Code binpack3d: User Manual

Silvano Martello*, David Pisinger†, Daniele Vigo*,
Edgar den Boef§, Jan Korst§

*DEIS, University of Bologna, Italy
†DIKU, University of Copenhagen, Denmark
§Philips Research Laboratories, Eindhoven, The Netherlands

1 The Code

Algorithm `binpack3d`, described in Martello et al. [2], was coded in ANSI-C and compiled on several platforms using `cc` and `gcc` compilers. In particular, the code has been tested with the `-pedantic` option of `gcc` to ensure that the ANSI-C standard is strictly respected. The n boxes are here numbered from 0 to $n-1$. Box positions are referred to a coordinate system having its origin in the lower-left-backward corner of the bin.

The algorithm comes with program `test3dbpp` that either reads an instance from an input file or constructs some randomly generated instances presented in Martello, Pisinger and Vigo [1].

A prototype of `binpack3d` appears as

```
void binpack3d(int n, int W, int H, int D, int *w, int *h, int *d,
               int *x, int *y, int *z, int *bno, int *lb, int *ub,
               int nodelimit, int iterlimit, int timelimit, int *nodeused,
               int *iterused, int *timeused, int packingtype)
```

where the parameters are:

`n`	size of the problem, i.e., number of boxes to be packed;
`W,H,D`	width, height and depth of every bin;
`w,h,d`	integer arrays of length `n`, where `w`[`j` − `1`] $= w_j$, `h`[`j` − `1`] $= h_j$, `d`[`j` − `1`] $= d_j$ are the dimensions of box j $(j = 1, \dots, n)$;
`x,y,z,bno`	integer arrays of length `n`, where the solution found is returned. For each box j $(j = 1, \dots, n)$, `bno[j-1]` is the bin number it is packed into, and `x[j-1]`, `y[j-1]`, `z[j-1]` are the coordinates of its lower-left-backward corner;
`lb`	best lower bound on the solution value obtained;

`ub`	objective value of the solution found, i.e., number of bins used to pack the n boxes. If an optimal solution was found, then `ub`=`lb`;
`nodelimit`	maximum number of decision nodes to be explored in the main branching tree (in thousands). If set to zero, the algorithm will run until an optimal solution is found (unless either `timelimit` or `iterlimit` is reached);
`iterlimit`	maximum number of iterations in the ONEBIN algorithm which packs a single bin (in thousands). If set to zero, the algorithm will run until an optimal solution is found (unless either `timelimit` or `nodelimit` is reached);
`timelimit`	time limit for solving the problem, expressed in seconds. If set to zero, the algorithm will run until an optimal solution is found (unless either `nodelimit` or `iterlimit` is reached); otherwise, no new branching node will be explored after `timelimit` seconds;
`nodeused`	a pointer to an integer where the number of nodes in the main branch and bound tree is returned;
`iterused`	a pointer to an integer where the total number of iterations in ONEBIN is returned;
`timeused`	a pointer to an integer where the used time in milliseconds is returned;
`packingtype`	desired packing type. If set to zero, the algorithm will search for an optimal general packing; if set to one, it will search for a robot packing.

The general structure of the code is shown in Figure 1. Initial lower bounds and heuristic solutions are computed at the root node. If the problem is not solved to optimality, the tree enumeration is recursively performed through procedure `rec_binpack`.

2 Implementation details

All parameters concerning the boxes are passed to algorithm `binpack3d` as integer arrays to form a simple interface to other programs. However, internally the algorithm stores the information of each box in a structure `box` that contains the following fields:

`no`	original (input) number of the box. This number follows the box throughout the algorithm for easy identification and debugging;
`w,h,d`	width, height and depth of the box;
`x,y,z`	coordinates of the current position of the lower-left-backward corner of the box;
`bno`	number of the bin the box is currently assigned to. If `bno` = 0 the box is not currently assigned to any bin;
`k`	boolean variable indicating whether the current box was selected when solving the single-bin subproblem;
`vol`	volume of the box.

```
1.   copy the input information to internal structures;
2.   compute lower bound $L_2$;
3.   execute heuristics $H1$ and $H2$, and set $u$ to the best solution value found;
4.   while no optimal solution is found or stopping criterion is met do
         comment: perform the tree enumeration;
4.1      assign the next box to an open bin or a new bin;
4.2      check feasibility of the assignment with onebin_decision;
4.3      if the assignment is not feasible then backtrack;
         else
             check the current solution for (non-)optimality;
             check node limit and time limit;
             for each open bin do
                 if the bin can be closed then
                     compute a new lower bound and possibly backtrack
             end for
         end if
5.   return the best solution found.
```

Figure 1: Structure of procedure `binpack3d`

All problem information is stored in a structure `allinfo` that contains the problem data and execution parameters, working data, bound values at the root node, as well as debugging, control and statistic information. Additional structures are used for the heuristic algorithms (`heurpair`) and the constraint programming algorithm (`domainpair`).

Different upper and lower bound values may be obtained by considering the instance according to the three different orientations. To this purpose, the computations are performed for a single orientation (procedures `bound_two_x`, `dfirst_heuristic` and `mcut_heuristic`): the overall lower and upper bound values are then obtained (through procedures `bound_two`, `dfirst3_heuristic` and `mcut3_heuristic`) making use of a routine `rotate_problem` that rotates the dimensions as $w \leftarrow h \leftarrow d \leftarrow w$ and $W \leftarrow H \leftarrow D \leftarrow W$. This routine is always called three times, so that all variables are restored to their original values after the three iterations. When deriving upper bounds, a routine `rotate_solution` is executed with `rotate_problem`, in order to ensure that solution coordinates follow the orientation of the instance.

References

[1] S. Martello, D. Pisinger, D. Vigo (2000), “The Three-Dimensional Bin Packing Problem”, *Operations Research* 48, 256–267.

[2] S. Martello, D. Pisinger, D. Vigo, E. den Boef and J. Korst (2006), “Algorithms for General and Robot-Packable Variants of the Three-Dimensional Bin Packing Problem”, *ACM Transactions on Mathematical Software* (to appear).