Box wrapping solver

Rodrigo Arias Mallo

June 18, 2018

1 Constraint programming

The solver `Gecode` was used to implement the variables and constraints, detailed in the `src/cp.cc` file.

1.1 Variables

Two integer variables x_i and y_i were used to set the *top-left* corner of each box i. The boolean array rot_i determines whether the box i is rotated (90 degrees) or not. In that case, the width and length are swapped for the box. The variable L measures the length of the roll, and is used to set a constraint for each solution found, such that the next one needs to be at most $L - 1$.

```
L = IntVar(*this, min_L, max_L);
x = IntVarArray(*this, num_boxes, 0, W);
y = IntVarArray(*this, num_boxes, 0, max_L);
rot = BoolVarArray(*this, num_boxes, 0, 1);
```

Some limits like $\min_L$ and $\max_L$ are computed from the total area of the boxes and the maximum length, when the boxes are vertical and stacked, to limit the range of the variables L and y. The boxes are placed by using the coordinates of the corners. So a box at $(0, 0)$ of width 2 and height 3 ends at $(2, 3)$.

1.2 Constraints

We enforce that L must be at least the lower part of each box, and W the right side, so each box is placed inside the roll. If the box is rotated, then we swap w_i and h_i.

```
rel(*this, (rot[i] && (L >= y[i] + w[i])) || (!rot[i] && (L >= y[i] + h[i])));
rel(*this, (rot[i] && (W >= x[i] + h[i])) || (!rot[i] && (W >= x[i] + w[i])));
```

When a box is a square, we don't test for rotations.

```
if(w[i] == h[i]) rel(*this, rot[i] == 0);
```

Finally, for each pair of boxes i and j with $i \neq j$, we add the following constraint.

```
rel(*this,
        (!rot[i] && (x[i] + w[i] <= x[j])) ||
        (!rot[j] && (x[j] + w[j] <= x[i])) ||
        (!rot[i] && (y[i] + h[i] <= y[j])) ||
        (!rot[j] && (y[j] + h[j] <= y[i])) ||
```

```
        ( rot[i] && (x[i] + h[i] <= x[j])) ||
        ( rot[j] && (x[j] + h[j] <= x[i])) ||
        ( rot[i] && (y[i] + w[i] <= y[j])) ||
        ( rot[j] && (y[j] + w[j] <= y[i]))
);
```

The last four parts are equal to the first four, but when a box is rotated. Each of the four test that there is a direction in which the distance within the boxes is at least 0, in that case the box is allowed to be placed there.

1.3 Branching policy

In order to determine how the variables are explored, some policies were tested to find the fastest one. First we branch by *rot* with the default value 0, (no rotation):

```
branch(*this, rot, BOOL_VAR_RND(r), BOOL_VAL_MIN());
```

Then we branch x and y, based on the policy selected. By default, the policy 0 is:

```
branch(*this, x, INT_VAR_RND(r), INT_VAL_RND(r));
branch(*this, y, INT_VAR_RND(r), INT_VAL_RND(r));
```

Whether the policy 1 is the following.

```
branch(*this, x, INT_VAR_SIZE_MIN(), INT_VAL_RND(r));
branch(*this, y, INT_VAR_SIZE_MIN(), INT_VAL_RND(r));
```

Finally, we branch on L using the minimum value posible:

```
branch(*this, L, INT_VAL_MIN());
```

The branching policy can be selected from the command line, using the argument `-p <number>` using 0 or 1. Use `-h` for more information.

1.4 Compilation

In order to compile the code, check that the `PREFIX` variable in the `src/Makefile` is set to the correct value. In my system Gecode is installed in `/usr` but you may want to use `/usr/local`. Then, to compile, execute `make`.

1.5 Running

The solver `cp` reads from the standard input, and writes to the standard output. To read an instance you can use:

```
% src/cp < in/bwp_10_4_1.in
3
9 2     9 2
3 0     5 2
6 0     8 2
0 0     2 2
```

Other options are available and are useful for debugging purposes, see `src/cp -h`. A `plot` program draws the box placement, but it needs to use an extended notation (`-e`). In the extended notation, the width of the paper roll is included in the first line, and the rotated boxes are indented. Then it can be used as:

```
% src/cp -e < in/bwp_10_4_1.in | src/plot
Plotting a roll of W = 10 and L = 3 with 4 boxes.
      0  1  2  3  4  5  6  7  8  9
   +------------------------------+
 0 |  4  4  4  2  2  2  3  3  3   |
 1 |  4  4  4  2  2  2  3  3  3   |
 2 |  4  4  4  2  2  2  3  3  3  1|
   +------------------------------+
```

Other options are documented in the builtin help:

```
% src/cp -h
Constraint programming solver for the BWP.
Usage

        src/cp [options] < file.in > file.out

Options

  -h       Show this help

  -t <s>   Set the time limit to <s> seconds. Default 120.

  -c <c>   Set the number of threads to <c>. Default 1.

  -e       Use the extended notation, printing the width
           of the paper roll in the first line along the length.
           Needed for plotting but not compatible with checker.

  -v       Be verbose.

  -p <p>   Set the policy number <p> of the branching method for x and y.
           By default is 1, available policies:
             0 - INT_VAR_RND(r) and INT_VAL_RND(r)
             1 - INT_VAR_SIZE_MIN() and INT_VAL_RND(r)
```

2 Linear programming

The CPLEX solver with the C++ interface was used to implement the variables and constraints, detailed in the `src/lp.cc` file.

2.1 Variables

The same variables were used, x_i and y_i to set the *top-left* corner of the box i. As the rot_i boolean array, and the length of the paper L.

```
IloIntVarArray x, y;
IloBoolVarArray rot;
IloIntVar L;
```

Also the same limits where used, like $\min_L$ and $\max_L$ for L.

```
L = IloIntVar(env, min_L, max_L);
x = IloIntVarArray(env, num_boxes, 0, W);
y = IloIntVarArray(env, num_boxes, 0, max_L);
```

2.2 Constraints

To enforce that the boxes should be inside the paper roll, we first compute the real width and height of each box, in case they are rotated:

```
IloExpr wi = (1 - rot[i]) * w[i] + rot[i] * h[i];
IloExpr hi = (1 - rot[i]) * h[i] + rot[i] * w[i];
```

Then, we limit the right and bottom side, as the right and top are at least 0:

```
model.add(W >= x[i] + wi);
model.add(L >= y[i] + hi);
```

Also, if a box is a square, we set rot_i to 0:

```
if(w[i] == h[i]) model.add(rot[i] == 0);
```

In order to avoid overlaping between two boxes i and j, with $i \neq j$, we first compute the real size of each box:

```
IloExpr wi = (1 - rot[i]) * w[i] + rot[i] * h[i];
IloExpr hi = (1 - rot[i]) * h[i] + rot[i] * w[i];
IloExpr wj = (1 - rot[j]) * w[j] + rot[j] * h[j];
IloExpr hj = (1 - rot[j]) * h[j] + rot[j] * w[j];
```

And then, we enforce that at least in one direction, the distance between the two boxes should be at least 0:

```
model.add(
                // x axis
                x[i] + wi <= x[j] ||
                x[j] + wj <= x[i] ||
                // y axis
                y[i] + hi <= y[j] ||
                y[j] + hj <= y[i]);
```

In order to avoid duplicate placements of boxes with equal size, we assign an index to each box i and j with $i < j$ based on their position:

```
IloExpr pi = x[i] + y[i]*W;
IloExpr pj = x[j] + y[j]*W;
```

And we only let the first box to have a smaller index:

```
model.add(pi <= pj);
```

Finally, we add the objective to minimize L:

```
model.add(IloMinimize(env, L));
```

2.3 Compilation

To compile, under the `src/` directory, type `make`.

2.4 Running

The program `src/lp` reads the input file from the standard input. To solve a specific instance use `src/lp < in/instance.in`. To run all the instances in `in/` use the script `./run.sh lp`.

3 Propositional satisfiability

The Lingeling solver was used to find a solution to the box wrapping problem, by encoding the constraints with logical clauses in CNF form. In order to generate the format required by the solver, a python program reads the input and writes to the solver standard input.

3.1 Variables

The rotation of each box i is determined by the R_i variable. In order to encode the position of the box, we use the notion of starting cell. The starting cell is the top-left coordinate of the box, and the variable $S[x, y, i]$ is true if and only if the box i is placed at (x, y). Also, another auxiliary variable, $P[x, y, i]$ determines if the cell (x, y) is occupied with the box i, in order to avoid overlapping.

The x value range is in $[0, W)$ and the y in $[\min_L, \max_L]$. The value $\min_L$ is computed from the area of the boxes, and determines the minimum length that could be used for the given area. The maximum length $\max_L$ is computed by placing all the boxes in a vertical line, rotated to reach the maximum length.

3.2 Constraints

Two main groups of constraints are used, one group for each box and another for each cell. An extra group is used to minimize L in each call to the solver. All the constraints are transformed to clauses in CNF form.

3.2.1 Box constraints

All these constraints are iterated for each box i and they assume that the rotation of the box is r, with $r = 0$ for no rotation, and $r = 1$ otherwise. We define the formula $RM(r, i)$ to be $\neg R_i$ if $r = 0$, and R_i if $r = 1$.

The following constraints are added first assuming r to be false. If the box is not square, then we add the constraints again, but assuming r true. Otherwise, we just fix R_i to be false, by adding $\neg R_i$ to the clausules.

Domain of the box Given a starting cell (x, y) for the box i with rotation r, all the cells occupied with the box are defined with the set $\Omega_P(x, y, i, r)$. We define the function $D(x, y, i, r)$ as a logic formula that enforces the cells occupied by the box in $P[x, y, i]$ to be true:

$$D(x, y, i, r) = \forall (u, v) \in \Omega_P(x, y, i, r) : S[x, y, i] \rightarrow P[u, v, i]$$

Then, we repeat the constraint for all the posible starting cells, given by the set $\Omega_S(i, r)$. The implication only enforces the constraint when the rotation of the box follows the assumption.

$$\forall (x, y) \in \Omega_S(i, r) : RM(r, i) \rightarrow D(x, y, i, r)$$

Force exactly one start cell In order to place the box, exactly one starting cell $S[x, y, i]$ must be true. Assuming the rotation r, we first compute the posible starting cells in $\Omega_S(i, r)$, then we enforce that at least one is true.

$$RM(r, i) \rightarrow \bigvee_{(x,y) \in \Omega_S(i,r)} S[x, y, i]$$

In the other way, we must restrict the number of start cells to be at most one. We can use the quadratic encoding to restrict $S[x, y, i]$. For all pairs of sorted starting cells $s_j = (x_j, y_j)$ and $s_k = (x_k, y_k)$, with $j < k$, we enforce that both cannot happen at the same time.

$$R_i \rightarrow \neg(S[x_j, y_j, i] \wedge S[x_k, y_k, i])$$

Deny start cell outside domain The cells in the roll outside the start domain $\Omega_S(i, r)$ should be enforced to be false.

$$\forall (x, y) \notin \Omega_S(i, r) : RM(r, i) \rightarrow \neg S(x, y, i, r)$$

3.2.2 Cell constraints

These constraints are placed for each cell c of the roll.

At most one cell occupied We enforce each cell $c = (x, y)$ to be occupied by at most one box. We use the quadratic encoding for each pair of boxes i, j, with $i < j$.

$$\neg(P[x, y, i] \wedge P[x, y, j])$$

3.3 Extra constraints

In order to minimize the length of the roll L, we use the following approach: We fix L to a value L' and use it as the length of the roll. If the solver finds a solution we decrease L' until the problem becomes unsatisfiable. The last satisfiable solution is the one with minimum L.

This search is performed by the binary search procedure, by spliting the values of L by half, and rejecting the half according to the result of the solver. The extra constraints are added and removed from the problem at each try, and disallow any box to be placed below the limit. We define $\Omega_L(L', \max_L)$ to be the set of cells below the limit L'. Then we enforce that no cell of any box can be placed at these cells.

$$\forall i, \forall (x, y) \in \Omega_L(L', \max_L) : \neg P[x, y, i]$$

3.4 Running

The solver `lingeling` must be installed and found via the `$PATH`, as well as python 3 with the `numpy` package. The program `sat` reads the instance from the standard input, executes the solver and writes the output to the standard output. To run all instances use the script `./run.sh sat`.

4 Automatic test

In order to execute the solvers in all instances, a script was prepared for the task. It reads the technologies to be used from the parameters. For instance to execute with `lp` only, use `./run.sh lp`. Or with `cp` and `lp`, use `./run.sh cp lp`.

The script checks before that the solution was not already computed, otherwise skips the computation. If you want to try again some instances that were solved, remove the solutions first.

The instances are read from the `in/` directory, and the solutions and logs are placed in the `out/` directory. Check the log file for a specific instance to see the details of the solver. The output file is named `bwp_`W`_`N`_`k`_`t`.out` as expected, and the log file as `bwp_`W`_`N`_`k`_`t`.log`.

Once a solution is produced by the solver, the `checker` test if it was correct. In case it detects some error, the solution is saved in a temporal file appending `.tmp` to the file name. The output of the checker can be read in `bwp_`W`_`N`_`k`_`t`.chk`.

This behavior allows to determine which instances were incorrectly solved. If the program doesn't terminate after the time limit of 120 seconds, it is killed, and the solution is also considered incorrect, and `.tmp` is also appended. In case you want to stop the script, use Control+C to abort the computation:

```
hop% ./run.sh cp
Using solver cp
bwp_3_2_1.in... SKIP
bwp_3_3_1.in... SKIP
bwp_3_4_1.in... SKIP
bwp_3_5_1.in... SKIP
bwp_3_6_1.in... SKIP
bwp_3_7_1.in... SKIP
bwp_3_8_1.in... SKIP
bwp_3_9_1.in... OK
bwp_3_10_1.in... OK
bwp_3_11_1.in... ^C
hop%
```