Box wrapping solver

Rodrigo Arias Mallo

May 30, 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

2.4 Running

For compiling, use `make` in the `src/` directory. 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 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%
```