Heuristics for Packing Semifluids

João Pedro Pedroso

Technical Report Series: DCC-2016-01

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL
Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

Heuristics for Packing Semifluids

João Pedro Pedroso

June 2015

Abstract

Physical properties of materials are seldom studied in the context of packing problems. In this work we study the behavior of semifluids: materials with particular characteristics, that share properties both with solids and with fluids. We describe the importance of some specific semifluids in an industrial context, and propose methods for tackling the problem of packing them, taking into account several practical requirements and physical constraints. Although the focus of this paper is on the computation of practical solutions, it also uncovers interesting mathematical properties of this problem, which differentiate it from other packing problems.

Keywords: Packing; Semifluid; Heuristics; Tree search.

1 Introduction

Semifluids are materials having characteristics of both fluids and solids. In the context of this paper, we will consider materials that cannot flow in one direction, though they are fluid in the other directions. As an example, consider tubes, which correspond to the industrial origin of this problem. Placed in a container, they can flow in the directions perpendicular to their length, but *not* in the direction of their length (see Figure 1). Assuming the tubes will be positioned perpendicularly to the Cartesian axes, depending on the direction of their placement they will flow either in the x or in the y dimension. Pipes, having positive radii, are imperfect semifluids, as they will not fully occupy the space available in the z dimension; however, they approximate a perfect fluid as the radii becomes smaller. We will consider that the material is a perfect semifluid, and hence the volume occupied is constant and divisible.

This paper describes several possibilities for packing semifluids in a container, and presents heuristics for the variant which closer corresponds to an industrial application.

2 Problem description

Even though packing problems may be generalized into a single problem, they are usually divided in two categories: minimizing the number of bins, and maximizing the load to pack in a bin (see, *e.g.*, [2]). Given an index set $\mathscr{S}$ of semifluid items, with each item i characterized by a fixed length ℓ_i and a volume v_i, and dimensions D, W, H of containers, these two variants for the problem of packing a semifluid are:

1. bin packing variant: find the minimum number of containers to accommodate all the items;
2. knapsack variant: given, additionally, a value w_i for the available volume v_i of each item i, find the packing of maximum value that can be inserted in a container.

In this paper we will focus on the knapsack variant.

2.1 Semifluid packing problems

There are several possibilities for packing a semifluid orthogonally in a container, as shown in Figure 2. Both the length ℓ (corresponding to the length of the tubes) and the volume v occupied by the semifluid are constant;

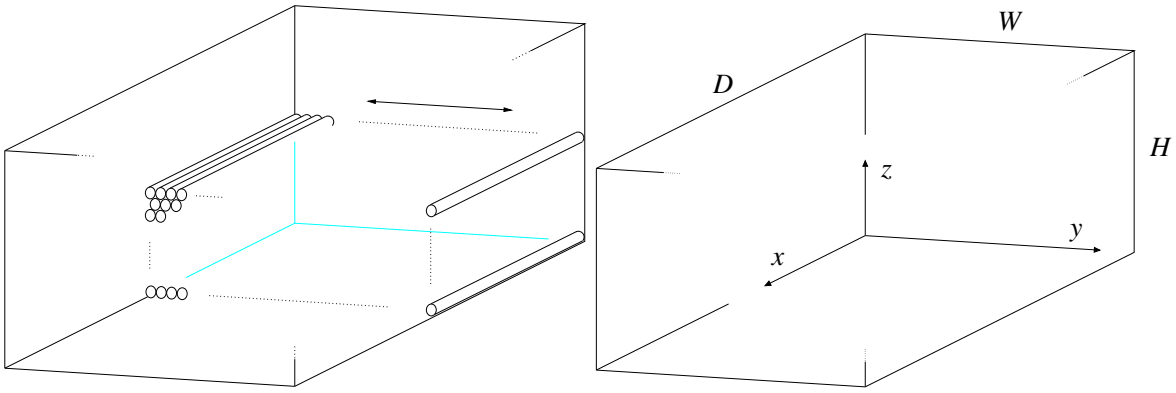

Figure 1: A container accommodating a semifluid: tubes (left); coordinate system used (right).

Figure 2: Two possibilities for accommodating a semifluid in a container.

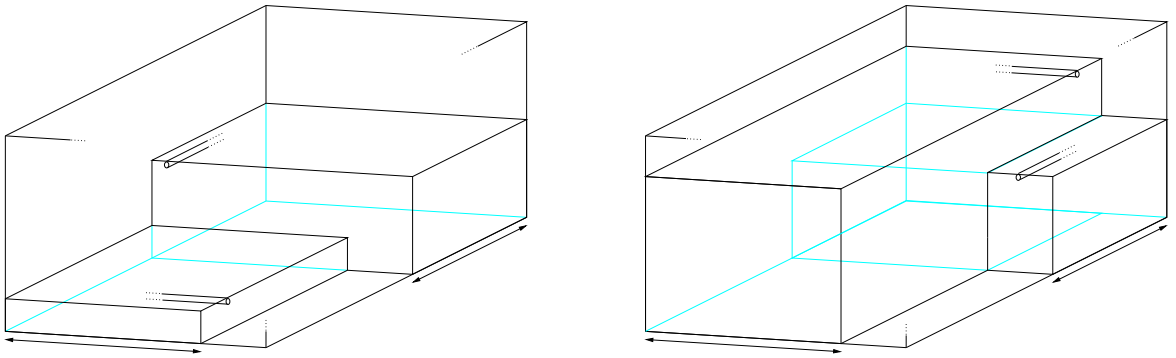

Figure 3: Packing a semifluid without overflowing another previously packed (left), and overflowing it (right).

in this figure, this means that $a \times b \times \ell = c \times d \times \ell = v$. Assuming that, except the container itself, there are no walls, a semifluid will take on all the available horizontal space in the direction where it freely flows. In the case presented, a would take the depth D of the container, and c would take its width W, and hence the corresponding heights are $b = \frac{v}{D\ell}$ and $d = \frac{v}{W\ell}$. After a semifluid is placed, others may be put on top of it, but they must not protrude (as detailed next). Hence, one may think of the space above a semifluid as a "container", which can be filled up with the same rules as the original container; in this sense, this is a recursive problem.

Depending on the application, it may be allowed or not that, when packing a semifluid, it overflows others previously packed, as illustrated in Figure 3. In general, allowing overflow makes packing solutions more difficult to implement in practice, and brings the problem more difficult to tackle; overflow will not be considered here. We will focus on packing semifluids by positioning the fixed dimension parallel to the x axis, as shown in Figure 1. This is the relevant variant when the container must be loaded from a lateral door at $x = D$: if the semifluids were rotated and be placed along the y axis, they would flow out of the door.

An important, practical packing rule restricts what can be placed on top of what. Indeed, for cargo stability and for facilitating loading, it is usually acceptable that shorter tubes are placed on top of longer tubes, but not the inverse; more precisely, there must be no holders protruding with respect to holders below them.

In semifluid packing, any fraction of an item's available volume may be packed; this is major difference with respect to other packing problems.

We call the problem of maximizing the value of semifluids packed in the container in these conditions the *basic semifluid packing problem*.

2.2 Background

Three-dimensional packing has recently been studied under several different perspectives; a recent survey can be found in [4]. The problem of allocating a given set of three-dimensional rectangular items to the minimum number of identical finite bins without overlapping has been addressed with tabu search in [10]: items are packed in several layers, the floor of the container being the first. A heuristic method for the situation where there is no requirement for packed boxes to form flat layers, keeping track of empty space seen from different perspectives and using a look-ahead scheme for positioning, is presented in [9]. However, the nature of the basic semifluid packing problem is rather different of these three-dimensional packing problems. As will be seen later, there is more similarity between our problem and two-dimensional cutting. The most closely related problem is the orthogonal two-dimensional knapsack problem with guillotine patterns. Methods for tackling this problem are often based on a discretization of possible positions for the rectangles in the Cartesian plane (see, *e.g.*, [14, 15, 5]). A different approach is proposed in [6], providing an exact algorithm for higher-dimensional orthogonal packing; the algorithm is based on bounding procedures which make use of dual feasible functions, within a tree search procedure. With respect to these problems, semifluid packing has the property that it is not required to pack all the available volume of each item; in rectangle packing, this would correspond to being able to cut some of the rectangles at the time of packing. Another difference between semifluid packing and previously studied problems concerns the requirement of no protuberance of items above others; this requirement is naturally respected in two-staged guillotine cuts, but usually is not enforced in general guillotine patterns.

To the best of our knowledge, basic semifluid packing or equivalent problems have not been studied before.

2.3 Mathematical model

We are not aware of previous attempts to formulate the semifluid packing problem as a mathematical optimization model, but there are some related problems. Integer programming models for two-dimensional two-stage bin packing problem have been proposed in [11] and extended by [14] to the three-stage problem. In both cases, decision variables are related to the assignment of the items to bins, stripes or stacks. Models for the related cutting stock problem, providing better linear relaxation bounds, are presented in [15], where a set of small rectangular items of given sizes is to be cut from a set of larger rectangular plates, in such a way that the total number of used plates is minimized. Despite some similarities, none of these models is adequate for our problem, mainly for two reasons: in semifluid packing the number of stages is in general much larger, and items may be partially assigned to a position (*i.e.*, a position may hold a fraction of the available volume of an item).

The formulation proposed next is not compact, as it requires an exponential number of variables; however, it hopefully conveys the characteristics of the problem. For the sake of clarity, we start with a simplified model, and later describe how it could be extended to the general case; the simplification consists of assuming that only one stack of each item is allowed on each layer. A layer, in this context, is either the floor of the container or the space above a previously packed item. Figures 5 and 8 may be of help for visualizing the model.

The first set of binary variables indicates which items are packed in the first layer: $y_i = 1$ if item i is packed directly on the container, $y_i = 0$ otherwise. To each variable y_i there is a corresponding continuous variable $0 \leq x_i \leq 1$ which represents the fraction of item i being packed at this place. Before introducing more variables, let us specify a constraint related to the length D of the container, which limits the length of items packed in this layer:

$$\sum_i \ell_i y_i \leq D.$$

Variable x_i may be positive only if $y_i = 1$:

$$x_i \leq y_i, \quad \forall i.$$

The height of the first layer is limited by the height of the container:

$$h_i x_i \leq H, \quad \forall i,$$

where $h_i = v_i/W$ is the total height that item i would take on a container of width W (this will be later be replaced by a stronger constraint).

We now introduce variables concerning the placement of items j on the second layer, *i.e.*, directly above some previously packed item i. Variables y_{ij} are the indicators for this, and the corresponding x_{ij} represent the fraction of j packed at this place. The solution must, therefore, observe:

$$\begin{aligned} y_{ij} &\leq y_i, && \forall i, j, \\ x_{ij} &\leq y_{ij}, && \forall i, j, \\ \sum_j \ell_{ij} y_{ij} &\leq \ell_i y_i, && \forall i, \end{aligned}$$

where the last constraint limits the length of items placed directly above item i. For each pair i, j the height of the corresponding stack is limited to the height of the container:

$$h_i x_i + h_j x_{ij} \leq H, \quad \forall i, j.$$

The fraction of i used in the two first layers is limited by one (this and the previous constraints will later be extended):

$$0 \leq x_i + \sum_j x_{ji} \leq 1.$$

We now have all the components to complete the model, by extending the number of layers. Notice that, as layers cannot protrude and items with identical length should be placed by decreasing value, there may be at most N layers, where N is the number of items. We may assume that the items are reversely ordered by length, *i.e.*, $\ell_1 \geq \ell_2 \geq \ldots \geq \ell_N$; this allows us to define variables with indices i, i' only for $i' > i$. Notice also that the number of indices indicates the level at which the item corresponding to a variable is being packed: variables for layer $1 \leq K \leq N$ will have K indices $i, j, \ldots, m, n$, with $i < j < \ldots < m < n$. The entire model is presented in Figure 4.

Equations (2) determines the total quantity of item i packed, which allows determining the total value packed in (1). Constraints (3) to (4) guarantee that the total length of what is packed on top of the container, or of a packed item, does exceed the respective lengths. Constraints (5) to (6) allow a positive quantity of an item to packed only if the corresponding indicator variable is equal to 1. Constraints (7) to (8) allow packing only on top of previously packed items. Inequalities (9) determines the height of all stacks, and limits it to the height of the container. Finally, (10) to (11) define the domain for each of the variables.

This model is rather clumsy, but it is not yet complete: it does not take into account the possibility of packing several stacks of each item on a given layer. For make the model complete one would have to create, for each layer and each compatible item, a number of variables equal to the number of times that item would fit in the layer, if it was packed alone. It is obvious that direct usage of this model is implausible, except for a rather small number of items; realistic usage would require a column generation approach.

3 Heuristic and complete search

For solving the basic semifluid packing problem, we firstly propose a heuristic method — which will later be improved — for dividing a container into smaller parallelepipeds, which we call *holders*. Each holder has a fixed depth, determined by the length of the semifluid it will accommodate. Due to the possibility for the semifluid to flow downwards, along the z dimension, and also along the y dimension, a semifluid will fully use the width of the physical container. The height of a filled holder is determined either by the volume of its

$$\text{maximize} \quad \sum_i w_i \tilde{x}_i \tag{1}$$

subject to:

$$\tilde{x}_i = x_i + \sum_j x_{ji} + \sum_{k,j} x_{kji} + \ldots + \sum_{n,\ldots,j} x_{n\ldots ji}, \quad \forall i \tag{2}$$

$$\sum_i \ell_i y_i \leq D \tag{3}$$

$$\sum_j \ell_{ij} y_{ij} \leq \ell_i y_i, \quad \forall i$$

$$\ldots$$

$$\sum_n \ell_{ij\ldots mn} y_{ij\ldots mn} \leq \ell_{ij\ldots m} y_{ij\ldots m}, \quad \forall i, j, \ldots, m \tag{4}$$

$$x_i \leq y_i, \quad \forall i \tag{5}$$

$$x_{ij} \leq y_{ij}, \quad \forall i, j$$

$$\ldots$$

$$x_{ij\ldots n} \leq y_{ij\ldots n}, \quad \forall i, j, \ldots, n \tag{6}$$

$$y_{ij} \leq y_i, \quad \forall i, j \tag{7}$$

$$\ldots$$

$$y_{ij\ldots mn} \leq y_{ij\ldots m}, \quad \forall i, j, \ldots, n \tag{8}$$

$$h_i x_i + h_j x_{ij} + \ldots + h_n x_{ij\ldots n} \leq H, \quad \forall i, j, \ldots, n \tag{9}$$

$$0 \leq \tilde{x}_i \leq 1, \quad \forall i \tag{10}$$

$$0 \leq x_i \leq 1, \quad \forall i$$

$$\ldots$$

$$0 \leq x_{ij\ldots n} \leq 1, \quad \forall i, j, \ldots, n$$

$$y_i \in \{0,1\}, \quad \forall i$$

$$\ldots$$

$$y_{ij\ldots n} \in \{0,1\}, \quad \forall i, j, \ldots, n \tag{11}$$

Figure 4: Mathematical optimization model.

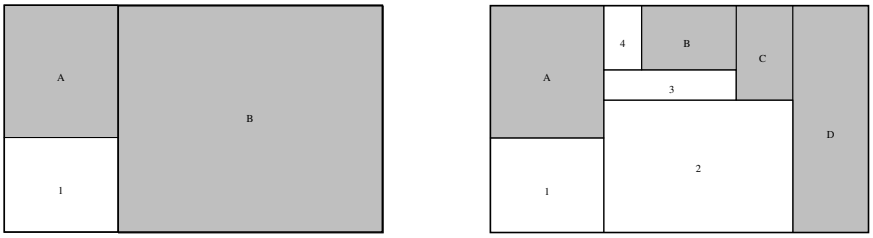

Figure 5: Section of a container through the $y = 0$ plane: open holders (shaded) after placing one item (left), and after placing four items (right).

semifluid or by the height of the physical container; in the latter case, the semifluid left over will possibly be packed in a different holder.

In this situation, one may think of the packing process as a division of, say, the container's wall at $y = 0$, into rectangles. Each rectangle corresponds to the volume of a particular item when projected into the $y = 0$ plane. For example, consider the placement of a semifluid as in Figure 1; a projection of the volume occupied is represented as a rectangle, alike 1 in the left diagram of Figure 5. Upon placing this item, the container is divided into three partitions: one where item 1 is held (which is *closed*, in the sense that it may not be used for other items), and the *open* holders above (A) and besides the item (B). Upon placing three more items in this example, the open holders are A, B, C, D in the right diagram of Figure 5.

3.1 Simple packing

A heuristic method for packing semifluids in these conditions can hence be though of as the process of choosing an item to pack, and an open holder for putting it (if some is available). For a semifluid of length ℓ, candidate holders j must have depth $D_j \geq \ell$. If the volume of a semifluid does not completely fit in the selected holder, the full height of the holder will be used (as for item 4 in the right diagram of Figure 5), and the remaining fluid is left to (possibly) pack later.

Given the characteristics of this problem, one might think of adapting known heuristics for bin packing and knapsack problems, as has been done for the two-dimensional knapsack problem (see, *e.g.*, [3, 5]); however, the geometric constraint forbidding longer lengths on top of shorter leads to possibly unexpected performance, as we will see shortly. Several alternative heuristic rules have been tried:

1. Best fit (BF): select the item/holder pair (i, j) which leads to the minimum difference $D_j - \ell_i$, *i.e.*, which leads to minimum currently unused space along x;

2. Longest item first, first fit (LFF): select the longest item that can be packed in some open holder (*i.e.*, item i with largest ℓ_i for which there exists a holder j such that $D_j - \ell_i \geq 0$), and insert it in the last open holder where it fits;

3. Longest item first, best fit (LBF): as LFF, but select the *smallest* open holder in which the item fits;

4. Worthiest item first, first fit (WFF): as LFF, but select most valuable items (per unit volume) first;

5. Worthiest item first, best fit (WBF): as LBF, but select most valuable items first.

These rules are used in the heuristic method detailed in Algorithm 1; we are abusing of notation, by allowing items and holders to be represented also by indices in their respective sets. The algorithm returns a map associating each item to the set of holders that contain it (which is empty for items that are not packed). The heuristic rule to be used is specified in line 5, and holders are created accordingly in the subsequent lines. The algorithm iterates as long as there is an open holder where some unpacked item fits.

The full description of the computational setup is deferred to Section 4; for the time being, we just present in Table 1 a comparison of the solutions obtained with these simple rules on a set of 3000 test instances. We have counted the number of times that heuristic construction with a rule is *strictly* better than with another, for all the combinations. The results obtained are rather surprising: rules based on the value of the items, very

Algorithm 1: Simple heuristic method for packing semifluids.

Data: instance:

- set $\mathscr{S}$ of items to pack
- item's length ℓ_i, volume v_i, and value w_i, $\forall i \in \mathscr{S}$
- physical container's width W, height H, and depth D;

Result:

- set of holders $\mathscr{H}$ and their dimensions and position inside the container;
- for each item i, the set x_i of holders where it is packed.

procedure `pack`$(D, W, H, \mathscr{S}, \ell, v, w)$
  $x_i \leftarrow \{\}, \quad \forall i \in \mathscr{S}$ *// initialize holders packing item i as empty sets*
  $\mathscr{H} \leftarrow \{\text{holder with dimensions } D \times W \times H\}$ *// open main holder*
  while *some item in $\mathscr{S}$ fits in an holder in $\mathscr{H}$* **do**
    $(i, j) \leftarrow \mathtt{h}(\mathscr{S}, \mathscr{H}, \ell, v, w)$ *// heuristic choice of item i and holder j*
    let D_j, W_j, H_j be the current dimensions of holder j
    $z \leftarrow v_i/(\ell_i W_j)$
    if $z \leq H_j$ **then** *// all volume of i fits*
      $v_i \leftarrow 0$
      $\mathscr{S} \leftarrow \mathscr{S} \setminus \{i\}$
      $(D_j, W_j, H_j) \leftarrow (\ell_i, W_j, z)$ *// adjust j's dimensions*
    else
      $v_i \leftarrow (v_i - \ell_i W_j H_j)$ *// update volume of i remaining unpacked*
      $(D_j, W_j, H_j) \leftarrow (\ell_i, W_j, H_j)$ *// adjust j's dimensions*
    $x_i \leftarrow x_i \cup \{j\}$ *// add j to set of holders packing i*
    $\mathscr{H} \leftarrow \mathscr{H} \setminus \{j\}$ *// remove j from open holders*
    if $D_j > \ell_i$ **then**
      $\mathscr{H} \leftarrow \mathscr{H} \cup \{\text{holder with dimensions } (D_j - \ell_i) \times W_j \times H_j\}$ *// open holder besides j*
    if $H_j > z$ **then**
      $\mathscr{H} \leftarrow \mathscr{H} \cup \{\text{holder with dimensions } \ell_i \times W_j \times (H_j - z)\}$ *// open holder on top of j*
  return x

Table 1: Comparison of simple rules for a data set of 3000 instances. Left table: n_{ij}, the number of times rule i was strictly better (*i.e.*, found a better solution) than rule j. Right table: $n_{ij} - n_{ji}$; positive values mean that rule on line i is better for more instances than the rule in column j.

	BF	LFF	LBF	WFF	WBF
BF	0	225	51	2526	2397
LFF	338	0	101	2525	2396
LBF	338	237	0	2529	2404
WFF	339	342	336	0	91
WBF	398	401	391	1737	0

	BF	LFF	LBF	WFF	WBF
BF	0	-113	-287	2187	1999
LFF	113	0	-136	2183	1995
LBF	287	136	0	2193	2013
WFF	-2187	-2183	-2193	0	-1646
WBF	-1999	-1995	-2013	1646	0

effective for the knapsack problem, are clearly outclassed by rules based on the length of the semifluid. The simple rule of selecting the longest semifluid, independently of its value, and placing it in the open holder that leads to less used space along the x axis (LBF) has generated the best results. This is the heuristic rule selected for comparison with more elaborate methods.

3.2 Local ascent

The previous packing algorithm can be easily extended to encompass local ascent, as proposed in Algorithm 2. The idea is very simple: after finding a packing with the previous heuristics, attempt another construction forbidding items packed in the current solution, one at a time. As soon as an improving solution is found, it is adopted as incumbent (*first-improve*). This process stops when all the neighbors of the current solution have been attempted, and they all lead to inferior solutions.

Algorithm 2: Local ascent for packing semifluids.

```
1  procedure ascent(D,W,H,𝒮,ℓ,v,w)
2      x ← pack(D,W,H,𝒮,ℓ,v,w)
3      let ℐ be the set of items packed in x
4      𝒯 ← {}
5      repeat
6          improved = false
7          for i ∈ ℐ \ 𝒯 do
8              𝒯 ← 𝒯 ∪ {i}
9              x′ ← pack(D,W,H,𝒮 \ {i},ℓ,v,w)
10             if value of x′ is greater than value of x then
11                 x ← x′
12                 let ℐ be the set of items packed in x
13                 improved = true
14                 break
15     until not improved
16     return x
```

This method is simple, and obviously finds a solution which is at least as good as that of Algorithm 1. As local ascent is still very fast, it is suitable for demanding situations (*e.g.*, interactive processes).

3.3 Complete search

There are two reasons why the previous methods may be unsatisfactory. The first reason concerns some rare, small instances for which a better solution can easily be found by inspection; the second reason concerns proving that the solution found is optimal. We next propose some variants for doing complete search, based on tree search.

Let us start with a caveat. In the packing process we are considering, division of the semifluid occurs only when it does not fit vertically, and the amount left is possibly packed in another holder. However, it may be optimal to fill only a part of the available amount of a semifluid. This case is illustrated in Figure 6; if item 2 is more valuable than 1, it would be optimal to fill all the volume of item 2 over a part of 1, and leave the remaining 1 unpacked, as shown in the rightmost diagram. However, visited solutions in a complete tree search are only the leftmost and the one in the center; hence, an *"optimum"* for tree search many not be truly optimal for the original problem.

Complete search is an extension of Algorithm 1 where, instead of considering only packing the item chosen by the heuristic rule in line 5, we consider all the possibilities of placing available items in open containers; each of these possibilities leads to a new node in the search tree. Notice that the branching factor is very large, and hence straightforward complete search is prohibitive even for small instances. Next, we present three relevant tree search alternatives for dealing with this difficulty; a visual insight of the differences between them is provided in Figure 7.

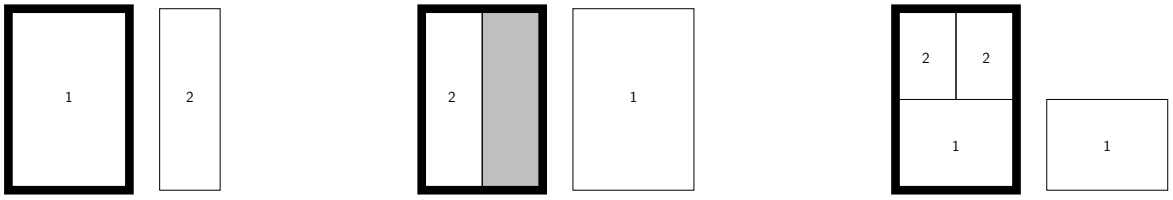

Figure 6: An instance for which complete search does not find the optimum (shown in the rightmost diagram). A vertical section of the container is represented with a bold line, and the item left over is shown beside it.

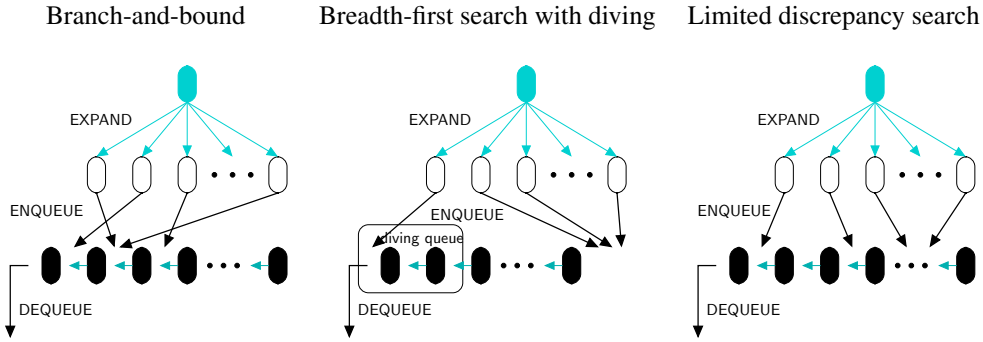

Figure 7: Queueing methods: branch-and-bound (left), where nodes in the queue are sorted by their upper bound; breadth-first search with diving (center), where no information about about the nodes entering the queue is used (at each expansion, one node generated is the diving node); and limited discrepancy search (right), where nodes are sorted by discrepancy (at each expansion, nodes are generated in this order).

3.3.1 Branch-and-bound

Branch-and-bound (BB) is the standard method for searching a tree in optimization (see, *e.g.*, [8] for an early survey). For a maximization problem, the comparison of an upper bound of the objective that can be reached from a given node, to a known lower bound of the objective, is used to eliminate from consideration parts of the search tree. The best solution visited so far is commonly used as the lower bound. In the case of the basic semifluid packing problem, an upper bound can be obtained by sorting the items by decreasing unit value, and filling the space still available in the container by this order, assuming no shape constraints (this is similar to the linear relaxation bound for the knapsack problem; see [12]). For a given partial solution, holders that cannot be filled due to having no unpacked items that fit inside them are withdrawn from the list of open holders; their volume is subtracted from the space available when computing the corresponding upper bound.

Another important factor for having a reasonably effective branch-and-bound concerns avoiding symmetric, or otherwise equivalent solutions. This is done with the following rules:

- items placed at the same horizontal level must have increasing indices in the set $\mathscr{S}$ of semifluids to pack;
- items placed on top of given item i having the same length as i cannot have a larger unit value than i.

The main steps of the branch-and-bound algorithm are outlined in Algorithm 3 (see also Appendix A). The algorithm is based on the iteration over elements in a queue (Q) until it becomes empty. Nodes whose upper bound is inferior to the objective value of the best known solution are discarded (line 4). Branching is carried out in lines 7–10. As all the possible assignments of yet unpacked items to open holders must be considered, the main limitation of the algorithm concerns the large number of nodes added in these lines.

The algorithm has two parameters, limiting CPU time and the size of the queue. The latter is used when restricting the number of open nodes is required for keeping memory usage acceptable; in such cases, we provide the possibility of removing a part of the queue (*chopping*, lines 11–13). When this occurs, as well as when the time limit is reached, the solution returned may be not optimal. In the experiment reported in

the Section 4, the maximum number of nodes is set to infinity, making CPU time the only factor limiting the search.

Algorithm 3: Main steps of the branch-and-bound algorithm.

```
create a queue Q with one node (the root relaxation)                    // Initialization
set upper bound UB ← ∞, lower bound LB ← −∞, optimality flag OPT ← true
repeat
    select and remove from Q node k with largest UB                     // Subproblem selection
    if UB^k ≤ LB or no items fit in open holders then                   // Pruning and fathoming
        continue
    foreach feasible assignment of unpacked items to open holders do    // Partitioning
        add new node n to Q
        if LB^n > LB then
            update LB ← LB^n
    while size of Q is larger than the allowed limit do                 // Chopping
        remove from Q node with smallest UB
        OPT ← false
    if time limit has been reached then                                 // Termination
        OPT ← false
until Q = {} or time limit has been reached
return solution that yielded LB, with optimality flag OPT
```

3.3.2 Breadth-first search with diving

As the branching factor is very large, standard branch-and-bound may not be allowed the time and space to produce a good solution, even for relatively small instances. Indeed, as will be seen in the next session, in a limited time the solution of branch-and-bound is often worse than that of the simple heuristics. For overcoming this issue, several alternatives have been proposed in the literature; these are usually based on *diving* (see, *e.g.*, [1, 13]). We firstly propose what we call *breadth-first with diving (BFD)*, which consists of the following:

1. Keep two search queues: the main queue Q and the *diving queue* R;
2. If R is not empty, at the current iteration explore the last element added to this queue (*i.e.*, explore R in a last in, first out manner);
3. If R is empty, at the current iteration explore the first element added to Q (*i.e.*, explore Q in a first in, first out manner);
4. When creating children of the current node, append the one that corresponds to the heuristic rule (LBF) to the queue R, and the remaining children (generated by decreasing item length) to Q.

Hence, R is searched in a last in, first out fashion, corresponding to the order of the LBF heuristic rule (longest item first, best fit container); therefore, the first leaf visited is the LBF solution. The exploration of Q in a breadth-first (first in, first out) fashion introduces diversity in the search, which balances well with the intensive search of the dive; this is important for time-limited executions, where parts of the tree are left unexplored. Furthermore, quickly finding solutions of good quality allows pruning more nodes in the search tree. Notice that as long as the item list is initially sorted by length, we can generate new nodes to add to Q without further sorting (however, sorting available items by value is required for computing the upper bound of a new node).

Diving does not interact well with the symmetry breaking rules: if the diving item was forbidden for avoiding symmetry, the first dive would be interrupted, and the corresponding heuristic solution would not be reached. In order to assure that we reach that solution, rules for avoiding symmetry are not enforced during diving.

In our implementation of BFS we are using the bounds described in Section 3.3.1, which in most cases allow pruning significant parts of the search tree.

3.3.3 Limited discrepancy search

Another alternative to standard branch-and-bound is *limited discrepancy search (LDS)*, where the tree is searched by increasing order of the *number of violations* of the heuristic rule, as proposed in [7]. This method has been attempted with the LBF heuristic rule, but the computation of discrepancy in this case requires sorting the moves available, using considerable computational time. A better alternative is to base the search in the longest item first, first fit rule (LFF); this allows a very quick expansion of nodes at each iteration, and exploring much larger parts of the tree in a limited time.

As in the standard version of LDS, this method uses a parameter specifying the discrepancy level above which search is abandoned. This usually allows adjusting the part of the tree that is explored to the resources available, as an alternative to simply interrupting the execution after a certain time has elapsed. We acknowledge that better solutions are often found with such an adjustment, and that memory usage will make the search impractical for long-running executions without limiting discrepancy; however, for an easier comparison with the other methods, we have set the discrepancy limit to infinity. Due to this choice, whenever LDS ends before reaching the limit CPU time, its solution is optimal.

In our implementation of LDS we are using the bounds described in Section 3.3.1, which in most cases allow pruning significant parts of the search tree.

4 Computational results

In order to assess the performance of the methods proposed, we have created a set of instances based on the characteristics of the real-world application. Practical instances we are aware of are small, as is the number of different semifluid lengths (tubes are usually cut in standard lengths). Instances with more than 20 items go beyond the application's requirement, but are useful for testing the behavior of the different algorithms. Instances are classified into two main families:

- Easy instances: generated in such a way that in the optimum there are no items left unpacked; for these instances, an optimal solution completely occupying the container is known.
- Hard instances: no optimum is known in advance; the volume of available items corresponds either to 100% of the container (as for easy instances, though now it is unlikely that all items can be packed), or to 150% of it.

The number of semifluids considered are 5, 10, 20, 50, and 100. Some instances have just a few distinct item lengths, other have more diverse lengths. For each combination of these characteristics, 100 different instances have been generated, totaling 3000 instances. A visualization of instances from the easy and hard subsets, with corresponding optimal and heuristic solutions, is provided in Figure 8 (details on the instance generator are available in Appendix A).

Our programs use exact arithmetic for all operations (hence, values in the instance files are written as fractions). All the executions were limited to 60 seconds of CPU time, and both the maximum number of nodes and the discrepancy limit were set to infinity.

We start recalling the comparison among simple heuristics (Table 1). Having selected LBF, we now compare it to more elaborate methods in Table 2. As expected, local ascent is always at least as good as LBF, being strictly superior for a massive share of instances. As the CPU time limitation is rather severe, local ascent is also often better than tree search methods. The best results overall have been obtained by limited discrepancy search.

Figure 9 graphically summarizes the results obtained. On each sub-figure, results for instances of type easy and hard are separated into two rows. Each bar (or curve, in the bottom sub-figure) represents percentages or averages considering all the instances of each size. Methods considered are, as before, the simple heuristic rule (LBF), local ascent based on this rule, branch-and-bound, breadth-first search with diving, and limited discrepancy search; to each method corresponds a column in the top three sub-figures, and a line in the bottom sub-figure. The abscissa for the three top sub-figures is the instance size, and for the bottom sub-figure is the

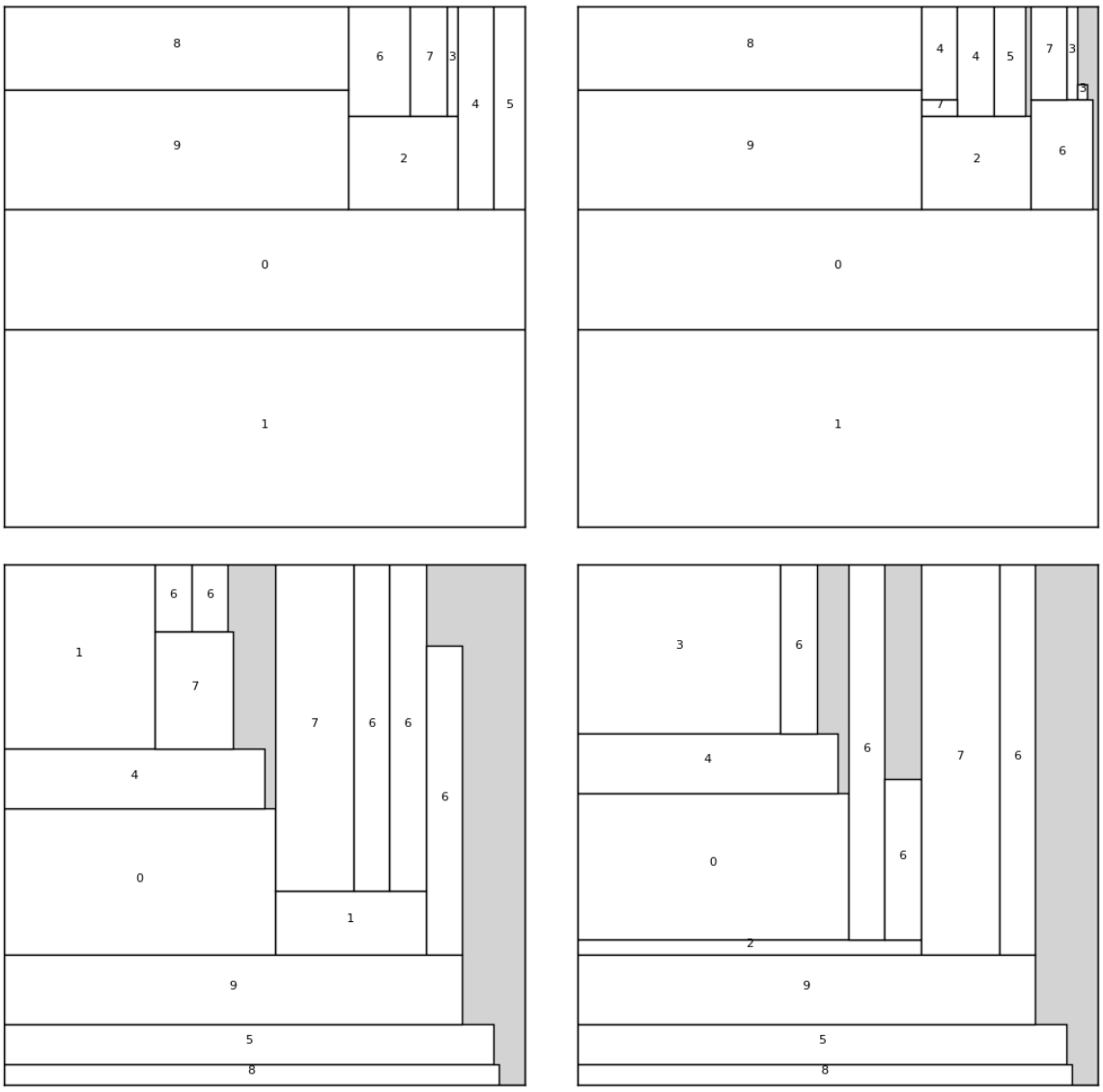

Figure 8: An optimal solution (left), and a heuristic solution (right) for instances with ten items: an easy instance (top) and a hard instance (bottom).

Table 2: Comparison of simple rule (LBF), local ascent (LA), and tree search — standard branch-and-bound version (BB), breadth-first search with diving (BFD), and limited discrepancy search (LDS) — for a data set of 3000 instances. Left table: n_{ij}, the number of times method i was strictly better (*i.e.*, found a better solution) than method j. Right table: $n_{ij} - n_{ji}$; positive values mean the method on line i is better for more instances than the method in column j.

	LBF	LA	BB	BFS	LDS
LBF	0	0	1627	306	42
LA	2041	0	1744	849	187
BB	909	633	0	108	101
BFS	1520	1092	1895	0	329
LDS	2007	1350	1915	1192	0

	LBF	LA	BB	BFS	LDS
LBF	0	-2041	718	-1214	-1965
LA	2041	0	1111	-243	-1163
BB	-718	-1111	0	-1787	-1814
BFS	1214	243	1787	0	-863
LDS	1965	1163	1814	863	0

Table 3: Average number of nodes explored, remaining in the queue at the end of the search, and created, for each of the tree search methods.

Instance		Nodes explored			Nodes in queue			Nodes created		
Type	Size	BB	BFS	LDS	BB	BFS	LDS	BB	BFS	LDS
easy	5	10.	5.	26.	0.	0.	0.	28.	17.	17.
easy	10	3193.	12.	103.	4379.	0.	0.	13790.	82.	78.
easy	20	8113.	166.	1896.	17431.	999.	3699.	37511.	3349.	5502.
easy	50	1974.	139.	6053.	17811.	6168.	109864.	23526.	8231.	115697.
easy	100	574.	54.	4488.	13400.	7131.	208455.	14545.	7812.	212709.
hard	5	1895.	4691.	9092.	1279.	2831.	4108.	4563.	9965.	12813.
hard	10	5586.	7454.	28896.	7979.	12363.	23271.	20654.	32701.	51551.
hard	20	4951.	3430.	34054.	25272.	22851.	73126.	40220.	40261.	106639.
hard	50	890.	447.	10488.	26710.	16993.	268792.	27603.	18229.	278513.
hard	100	290.	84.	4650.	18535.	7565.	256590.	18825.	7676.	260851.

CPU time used. For each instance we have identified the best solution found by all the methods (which in some cases is optimal); on the top sub-figure, the ordinate is the percentage of instances for which each method finds such solution. The next set of plots shows the percentage of instances for which the search tree was completely explored (for the relevant methods). Follows a plot of the average CPU time used in the solution process, for all the instances of each size/type; the time for each run was limited to 60 seconds, but in many cases was smaller. Finally, the sub-figure in the bottom shows the evolution of the average value of the objective for the best solution found by each method, in terms of the CPU time used; here we can observe how the gap between the different methods progresses.

As can be seen in Figure 9, "standard" branch-and-bound (taking the node with the highest upper bound at each iteration) quickly becomes very limited, when the instance size increases; this is due to the very high branching factor. Crossing information on that figure with that of Table 3, we see that when instance size increases, a very large number of nodes are open, but, due to the time limitation, only a small part of them can be explored. This can be observed for all except smallest, easy instances. For finding good solutions in a limited time, methods fully exploiting the heuristics (LA, BFS and LDS) have a much better performance. Note that, for larger values of the CPU limit, the number of open nodes may have to be limited for avoiding memory overflow.

We have seen in Table 2 that the method that is able to find strictly better solutions than the others for more instances is limited discrepancy search. This is corroborated by the evolution over time of the average solution, for all instances of a given size, presented at the bottom of Figure 9. The general tendency is to have LDS finding good solutions more quickly than the other tree search methods; however, near the CPU limit imposed, LDS is closely followed by BFD (*e.g.*, for hard instances of size 50). In terms of the ability to complete the search, and hence to prove optimality, BFD and LDS are roughly equivalent; these methods appear to be considerably better than BB for easy instances, though slightly inferior for hard instances.

The main factor for LDS to be able to explore much more nodes than BFD is the ability to easily keep nodes organized by increasing discrepancy; for technical details, please consult the implementation code (see Appendix A).

Another interesting observation concerns the performance of local ascent. For small instances, LA quickly finds the best solution (often proven optimal by tree search); however, LA is outclassed by tree search methods for mid-sized instances, to regain a relative good performance for large instances, as can be seen in the top graphic of Figure 9. This is because local ascent is very fast, and hence the time constraint is not limiting it in our experiment, even for large instances.

5 Conclusions

Semifluids are materials having both fluid and solid characteristics. In this paper, we studied the problem of packing a particular type of semifluid which cannot flow in one direction, though it is fluid in the other directions; this is the case when tubes of a small diameter are packed in parallel. In this context, a packing item — a semifluid — is a set of identical tubes. Different items have different length and/or value, and any

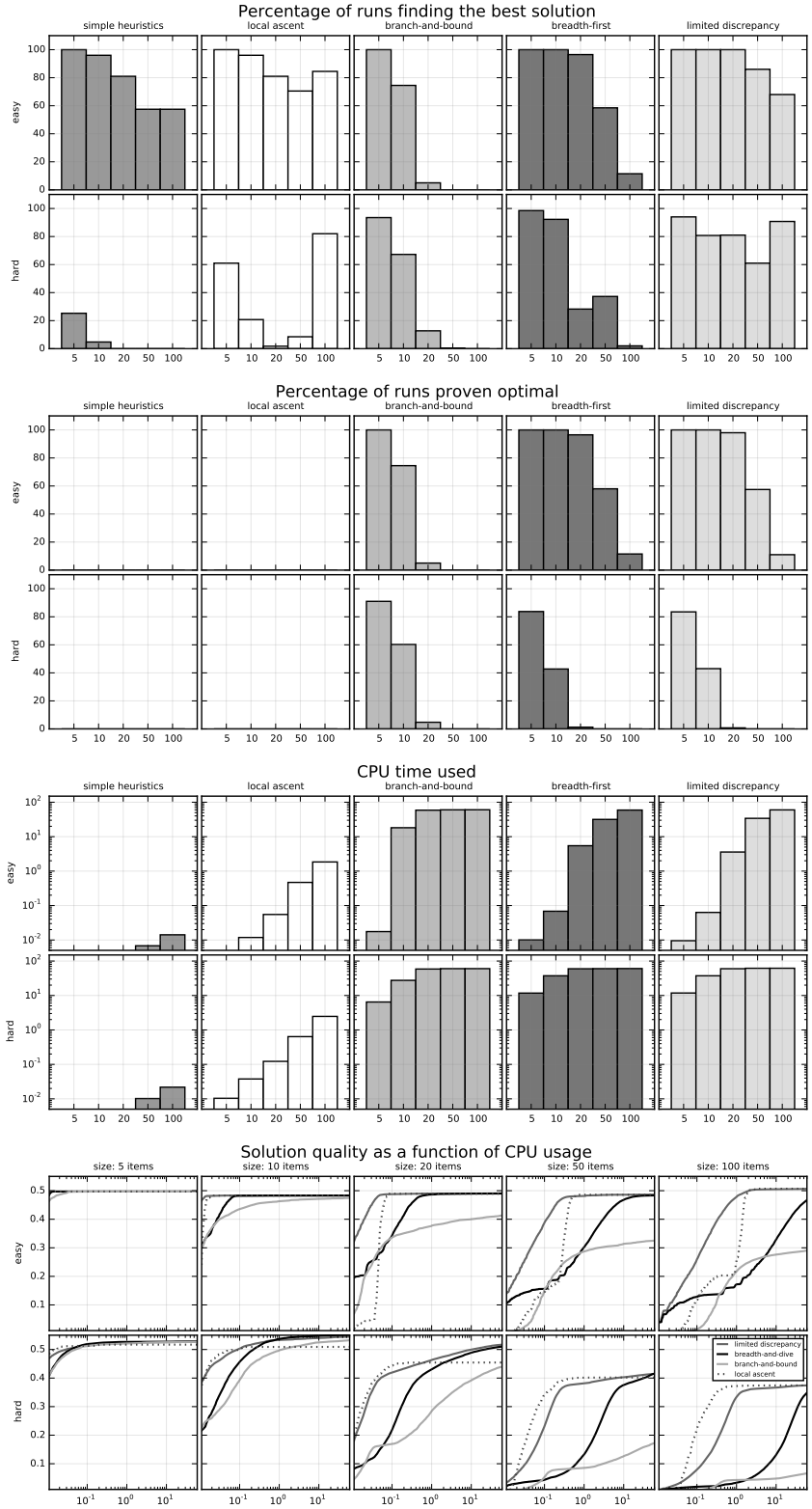

Figure 9: Overall aperçu of the methods' performace: percentage of best solutions found (top), percentage of optimal executions (upper-center) and CPU time (lower-center) used, in terms of type (easy/hard) and number of items of the instance (measures are percentages/averages considering all instances of each size/type). Bottom: evolution of the average of the objective value for all the instances of given size/type in terms of CPU time.

fraction of an item may be used, with the objective of obtaining the maximum value packed.

Given the assumption of continuity, *i.e.*, that one may arbitrarily divide a given volume, the problem of packing a set of tubes of different lengths in a container is surprisingly difficult. This paper presents heuristics and complete search for the variant which closer corresponds to the industrial application: all the semifluids must be packed in the same direction, and a semifluid placed on top of another must not protrude. In this paper, we have considered divisions of the volume of an item only when it reaches the ceiling of the container.

Several methods, from simple heuristics to complete search, are proposed. The choice among them depends on the application. Simple heuristics are very quick, but often fail to find good solutions. Local ascent based on simple heuristics often finds very good solutions, and is likely to be the best method for large instances and limited CPU time. Among tree search methods, limited discrepancy search is often superior to the others, finding solutions of very good quality and frequently proving them optimal.

Semifluid packing under assumptions not considered in this paper is an interesting subject for future research; in particular, exploring different packing directions and the possibility of overflow. The complexity class of this problem is unknown; determining it is an interesting research topic. Another interesting research direction concerns developing compact mathematical models for optimization, taking full consideration of the possibility of packing fractions of each item. The proposed heuristics could be extended and refined, in particular for taking into account the possibility of diversifying the point of division of an item into several packing places, further than the top of the container. Yet another unexplored possibility for improvement concerns using the objective value of a heuristic solution as a lower bound, at the root node.

Acknowledgements

This work is partly funded by FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2013. We would like to thank Benjamin Müller, from Zuse Institute Berlin, for important suggestions. We would also like to thank three anonymous reviewers for their constructive comments on a previous version of this paper.

A Supplementary programs and data

Supplementary programs and data associated with this article can be found online at `http://www.dcc.fc.up.pt/~jpp/code/semifluid`. That page contains an implementation of all the algorithms described in this paper and the program used for generating the instances, as well as the generated data.

In the real-world application of this problem the number of different tube lengths in catalog is small. To a smaller extent, this is also true for other numeric values in the required data. We simulate this by limiting the number of digits in the random numbers generated: 3 digits in general, 2 or 3 digits for tube lengths. All the values are normalized, so that the container dimensions are $1 \times 1 \times 1$. As we use exact arithmetic for all operations, values generated and stored in the files are fractions. The combinations of parameters used for instance generation are summarized in Table 4.

Table 4: Characteristics of benchmark instances used: for each set of parameters 100 independent random instances have been generated, totaling 3000 instances.

Type	Number of items	Digits in ℓ_i	Volume of items (% of $D \times W \times H$)	Total
easy	5, 10, 20, 50, 100	2, 3	100%	1000 instances
hard	5, 10, 20, 50, 100	2, 3	100%, 150%	2000 instances

For hard instances no optimum is known in advance. These instances have been generated by simply drawing random numbers for lengths and volumes with the required number of digits, and afterwards updating the volumes so that the total volume will be the desired factor of the container's volume (in our data set, 1 or 3/2).

Easy instances have the space of the container completely filled. This is done by successive divisions of the container, as shown in Algorithm 4. To each holder generated this way there will correspond a different item. Using this procedure, the total volume of items will always equal the volume of the container.

Instances closer to the real-world application that motivated this paper are hard instances with 10 to 20 items, two digits in ℓ_i and items occupying 100% to 150% of the container's volume.

Algorithm 4: Main steps for generating an easy instance.

```
create a holder h with the size of the container
𝒦 ← [h]
repeat
    randomly select a holder h from 𝒦
    randomly select r with uniform distribution in [0, 1]
    if r < 1/2 then                                   // With 50% probability
        if h has no other holders on top then
            divide h vertically
            replace h by the two newly created holders
    else
        divide h horizontally
        replace h by the two newly created holders
until number of holders is equal to the number of desired items
```

References

[1] Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: A new approach to integrate CP and MIP. In: Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'08, pp. 6–20. Springer-Verlag, Berlin, Heidelberg (2008)

[2] Baldi, M.M., Crainic, T.G., Perboli, G., Tadei, R.: The generalized bin packing problem. Transportation Research Part E: Logistics and Transportation Review **48**(6), 1205 – 1220 (2012)

[3] Coffman, Jr., E.G., Garey, M.R., Johnson, D.S., Tarjan, R.E.: Performance bounds for level-oriented two-dimensional packing algorithms. SIAM Journal on Computing **9**(4), 808–826 (1980)

[4] Crainic, T., Perboli, G., Tadei, R.: Recent advances in multi-dimensional packing problems. In: C. Volosencu (ed.) New technologies — trends, innovations and research. InTech (2012)

[5] Dolatabadi, M., Lodi, A., Monaci, M.: Exact algorithms for the two-dimensional guillotine knapsack. Computers & Operations Research **39**(1), 48 – 53 (2012). Special Issue on Knapsack Problems and Applications

[6] Fekete, S.P., Schepers, J., van der Veen, J.C.: An Exact Algorithm for Higher-Dimensional Orthogonal Packing. Operations Research **55**(3), 569–587 (2007)

[7] Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes, pp. 607–615. Morgan Kaufmann (1995)

[8] Lawler, E.L., Wood, D.E.: Branch-and-bound methods: a survey. Operations Research **14**, 699–719 (1966)

[9] Lim, A., Rodrigues, B., Wang, Y.: A multi-faced buildup algorithm for three-dimensional packing problems. Omega **31**(6), 471 – 481 (2003)

[10] Lodi, A., Martello, S., Vigo, D.: Heuristic algorithms for the three-dimensional bin packing problem. European Journal of Operational Research **141**(2), 410 – 420 (2002)

[11] Lodi, A., Martello, S., Vigo, D.: Models and bounds for two-dimensional level packing problems. Journal of Combinatorial Optimization **8**(3), 363–379 (2004)

[12] Martello, S., Toth, P.: Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Chichester (1990)

[13] Pochet, Y., Wolsey, L.A.: Production Planning by Mixed Integer Programming. Springer (2006)

[14] Puchinger, J., Raidl, G.R.: Models and algorithms for three-stage two-dimensional bin packing. European Journal of Operational Research **183**(3), 1304–1327 (2007)

[15] Silva, E., Alvelos, F., de Carvalho, J.V.: An integer programming model for two- and three-stage two-dimensional cutting stock problems. European Journal of Operational Research **205**(3), 699–708 (2010)