A Tree Search Algorithm for Solving the Container Loading Problem

Tobias Fanslau, Andreas Bortfeldt

Diskussionsbeitrag Nr. 426
Mai 2008

Diskussionsbeiträge der Fakultät für Wirtschaftswissenschaft
der FernUniversität in Hagen

Herausgegeben von der Dekanin der Fakultät

A Tree Sarch Algorithm for Solving the Container Loading Problem

Tobias Fanslau, Andreas Bortfeldt

Abstract:

The paper presents a tree search algorithm for the three-dimensional container loading problem (3D-CLP). The 3D-CLP is the problem of loading a subset of a given set of rectangular boxes into a rectangular container so that the packed volume is maximized. The method has two variants: the packing variant guarantees full support from below for all packed boxes, while this constraint is not taken into account by the cutting variant. The guillotine cut constraint is considered by both variants. The method is mainly based on two concepts. On the one hand the block building approach is generalized. Not only blocks of identical boxes in the same spatial orientation are applied but also blocks of different boxes with small inner gaps. On the other hand the tree search is carried out in a special fashion called a partition-controlled tree search (PCTRS). This makes the search both efficient and diverse, enabling a sufficient search width as well as a suitable degree of foresight. The approach achieves excellent results for the well-known 3D-CLP instances suggested by Bischoff and Ratcliff with reasonable computing time.

Keywords:

Container loading; 3D packing; Single Large Object Placement Problem (SLOPP); Single Knapsack Problem (SKP); heuristic; tree search.

Fakultät für Wirtschaftswissenschaft, FernUniversität in Hagen
Profilstr. 8, D-58084 Hagen, BRD

Tel.: 02331/987–4433
Fax: 02331/987–4447
E-Mail: andreas.bortfeldt@fernuni-hagen.de

A Tree Search Algorithm for Solving the Container Loading Problem

Tobias Fanslau, Andreas Bortfeldt

1. Introduction

Cutting and packing problems (C&P; cf. Dyckhoff and Finke 1992) represent problems of the optimal use of resources. While cutting problems are concerned with the best possible use of materials such as steel, glass or wood, packing problems involve the best possible capacity utilisation of packaging space. The effective utilization of material and transport capacities is of great economic importance in production and distribution processes. It also contributes to using natural resources economically, limiting the growth in traffic and as a whole to treating the environment with greater care. The development of even more effective cutting and packing methods remains an important task because, in view of the size of today's productions and distribution processes, even relatively minor growth in the utilization of material and space capacities can result in considerable material savings and reductions in costs.

The subject of the present paper is the three-dimensional container loading problem (3D-CLP), which is formulated as follows: a large cuboid, the container, and a set of smaller cuboids, the boxes, are given. In general, the total volume of the boxes exceeds the container volume. A permissible arrangement of subset of the given boxes is to be determined in such a way that the packed box volume is maximized and, where applicable, additional constraints are complied with. A box arrangement is deemed to be permissible if

- each box lies completely in the container, i.e. does not penetrate its boundary surfaces,
- no two boxes overlap and
- each box is parallel to the container's boundary surfaces.

A box type is defined in principle through the three side dimensions of a box. If there is only one box type in a box set, it is described as homogeneous. If there are only a few box types with a relatively large number of specimens per type, this is a weakly heterogeneous box set; on the other hand, if there are many box types with only a few exemplars per type, the box set is strongly heterogeneous. This paper presents a tree search method for the 3D-CLP which is just as suitable for weakly heterogeneous box sets as for strongly heterogeneous. Both variants are explicitly differentiated in the new typology of the C&P problems (cf. Wäscher et al. 2007). Using this typology the method presented here can be applied to both the Single Large Object Placement Problem (SLOPP, weakly heterogeneous) and to the Single Knapsack Problem (SKP, strongly heterogeneous).

Cutting and packing problems are known to be dual: a ("pure") cutting problem can also be formulated as a packing problem, and vice versa (cf. Dyckhoff and Finke 1992). However, the duality of C&P problems should not make us lose sight of the fact that the two application situations, cutting or packing items, are characterized by fundamentally different constraints. Where items of desired dimensions are cut, on the basis of the most used cutting technology the guillotine cutting constraint is of primary importance:

(C1) Guillotine cutting constraint

Only (guillotine) cuts are permitted where the cutting area lies parallel to a boundary surface of the container and the cut piece is completely separated in two smaller parts.

In contrast, when items are packed, stability requirements above all play a part (cf. e.g. Bischoff et al. 1995). The following support constraint is of fundamental importance here:

(C2) Support constraint
The area of each box of a packing plan that is not placed on the floor of the container must be supported completely (i.e. 100%) by other boxes.

Finally, the following orientation constraint is included in the problem:

(C3) Orientation constraint
For certain box types, up to five of the maximal six possible orientations are prohibited.

While constraint (C1) is only important in a cutting context, and constraint (C2) only in a packing context, it is known that constraint (C3) is found in both application situations. In this paper a tree search method with two variants is introduced: while the packing version observes the support constraint (C2), this is not the case with the cutting version; constraints (C1) and (C3) are fulfilled by both variants. A comparison of the results of both variants reveals the considerable influence of the required full box support on the full use of space as the primary optimization objective. In accordance with the duality of C&P problems, this paper will use the terminology of the packing problem throughout, although both the packing problem and the cutting problem are dealt with.

The rest of the paper is divided as follows: Section 2 provides an overview of the literature and Section 3 presents the tree search method. Section 4 is dedicated to the numerical test of the method and Section 5 summarizes the paper.

2. Literature Overview

The 3D-CLP is NP-hard in the strict sense and is also regarded in practice as extremely difficult (cf. Pisinger 2002). Up to now, only a few exact methods were suggested. Fekete and Schepers (1997) develop a general framework for the exact solution of multi-dimensional packing problems. Martello et al. (2000) present an exact branch-and-bound method (B&B) for the 3D-CLP. This is part of a method for the three-dimensional bin packing problem with which instances with up to 90 boxes could be solved.

Heuristic methods without an optimality guarantee for the 3D-CLP can be divided into three groups with regard to the method type:

(1) Conventional heuristics
These include construction methods (e.g. greedy algorithms) and improvement methods (e.g. local search algorithms). Conventional heuristics for solving the 3D-CLP were suggested, e.g., by Bischoff et al. (1995), Bischoff and Ratcliff (1995) and Lim et al. (2003).

(2) Metaheuristics
In the last ten years metaheuristic search strategies constituted the preferred method types for the 3D-CLP. Genetic algorithms (GA) were suggested among others by Hemminki (1994), Gehring and Bortfeldt (1997, 2002) and Bortfeldt and Gehring (2001). Tabu search algorithms (TS) came from Sixt (1996) and Bortfeldt et al. (2003). Simulated annealing methods (SA) were developed by Sixt (1996) and by Mack et al. (2004). A method of local search based on the Nelder and Mead approach dates from Bischoff (2004). Moura and Oliveira (2005) as well as Parreño et al. (2007) introduce a GRASP method (Greedy Randomized Adaptive Search Procedure).

(3) Tree search methods
Incomplete tree search or graph search methods (in brief TRS) were applied successfully to the 3D-CLP. Mention should be made of the method of the And/Or graph search by Morabito and Arenales (1994), the tree search methods from Eley (2002) and Hifi (2002)

and the B&B method from Terno et al. (2000) (the integrated CLP method is meant) and Pisinger (2002).

The existing 3D-CLP methods are based on different heuristic packing approaches that determine the structure of generated packing plans (cf. Pisinger 2002):

(1) Wall building approach
The container is filled by vertical cuboid layers ("walls") that mostly follow along the longest side of the container. The approach is realized among others in Bortfeldt and Gehring (2001) and in Pisinger (2002).

(2) Stack building approach
The boxes are packed in a suitable manner in stacks, which are themselves arranged on the floor of the container in a way that saves the most space. Characteristic of this approach is that the stacks do not themselves form walls as defined before. Advocates of this approach are, among others, the heuristic from Bischoff and Ratcliff (1995) and the GA from Gehring and Bortfeldt (1997).

(3) Horizontal layer building approach
The container is filled from bottom to top through horizontal layers that are each intended to cover the largest possible part of the (flat) load surface underneath. This approach is realized in the methods from Bischoff et al. (1995) and Terno et al. (2000).

(4) Block building approach
The container is filled with cuboid blocks that mostly contain only boxes of a single type with the same spatial orientation. The approach is related to approach (3), but the main motive here is to fill the largest possible sub-spaces of the container without internal losses; cf. as well the characterisation of this approach in Eley (2002). Representatives of the approach are the TS method from Bortfeldt et al. (2003), the tree search method from Eley (2002) and the SA/TS hybrid respectively from Mack et al. (2004).

(5) Guillotine cutting approach
This approach is based on a slicing tree representation of a packing plan. Each slicing tree corresponds to a successive segmentation of the container into smaller pieces by means of guillotine cuts, whereby the leaves correspond to the boxes to be packed. Morabito and Arenales's graph search method (1994) is based on this approach.

In general it can be said that research on the 3D-CLP has up to now concentrated very one-sidedly on the packing application context. The great majority of the methods mentioned observe the support constraint (C2) and the orientation constraint (C3) as well, and are suitable for supporting 3D packing processes. A considerable part of the listed methods also include further constraints from the packing context in the problem, e.g. a weight constraint for the freight; cf. among others Terno et al. (2000), Bortfeldt and Gehring (2001), Eley (2002), Mack et al. (2004). The heuristic developed by Bischoff et al. (1995) for loading pallets and the GRASP method from Moura and Oliveira (2005) prove to be particularly sensitive methods for the stability of the load. Bischoff (2004) considers a complicated overstacking constraint, which limits the pressure applied to the surface covering of boxes.

Only a few contributions refer explicitly up to now to the cutting application context of the 3D-CLP (cf. Hifi 2002). Other contributions present methods that also comply with the guillotine cut constraint (C1) and can therefore be used for cutting tasks; cf. among others Morabito and Arenales (1994), Bortfeldt and Gehring (2001) and Pisinger (2002). Mack et al. (2004) examine to an extent the influence of the support constraint (C2) on the achievable volume utilisation.

3. The Tree Search Algorithm

The proposed tree search method is referred to as CLTRS (Container Loading by Tree Search). It is based in the main on two concepts:

- The block building approach is generalized. Along with the usual compact blocks from identical boxes with the same spatial orientation, blocks with small inner gaps are now used as well to construct packing plans.
- A special form of the tree search is used that tries to establish a suitable balance between the search width and the foresight during the search.

Differences between the cutting and the packing variants of CLTRS will be discussed at the appropriate points. In the same way, references will be made to method components that are specially tailored to a weakly heterogeneous or strongly heterogeneous stock of boxes.

3.1. The method's data structures

The data structures used by CLTRS are introduced informally below and terminological agreements made.

a) Container and box data

The cuboid container is given by its three internal dimensions and let it be embedded in the first octant of a 3D coordinates system (3D-CS). This is shown in Figure 1, which also illustrates terms such as "left", "in front of", etc. The given box set is represented by a box type vector. This contains for each type the three side dimensions that define the type and the number of appropriate boxes. The indexing of the box types starts from the sorting of the vector in descending order in accordance with the volume of the boxes. The container dimensions and the box type vector are global data. During the search, the set of the still free (i.e. still not packed) boxes is mainly given by a vector *Bres*, which contains the number of free boxes per type index. A cuboid (box, empty part space in the container, envelope cuboid of a box arrangement) is referred to as oriented if it may no longer be turned with regard to the 3D-CS (but possibly may still be displaced). Let it be agreed that an oriented cuboid always lies "somewhere" in the first octant of the 3D-CS and parallel to its axes. The reference corner of a oriented cuboid is then understood as the corner nearest to the origin of the 3D-CS. The dimensions of an oriented cuboid in the three coordinate directions are generally designated with *mx*, *my* and *mz* respectively.

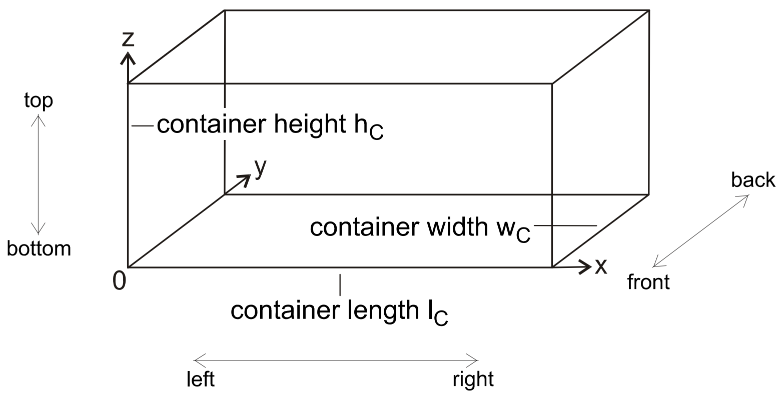

Figure 1: Container in a 3D coordinates system

b) Residual space and residual space stack

A residual space is an oriented cuboid space in the container. It is described through its three side dimensions *mx*, *my*, *mz* and its reference corner *rc* with the coordinates *x*, *y*, *z*. The container is itself a residual space with the coordinates' origin as the reference corner. During the search, residual spaces are constructed continuously and later processed, i.e. filled with boxes or detected as not fillable. Residual spaces that are already constructed but have not yet been processed are managed in a residual space stack *Rs_stack*.

c) Block and block list

A block is an arrangement made up of one or more oriented boxes (cf. Section 3.3). During the search, a block is represented above all through the dimensions of the oriented envelope cuboid of the arrangement and by the packed box set. However, the block data also clearly stipulate the spatial orientation and position of all boxes relative to the reference corner of the envelope cuboid. The block list *Bll* contains all blocks that can be used for the search, or more precisely, for a stage of the search.

d) Placed block

The container is filled through the successive placing of blocks. To place a block *bl* means that it is arranged in the reference corner *rc* of a residual space *rs*. A placed block *pbl* is therefore given by a pair (*bl*, *rs*.*rc*). The block data and the reference corner stipulate clearly the position of all boxes of the block in the container. Figure 2 shows the placing of a block, represented by its envelope cuboid, in a residual space.

e) Packing plan

A packing plan is an incomplete or complete solution of the given problem instance. A solution is complete when all boxes are packed or the residual space stack is empty. A packing plan has as components a vector *Pbl*(*i*), *i* = 1,...,|*Pbl*|, of placed blocks and the total stowed box volume *v*. By means of the appropriate block list *Bll* the orientation and position of the placed boxes of all blocks can be subsequently clearly determined. The better of two complete packing plans is the one with the greater *v* value.

f) State

A variable of the type 'state' summarises the state of the search after some blocks have been placed. A state variable *s* contains the following components:

- the (usually incomplete) packing plan *p*,
- the volume value *vc* (cf. Section 3.4),
- the vector of free boxes *Bres* and
- the residual space stack *Rs_stack*.

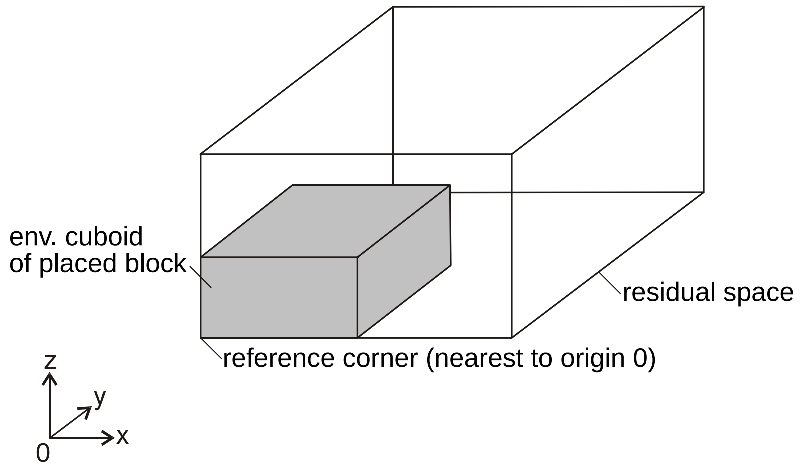

Figure 2: A residual space with a placed block

The component notation familiar from the programming language C is used for structures, e.g. *s.vc* for the *vc* value of the state variable *s* and *s.p.Pbl[i]* for the *i*-th placed block of the solution *p* of the state variable *s*. Vectors are in some cases noted only as sets, for the sake of simplicity. Along with the container and box dimensions, the method parameters (which will be explained later) and other appropriately designed variables are available globally.

3.2. Overall algorithm

The overall algorithm of the method is shown in Figure 3. The total search is divided into two large sections, known as stages. A specific block list is generated at the start for each stage. In the first stage, only blocks from identical boxes with the same spatial orientation are provided. In the second stage, blocks are generated whose box arrangements usually do not completely fill up their envelope cuboid. While the block list for stage 1 is customized for weakly heterogeneous box sets, the block list in stage 2 proves to be particularly suitable for strongly heterogeneous box sets. However, this allocation does not apply without restriction: for example, in many cases the best utilization for weakly heterogeneous instances as well is achieved with the block list for the second stage. For this reason, a two-phase search approach that diversifies the search is selected.

```
algorithm CLTRS
input (container dimensions, boxes data, parameters);
// initialize (global) best packing plan
pbest.v := 0;
// carry out search in two stages
for i := 1 to 2 do
      generate special block list Bll for current stage;
      search_effort := 1; // internal parameter of the search efforts
      // generate successive packing plans with increasing search effort
      repeat
          init_state(s); // initialize state variable s
          // process successive residual spaces
          while (s.Rs_stack ≠ ø) do
                find_best_block(s, found, bl);
                update_state(s, found, bl, rs);
          endwhile;
          // update search effort for next iteration
          search_effort := search_effort * 2;
      until (time_limit(i) exceeded);
endfor;
// output best packing plan over both stages
output(pbest);
end.
```

Figure 3: Overall algorithm of method CLTRS

Within each stage one complete packing plan after the other is generated until the given time limit specific for each stage is exceeded. Each packing plan consists exclusively of blocks from the block list specific to the stage. Finally, the best packing plan generated over both stages is output.

The generation of a packing plan in a stage is referred to as an iteration. At the start of an iteration the state variable *s* is initialized. The appropriate residual space stack then contains the container as the sole residual space, the current solution is empty and all boxes are still free. Following this, the uppermost residual space of the stacks is processed until it is empty and the next solution is thus complete. To fill a residual space *rs* a suitable block *bl* is specified where possible. If this is successful, following this the current solution *s.p*, the set of free boxes *s.Bres* and the residual space stack *s.Rs_stack* are updated. In doing this, residual space *rs* is replaced by three daughter residual spaces of *rs* (cf. Section 3.5, c)). If no block fits into the residual space *rs*, only *rs* is removed from *s.Rs_stack*. Updating of the best found packing plan *pbest* is also carried out where applicable when specifying the best block for a residual space (cf. Section 3.4).

The method *find_best_block* carries out a tree search for the specification of a block *bl* whose effort is defined through the internal parameter *search_effort*. This is doubled from solution to solution and thus better and better packing plans can be calculated.

In favour of the selected method is that it is hardly possible to assess a priori the scope that the tree search should have to generate a solution with highest possible volume utilisation with a moderate amount of time spent. In addition, very little time is spent generating the first packing plans, i.e. for low values of *search_effort*.

To a certain extent, CLTRS behaves like a construction heuristic in each iteration: if a block was specified, this decision is not revised when further blocks are determined.

3.3. The generalized block building approach

If excellent results for weakly heterogeneous instances were already achieved by means of the block building approach, this has not been the case up to now for strongly heterogeneous instances. As was said above, the motive of the block building approach consists in filling relatively large part spaces of the container without loss. Its application is, however, linked to the precondition that a relatively large number of exemplars are available per box type. This is not given for strongly heterogeneous box stocks, so that no blocks can be formed, or only very small ones. In order to make the approach fruitful for the strongly heterogeneous case as well it has to be generalized: while on the one hand blocks are to fill relatively large volumes without loss, or practically without loss, they may now consist as well of boxes of several types. The generalization is brought about through the introduction of new block types, which will be explained in detail below.

3.3.1. Block types of the generalized block building approach

a) "Simple" type blocks

Blocks in the traditional sense are referred to as "simple" type blocks or as simple blocks. A simple block is therefore a cuboid arrangement from $nx \times ny \times nz$ boxes of the same type in the same spatial orientation, whereby *nx* boxes lie in the x direction, *ny* boxes in the y direction and *nz* boxes in the z direction (cf. Figure 4).

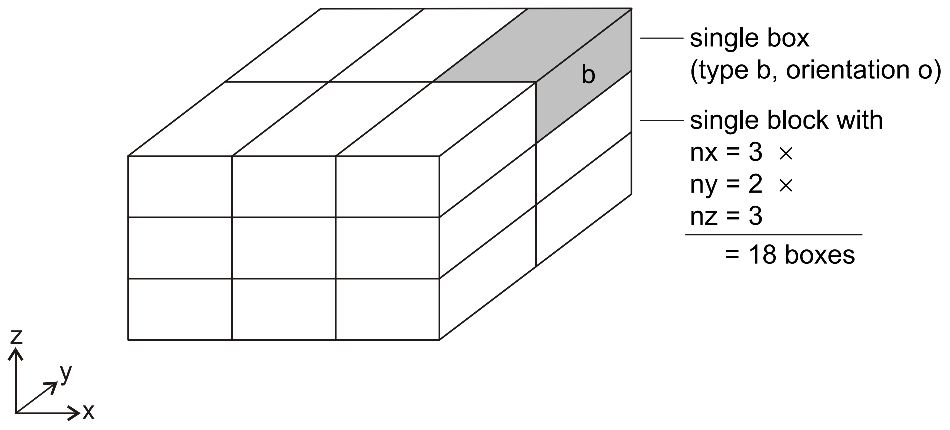

Figure 4: Block of type “Simple”

b) "General" type blocks for the cutting variant

A block of the "general" type for the cutting variant of the CLTRS method is also referred to as a general cutting block (GC block for short). It contains an arrangement of one or more boxes and the appropriate oriented envelope cuboid C_{env}. The envelope cuboid is defined as usual, i.e. it is touched on each of its inner sides by the box arrangement.

General cutting blocks can be defined recursively (but rather informally) as follows:

(i) A single box of any type in any permissible spatial orientation is a GC block.

(ii) Let bl_1 and bl_2 be GC blocks. Let the box arrangements be observed that are obtained if the envelope cuboids $C_{env}(bl_1)$ and $C_{env}(bl_2)$ of both GC blocks with the enclosed boxes are arranged contiguous in x direction or contiguous in y direction or contiguous in z direction, as sketched in Figure 5. Each of the three combined box arrangements bl_c forms a GC block with the appropriate envelope cuboid $C_{env}(bl_c)$, if the following conditions are fulfilled:

(ii.1) Box availability
The box set of the combined box arrangement bl_c, i.e. the merger of the box sets of the GC blocks bl_1 and bl_2, is a subset of the given stock of boxes.

(ii.2) Space availability
The envelope cuboid $C_{env}(bl_c)$ can be arranged completely in the given container.

(ii.3) Contact surfaces comparison
If the GC blocks bl_1 and bl_2 are combined in x-direction bl_1 may only lie to the left of bl_2 if the following applies for the contact surfaces:

$$my(C_{env}(bl_1)) \times mz(C_{env}(bl_1)) \geq my(C_{env}(bl_2)) \times mz(C_{env}(bl_2)).$$

On a combination in y-direction bl_1 may only be in front of bl_2 if:

$$mx(C_{env}(bl_1)) \times mz(C_{env}(bl_1)) \geq mx(C_{env}(bl_2)) \times mz(C_{env}(bl_2)).$$

On a combination in z-direction bl_1 may only be underneath bl_2 if:

$$mx(C_{env}(bl_1)) \times my(C_{env}(bl_1)) \geq mx(C_{env}(bl_2)) \times my(C_{env}(bl_2)).$$

(ii.4) Space utilization
The percentage space utilization of the envelope cuboid $C_{env}(bl_c)$, given by the quotient fr := (volume of all boxes of bl_c) / (volume of $C_{env}(bl_c)$ × 100, does not fall below the given minimal space utilization min_fr (in %), i.e. $fr \geq min\_fr$.

(iii) Only a box arrangement, which is identifiable in accordance with (i) and (ii) as a GC block, represents a GC block. □

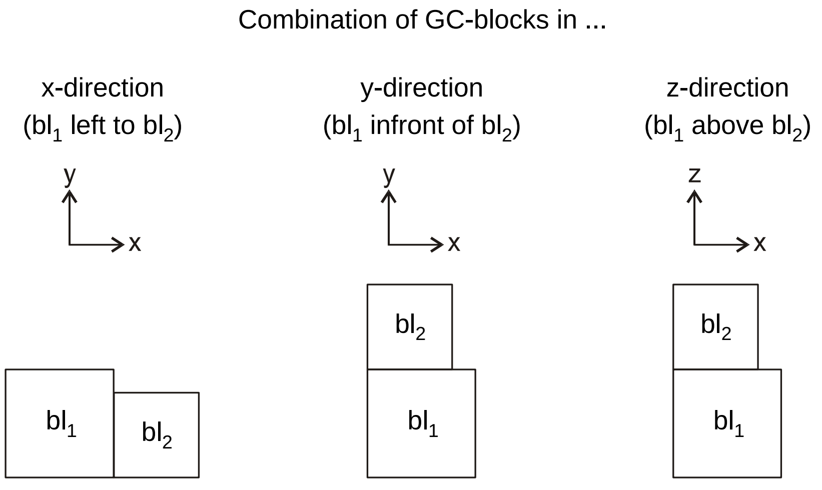

Figure 5: Combining GC blocks in x-, y- and z-direction

The definition of a GC block may be commented as follows. The blocks bl_1 and bl_2 are referred to where applicable as the parent blocks of a combined block bl_c. The infringement of the conditions (ii.1) and (ii.2) would lead to blocks that are of no use for the search. The contact surfaces condition (ii.3) stipulates which of the two parent blocks lies in the reference corner of a combined block bl_c and helps to avoid redundant GC blocks. Condition (ii.4) is decisive: if the parameter *min_fr* for block generation is selected large enough (for example *min_fr* = 98%), only blocks that are practically without loss are generated, while at the same time, the combination of boxes of different types in GC blocks is not ruled out. The definition therefore conforms to the above-mentioned requirements for a generalization of the original block definition. In addition, it is obvious that every simple block is at the same time a general block. Figure 6 shows a GC block that is created through the combination of two simple blocks in z-direction. At the same time, the example makes clear that GC blocks in general do not comply with the support constraint (C2). This means that an additional block type is required for the packing variant of the CLTRS, if the block building approach is to be generalized in compliance with the support constraint (C2) as well.

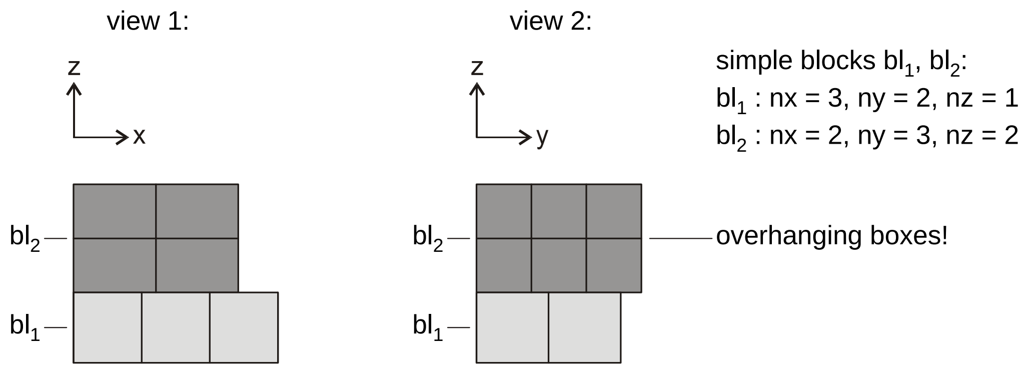

Figure 6: GC block with overhanging boxes

c) "General" type blocks for the packing variant

A "general" type block for the packing variant is also referred to as a general packing block (GP block for short). Together with a box arrangement and the appropriate envelope cuboid C_{env} it contains a packing area. This is part of the top area of the envelope cuboid with the following characteristic: if a box lies completely on the packing area, its area is supported completely by other boxes. GP blocks are now defined by stating that the packing area is always rectangular and the front left-hand corner (i.e. the corner nearest to the origin of the 3D-CS) is incidental to the front left-hand corner of the covering area of the envelope cuboid. Consequently, the packing area of a GP block is clearly determined by the coordinates x_2, y_2 of its rear right-hand corner relative to the reference corner of the envelope cuboid (cf. Figure 7).

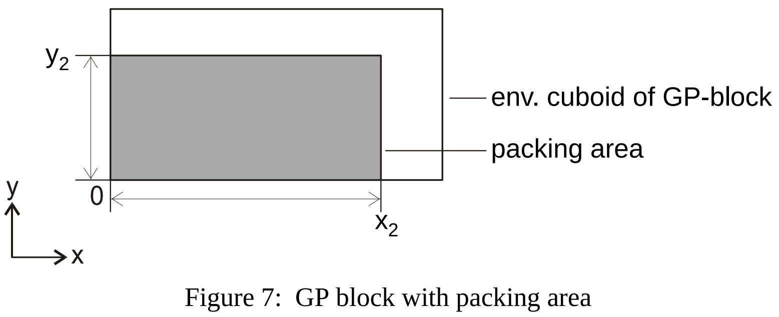

Figure 7: GP block with packing area

General packing blocks are defined analogously to general cutting blocks. While parts (i) and (iii) of the above definition can be taken over, part (ii) is to be extended by two conditions. The packing area condition shown in Table 1 stipulates requirements for the packing areas of two GP blocks to be combined. At the same time the packing area of the new combined GP block is fixed. The height condition requires that on a combination of two GP blocks bl_1 and bl_2 in a horizontal direction (x or y) the heights of the envelope cuboids of bl_1 and bl_2 coincide.

Table 1: Packing area condition as part of the definition of GP blocks

Given combination direction, position of parent blocks bl_1 and bl_2	Conditions for packing areas of parent blocks bl_1 and bl_2	New packing area: rear right-hand corner
x-direction bl_2 on right of bl_1	$x_2(bl_1) = mx(bl_1)$ $x_2(bl_2) = mx(bl_2)$ $y_2(bl_1) \geq y_2(bl_2)$	$x_2 = mx(bl_1) + mx(bl_2)$ $y_2 = y_2(bl_2)$
y-direction bl_2 behind bl_1	$y_2(bl_1) = my(bl_1)$ $y_2(bl_2) = my(bl_2)$ $x_2(bl_1) \geq x_2(bl_2)$	$x_2 = x_2(bl_2)$ $y_2 = my(bl_1) + my(bl_2)$
z-direction bl_2 above bl_1	$x_2(bl_1) \geq mx(bl_2)$ $y_2(bl_1) \geq my(bl_2)$	$x_2 = x_2(bl_2)$ $y_2 = y_2(bl_2)$

The packing area condition is made clear by means of the combination of two GP blocks in x-direction in Figure 8.

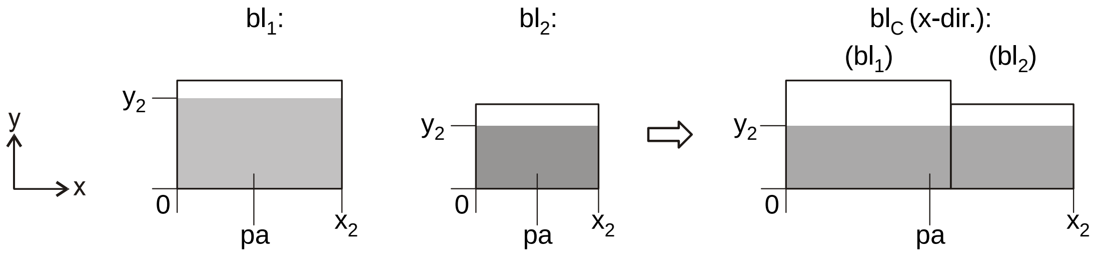

Figure 8: Combining two GP blocks in x direction (pa: packing area)

According to part (i) of the definition, individual boxes in a defined spatial orientation are special simple GP blocks whose packing area coincides with the top area of the box. The conditions for the packing areas of GP blocks that are to be combined guarantee that the packing area of a new, combined GP block is also rectangular and starts in the front left-hand corner of the envelope cuboid's top area. It is also guaranteed that each box that lies fully on the packing area is supported completely from below. This means that the requirement for the inclusion of the support constraint (C2) in the generalization of the original block concept is fulfilled. Note that each simple block is also a GP block.

3.3.2. Generating blocks and block data

Simple blocks are generated at the start of the first stage of the search and general blocks are made available in the block list *Bll* at the start of the second stage. The cutting and the packing variant differ as explained with regard to general blocks (GC blocks versus GP blocks).

The block generation may be characterized in detail as follows:

- Starting from individual boxes, the blocks for the different method variants and stages of the search are generated systematically on the basis of the recursive definitions shown above (an algorithmic representation is waived for this reason).
- If the envelope cuboids and the packed box sets of two generated blocks coincide, only one of the blocks is included in the block list. This reduces the size of the block list as well as the redundancy of the whole search.
- The number of places in the block list is restricted by the parameter *max_bl*. Accordingly, the block generation is interrupted once *max_bl* blocks have been included in the block

list for a search stage. Experiments must be carried out to determine suitable values for the parameters *max_bl* and the minimal block utilization *min_fr*.
- The filled block list *Bll* is sorted in descending order according to the block volume.

Data that are relevant during the search are recorded for each block. These include the dimensions and the volume of the envelope cuboid, the arranged box set and, with the packing variant, the coordinates of the rear right-hand corner of the packing area. Other data are only of interest for the concluding output of the calculated packing plan and fix the orientation and relative position of the individual boxes within the block. In order to determine the final position of the individual placed boxes in the container, its generation process is, so to speak "reversed". Accordingly, indices of its parent blocks and their relative position are recorded for each block.

3.4. Determining the best block for a residual space

The core of the method is determining the best block in the current block list for the next residual space to be loaded. Roughly speaking, the best block is determined as follows:
- Let a search state *s* = (*p*, *vc*, *Bres*, *Rs_stack*) be given with a solution from *i* ($i \geq 0$) placed blocks.
- In order to determine a block for the uppermost residual space *rstop* in *s.Rs_stack*, starting from *s*, i.e. retaining the *i* blocks already placed, different complete solutions *p*' with *i*' > *i* placed blocks are generated experimentally.
- Let *ptmpbest* be the best complete temporary solution with maximal packed volume. Of course the (*i*+1)th block of all temporarily generated solutions fills the residual space *rstop*. Finally, the (*i*+1)th block in *ptmpbest* is returned as the best block for *rstop*.

A multiple tree search is carried out to generate complete solutions starting each time from search state *s*. This means that a differently configured tree search, a so-called partition search, is started from *s* several times and *ptmpbest* is determined as the best solution over all partition searches.

All partition searches starting at state *s* are based on a uniform additional search depth *total_add_blocks*, which indicates the maximal number of additionally placed blocks (to the *i* blocks in *s*). The additional search depth is stipulated in accordance with:

$$total\_add\_blocks = \max\{1, \max\{d \mid d \geq 0 \text{ and } min\_ns ** d \leq search\_effort\}\} \qquad (1)$$

For *min_ns* = 2, *search_effort* = 4, e.g., the result is *total_add_blocks* = 2. The parameter *min_ns* indicates the desired minimal number of successors of a partial solution (or node) throughout the search.

Solutions generated during a partition search that already show the maximal number of blocks (*i* + *total_add_blocks*) are generally still incomplete. Each solution of this type is transformed by means of the so-called standard completion into exactly one complete solution: starting from the last achieved search state *s*', the uppermost residual space (if possible) is repeatedly filled with a block with the maximal volume until a complete solution is achieved. These complete solutions are then used to determine *ptmpbest*.

An ordered partition π of a whole number *n* ($n \geq 1$) is given by a (*l*+1)-tuple of integers:

$$\pi = (l, d_1, \ldots, d_l), \text{ with } l \geq 1,\ d_j > 0,\ j = 1,\ldots,l \text{ and } \sum_{j=1}^{l} d_j = n.$$

In the framework of the multiple tree search a partition search, i.e. a tree search controlled by a partition π, is carried out for each partition π of the given additional search depth *total_add_blocks*. The partition $\pi = (l, d_1, \ldots, d_l)$ stipulates that the appropriate search is broken down into *l* search phases and that on the *j*-th search phase maximal d_j blocks are placed in addition ($j = 1,\ldots,l$).

Let the first search phase of a partition search be considered first:

- Starting from the search state *s* with *i* placed blocks (cf. above), a best search state *stmpbest* with maximal $i + d_1$ blocks is determined.
- For this purpose, for each occurring partial solution *p*, starting with *s.p*, ns_1 successor solutions *p*' including the appropriate search states *s*' are generated. The number of successors ns_1 is calculated in accordance with

  $ns_1 = \max\{min\_ns, \max\{n \mid n \geq 0 \text{ and } n ** d_1 \leq search\_effort\}\}$ (2)

  For $min\_ns = 2$, $search\_effort = 4$ and $d_1 = 1$ or $d_1 = 2$ the result is $ns_1 = 4$ or $ns_1 = 2$.
- The ns_1 successors of a solution are generated by filling the next residual space to be loaded alternately with one of the largest volume ns_1 matching blocks.
- The usual case is that a solution with $i + d_1$ blocks is not yet complete and is then completed by means of the standard completion sketched above.
- The best phase solution *ppbest* is determined from among all the complete solutions generated during the first search phase. Where applicable, the best solution of the current partition search *ptmpbest* and the best solution of the complete search *pbest* are updated
- The best search state of the first phase *stmpbest* results as search state after the first $i + d_1$ blocks of the best phase solution *ppbest* are placed. Put another way: *stmpbest* is the search state with $i + d_1$ blocks from which the best phase solution results by standard completion.

The following search phases of a partition search run analogously to the first phase. The best achieved search state of the *j*-th phase *stmpbest* serves as the initial state of the (*j*+1)th phase and already contains $i + d_1 + \ldots + d_j$ blocks finally placed in the framework of the partition. The number of successors results from equation (2), if d_1 is replaced by d_{j+1}.

An example may be considered to illustrate the partition search and its search phases. Let $min\_ns = 2$, $search\_effort = 8$ (4th iteration), i.e. $total\_add\_blocks = 3$. Consequently, separate tree searches must be carried out for the following partitions: (1) $\pi_1 = (1, 3)$, (2) $\pi_2 = (2, 2, 1)$, (3) $\pi_3 = (2, 1, 2)$ and (4) $\pi_4 = (3, 1, 1, 1)$. The following figures 9a to 9d illustrate the four partition searches.

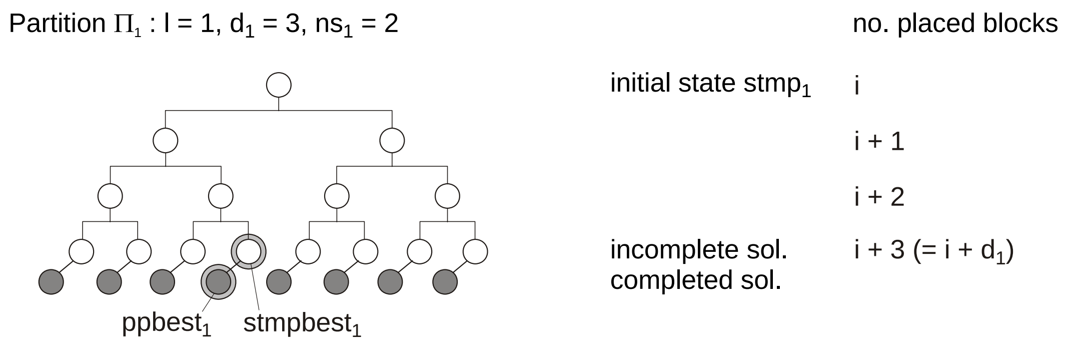

Figure 9a: Partition search for total_add_blocks = 3 (Partition π_1)

Some of the heuristic elements used in the serach for the best block are known. These include the restriction of the search per residual space to the blocks with the largest volumes (cf. Bortfeldt et al. 2003). The restriction of the (additional) search depth saves effort on the one hand, while the supplementing standard completion of solutions permits a certain assessment of the quality of the previously achieved partial solution (cf. Eley 2002). However, the heuristic core concept is the design of the search as a (multiple) partition-controlled tree search (PCTRS) in the sense defined above.

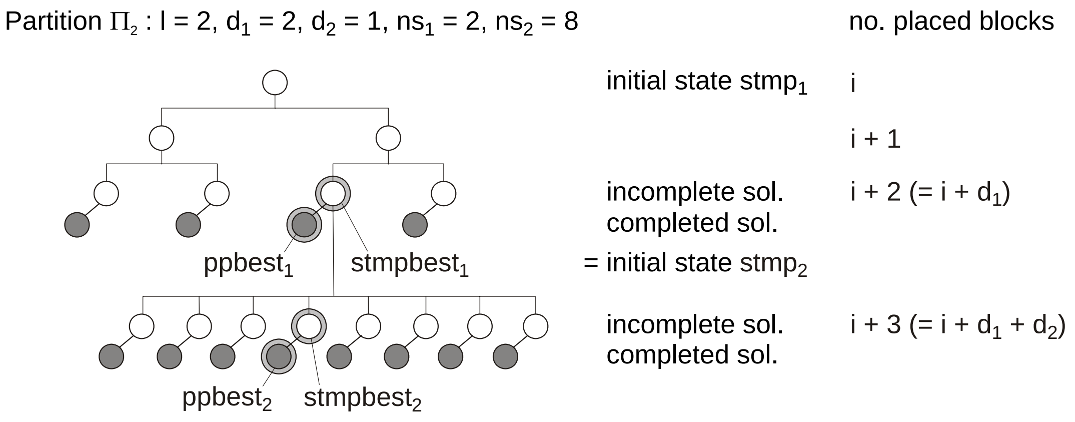

Figure 9b: Partition search for total_add_blocks = 3 (Partition π_2)

The PCTRS may be characterized in greater detail by means of an example. Starting from *i* already placed blocks, complete solutions are to be generated through a tree search with an additional search depth *total_add_blocks* = 3 and subsequent standard completion. If a tree search is now carried out with a constant number of successors *ns* = 8, then 8^3 = 512 complete solutions are to be generated. In contrast, in the example under consideration, only 8+12+12+24 = 56 complete solutions are generated with PCTRS, this is only 11% of the solutions generated with constant *ns* = 8 (cf. Figure 9a–9d).

This saving is achieved in two ways: firstly, through a reduction of the number of successors *ns* in the search phases for the different partitions. This applies in particular for partition π_1, where *ns* = 2 is used throughout instead of *ns* = 8 (cf. Figure 9a). However, the described partitioning of the search also leads in itself to a reduction on the number of complete solutions. For example, in partition π_4 *ns* has the maximum value 8 throughout, while at the same time only 24 complete solutions are generated (cf. Figure 9d). The reduction is formed by a certain decision already being finally taken in each phase. For example, the first search phase of the partition search for π_4 decides on the (*i*+1)th block, the second search phase decides on the (*i*+2)th block of the best solution generated by this partition search.

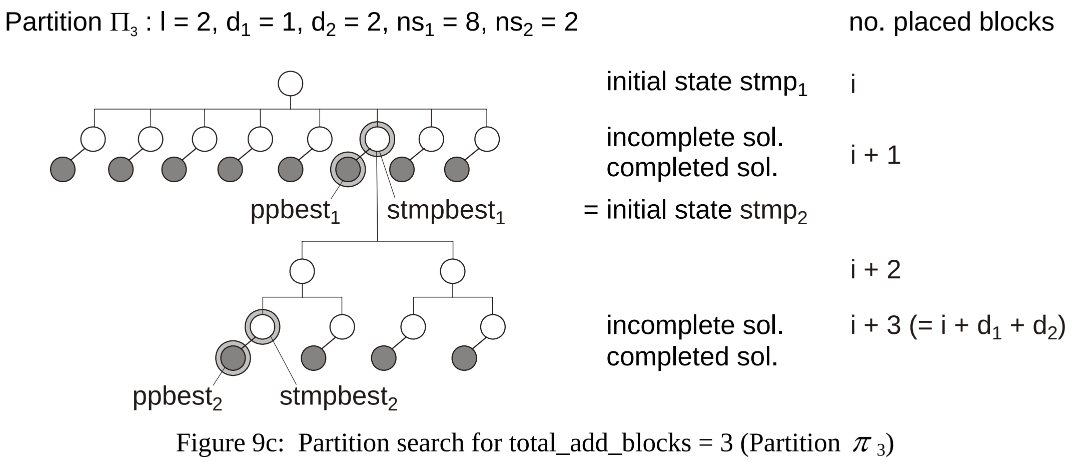

Figure 9c: Partition search for total_add_blocks = 3 (Partition π_3)

The number of successors of a search phase of a partition search determines the (local) search width or diversity. The search depth d_j stipulates at the same time the number of consecutive search levels that are included in the decision made by the search phase and in this way determines its degree of foresight. This means that the only search phase of partition π_1 has a greater degree of foresight with the search depth 3 than the three phases of π_4 each with the search depth 1.

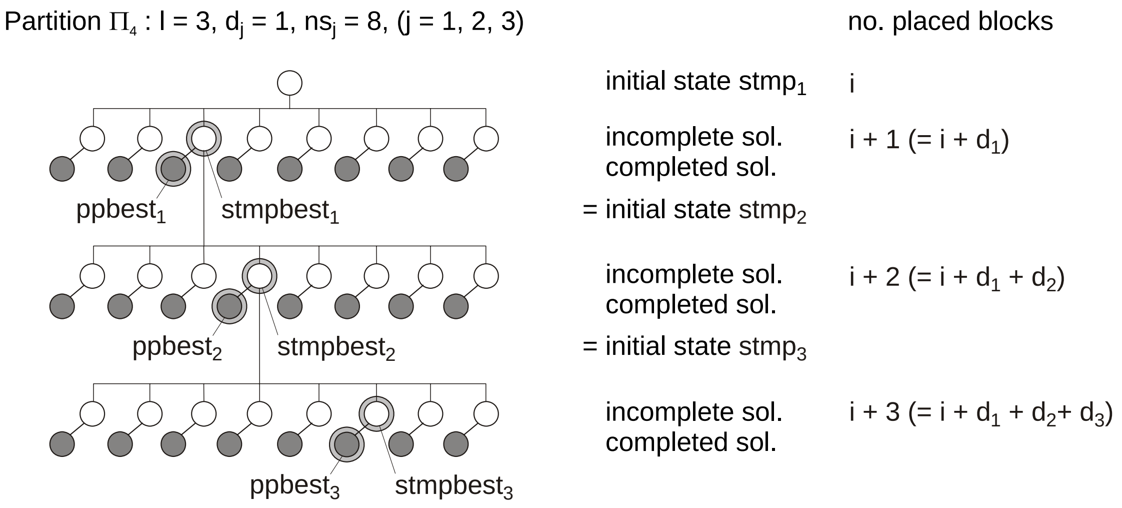

Figure 9d: Partition search for total_add_blocks = 3 (Partition π_4)

With the PCTRS a separate partition search is carried out for each partition of a given total additional search depth (*total_add_blocks*), whereby the search widths and the degree of foresight differ greatly for the individual search phases. In this way, a too one-sided stress of one factor of success with is (hopefully) avoided, and a more balanced consideration of the search width and the foresight is achieved.

Block determination for a residual space is now described formally. The procedure *find_best_block* (cf. Figure 10) receives the initial state *s* as a (pure) input parameter.

```
procedure find_best_block(in: s, out: found, out: bl)
found := TRUE; // block found
// generate block list for top residual space to deal with special cases
generate_rs_blocklist(top(s.Rs_stack),s.Bres,2,nbl,Bll_rs);
if (nbl = 0) then found := FALSE; return; endif; // no suitable block
if (nbl = 1) then bl := Bll_rs(1); return; endif; // only one suitable block
total_add_blocks := max{1, max{d | d ≥ 0, min_ns**d ≤ search_effort}}; // total additional search depth
ptmpbest := s.p; // best temporary packing plan
s.vc := s.p.v;

// carry out multiple tree search
for each ordered partition (l,d_1,…,d_l) of total_add_blocks do
        // initialize global state variable stmpbest
        stmpbest := s;
        for j : =1 to l do
             // provide global initial state stmp for next search phase
             stmp := stmpbest;
             // determine maximal number of successors for j-th search phase
             ns := max{min_ns, max{n | n ≥ 0, n ** d_j ≤ search_effort}};
             // carry out j-th search phase starting from state stmp with search depth d_j
             extend_solution(d_j, 0, ns);
        endfor;
endfor;
i := |s.p.Pbl|; // Index of the last old block
output(ptmpbest.Pbl(i+1)); // (i+1)th block of the best temporary complete solution
end.
```

Figure 10: Procedure find_best_block

First of all, the special cases are dealt with, i.e. there is no block, or only one, for the current residual space (*nbl* = 0, 1). If neither of these special cases is found, the multiple tree search is carried out for all partitions π of the previously determined additional search depth *total_add_blocks*. For each partition search the state variable *stmp* contains the respective current state of a search phase and is initialized with the best state of the preliminary phase *stmpbest*, at the start with the initial state *s*. After the multiple tree search, the block of the best solution *ptmpbest* packed in the uppermost residual space of *s* is returned.

The recursive procedure *extend_solution* (s. Figure 11) serves to extend the solution *stmp.p* by one block in each case. Whenever the procedure is called within a search phase the parameters *max_add_blocks* (conforms to a d_j) and *no_succ* (number of successors) are fixed, while the current number of additional blocks *no_add_blocks* increases by 1 per call.

```
procedure extend_solution(in: max_add_blocks, in: no_add_blocks, in: no_succ)
// deal with special cases …
if (stmp.Rs_stack = ø or stmp.Bres = ø or              // temporary solution already complete or…
   no_add_blocks = max_add_blocks) then                // max. depth of search phase reached
        scompl := stmp;
        complete_solution(scompl); // perform standard completion
        stmp.vc := scompl.p.v;
        if (stmp.vc > stmpbest.vc) then stmpbest := stmp; endif; // best phase state
        if (scompl.p.v > ptmpbest.v) then ptmpbest := scompl.p; endif; // best compl. temp. solution
        if (scompl.p.v > pbest.v) then pbest := scompl.p; endif; // best global solution
        return;
endif;
// generate block list for top residual space
generate_rs_blocklist(top(stmp.Rs_stack), no_succ, stmp.Bres, nbl, Bll_rs);
// deal with special case of empty block list
if (nbl = 0) then
        update_state(stmp, FALSE, dummy_block, rs);
        extend_solution(max_add_blocks, no_add_blocks, no_succ);
        restore_state(stmp, FALSE, dummy_block, rs);
        return;
endif;

// extend current solution alternatively by one block of current block list
for i := 1 to nbl do
        update_state(stmp, TRUE, Bll_rs(i), rs);
        extend_solution(max_add_blocks, no_add_blocks + 1, no_succ);
        restore_state(stmp, TRUE, Bll_rs(i), rs);
endfor;
end.
```

Figure 11: Procedure extend_solution

At the beginning a test is made to see whether the solution *stmp.p* is still to be extended. This is not the case if *stmp.p* is already complete or if *no_add_blocks* has already reached the maximal value *max_add_blocks*. In the latter case the standard completion is applied to *stm.p*. In both cases the best solutions (*ptmpbest*, *pbest*) and the best state *stmpbest* are updated where necessary. The component s*tate.vc* of a state variable *state* with the generally incomplete solution *state.p* saves the volume value of the complete solution obtained by standard completion from *state.p* and is required for comparison purposes.

Another special case occurs when the block list for the current residual space of the phase search is empty. Residual spaces that cannot be filled are simply by-passed, i.e. removed from the stack.

If none of the special cases are found, the current solution is extended alternatively by one of the first blocks of the block list and the method *extend_solution* is called again.

Figure 12 shows the procedure *complete_solution* for the standard completion of the solution *sc.p* integrated in a search state *sc*.

```
procedure complete_solution(inout: sc)
while (sc.Rs_stack ≠ ø and sc.Bres ≠ ø) do
        rs := pop(sc.Rs_stack);
        generate_rs_blocklist(rs, sc.Bres, 1, nbl, Bll_rs);
        if (nbl > 0) then update_state(sc, TRUE, Bll_rs(1), rs); endif; // pack largest block
endwhile;
end.
```

Figure 12: Procedure complete_solution.

3.5. Other procedures

The remaining procedures of the CLTRS method are specified below.

a) Procedures for the maintenance of the search state

Figure 13 shows procedures for the maintenance of the search state given by a state variable *s*. The procedure *init_state* initializes the search state for generating a packing plan. The procedure *update_state* updates the search state after a residual space processing, while the procedure *restore_state* reverses this updating. With *update_state* and *restore_state* the cases of the availability or not of a block *bl* for the relevant residual space are to be differentiated.

```
procedure init_state(inout: s)
s.p.Pbl := ø; s.p.v := 0; s.Bres := {all boxes}; s.Rs_stack := {container};
end.

procedure update_state(inout: s, in: block_available, in: bl, out: rs)
rs := pop(s.Rs_stack);
if (block_available) then
        s.p.Pbl := s.p.Pbl U {(bl, rs.rc)}; s.p.v := s.p.v + volume(bl);
        s.Bres := s.Bres \ {boxes in bl};
        generate_daughter_spaces(s, rs, bl);
endif;
end.

procedure restore_state(inout: s, in: block_available, in: bl, in: rs)
if (block_available) then
        s.p.Pbl := s.p.Pbl \ {(bl, rs.rc)}; s.p.v := s.p.v − volume(bl);
        s.Bres := s.Bres U {boxes in bl};
        for i := 1 to 3 do pop(s.Rs_stack) endfor;
endif;
push(s.Rs_stack, rs);
end.
```

Figure 13: Procedures init_state, update_state and restore_state

b) Procedure generate_rs_blocklist

The procedure *generate_rs_blocklist* (in: *rs*, in: *Bres*, in: *no_succ*, out: *nbl*, out: *Bll_rs*) provides a residual space block list *Bll_rs* of *nbl* (*nbl* ≥ 0) admissible blocks for the residual space *rs* and the set *Bres* of free boxes. For this purpose the block list *Bll* of the current search stage is run through from the front to the rear and admissible blocks are taken over consecutively in the list *Bll_rs*, i.e. in descending order of the volume of the envelope cuboid. In this way, maximal *no_succ* blocks are provided for the residual space *rs*. A block is deemed to be admissible if its oriented envelope cuboid fits in the residual space *rs* and all boxes of the block are still free.

c) Procedure generate_daughter_spaces

The procedure *generate_daughter_spaces* (inout: *s*, in: *rs*, in: *bl*) presupposes that the block *bl* was previously placed in the residual space *rs* and *rs* was removed from the residual space stack *s.Rs_stack*. Three daughter residual spaces are now to be generated within *rs* and inserted in *s.Rs_stack*. The three new residual spaces are to completely fill the space remaining after placing *bl*. The cutting variant will be considered first.

Figure 14 shows residual space *rs* and block *bl* each with their dimensions and the differences *rmx* = *rsx* – *blx*, *rmy* = *rly* – *bly* sowie *rmz* = *rsz* – *blz* called reduction measures.

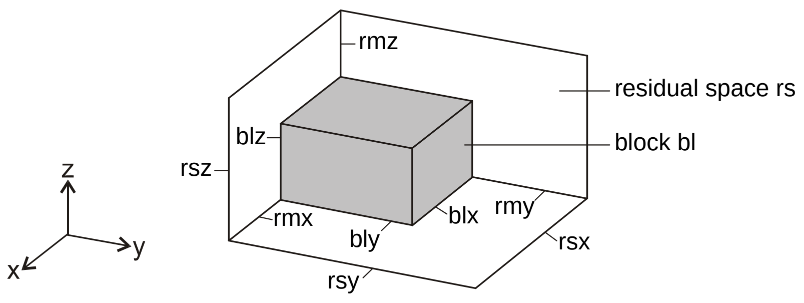

Figure 14: Residual space rs, block bl and reduction measures

If the free space in *rs* is cut into three new residual spaces, the following applies:

- A new residual space, referred to as *drs_x*, lies to the right next to the block *bl* and has the x-dimension *rmx*. A further residual space, referred to as *drs_y*, lies behind *bl* and has the y-dimension *rmy*. The third new residual space, referred to as *drs_z*, lies above *bl* and has the z-dimension *rmz*.
- Exactly one of the new residual spaces coincides with the residual space *rs* in two dimensions and is designated as maximal (max for short). Exactly one of the new residual spaces coincides with the block *bl* in two dimensions and is designated as minimal (min). The third residual space is then designated as medium (med).
- The three residual spaces are generated by three guillotine cuts, the first of which subdivides the mother residual space *rs*.

The result is the following 6 possible partitioning variants shown in Table 2.

Table 2: Partitioning variants for a residual space

No.	Residual space definition			To be used with
	drs_x	drs_y	drs_z	ranking of the reduction measures
1	max	med	min	rmx, rmy, rmz
2	med	max	min	rmy, rmx, rmz
3	max	min	med	rmx, rmz, rmy
4	med	min	max	rmz, rmx, rmy
5	min	max	med	rmy, rmz, rmx
6	min	med	max	rmz, rmy, rmx

In the case of partitioning variant 1 the residual spaces have the following dimensions (cf. Figure 15):

- *drs_x: mx = rmx, my = rsy, mz = rsz;*
- *drs_y: mx = blx, my = rmy, mz = rsz;*
- *drs_z: mx = blx, my = bly, mz = rmz.*

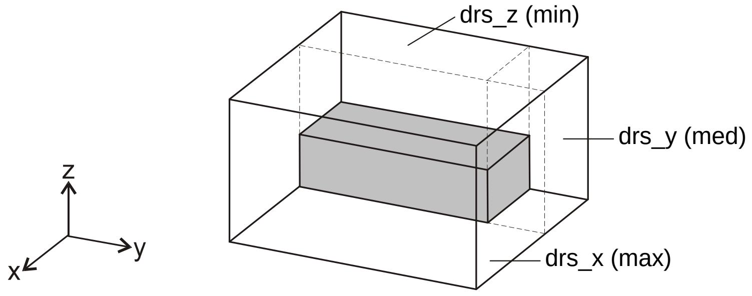

Figure 15: Residual spaces of partitioning variant 1

The partitioning variant to be used is selected by means of the reduction measures for a residual space *rs* and block *bl*. First of all it is stipulated: one reduction measure *rm1* has a higher rank than another reduction measure *rm2*, if:

- *rm1* ≥ *rm2* or
- *rm2* is smaller than the minimal box measure placeable in *rm2* direction.

Based on this definition a ranking of the reduction measures is determined for a given residual space *rs* and block *bl*; with several possible rankings, any one is selected. The selection of the partitioning variant is then carried out in accordance with Table 2 (last column). If e.g. *rmx* = 70, *rmy* = 50 and *rmz* = 60 and if none of the dimensions falls below the minimal box dimension in the direction concerned, the result is the order of preference *rmx*, *rmz*, *rmy* and partitioning variant 3 is to be applied. In contrast, if *rmz* (and only *rmz*) falls below the minimal box dimension in z direction, the order of preference is *rmx*, *rmy*, *rmz* and partitioning variant 1 is to be selected.

The reduction measures are the initially critical dimensions of the new residual spaces, i.e. they can above all have the effect that new residual spaces can no longer be filled. The described partitioning rule prevents relatively small reduction measures being paired with relatively large other residual space measures and thus large volume losses being risked.

The still outstanding rule for inserting new residual spaces into the residual space stack is to be introduced together with a further element of residual space management, namely the transfer of unused packing volume.

Let, for example, residual space *rs* be partitioned in accordance with variant 1 (cf. Figure 15). If the maximal residual space *drs_x* now proves not to be fillable, the minimal residual space *drs_z* can be enlarged by setting its x-dimension to *rsx*. Thus it may still be possible to use the cuboid with the dimensions *mx* = *rmx*, *my* = *bly* and *mz* = *rmz* that is transferred from *drs_x* to *drs_z*.

In general, the procedure is as follows:

- Insertion rule: the new residual spaces are inserted into the residual space stack in the order of preference "min – max – med", i.e. minimal residual space according to selected partition variant first, etc.
- However, if a residual space cannot be filled from the start because the reduction measure is too small, it is not inserted in the stack in the first place.
- If the medium or maximal residual space proves not to be fillable after being removed from the stack, a suitable part cuboid is transferred subsequently to the minimal residual space. This operation was not shown in the procedure *update_state*, and neither was the reverse operation shown in the procedure *restore_state*. Packing space is not transferred from the maximal to the medium residual space.

The insertion rule guarantees that packing space can be transferred subsequently to the minimal new residual space. In addition, the maximal new residual space is placed on the stack before the medium space because the maximal residual space can in general be filled more easily.

The modifications that are required for the packing variant are very simple. Firstly, only the first two partitioning variants are permissible in which the residual space *drs_z* above block *bl* is minimal and therefore lies fully on the top area of *bl*. Secondly, the bottom area of *drs_z* is reduced to the packing area of the block *bl*. In addition, only residual spaces *drs_x* and *drs_y* are permissible as receivers of a space transfer.

3.6. Handling the selected constraints

Finally, handling the selected constraints (C1) to (C3) is dealt with briefly.

The guillotine cut constraint (C1) fulfils itself in both the cutting variant and in the packing variant of the CLTRS method. The boxes of any block can obviously be reproduced by guillotine cuts. Each placed block lies fully in a residual space and finally, starting from the container, additional residual spaces are created only through guillotine cuts.

The support constraint (C2) is fulfilled in the packing variant of CLTRS. On the one hand with this variant only blocks are formed in which all boxes that do not lie on the floor of the block are supported 100% from below (cf. Section 3.3.1); on the other hand, residual spaces above blocks are constructed in such a way that no overhanging boxes can occur (cf. Section 3.5, c)).

The orientation constraint (C3) is complied in that impermissible box orientations are avoided per block and box type.

4. Numerical test

The CLTRS method was implemented in C. The 1600 3D-CLP instances from Bischoff and Ratcliff (1995) and from Davies and Bischoff (1998) respectively were included for the test; they are referred to below as Bischoff-Ratcliff instances or BR instances.

The 1600 BR instances are subdivided into 16 test cases with 100 instances each that are numbered BR0 to BR15. The 100 instances of each test case coincide in the number of box types. The box sets of the different test cases vary from homogeneous, through weakly heterogeneous to strongly heterogeneous; details can be seen in Table 3. Each instance includes the orientation constraint (C3), which prohibits the use of certain larger side dimension as height dimension. With the test of the cutting variant of the CLTRS it is assumed that the guillotine cut constraint (C1) is to be complied with in addition. With the test of the packing variant it is presupposed that the support constraint (C2) has to be complied with in addition.

A PC with a 2.6 GHz Intel processor was used for the test. The following set of parameters was used uniformly for all tested instances and both CLTRS variants: maximal number of blocks *max_bl* = 10000, minimal space utilization for blocks *min_fr* = 98%, minimal search width *min_ns* = 3, time limits for stages 1 and 2: 60 and 180 seconds respectively.

Table 3 contains the test results for both CLTRS variants. For the 100 instances of each test case the mean volume utilization is given as a percentage of the container volume and the standard deviation (in %) is also given.

Table 3: Test results of method CLTRS for the Bischoff and Ratcliff test cases

Test case	No. of box types	Avg. no. of boxes per type (rounded)	Cutting variant Volume utilization (%)	Std. Deviation (%)	Med-Supp (%)	Packing variant Volume utilization (%)	Std. Deviation (%)	St1	St2 (%)
BR0	1	206	89.95	6.5	94.91	89.83	6.5	1.16	3.30
BR1	3	50	95.05	1.9	94.19	94.51	2.2	1.37	5.45
BR2	5	27	95.43	1.2	92.93	94.73	1.5	1.40	8.05
BR3	8	17	95.47	0.8	90.91	94.74	1.0	1.37	10.78
BR4	10	13	95.18	0.7	89.73	94.41	1.0	1.39	11.78
BR5	12	11	95.00	0.6	88.96	94.13	0.8	1.39	13.73
BR6	15	9	94.79	0.6	86.37	93.85	0.7	1.41	15.76
BR7	20	7	94.24	0.6	82.21	93.20	0.7	1.36	19.50
BR8	30	4	93.70	0.5	79.96	92.26	0.8	1.38	24.76
BR9	40	3	93.44	0.5	76.88	91.48	0.7	1.38	30.17
BR10	50	3	93.09	0.5	76.56	90.86	0.7	1.36	33.21
BR11	60	2	92.81	0.5	76.01	90.11	0.8	1.33	38.17
BR12	70	2	92.73	0.5	74.37	89.51	0.7	1.32	40.83
BR13	80	2	92.46	0.5	74.07	88.98	0.9	1.30	42.89
BR14	90	1	92.40	0.5	73.59	88.26	0.8	1.27	46.67
BR15	100	1	92.40	0.5	59.71	87.57	0.8	1.25	50.45
Avg.	–	–	**93.6**	1.1	–	**91.8**	1.3	1.3	24.7

In addition, the indices for stability St1 and St2 (cf. Bischoff and Ratcliff 1995) are shown for the packing variant, and the mean box support Med-Supp for the cutting variant:

- The index St1 indicates the mean number of boxes that support a box that does not lie on the floor of the container. St1 should be as large as possible.
- The index St2 indicates the percentage of boxes in relation to all placed boxes that are not touched on at least three sides by another object (a box or container). St2 should be as small as possible.
- The mean box support Med-Supp shows the mean share of the bottom area (in %) that is supported from below for the boxes that do not lie on the floor of the container.

The test results are evaluated first of all with regard to the generalized block building approach. Averaged over all 1600 instances, 1395 simple blocks, but 4137 general packing blocks and 9081 general cutting blocks, were generated per instance. The supply of GP and GC blocks is therefore much more extensive in comparison with simple blocks. This pays off. In both the cutting and the packing version the best solution is found for about 70% of all instances in stage 2, i.e. using general blocks as well. Without the second stage of the search the mean volume utilization would worsen by about 0.8% points in the cutting variant and by around 1.1% points in the packing variant.

However, the usefulness of the general blocks depends greatly on the heterogeneity of the box sets. In the packing variant, for the first 8 test cases BR0 to BR7 the best solution is achieved for good 44% of the instances in stage 1, that is with simple blocks only, while for the following 8 test cases BR8 to BR15 the best solution is determined in stage 1 for just 9% of the instances. Similar figures apply for the cutting variant. If the three test cases BR13 to BR15 with the most heterogeneous box sets are taken, volume utilization there is increased through the second search stage with the use of general blocks by around 2.7% points on average in the packing variant and by around 2.0% points on average in the cutting variant! All in all it is found, on the one hand, that the general blocks prove to be a key concept for achieving higher volume utilizations for strongly heterogeneous box sets; on the other hand, the selected two-stage search approach conforms to the different suitability of simple or general blocks for weakly and strongly heterogeneous instances.

For both CLTRS variants the mean volume utilizations increase to test case BR3, and then decrease monotonically. An analogous dependence of the volume utilization on the heterogeneity of the supply of boxes can also be ascertained for other methods (cf. e.g. Eley 2002, Mack et al. 2004). It therefore seems obvious to search for causes of this change in trend (monotonic growth, following by monotonic decrease) that are independent of the method applied. An approximate qualitative explanation can be outlined as follows:

- With a weakly heterogeneous box set large box arrangements, in particular with simple blocks, can be formed with few internal losses. The internal loss of a box arrangement is understood as the volume of its envelope cuboid that is not filled with boxes. On the other hand, there are only a few boxes available and therefore only a few different box dimensions. For this reason, the container dimensions can often not be represented to a sufficient extent through a combination of box dimensions. Consequently, there are space losses at the container walls (side walls, ceiling), that can be designated as boundary losses. In short: in the weakly heterogeneous case boundary losses dominate. This occurs to a particularly crass extent with homogeneous instances. For example, capacity utilization of only 65.53% is achieved for instance 23 of the test case BR0. In spite of this, the solution, as a manual test shows, is global-optimal. It consists of a single simple block, the total loss is therefore a boundary loss.
- Strongly heterogeneous box sets lead to contrary effects. On the one hand, larger box arrangements typically have considerable internal losses. On the other hand, there are many variants, to combine the container dimensions by box dimensions (or dimensions of part arrangements). In short: in the strongly heterogeneous case internal losses dominate.
- The trend change that was observed for many methods can now be understood as an effect of the interaction of the two outlined influencing factors. One may expect a peak in (average) volume utilization for instances of a certain degree of heterogeneity where the internal losses tend to be "still small" while at the same time the boundary losses tend to be "already small".

The achieved mean volume utilizations state that for the CLTRS method compliance with the support constraint (C2) "costs" about 2% points volume utilization. Similar results were already found for the method from Bortfeldt et al. (2003) and Mack et al. (2004). It is therefore suspected that, even independently of the power of a heuristic method, the support constraint (C2) ha a considerable effect on the (achievable) volume utilization. However, a detailed comparison of the volume utilizations for both CLTRS variants shows immediately that the negative influence of constraint (C2) on the volume utilization depends decisively on the heterogeneity of the box set. As Table 4 teaches, the difference between the volume utilizations for the cutting and packing variant in test case BR0 is only 0.12% points, while it is almost 5% points for BR15! The question arises here as well whether this phenomenon can be understood independently of the method used. In fact, it can be argued as follows:

- If the stock of boxes is weakly heterogeneous, packing plans with high volume utilization will consist of few part arrangements with low or no internal loss (simple blocks). However, the top surfaces of such part arrangements provide almost complete support for the upper boxes. Compliance with the support constraint therefore collides only slightly with the demand for greater volume utilization.
- If the stock of boxes is strongly heterogeneous, compliance with the support constraint leads to it no longer being possible to a greater extent to fill parts of the space above boxes. In contrast, without the constraint (C2) at every height the complete horizontal container area is available for further box placings. Here the constraint (C2) and the demand for greater volume utilization collide.
- The mean box support Med-Supp for the cutting variant shown in Table 3 achieves high values above 90% for weakly heterogeneous test cases, decreases monotonically with increasing heterogeneity and is less than 60% for the test case BR15. This speaks for the given explanation: the optimization of the volume utilization appears for weakly heterogeneous box sets to be largely consistent with the support constraint (C2), but hardly consistent in the strongly heterogeneous case.

  Some additional test results can be addressed briefly:
- The standard deviations are in general comparatively low in both the cutting and the packing variant (cf. e.g. Eley 2002). The CLTRS method behaves extremely stable in this sense. The homogenous instances form an exception. With favourable combinations of the dimensions of the containers or of the individual box types, the result there is quite high capacity utilizations, and extremely poor capacity utilizations with unfavourable combinations, which was already made clear in examples.
- With the block building approach, particularly good values are naturally not achieved for stability criterion St1. This is not changed either by the use of general blocks, in particular as these are put together from simple blocks.
- However, with regard to stability criterion St2 CLTRS achieves much better values for the test cases BR1 to BR7 than many comparable methods including the heuristic from Bischoff et al. (1995) (cf. Eley 2002).

Both CLTRS variants are compared with other methods from the literature in Table 4. The only values taken into account are average values for volume utilization (as a percentage of the container volume) or the computing time (in seconds) for the test cases BR1 to BR7 (700 BR instances) and (where present) for the test cases BR1 to BR15 (1500 BR instances). Where known, the cycle frequency of each CPU used is also indicated. Test case BR0 is not taken into account, because up to now there have hardly been any results for other methods.

A fair comparison of methods is only possible if it is also noted whether the methods fulfil constraints (C1) and (C2). This applies in particular for the support constraint (C2) and thus column 2 of Table 4 indicates whether the different methods fulfil the constraint (C2) or not ("y" or "n"). As a consequence, the cutting variant of the CLTRS is to be compared only with the methods from Bortfeldt et al. (2003), Mack et al. (2004) and Parreño et al. (2007), whereas the packing variant should be compared with the other methods listed in Table 4.

The CLTRS method achieves in both variants on the whole a considerably better volume utilization for the Bischoff-Ratcliff instances, with acceptable computing times, than all the respective relevant comparative methods. The CLTRS is apparently just as suitable as a cutting method as it is as a packing method. The dominance with regard to the comparative methods is just as clear with strongly heterogeneous as with weakly heterogeneous instances. The CLTRS is therefore also equally suitable for the 3D-CLP variants SLOPP and SKP.

Table 4: Comparison of the CLTRS method with other 3D-CLP methods

Source / authors / type of method	Constr. (C2) resp.	CPU-frequ. (MHz)	Test cases BR1 – BR7 Volume util. (%)	Test cases BR1 – BR7 CPU time (Sec.)	Test cases BR1 – BR15 Volume util. (%)	Test cases BR1 – BR15 CPU time (Sec.)
Terno et al. (2000), B&B	y	–	88.5	–	–	–
Bortfeldt/Gehring (2001), GA	y	400	90.1	316	88.6	316
Gehring/Bortfeldt (2002), par. GA	y	4 x 400	90.4	183	89.0	183
Eley (2002), TRS	y	200	88.8	600	–	–
Lim et al. (2003), greedy heuristic	n	600	87.6	–	–	–
Bortfeldt et al. (2003), par. TS	n	4 x 2000	92.7	121	–	–
Bischoff (2004), Nelder-Meat Proc.	y	1700	90.5	210	–	–
Mack et al. (2004), par. SA/TS	n	4 x 2000	93.2	222	–	–
Moura/Oliveira (2005), GRASP	y	2400	89.7	34	86.7	69
Parreño et al. (2007), GRASP	n	1500	93.8	302	91.1	101
CLTRS, Packing variant	y	2600	**94.2**	320	**91.9**	320
CLTRS, Cutting variant	n	2600	**95.0**	319	**93.9**	320

Finally, the special variant of the heuristic tree search, PCTRS, used in the CLTRS is to be evaluated. A comparison with the TS method from Bortfeldt et al. (2003) and with the SA/TS hybrid from Mack et al. (2004) lends itself for this purpose. Both methods are based on the traditional block building approach: generated packing plans consist of simple blocks. For the test cases BR1 to BR7 the CLTRS now achieves a mean capacity utilization of 94.7% (cutting variant) right away in the first search stage (simple blocks only); for the TS method and the SA/TS hybrid respectively the corresponding values are 92.7% and 93.2%. The result is that the CLTRS clearly dominates the above-mentioned comparative methods even with a restriction to simple blocks, which, with correspondence of the block types used, is due above all to the special approach of the tree search.

5. Summary

This paper presents the tree search method CLTRS for the three-dimensional container loading problem (3D-CLP). The CLTRS packing variant guarantees the full support from below of all packed boxes, while the cutting variant (and the packing variant as well) observes the guillotine cut constraint that is indispensable for cutting applications. The CLTRS method is based above all on two concepts. On the one hand, the traditional block building approach is generalized, so that blocks with small gaps are now permissible as well as structural elements of packing plans. On the other hand, an (as far as we know) innovative form of tree search, called here partition-controlled tree search (PCTRS), is used to pack blocks. The PCTRS makes the search efficient and diverse, in that it generates solutions economically and at the same time ensures both the required width and the necessary foresight for the tree search.

On the test of the 1600 3D-CLP instances from Bischoff and Ratcliff CLTRS achieves the up to now best volume utilizations for both weakly and for strongly heterogeneous test cases with acceptable computing times and proves to be equally suitable for the problem types SLOPP and SKP. Results were presented as well for instances with homogeneous stocks of boxes as well, i.e. for the up to now rather neglected problem type 3D identical item packing problem (IIPP).

In the future, additional constraints (e.g. weight distribution) are to be implemented in CLTRS. A 2D variant is also planned, as well as the application of CLTRS as a core module of a bin packing method.

References

Bischoff, E.E., M.S.W. Ratcliff. 1995. Issues in the Development of Approaches to Container Loading. *Omega* **23** 377–390.

Bischoff, E.E., F. Janetz, M.S.W. Ratcliff. 1995. Loading Pallets with Non-identical Items. *Eur. J. of Oper. Res.* **84** 681–692.

Bischoff, E.E. 2004. Three dimensional packing of items with limited load bearing strength. *Eur. J. of Oper. Res.*, **168** 952–966.

Bortfeldt, A., H. Gehring. 2001. A Hybrid Genetic Algorithm for the Container Loading Problem. *Eur. J. of Oper. Res.* **131** 143–161.

Bortfeldt, A., H. Gehring, D. Mack. 2003. A Parallel Tabu Search Algorithm for Solving the Container Loading Problem. *Parallel Computing* **29** 641–662.

Davies, A.P., E.E. Bischoff. 1998. Weight Distribution Considerations in Container Loading. European Business Management School, University of Wales, Swansea, Statistics and OR Group, Technical Report.

Dyckhoff, H., U. Finke. 1992. *Cutting and Packing in Production and Distribution*. Heidelberg, Physica-Verlag.

Eley, M. 2002. Solving Container Loading Problems by Block Arrangements. *Eur. J. of Oper. Res.* **141** 393–409.

Fekete, S.P., J. Schepers. 1997. On more-dimensional packing III: Exact algorithms. Technical Report ZPR97-290, Mathematisches Institut, Universität zu Köln.

Gehring, H., A. Bortfeldt. 1997. A Genetic Algorithm for Solving the Container Loading Problem. *Internat. Trans. in Oper. Res.* **4** 401–418.

Gehring, H., A. Bortfeldt. 2002. A Parallel Genetic Algorithm for Solving the Container Loading Problem. *Internat. Trans. in Oper. Res.* **9** 497–511.

Hemminki, J. 1994. Container loading with variable strategies in each layer. Presented at ESI-X, EURO Summer institute, Jouy-En-Josas, France, July 2–15, 1994.

Hifi, M. 2002. Approximate Algorithms for the Container Loading Problem. *Internat. Trans. in Oper. Res.* **9** 747–774.

Lim A., B. Rodrigues, Y. Wang. 2003. A multi-faced buildup algorithm for three-dimensional packing problems. *Omega* **31** 471 – 481.

Mack, D., A. Bortfeldt, H. Gehring. 2004: A parallel hybrid local search algorithm for the container loading problem. *Internat. Trans. in Oper. Res.* **11** 511–533.

Martello, S., D. Pisinger, D. Vigo. 2000. The three-dimensional bin packing problem. *Oper. Res.* **48** 256–267.

Morabito, R., M. Arenales. 1994. An AND/OR-graph Approach to the Container Loading Problem. *Internat. Trans. in Oper. Res.* 1 59–73.

Moura, A., J.F. Oliveira. 2005. A GRASP approach to the Container-Loading Problem. *IEEE Intelligent Systems* **20** 50–57.

Parreño, F., R. Alvarez-Valdes, J.F. Oliveira, J.M. Tamarit. 2007. A maximal-space algorithm for the container loading problem. *INFORMS J. on Computing*, in press.

Pisinger, D. 2002. Heuristics for the Container Loading Problem. *Eur. J. of Oper. Res.* **141** 143–153.

Sixt, M. 1996. *Dreidimensionale Packprobleme. Lösungsverfahren basierend auf den Metaheuristiken Simulated Annealing und Tabu-Suche*. Frankfurt am Main, Peter Lang, Europäischer Verlag der Wissenschaften.

Terno, J., G. Scheithauer, U. Sommerweiß, J. Rieme. 2000. An efficient approach for the multi-pallet loading problem. *Eur. J. of Oper. Res.* **123** 372–381.

Wäscher, G., H. Haussner, H. Schumann. 2007. An Improved Typology of Cutting and Packing Problems. *Eur. J. of Oper. Res.*, **183** 1109–1130.