all $b \in \{1, \cdots, n\}$. Then, the objective function is simply:

$$\min M$$

However, S_b still needs to be defined. This is also rather simple. For each box b and for each cell of the roll (x, y) impose the following two constraints:

$$(R_b = 0) \wedge (X_{b,x,y} = 1) \Longrightarrow (S_b = y + b_l), \qquad (R_b = 1) \wedge (X_{b,x,y} = 1) \Longrightarrow (S_b = y + b_w)$$

This is implemented with the following piece of code:

[4]The actual solution to the problem, as specified in the statement, is not the roll itself but only the top-left and bottom-right corners of each box.

```
Sb = IloNumVarArray(env, N, 0, L, ILOINT);
M = IloNumVar(env, 0, L, ILOINT);
for (int b = 0; b < N; ++b) {
    for (length y = 0; y < L; ++y) {
        for (width x = 0; x < W; ++x) {
            model.add(
                IloIfThen(env, ((R(b) == 0) && (X(b,x,y) == 1)), (S(b) == y + b_length))
            );
            model.add(
                IloIfThen(env, ((R(b) == 1) && (X(b,x,y) == 1)), (S(b) == y + b_width))
            );
        }
    }
    model.add(M >= S(b));
}
model.add(
    IloMinimize(env, M)
);
```

where the function "S(b)" is defined as follows Assuming the definition of the two following functions:

```
IloNumVar S(size_t b) const { return Sb[b]; }
IloNumVar S(size_t b) { return Sb[b]; }
```

Despite the existence of an explicit objective function one could still perform some basic transformations on the input data, similar to the ones described in section 3.2.1, hoping that it can help the solver find better solutions faster. However, a few basic experiments show that this is not the case. In fact, the short execution time the solver is allowed[5] does not let it find solutions that are as good as the ones found when the solver is allowed to run for 45 seconds straight without any kind of transformation.

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

However, the "Makefile.ibm" file in the *build-rules/* directory may have to be modified. In particular, the variables that contain the paths containing the headers used in the include path for the compiler:

```
SYSTEM       = x86-64_linux
LIB_FORMAT   = static_pic
CPLEX_DIR    = /opt/ibm/ILOG/CPLEX_Studio1271/cplex
CONCERT_DIR = /opt/ibm/ILOG/CPLEX_Studio1271/concert
```

Once the compilation has finished we can execute the binary file generated[6]. The complete usage can be seen by issuing the command:

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

The solver accepts few parameters. One of them is mandatory: the input file containing the instance to be solved passed with "-i". The file to store the output is optional and is passed with "-o". In order to obtain a solution that minimises the roll length used, use "–optim". Two extra options can be passed to affect the way the solver is executed: determine the number of threads with "–n-threads" and use "–stop-when" to indicate a timeout. An example can be seen in figure 6.

```
$ ./wrapping-boxes --optim -i bwp_3_3_1.in -o bwp_3_3_1.out --n-threads 8
```

Figure 6: An example of how to use the Linear Programming solver.

[5] Since we want to apply several heuristics, it is important not to let the solver for too long, or it might take more than 120 seconds which the time limit imposed in the statement.

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

5 Satisfiability

This part of the project is aimed at solving the BWP using the satisfiability paradigm, that is, by constructing a Boolean formula in Conjunctive Normal Form (CNF), using a number of Boolean variables, and then solving it. The solution is a truth assignment to the variables. The way this solution is obtained is by means of a SAT solver.

This time, this paradigm has mostly only disadvantages: not only we cannot find an optimal solution directly just by finding a truth assignment to the variables, but also the "language" is less expressive, or, rather, more rigid since it only allows Boolean clauses in CNF than the one used in Constraint or Linear Programming. However, the constraints and variables that have to be imposed are exactly the same as the ones formalised in section 2 and, as will be explained in the coming sections, optimality can be reached using fairly simple methods.

By less expressive we mean the following: take, for example, the constraint 1 formalised in section 2.2. This constraint formalised the idea that each box has to have its top-left corner assigned to exactly one cell, and, given the Boolean variables detailed in 2 for each cell on the roll for a certain box, we only had to add up all the variables and impose that the sum be equal to 1. This was implemented in sections 3 and 4 following equation 4 literally. This time we can not do this. In Boolean satisfiability this can be seen as "given n Boolean variables, exactly one of the n variables $x_1, \cdots, x_n$ has to be true". Encoding this constraint requires two types of constraints: "at least one" and "at most one". If we manage to encode them using two Boolean expressions in CNF each, the conjunction of the two will give us the "exactly one" constraint:

- At least one: given variables $x_1, \cdots, x_n$ at least one of them must be true. This is straightforward, we only need to add the following clause to the Boolean formula:

$$x_1 \vee x_2 \vee \cdots \vee x_n \tag{15}$$

  Since all clauses of the formula have to evaluate to true to make it satisfiable then at least one of the variables x_i must be set to true.

- At most one: given variables $x_1, \cdots, x_n$ at most one of them must be true. This is the most difficult constraint to implement using a Boolean formula in CNF. To begin with, there are different ways to encode this constraint. Here are listed three of them:

  1. Quadratic encoding ($\mathcal{Q}$): the simplest of the encodings, we only have to add the clauses

$$\mathcal{Q}(x_1, \cdots, x_n) = \bigvee_{1 \leq i < j \leq n} (\overline{x_i} \vee \overline{x_j}) \tag{16}$$

     which will produce $\binom{n}{2} = O(n^2)$ clauses.

  2. Logarithmic encoding ($\mathcal{L}$): this encoding uses $m = \lceil \log n \rceil$ extra variables to encode the intended meaning. Given n variables, introduce variables $y_0, \cdots, y_{m-1}$ and add the following clauses:

$$\mathcal{L}(x_1, \cdots, x_n) = \bigwedge_{i=1}^{n} \left(\bigwedge_{j} (\overline{x_i} \vee y_j) \right) \wedge \left(\bigwedge_{k} (\overline{x_i} \vee \overline{y_k}) \right) \tag{17}$$

     for the bits j set to 1 in the binary representation of i and for the bits k set to 0 in the binary representation of i, respectively. Since for each variable $O(\log n)$ clauses are added, the logarithmic encoding produces in total $O(n \log n)$ clauses.

3. Heule encoding ($\mathcal{H}$): this encoding is based on the quadratic encoding. The set of clauses produced $\mathcal{H}(x_1, \cdots, x_n)$ is defined with the following recurrence:

$$\mathcal{H}(x_1, \cdots, x_n) = \begin{cases} \mathcal{Q}(x_1, \cdots, x_n) & \text{if } n \leq 3 \\ \mathcal{Q}(x_1, x_2, y) \wedge \mathcal{H}(x_3, \cdots, x_n, \overline{y}) & \text{otherwise} \end{cases} \tag{18}$$

where y denotes a new variable that was not used before.

The performance of the SAT solver will depend largely on the encoding chosen for constraints of type "at most one". Further details are given in section 6.3.

5.1 Implementation

This time we did not need to implement any array of variables, since we only have to output the clauses into a file that will be used by a SAT solver. However, each variable needs an identifier (starting at 1). Generating these identifiers is done via the following functions:

1. Variables in 1 ($C_{b,x,y}$):

```
// 0 <= b < N, 0 <= x < W, 0 <= y < L
int C(int b, int x, int y) {
    return nXvars + 1 + b*W*L + y*W + x;
}
```

2. Variables in 2 ($X_{b,x,y}$):

```
// 0 <= b < N, 0 <= x < W, 0 <= y < L
int X(int b, int x, int y) {
    return 1 + b*W*L + y*W + x;
}
```

3. Variables in 3 (R_b):

```
// 0 <= b < N
int R(int b) {
    return nXvars + nCvars + 1 + b;
}
```

The variables $nXVars$, $nCvars$ and $nRvars$ store the amount of variables for $X_{b,i,j}$, $C_{b,i,j}$ and R_b respectively. The amounts of each type of variables is given at the end of section 2.1. Variables W and L store the roll's width and length, respectively.

While keeping in mind the implementation of these functions, and assuming the existence of a class **clause** which represents a list of literals, and the existence of an object **CE** which is capable of encodign the "at least/most one" constraints in the three different ways detailed above ($\mathcal{Q}$, $\mathcal{L}$, and $\mathcal{H}$), the constraints in section 2.2 have been implemented 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 (size_t b = 0; b < N; ++b) {
    clause cl;
    for (length y = 0; y < L; ++y) {
        for (width x = 0; x < W; ++x) {
            cl += X(b,x,y);
        }
    }
    CE.exactly_one(cl, out);
}
```

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

```
for (length y = 0; y < L; ++y) {
    for (width x = 0; x < W; ++x) {
        clause cl;
        for (size_t b = 0; b < N; ++b) {
            cl += C(b,x,y);
        }
        CE.amo(cl, out);
    }
}
```

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) {
           out << -X(b,x,y) << " " << C(b,j,i) << " 0" << endl;
       }
   }
   out << -R(b) << " 0" << endl;
   ```

   - 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) {
           out << R(b) << " " << -X(b,x,y) << " " << C(b,j,i) << " 0" << endl;
       }
   }

   for (length i = y; i <= y + b_width - 1; ++i) {
       for (width j = x; j <= x + b_length - 1; ++j) {
           out << -R(b) << " " << -X(b,x,y) << " " << C(b,j,i) << " 0" << endl;
       }
   }
   ```

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):

   ```
   out << -X(b,x,y) << " 0" << endl;
   ```

   - For rectangular boxes (constraints 10 and 11):

   ```
   out << R(b) << " " << -X(b,x,y) << " 0" << endl;

   out << -R(b) << " " << -X(b,x,y) << " 0" << endl;
   ```

5.2 Finding the optimum solution

Likewise the Constraint Programming paradigm, Satisfiability is not defined to solve optimisation problems either. However, we can use a SAT solver as a black box to find optimal solutions. To achieve this, the ideas explained in section 3.2 will be applied similarly within a framework that consists of three independent modules: the clause generator, the SAT solver, and the solution generator. The first will generate the necessary clauses in CNF for a given instance of the Box Wrapping Problem, which will be passed on to the SAT solver, which in turn will compute a truth assignment (or decide that the Boolean formula is unsatisfiable). If the formula is satisfiable then the solution generator will process this truth assignment and build a solution, that is, find the position of each top-left and bottom-right corners for each box. Now, this solution generator returns the length that was used and sends it to the clause generator which will compute the minimum between the length's upper bound (using equation 3) and this value (minus 1). Basically, the ideas behind equation 13 are applied so as to force the SAT solver find a solution with a shorter roll length. This process is applied until the

SAT solver decides that the Boolean formula generated is unsatisfiable, which implies that optimality was reached. See figure 7 for a graphical representation of this process. Therefore, the optimal value of the roll's length L is such that the Boolean formula produced is satisfiable, but the one produced with $L-1$ is unsatisfiable.

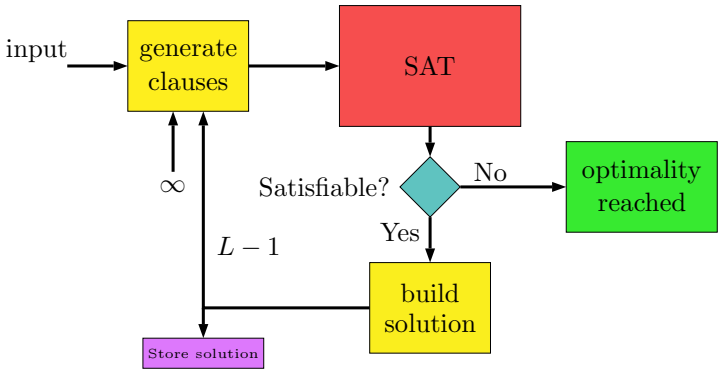


Figure 7: Finding an optimal solution using a SAT solver. The initial value of the length used in equation 13 is ∞.

Other methods could have been used. For example, binary search could have been applied. However, deciding whether a Boolean formula is satisfiable (and computing a truth value in case it is) is far easier than concluding unsatisfiablity. This way we avoid sending values of L that will certainly lead to producing an unsatisfiable formula hence facing the challenge of deciding unsatisfiability only once (for that one value of L that will certainly define the optimal value of the roll's length).

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

This will generate two executable files: *clause-generator* and *solution-generator*. The former generates the clauses and the latter takes the solution of the SAT solver and builds the solution. The clauses generated have been tested only on the **lingeling** SAT solver (see [1]). Moreover, the file generated has to be reversed. The solution generated by the solver has to be processed so that only the truth assignment is left on the file (the numeric values representing this truth assignment: if variable x_1 is assigned to true then the numerical value is "1". If it is assigned to false then it is "-1").

The clause generator has been designed to support two ways of solving an instance: when we allow the boxes to rotate, and when we do not. Moreover, the three types of encodings ($\mathcal{Q}$, $\mathcal{L}$, and $\mathcal{H}$) explained at the beginning of this section. In figures 8 and 9 are shown two examples of usage of these modules.

```
$ ./clause-generator -i bwp_3_3_1.in -o bwp.cnf.rev \
                --solver rotate --amo-encoder heule --max-L 1
```

Figure 8: An example of how to use the *clause-generator* module.

```
$ ./solution-generator --boxes bwp_3_3_1.in --variables bwp.vars \
                       -o bwp_3_3_1.out --solver rotate --max-L 1
```

Figure 9: An example of how to use the *solution-generator* module.

Since the process of specifying the parameters correctly, while preprocessing the files for and produced by the SAT solver, is rather cumbersome, the process depicted in figure 7 is governed by the script *exe-SAT.sh* in the *box-wrapping/SAT/* directory. It requires the file with the description of the instance, the name of the file containig its solution, what solver should be used, and, optionally, what encoder has to be used for the "at most one" constraints ($\mathcal{Q}$, $\mathcal{L}$, and $\mathcal{H}$), and two timeouts, one for the whole script and another for the SAT solver. An example can be seen in figure 11.

```
$ ./exe-SAT.sh  -i=bwp_3_3_1.in -o=bwp_3_3_1.out \
                --solver=rotate --amo-encoder=logarithmic
```

Figure 10: An example of how to use the *exe-SAT.sh* script.

The output of this script for the example in figure 11 is the following:

```
Try max length: 999999
    At iteration: 0
  | W
L | 0 1 2
----------
0 | 1 . .
1 | . 3 .
2 | . . 2

Boxes' corners:      (w,l)
    Box 1 is at
        top left     (0,0)
        bottom right (0,0)

    Box 2 is at
        top left     (2,2)
        bottom right (2,2)

    Box 3 is at
        top left     (1,1)
        bottom right (1,1)

    Roll length used: 3
```

```
Try max length: 2
    At iteration: 1
  | W
L | 0 1 2
----------
0 | . 3 2
1 | 1 . .

Boxes' corners:      (w,l)
    Box 1 is at
        top left     (0,1)
        bottom right (0,1)

    Box 2 is at
        top left     (2,0)
        bottom right (2,0)

    Box 3 is at
        top left     (1,0)
        bottom right (1,0)

    Roll length used: 2
```

```
Try max length: 1
    At iteration: 2
  | W
L | 0 1 2
----------
0 | 3 2 1

Boxes' corners:      (w,l)
    Box 1 is at
        top left     (2,0)
        bottom right (2,0)

    Box 2 is at
        top left     (1,0)
        bottom right (1,0)

    Box 3 is at
        top left     (0,0)
        bottom right (0,0)

    Roll length used: 1

Try max length: 0
    At iteration: 3
    lingeling determined unsatisfiability
    of the Boolean formula
       Optimality reached!
```

Figure 11: Edited output of the script *exe-SAT.sh* using the parameters of the example in figure 11.

6 Benchmarking

A quick shortcut for the execution of any solver on all files inside a directory, describing an input each, 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 12.

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

Figure 12: How to use the benchmarking script to execute the solver for Constraint Programming on the inputs in the *inputs/material/* directory.

All the other solvers available are CP, LP and SAT. This script will, for every input in the directory, 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 one then it will output a message similar to the one in figure 13. If it is worse, a message similar to the one in figure 14.

```
Optimal solution reached
In 34.488 seconds
Current progress:
           Solved optimally : 90 / 105 ( 85.71% )
       Solved sub-optimally : 15 / 105 ( 14.28% )
                  Unsolved : 0 / 105 ( 0.00% )
                 Remaining : 105 / 108 ( 97.22% )
     Total time elapsed : 2136.317 seconds
```

Figure 13: 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 13 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 36 minutes to solve a total of 105 inputs, 90 of them optimally and 15 of them suboptimally. The inputs used to get the message in figure 13 were all the inputs sorted lexicographically from the "bwp_3_3_1.in" to the "bwp_11_10_1.in". None of them were left unsolved. The total output for the same CPU can be found in the file "scripts/CP-p4-1-benchmark-log"[7].

```
Suboptimal solution:
        Optimal : 12
             CP : 15
In 45.025 seconds
Current progress:
        Solved optimally : 88 / 103 ( 85.43% )
    Solved sub-optimally : 15 / 103 ( 14.56% )
                Unsolved : 0 / 103 ( 14.56% )
               Remaining : 103 / 108 ( 95.37% )
      Total time elapsed : 2091.806 seconds
```

Figure 14: 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 optimal output. Obviously, the output must be in the format specified in the statement. Using the Box Wrapper user interface could be of help (see section A). This simple GUI was used to make the optimal outputs for all the inputs in directory *inputs/material/*.

In the following sections are presented the results for each solver, on two different CPUs, if possible. The CPUs are an Intel Pentium IV 1.8 GHz ($p4$), and an Intel i7-6700 HQ 3.5 GHz ($i7$). We also present results specific for the $i7$ using multiple threads. Since the tables will contain what instances were solved optimally, and which suboptimally, it is worth mentioning that this is known for two reasons.

[7] In a Unix-like environment use the command *less -r* to visualise the contents with colour.

One, because the solver terminated of its own accord, that is, at some point it either decided that it found the optimal solution (the case for the Linear Programming solver) or that no better solutions can possibly be found using applying the same algorithm (the case for the Constraint Programming solver and the Satisfiability framework). The other, although the solver was timeout-terminated, the solution's optimality is known because an optimal solution is known (the "hand-made" solutions). In the case this "hand-made" solution did not exist, the optimality of the solver's solution could not be determined. The amount of "optimal" solutions found in these circumstances is indicated in red.

6.1 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 3.2.1).
- The solver will stop as soon as it finds a solution or 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 seconds/execution $\times$ 9 executions = 45 seconds of time limit.

Needless to say that, 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. The "(x1)" and "(x4)" refer to the number of threads used for that CPU. However, the improvement is not as good as one could expect.

	p4	i7 (x1)	i7 (x4)
Optimal solutions	93 (19) / 108	97 (13) / 108	100 (12) / 108
Sub-optimal solutions	15 / 108	11 / 108	8 / 108
Unsolved instances	0 / 108	0 / 108	0 / 108
Total time	37 mins	33 mins	31 mins

Table 1: Summary presenting the amount of optimally and suboptimally solved instances, and unsolved instances over all 108 instances in the *inputs/material/* directory. Also, the total execution time is given for the CP solver in two different CPUs.

The outputs can be found in the *outputs/* directory, inside folders *CP-p4/*, *CP-i7-1/* and *CP-i7-4/* respectively. Table 2 gives the list of instances solved suboptimally for each CPU.

Instance name			P4	i7 (x1)	i7 (x4)	Optimal value
3	7	1	13			12
3	11	1	31	31	30	29
3	12	1	16	16		15
3	13	1	23	23		22
4	11	1	12	12	12	11
5	12	1	27	27	27	24
6	11	1	15	15		13
6	13	1	36	36	36	35
7	13	1	15	15	15	14
8	13	1	20	20	20	17
9	11	1	7			6
9	13	1	16			15
10	11	1	6			5
10	13	1	15	15	15	14
11	8	1	15	15	14	12

Table 2: Instances in the *inputs/material/* directory solved suboptimally and the roll's length found. An empty cell in the table indicates that the instance was solved optimally.

For further details, see the output of the "benchmark.sh" script in the files "scripts/CP-p4-benchmark-log", "scripts/CP-i7-1-benchmark-log" and "scripts/CP-i7-4-benchmark-log", respectively.

6.2 Linear Programming

The parameters used to execute the Linear Programming solver are the following:

- The solver will stop as soon as it finds the optimal solution or not much later than **45** seconds have passed since the start of its execution.
- The solver will use as many available threads as there are in the system.

Although the solver was executed on Ubuntu 16.04 its timeout was not applied using the "time" command because the class IloCplex uses an internal clock for it. In spite of using this timeout, whether executed using 1 or 4 threads, the solver's execution sometimes took more time: usually around 15 extra seconds, with one single exception where the solver took around 200 seconds more (lasting up to 260 seconds).

The CPLEX library was obtained through an academical-use only license for a 64-bit system. For this reason, we can only present results for one CPU only. See table 3 for a summary on the amount of optimally and suboptimally solved instances for *i7*.

	i7 (x1)	i7 (x4)
Optimal solutions	84 (6) / 108	88 (8) / 108
Sub-optimal solutions	11 / 108	17 / 108
Unsolved instances	13 / 108	3 / 108
Total time	34 mins	34 mins

Table 3: Summary presenting the amount of optimally and suboptimally solved instances, and unsolved instances. Also, the total execution time is given for the LP solver.

The outputs can be found in the *outputs/* directory, inside folders *LP-i7-1/* and *LP-i7-4/*. Table **??** gives the list of unsolved instances and instances solved suboptimally.

Instance name			i7 (x1)	i7 (x4)	Optimal value
4	13	1	37		13
5	10	1	16		14
5	13	1	11		6
6	11	1	27	14	13
6	12	1	23		8
6	13	1	x	x	35
7	12	1		11	5
7	13	1	36	36	14
8	7	1	13		9
8	9	1	25		7
8	10	1	x	28	9
8	12	1	x	11	7
8	13	1	x	x	17
9	11	1	x	x	6
9	12	1	x	10	3
9	13	1	x	34	15
10	9	1	22	24	6
10	10	1	x	25	4
10	11	1	19	9	5
10	12	1	x	16	5
10	13	1	47	47	14
11	6	1		5	4
11	8	1	x	32	12
11	10	1		30	5
11	11	1	x		3
11	12	1	x	28	6
11	13	1	x	24	4

Table 4: Instances in the *inputs/material/* directory solved suboptimally and the roll's length found. An empty cell in the table indicates that the instance was solved optimally, and an x that no solution was produced.

In table 4 we can see that using more computation power helps, but some odd behaviour can be spotted. For example, instances $(7, 12, 1)$, $(10, 9, 1)$, $(11, 6, 1)$ and $(11, 10, 1)$ where using four threads led to a worse solution. However, this is a minor issue since these are outnumbered by those instances not solved when using one thread but solved, although with a bad-quality solution, when using four threads.

For further details, see the output of the "benchmark.sh" script in the files "scripts/LP-i7-1-benchmark-log" and "scripts/LP-i7-4-benchmark-log", respectively.

6.3 Satisfiability

The parameters used to execute the Satisfiability solver are the following:

- The execution of the whole framework depicted in figure 7 has a time limit of 120 seconds.
- The SAT solver module has a time limit of 120 seconds within the total 120 seconds available (if the framework has been executed for 100 seconds and the SAT solver needs to be executed again then it is allowed to run for only 20 seconds).

- The encoding used for the "at most constraint" is the Heule encoding (see equation 18) since it seems to be the one that makes the framework run the fastest (see table 7).

As mentioned in section 5 the solver used is **lingeling**[1]. This solver has several command line options. However, none of them was used. The results are presented in table 5.

	i7	p4
Optimal solutions	108 (9) / 108	106 (12) / 108
Sub-optimal solutions	0 / 108	2 / 108
Unsolved instances	0 / 108	0 / 108
Total time	27 mins	45 mins

Table 5: Summary presenting the amount of optimally and suboptimally solved instances, and unsolved instances. Also, the total execution time is given for the SAT framework.

The only machine not powerful enough to solve (presumably) all instances optimally was the Pentium IV (*p4*). These instances are shown in table 6.

Instance name			P4	Optimal value
6	13	1	38	35
10	13	1	42	14

Table 6: Instances in the *inputs/material/* directory solved suboptimally and the roll's length found. An empty cell in the table indicates that the instance was solved optimally.

Table 7 shows a performance comparison of the framework when solving several instances using the different encodings. The performance is the time, in seconds, of a single execution. Although there is not much statistical significance in the measurement of the time of a single execution (see for example the execution times for instance $(4, 12, 1)$), some of the differences are significant enough to determine that the Heule encoding is the best for the scope of this project (see for example the execution times for instances $(3, 12, 1)$, $(6, 12, 1)$, $(11, 12, 1)$).

Instance name			$\mathcal{Q}$	$\mathcal{L}$	$\mathcal{H}$
3	12	1	3.86	5.71	3.91
4	12	1	2.86	2.46	2.21
5	12	1	6.50	4.03	4.10
6	12	1	17.86	14.93	10.03
7	12	1	6.97	4.09	5.47
8	12	1	87.03	42.10	56.76
9	12	1	7.69	7.49	5.76
11	12	1	15.38	9.11	7.6

Table 7: Time in seconds that the framework in figure 7 needed to solve the different instances for the different encodings of the "at most one" constraint. $\mathcal{Q}$ for the quadratic, $\mathcal{L}$ for the logarithmic, and $\mathcal{H}$ for the Heule encoding, as formalised in section 5, equations 16, 17 and 18, respectively.

7 Conclusions

The aim of this project was to use three different strategies to solve the Box Wrapping Problem, an optimisation problem with a huge combinatorial space. Two of the strategies applied are not

defined to solve optimisation problems (the Constraint Programming paradigm, in section 3, and the Satisfiability paradigm, in section 5) and the other was defined for this particular purpose (the Linear Programming paradigm, in section 4). The latter, in spite of being specifically made to solve such type of problems, lacks the ability to solve problems with a significant constraint-solving component, one of the reasons we had to apply, not the "pure" Linear Programming approach, but the Mixed-Integer Linear Programming.

This project has shown that all of these strategies can be applied to solve this problem, though some of them more easily than others: the Constraint Programming technique only required the addition of extra constraints (see section 3.2), the Linear Programming technique the definition of an objective function (see 4.2) and Satisfiability required the implementation of a whole framework that uses a SAT solver as a black box to find an optimal solution (see section 5.2). The difficulty in finding the optimal solution increases with every technique.

Despite of this, the most difficult technique seems to be most efficient. Table 5 shows that with the Satisfiability paradigm we could find the optimal solution to 99 instances (the other 9 are known to be optimal due to the existence of a hand-made solution to the corresponding instance, see discussion at the beginning of section 6), while tables 1 and 3 show, respectively, that the Constraint Programming approach could find 88 optimal instances (and 12 presumably optimal) and that the Linear Programming approach could find 80 optimal instances (and 8 presumably optimal). Therefore, it seems that Satisfiability seems to be the best approach to tackle the BWP.

However, it is worth mentioning that the license of CPLEX (see [3]), used for the Linear Programming part, is "academical": it is quite likely that a complete license of this software might be more efficient and even outperform the Gecode library (see [5]) and/or **lingeling** (see [1]).

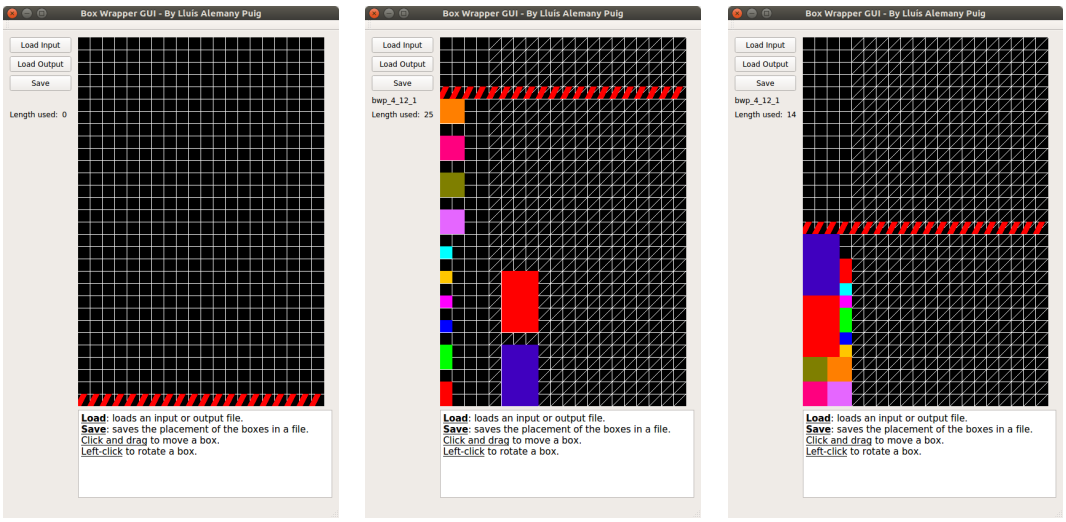
It could be argued that the mathematical modelling of the problem (see section 2) is rather simple. For example, constraints to stop the solvers from trying to improve a soluion by swaping two identical boxes (avoidance of symmetries) should have been implemented. Admittedly, this could have speeded up the solvers, but it has been proven that a few and simple heuristics can also help, even a great deal, in finding optimal solutions. Furthermore, the lack of these constraints does not seem to be an issue in the Satisfiability part. However, these solvers were only tested on rather small instances.

Taking everything into account, it seems that the Satisfiability approach seems to be the best for solving this problem, as long as there is no more evidence in favour of the Linear Programming approach.

A Box wrapper GUI

The Box Wrapper is a very simple user interface implemented with Qt 5.5 (see [4]), and OpenGL (see [2]), 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 15 for three screenshots of the software.



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

Figure 15: The different states of the software. We can load an instance (15b) and edit the positions of the boxes and their rotation to obtain a solution (15c).

B Conversion of non-linear linear expressions

In this section are shown two simple examples on how to transform logical constraints that relate Boolean variables into linear expressions.

B.1 3 or

Assume three Boolean variables a, b, c related as follows:

$$(a = 1) \vee (b = 0) \vee (c = 1) \tag{19}$$

and we want to express the same condition but with a series of linear constraints. First, use the associative property of the logical $\vee$ to obtain: $((a = 1) \vee (b = 0)) \vee (c = 1)$. Now, introduce a new Boolean variable δ so that

$$(\delta = 1) \iff ((a = 1) \vee (b = 0)) \;\equiv\; (\delta = 1) \iff (b \leq a)$$

Now, using the transformations explained in section 4 we obtain two new linear expressions

$$b - a \leq \mathcal{U}(1 - \delta), \quad b - a \geq (\mathcal{L} - 1)\delta + 1$$

Taking $\mathcal{U} = 1$ and $\mathcal{L} = -1$ as upper and lower bounds for $b - a$ we get

$$b - a \leq 1 - \delta, \quad b - a \geq -2\delta + 1 \tag{20}$$

Our claim is that if the three linear expressions in 21 are satisfied then the logical constraint in 19 will be true.

$$\delta + c \geq 1, \qquad b - a \leq 1 - \delta, \qquad b - a \geq -2\delta + 1 \tag{21}$$

For this, we can test all 8 possible combinations of values for a, b, c. For each pair of values for a, b the algorithm that solves the linear constraints gives a value to variable δ that satisfies the expressions in 20. This value is then used to check that the constraint $\delta + c \geq 1$ is satisfied for the given value of c. If it is, then the logical constraint in 19 is satisfied, that is, it is true that either $a = 1$, $b = 0$, $c = 1$. Table 8 shows the evaluation of all the linear expressions for all possible combinations of $\{0, 1\}^3$.

	a	b	c	$(b - a \leq 1 - \delta) \wedge (b - a \geq -2\delta + 1)$	δ	$\delta + c \geq 1$
*	0	0	0	$(0 \leq 1 - \delta) \wedge (0 \geq -2\delta + 1) \equiv 1/2 \leq \delta \leq 1$	1	•
*	0	0	1		1	•
	0	1	0	$(1 \leq 1 - \delta) \wedge (1 \geq -2\delta + 1) \equiv 0 \leq \delta \leq 0$	0	
*	0	1	1		0	•
*	1	0	0	$(-1 \leq 1 - \delta) \wedge (-1 \geq -2\delta + 1) \equiv 1 \leq \delta \leq 2$	1	•
*	1	0	1		1	•
*	1	1	0	$(0 \leq 1 - \delta) \wedge (0 \geq -2\delta + 1) \equiv 1/2 \leq \delta \leq 1$	1	•
*	1	1	1		1	•

Table 8: The cases where the constraint 19 is satisfied (marked with "*") and the cases where the constraints in 21 are satisfied (marked with •).

B.2 Several "and"

Transforming a series of logical "and" ($\wedge$) operations is fairly trivial. Given Boolean variables $x_1, \cdots, x_n$ each equal to some Boolean value $b_1, \cdots, b_n$, the logical constraint

$$\bigwedge_{i=1}^{n} (x_i = b_i) \tag{22}$$

is transformed into the following equality:

$$\sum_{i:b_i=1} x_i + \sum_{i:b_i=0} 1 - x_i = n \tag{23}$$

which in turn is transformed into two inequalities $\geq$,$\leq$.

The only assignment to the variables $x_1, \cdots, x_n$ that satisfies equation 23 is $b_1, \cdots, b_n$. The proof is done by contradiction. Assume there are p variables that must be 1 and z variables that must be 0. If the assignment is $\phi = b_1, \cdots, b_n$ then:

$$\sum_{i:b_i=1} \phi_i + \sum_{i:b_i=0} 1 - \phi_i = p + z = n$$

Let ϕ' the same assignment as ϕ but, if $b_i = 1$ then $\phi'_i = 0$. In this case we have that:

$$\sum_{i:b_i=1} \phi'_i + \sum_{i:b_i=0} 1 - \phi'_i = p - 1 + z \neq n$$

Conversely, if $b_i = 0$ let $\phi'_i = 1$. Then, we have

$$\sum_{i:b_i=1} \phi'_i + \sum_{i:b_i=0} 1 - \phi'_i = p + z - 1 \neq n$$

References

[1] Armin Biere. *Lingeling, Plingeling and Treengeling*. URL: http://fmv.jku.at/lingeling/ (visited on 06/02/2018).

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

[3] IBM. *IBM CPLEX Optimizer for z/OS*. URL: https://www.ibm.com/support/knowledgecenter/SS9UKU_12.7.0/com.ibm.cplex.zos.help/CPLEX/homepages/CPLEX_Z.html (visited on 05/16/2018).

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

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