Computers & Operations Research 39 (2012) 2267–2276

Contents lists available at SciVerse ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

A heuristic block-loading algorithm based on multi-layer search for the container loading problem

Defu Zhang [a], Yu Peng [b,*], Stephen C.H. Leung [c]

[a] *Department of Computer Science, Xiamen University, 361005, China*
[b] *Department of Computer Science, Hong Kong University, Hong Kong*
[c] *Department of Management Sciences, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong*

ARTICLE INFO

Available online 25 October 2011

Keywords:
Container loading problem
Heuristic algorithm
Depth-first search

ABSTRACT

This paper presents an efficient heuristic block-loading algorithm based on multi-layer search for the three-dimensional container loading problem. First, a basic heuristic block-loading algorithm is introduced. This algorithm loads one block, determined by a block selecting algorithm, in one packing phase, according to a fixed strategy, until no blocks are available. Second, the concept of composite block is introduced, the difference between traditional block and composite block being that composite block can contain multiple types of boxes in one block under some restrictions. Third, based on the depth-first search algorithm, a multi-layer search algorithm is developed for determining the selected block in each packing phase, and making this result closer to the optimal solution. Computational results on a classic data set show that the proposed algorithm outperforms the best known algorithm in almost all the test data.

1. Introduction

In logistics and transportation industries, three-dimensional packing problems often occur. Improving the efficiency of packing (space utilization) has become a very important issue as the global economy is becoming ever more competitive. An effective method for solving three-dimensional packing problems is not only of great economic importance in logistics and transportation processes but also has environmental implications since lower fuel consumption helps to reduce pollution.

The container loading problem is a typical three-dimensional packing problem that involves several practical applications having different optimization objectives and loading constraints. Thus there exist many variants of the problem. Dyckhoff and Finke [1] outline the different types of loading problems, and give the corresponding classification, such as a single container loading problem, multi-container loading problem, homogeneous loading problem (containing only one type of box) and heterogeneous loading problems (including many types of boxes). The single container loading problem considered by this paper can be described as follows.

Assume a container (of volume V) and a series of boxes to be packed into the container. The container and the boxes are of cuboid shape. The objective is to determine a feasible loading plan that meets the given loading constraints and maximizes utilization of space in the container, i.e. the filling rate ($S/V \times 100\%$) is as large as possible, where S is the total volume of boxes loaded in the container. A feasible loading plan must meet the following criteria:

(1) Any loaded box cannot overlap with the container, and no two boxes can overlap each other;
(2) The surface of the loaded boxes is parallel to the surface of the container.

When solving real-world container loading problems, one has to consider some practical constraints, such as orientation constraints, loading stability, load bearing strength of boxes, handling constraints, container weight limit and so on [2]. This paper considers the following two constraints:

(C1) Orientation constraint: for certain box types, up to five of the maximum six possible orientations are prohibited.
(C2) Support constraint: in a given packing plan the area of each box not placed on the floor of the container must be supported completely (i.e. 100%) by other boxes.

The remainder of this paper is organized as follows. In the next section an overview of literature is provided. In Section 3 main parts of the proposed algorithm are introduced. In Section 4 computational experiments are described and analyzed. Finally, in Section 5 conclusions are drawn.

* Corresponding author. Tel.: +852 28578465; fax: +852 25598447.
E-mail address: ypeng@cs.hku.hk (Y. Peng).

0305-0548/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2011.10.019

2. Literature review

The container loading problem is a typical NP-hard problem [3], so there is no polynomial-time optimal algorithm for solving it. Exact algorithms often encounter the "combinatorial explosion" phenomenon as the problem size increases. Studies on the exact algorithms show that they solve problems of limited size only. Therefore, heuristic methods become the first choice of theoretical studies and practical applications. George and Robinson [4] first presented a wall-building heuristic algorithm. Bischoff and Marriott [5] compared 14 kinds of layer-based approaches. Based on the idea of plane, Bischoff et al. [6] presented a heuristic algorithm for the heterogeneous loading problem. Constructive algorithms have also been developed by Bischoff and Ratcliff [2] and Bischoff [7]. Lim et al. [8] developed a heuristic algorithm. Juraitis et al. [9] presented a randomized heuristic algorithm. Huang and He [10,11] proposed two effective heuristic algorithms based on the idea of caving degree for a class of container loading problems.

By combining with constructive algorithms, many meta-heuristic algorithms have been proposed. A series of research results have been obtained using tabu search [12], genetic algorithm [13] and hybrid algorithms [14]. In order to improve the performance of metaheuristics, the parallelization technique is used by Gehring and Bortfeldt [15], Bortfeldt et al. [16] and Mack et al. [17]. Based on the idea of free space, Moura and Oliveira [18] proposed a greedy random adaptive search algorithm (GRASP). Parreño et al. [19] further developed GRASP using the concept of maximal space, a nondisjoint representation of free space in a container. Zhang et al. [20] presented a combinational heuristic algorithm that combined personification heuristics and simulated annealing algorithm. Zhang et al. [21] further developed a hybrid simulated annealing algorithm based on the block heuristic idea. Recently, Parreño et al. [22] presented a variable neighborhood search (VNS) algorithm that combines a constructive procedure based on the concept of maximal-space. He and Huang [23] developed a fit degree algorithm by combining the constructive algorithm with local search for the container loading problem.

In addition, incomplete tree search or graph search methods have also been applied successfully to the 3D-CLP. Morabito and Arenales [24] presented a search method based on And/Or graph. Eley [25] designed an algorithm based on the same block. Hifi [26] proposed a tree search method. Pisinger [27] presented an algorithm that first divides the whole container space into several vertical layers, then divides layers into a number of horizontal or vertical strips and then uses an algorithm for the knapsack problem. Bortfeldt and Mack presented a heuristic algorithm that was derived from a branch-and-bound approach [28]. Fanslau and Bortfeldt [29] proposed an effective tree search algorithm based on the idea of composite block, which is the best algorithm in existing literature.

In this paper, a heuristic block-loading algorithm based on multi-layer search is proposed; experimental results show that the proposed algorithm outperforms the existing algorithm in the literature.

3. Heuristic block-loading algorithm based on multi-layer search

3.1. Basic heuristic block-loading algorithm

Heuristic algorithms are preferred because of practical needs and the resultant complexity of the loading problem. A good heuristic algorithm should not only be able to quickly find a solution close to the optimal solution, but also provide the necessary flexibility for different situations and needs. In this paper, a block-loading heuristic algorithm is proposed for solving the container loading problem. This algorithm works as follows. First generate a list of all feasible blocks, and initialize the residual space in the stack to contain the container as the sole residual space, and then start the iteration process. Each iteration takes

```
algorithm BasicHeuristic (isComposite, searchParams, problem)
if isComposite then
  blockTable := GenSimpleBlock (problem.container, problem.box List, problem.num)
else then
  blockTable := GenSimpleBlock (problem.container, problem.box List, problem.num)
endif
set search parameters according to search Params
ps.avail := problem.num
ps.plan := {}
ps.volume = 0
ps.spaceStack := {}
ps.spaceStack.push (problem.container)
while ps.spaceStack ≠ {} do
    space := ps.spaceStack.top ()
    blockList := GenBlock List ( ps.space, ps.avail )
    if blockList ≠ {} then
        block := FindNextBlock ( ps, blockList )
        ps.spaceStack.pop ()
        ps.avail := ps.avail - block.require
        ps.plan := ps.plan + (space, block)
        ps.plan.volume := ps.plan.volume + block.volume
        ps.spaceStack.push (GeResidulSpace(space, block))
    else then
        TransferSpace(space, ps.spaceStack)
    endif
endwhile
return ps.plan
```

Fig. 1. Basic block-loading heuristic algorithm.

one residual space from the top of the stack, and then generates a list of feasible blocks for it; if the list is not empty, then the block selection algorithm is used to determine a block and pack the block into this residual space to generate a loading, and then this space is divided into new spaces which are added to the stack. Otherwise, give up the current residual space, and try to reuse its space the algorithm repeats the loading process until the stack is empty. Where a feasible block denotes a block can be packed into one residual space. A block is an arrangement made up of one or more oriented boxes. A residual space denotes an oriented cuboid space in the container. The idea of this algorithm is somehow similar to the basic heuristic algorithm developed by Bortfeld et al. [16], but it is a more efficient and effective heuristic algorithm.

Fig. 1 gives detailed description of the basic block-loading heuristic algorithm. This algorithm first generates all possible blocks according to the value of input parameter isComposite and initializes the current partial loading plan and then starts the loading process. At each loading stage, the algorithm takes a residual space from the stack, and then uses GenBlockList to generate a list of feasible blocks. If the list is not empty, FindNextBlock algorithm selects a block to load and adds it to the current partial loading plan, and then uses GenResidulSpace algorithm to divide the unfilled space into residual spaces and adds them into the stack. If the list is empty, the algorithm TransferSpace tries to transfer the available part of the residual space to the corresponding space in the stack.

The *space* denotes a cuboid space and is determined by the reference point and its three side dimensions, *problem* denotes a loading problem and is defined by the container, Box list the available boxes and *block* stands for a simple or a composite block and records the number of boxes in this block. Moreover, *block* records one rectangle region that can support other boxes while considering Constraint C2. Since the block may contain gaps, other blocks cannot be loaded if these blocks lose the support of the box. In addition, *block* records the number of combinations that limits the maximum number of composited blocks in order to decrease the complexity of generating the composite block. This algorithm includes four core parts that are introduced in the next sections.

3.2. Generation of block

Simple block is a cuboid generated by boxes of the same type with the same orientation and has no wasted space between boxes. GenSimpleBlock algorithm enumerates all possible combinations of boxes of the same type and the corresponding block is added to the block list. Composite block is a combination of simple blocks and is defined as follows:

(1) A simple block is a basic composite block;
(2) There are two composite blocks *a* and *b*. Composite block *c* can be obtained in *x*-axis, *y*-axis and *z*-axis directions; details can be found in [21,29].

Clearly, according to the above definition, the number of composite blocks will be exponential, and a randomly generated composite block may contain many holes, which are not conducive to loading. Therefore, it is necessary to impose certain restrictions on the composite block, which are as follows:

(1) The composite block size is not larger than the size of the container.
(2) The number of each type of boxes for the composite block is less than the number of the corresponding available boxes.
(3) The composite block may contain holes, but its filling rate is not less than *MinFillRate*.
(4) In order to control the complexity of generating the composite block, the composite number of composite blocks is defined as follows: complexity of a simple block is 0, and complexity of other composite blocks is the complexity of its subblock plus one. The composite number of generated blocks is limited to at most *MaxTimes*.
(5) For composite blocks with the same length of three edges, same boxes and same supported rectangle are considered as equivalence. Repeated equivalent blocks will be ignored.
(6) While considering C2, composite of subblocks in *x*-axis and *y*-axis directions must ensure that supported rectangles of subblocks can be merged into one supported rectangle, namely, the subblocks have the same height and their supported rectangles are adjacent. In addition, the composite of subblocks in *z*-axis direction must meet C2, namely, the above subblock must be placeable on the supported rectangle of the subblock below. Fig. 2 describes the mergence of supported rectangles, where the shaded region denotes the supported rectangles. For example, in Fig. 2(b), composite blocks B and C have the same height, and their supported rectangles are merged into one supported rectangle pqsr.

While meeting the above constraints, the number of blocks is still large, and we limit the block generation algorithm to stop when the number of blocks increases to *MaxBlocks*.

According to the definition of the composite block, Fig. 3 gives a composite block generation algorithm. This algorithm first calls in a simple block generation algorithm to generate all possible simple blocks and then, it iterates *MaxTimes* times, in each iteration, for any two generated composite blocks, and tries to combine them together in *x*-axis, *y*-axis and *z*-axis directions. If *a* and *b* meet the composite conditions described above, then the new generated composite block *c* is added into the block list.

3.3. Generation of feasible block list

GenBlockList is used to generate a list of feasible block from *blockTable* for the current residual space where *blockTable* is a list of all feasible blocks, and is pre-generated before the algorithm begins, in order to avoid the repetitive computation. Therefore, the basic block-loading heuristic algorithm has nothing to do with

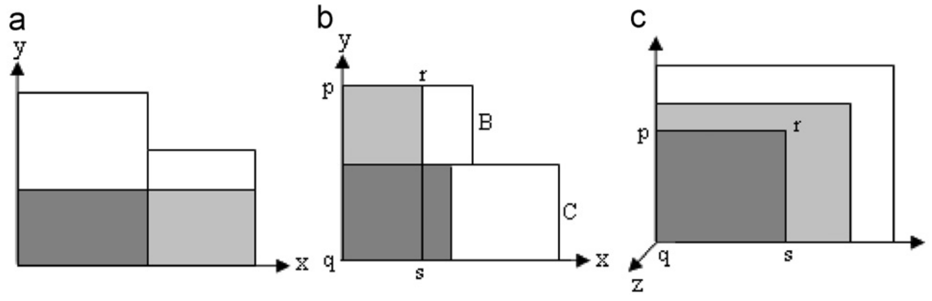

Fig. 2. Mergence of supported rectangles; (a–c) merging in *x*-axis.

```
algorithm GenComplexBlock (container, boxList, num)
blockTable := GenSimpleBlock(space, boxList, num)
for times := 0 to MaxTimes-1 do
    newBlockTable := { }
for each a, b in blockTable do
    if a. times = times or b. times = times then
        if a and b satisfied the constraints in x-direction then
            c := combine a, b in x-direction
            add c to newBlockTable
        endif
        if a and b satisfied the constraints in y-direction then
            c := combine a, b in y-direction
            add c to newBlockTable
        endif
        if a and b satisfied the constraints in z-direction then
            c := combine a, b in z-direction
            add c to newBlockTable
            endif
        endif
endfor
    blockTable := blockTable + newBlockTable
    reduce duplicated block in blockTable
endfor
sort blockTable by decreasing volume of blocks.
return blockTable
```

Fig. 3. Composite block generation algorithm.

```
algorithm GenBlockList (space, avail)
blockList := { }
for each block in blockTable do
    if block.require ≤ avail
        and block.lx ≤ space.lx
        and block.ly ≤ space.ly
        and block.lz ≤ space.lz then
        blockList := blockList + block
    endif
endfor
return blockList
```

Fig. 4. A feasible block list generation algorithm.

GenBlockList, so that GenBlockList can be changed according to the users' demand.

Fig. 4 describes a feasible block list generation algorithm. This algorithm scans *blockTable*, and returns all the blocks that can be placed into the residual space and still have enough residual boxes where blocks in *blockTable* are sorted by the total volume of boxes in blocks in descending order, and a list of feasible blocks returned has the same descending order of boxes' volume.

3.4. Block selection based on multi-layer search

Based on integer decomposition, Fanslau and Bortfeldt [29] presented a tree search algorithm (CLTRS) for selecting one block and obtained excellent results; it is the best algorithm so far. However, after carefully analyzing CLTRS, we found that CLTRS is still worthy to be further improved as

(1) For the specified integer decomposition, CLTRS may select for each stage only the local optimal solution in the current state, and then execute the next stage search. However, the local optimum, after all, is not the global optimum; if there occurs the inferior choice at some stage, the subsequent evaluation may have bias. Although the algorithm can search more states by enumerating all integer decomposition, some better blocks that need deeper search are difficult to be found because of local choice.
(2) During the search process, a large part of computing is actually repeated, not only because different decomposition ways overlap, but also because the node searched by different decompositions is likely to be the same. In these cases, repetitive computing is not necessary.

Based on the above analysis, we propose a multi-layer search approach where one layer is equivalent to an integer decomposition in which the depth of each layer is 1. i-level node is the initial partial loading plan after i blocks is placed. This algorithm is different from the tree search algorithm in that it does not select one block to continue search, but selects the best *MaxHeap* partial loading plans and uses different depths d to call the depth-first search algorithm, and inserts all search results into the results of the corresponding $i+d$ level. In each layer, the algorithm uses the heap to save *MaxHeap* optimal results. Iterative algorithm starts from 0 level until *depth* layers. This algorithm is similar to the tree search, but in each layer *MaxHeap* nodes are extended and the upper nodes that have been computed will not be recalculated.

Fig. 5 describes the multi-layer search algorithm. *depth* describes the number of blocks to be loaded, *maxD* specifies the largest depth to be tried in each layer, *MaxHeap* is the maximum number of optimal partial loading plans that are maintained by the heap in each layer, *effort* denotes the complexity of search and means that depth first search in any stage can at most visit *effort* leaf nodes. DepthFirstSearch is a depth first search algorithm. This algorithm not only records the local optimal solution, but also adds the evaluation results of all leaf nodes to the corresponding layer; its details are given in Fig. 6.

Fig. 7 describes a block placement algorithm. This algorithm combines the block and the current residual space into the current plan and manages the residual space according to Table 1 and Table 2.

Fig. 8 describes the block removal algorithm, which is the inverse process of Fig. 7. This algorithm removes the placement of the current block from the current partial loading plan, recovers the used boxes and original residual space.

Fig. 9(a) and (b) shows two examples of the multi-layer search, where the route and the branch relating to the optimal node are listed, and other details are ignored. Fig. 9(a) shows a typical running result. This algorithm starts from level 0, and executes depth-first search with depths 1 and 2. This algorithm extends the two optimal nodes for level 1, where one node finds the other optimal node in the whole process after another depth-first search with depth 2 being finished. Fig. 9(b) describes a similar

```
algorithm MultiLayerSearch (ps, depth, maxD, MaxHeap, effort)
result.volumeComplete := 0
add ps to heap[0]
for layer := 0 to depth-1 do
    keep heap [layer] containing only the best MaxHeap elements
    for each ps in heap[layer] do
        for d := 1 to maxD do
            branch := max{b | $b^d$ ≤ effort}
            DepthFirstSearch1(ps, d, branch, layer + d)
        endfor
    endfor
endfor
return the maximum in heap [depth]
```

Fig. 5. Multi-layer search algorithm.

instance. The difference is that the optimal node is found after the depth-first search with depth 2 followed by a depth-first search with depth 1.

The multi-layer search is used to design the block selection algorithm. Since optimal *N* solutions in level $k+1$ may come from optimal *N* solutions in level *k*, it is not necessary to execute multi-layer search with the largest depth for all the feasible blocks; we can search in different depths and use the greedy algorithm to filter out some worse feasible blocks. However, *N* needs to be large enough so that the final optimal solution can be included with large probability.

Fig. 10 gives the block selection algorithm, which uses the incremental search depth to execute the multi-layer search; each time half of the worse blocks are filtered out until the number of feasible blocks is not greater than *N*.

```
algorithm DepthFirstSearch(ps, depth, branch, layer)
if depth ≠ 0 then
    space := ps.spaceStack.top()
    blockList := GenBlockList(ps.space, ps.avail)
    if blockList ≠ {}  then
        for i := 0 to branch-1 do
            PlaceBlock(ps, blockList[i])
            DepthFirstSearch(ps, depth-1, branch, layer)
            RemoveBlock(ps, blockList[i], space)
        endfor
    else then
        TransferSpace(space, ps.spaceStack)
        DepthFirstSearch(ps, depth, branch, layer)
        TransferSpaceBack(space, ps.spaceStack)
    endif
else then
    Complete(ps)
    add ps to heap[layer]
endif
```

Fig. 6. An improved depth first search algorithm.

```
algorithm PlaceBlock(ps, block)
space := ps.spaceStack.top()
ps.spaceStack.pop()
ps.avail := ps.avail - block.require
ps.plan := ps.plan + (space, block)
ps.plan.volume := ps.plan.volume + block.volume
ps.spaceStack.push(GenResidulSpace(space, block))
```

Fig. 7. Block placement algorithm.

3.5. Partition and transfer of residual space

At each loading stage, a residual space is loaded either with feasible blocks or without feasible blocks. If there are feasible blocks available for the residual space, the algorithm selects one feasible block according to the block selection algorithm, and then loads this block and divides the residual space into three new spaces and inserts them into the stack. Otherwise, the current residual space is deleted if its partial spaces can be incorporated into the corresponding spaces in the stack, then transfer these partial spaces so that they can be reused.

Figs. 11 and 12 show the partition and transference of the residual space for loading problems with and without C2, respectively. The residual space is divided into three spaces (spaceX, spaceY and spaceZ) according to different cases where spaceX, spaceY and spaceZ denote spaces along *x*-axis, *y*-axis and *z*-axis directions, respectively. It is noted that while considering C2, spaceZ is divided along the placing rectangle ($ax \times ay$ as shown in Fig. 11(a)) in order to guarantee that spaceZ gets adequate support; so there are two partition ways for this case, as shown in Fig. 11. For without C2 case, there are six partition ways as shown in Fig. 12.

As shown in Figs. 11 and 12, the difference between partition ways is spaces that will be transferred in different way. We hope to ensure the integrity of the space as much as possible during the partition process. There are many ways to measure the space

```
algorithm RemoveBlock (ps, block, space)
ps.avail := ps.avail + block.require
ps.plan := ps.plan - (space, block)
ps.plan.volume := ps.plan.volume - block.volume
remove 3 top spaces from ps.spaceStack
ps.spaceStack.push(space)
```

Fig. 8. Block removal algorithm.

Table 1
The partition way of the residual space (with C2).

Partition condition	Partition way	Order of entering stack	Transferring condition	Transferring result
$my \geq mx$	Fig. 11(a)	spaceZ, spaceX, spaceY	spaceY has no feasible block	Fig. 11(b)
$mx \geq my$	Fig. 11(b)	spaceZ, spaceY, spaceX	spaceX has no feasible block	Fig. 11(a)

Table 2
The partition way of the residual space (without C2).

Partition condition	Partition way	Order of inserting stack	Transferring condition	Transferring result
$my \geq mx \geq mz$	Fig. 12(a)	spaceZ, spaceX, spaceY	spaceY has no feasible block	Fig. 12(e)
			spaceX has no feasible block	Fig. 12(c)
$mx \geq my \geq mz$	Fig. 12(b)	spaceZ, spaceY, spaceX	spaceX has no feasible block	Fig. 12(c)
			spaceY has no feasible block	Fig. 12(e)
$my \geq mz \geq mx$	Fig. 12(c)	spaceX, spaceZ, spaceY	spaceY has no feasible block	Fig. 12(f)
			spaceZ has no feasible block	Fig. 12(a)
$mz \geq my \geq mx$	Fig. 12(d)	spaceX, spaceY, spaceZ	spaceZ has no feasible block	Fig. 12(a)
			spaceY has no feasible block	Fig. 12(f)
$mx \geq mz \geq my$	Fig. 12(e)	spaceY, spaceZ, spaceX	spaceX has no feasible block	Fig. 12(d)
			spaceZ has no feasible block	Fig. 12(b)
$mz \geq mx \geq my$	Fig. 12(f)	spaceY,spaceX,spaceZ	spaceZ has no feasible block	Fig. 12(b)

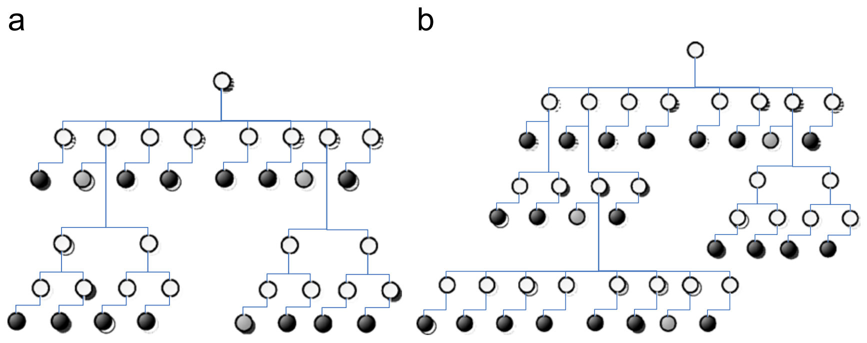

Fig. 9. Multi-layer heuristic search algorithm. (a) *depth*=3, *maxD*=2, *MaxHeap*=2, (b) *depth*=3, *maxD*=2, *MaxHeap*=2.

```
algorithm FindNextBlock(ps, blockList)
for depth := 1 to maxDepth do
    for each block in blockList do
        space := ps.spaceStack.top()
        PlaceBlock(ps, block)
        block.fitness := MultiLayerSearch(ps, depth, maxD, MaxHeap, effort)
        Remove(ps, block, space)
    endfor
    sort the blockList by decreasing fitness
    if size(blockList) > 2N then
        blockList := the first half of blockList
    else then
        blockList := first N in blockList
    endif
endfor
return the block with maximum fitness
```

Fig. 10. Block selection algorithm.

integrity. The strategy in this paper is to make the divided residual space as large as possible. The size of the divided residual space is determined by the residual length *mx*, *my* and *mz* in the *x*-axis, *y*-axis and *z*-axis, after one block is placed into the residual space as shown in Fig. 13, and the transferable space is the space with the largest residual length. GenResidualSpace implements the partition of the unfilled space, the returned residual spaces (spaceX, spaceY, spaceZ) are sorted by non-decreasing of *mx*, *my*, *mz*, and ensures that the space including the transferable space is the last to be inserted into the stack. Tables 1 and 2 show the partition ways of the residual spaces.

As the one with transferable space is the last one added to the stack, once there is no feasible block for it, the transferable space in it can be reused. As shown in Fig. 13, we can observe that re-allocation of the transferable space is actually to recut the unfilled space. Recutting of the unfilled space is implemented by the TransferSpace algorithm. TransferSpace first judges whether one or two residual spaces in the top of the stack come from the same partition as the current space; if that is the case, transferable spaces of the current residual space are transferred to them. The details are described in Tables 1 and 2.

4. Computational results

In order to verify the performance of the heuristic block-loading algorithm based on multi-layer search (HBMLS) for different problems, HBMLS is run on an Intel® Xeon® X5460 @ 3.16GHz. The running and compiling environment is Debian Linux and gcc 4.3.2. Settings of constants in HBMLS are given as follows: $MinFillRate=0.98$, $MaxTimes=5$, $MaxBlocks=10000$, $N=16$, $MaxD=2$, $MaxHeap=6$. The 6 group parameter settings used in our experiment, group i is set as $MaxDepth=i+1$, $effort=3^{i-1}$, $i=1,2,\dots,6$, $Time\ Limit=500$. HBMLS chooses the best results among the tested groups as the final results. When the execution exceeds the *Time Limit*, the next group parameter will not be run. In addition, while the algorithm uses one group parameter to calculate, two algorithms based on simple and composite blocks are calculated in parallel by the two threads and the corresponding results are recorded for comparison.

The test data comes from [2], including 16 classes from BR0 to BR15 each class includes 100 instances. There are 1600 instances in total. These instances can be downloaded from OR-Library [30] or http://59.77.16.8/Download.aspx#p4. The number of types in 16 classes ranges from 1 to 100, heterogeneous, from weak to strong. So it is appropriate to test the performance of loading algorithms where BR0 only contains one kind of box type, which means purely homogeneous loading instances. BR1–BR7 belongs to a weak heterogeneous loading problem, while BR8–BR15 is a strongly heterogeneous loading problem.

4.1. Evaluation of composite block and multi-layer search

In order to verify the effectiveness of composite block and multi-layer search, we run HBMLS for 1600 instances. Tables 3 and 4 report the computational results of HBMLS where *Meann* denotes the average number of boxes, AS denote the algorithm using simple block and AC denotes the algorithm using composite block. *Combine* denote the algorithm combining simple block with composite block. *Compositebetter* denotes the number of times AC performs better than AS. From Tables 3 and 4, we can observe that irrespective of whether we take C2 into account or not, AC is generally better than AS. AC performs better as the heterogeneous instances become strong.

From Tables 3 and 4, the filling rate of different algorithms will decrease as the heterogeneous classes become strong, no matter whether C2 is considered or not. The reason is that search space will become larger as the number of boxes increases so that these algorithms need more time to find a better solution. The advantage of composite block is that it can combine more boxes together in a compact way so that one loading can load more without lossing too much filling rate. Therefore, composite block decreases the size of the solution space without influencing the solution quality, and thus it obtains better results than a simple block, and this advantage increases as the number of types of boxes increase.

On the other hand, when taking into account C2, the filling rate decreases by about 0.5% to 4.0%, because C2 makes many loading cases impossible and loses the loading flexibility. At the same time, this is the reason why algorithms for C2 need shorter running time.

4.2. Comparison with other algorithms

For BR1–BR15, many researchers have tested their algorithms for these instances. The compared algorithms include H_E by Eley [25], a parallel tabu search algorithm (PTSA) by Bortfeldt et al. [16], parallel hybrid algorithm (PHYB) by Mack et al. [17], MFB by Lim

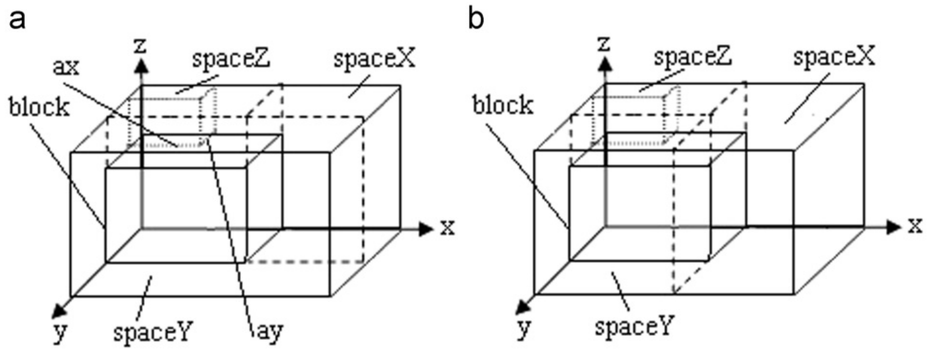

Fig. 11. Partition of the residual space with supported constraint C2.

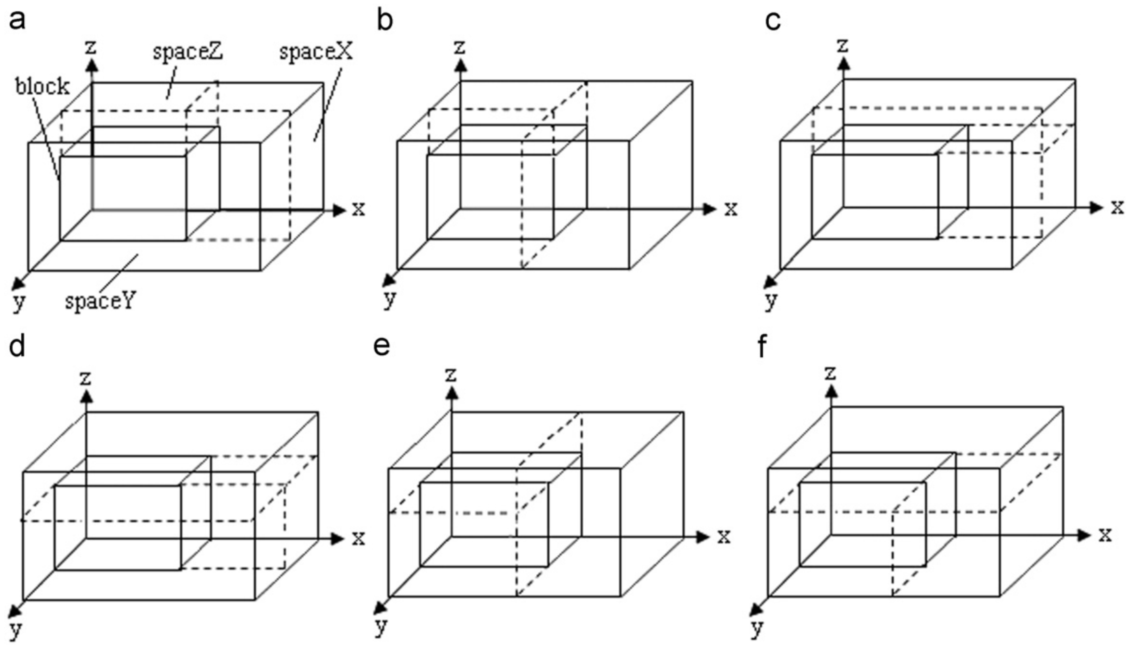

Fig. 12. Partition of the residual space without supported constraint C2.

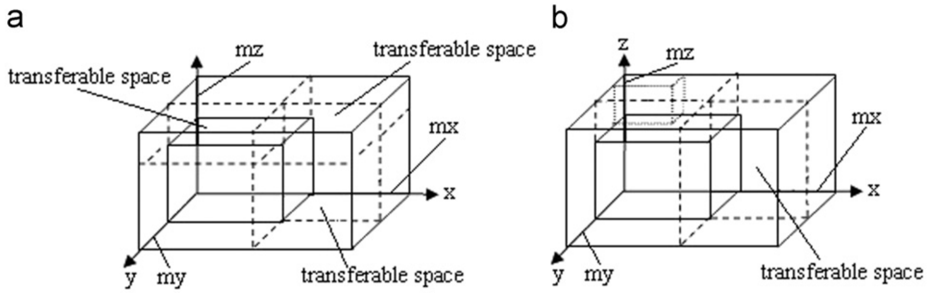

Fig. 13. Transferable space.

et al. [8], random heuristic algorithm (RHA) by Juraitis et al. [9], a new heuristic algorithm (H_B) by Bischoff [7], SPBBL-CC4 by Bortfeldt and Mack [28] and the combination heuristic (CH) [20]. The algorithms mentioned above only considered weak heterogeneous loading problem and only tested classes BR1–BR7. In particular, hybrid genetic algorithms (HGA_GB) by Bortfeldt and Gehring [14] and parallel genetic algorithm (PGA_GB) [15], the greedy random adaptive search algorithm (GRASP1) [18], GRASP2 [19], hybrid simulated annealing algorithm (HSA) [21], variable neighborhood search (VNS) [22], fit degree algorithm (FDA) [23] and tree search algorithm (CLTRS) [29] are included. These algorithms tested BR1–BR15. In addition, A_2 by Huang and He [11] also reports the results of BR8–BR15.

The results of these algorithms mentioned above are directly from the literature and are published after 2000. We obtained a CLTRS program from Bortfeldt and Fanslau [29] and ran it in the same computer as HBMLS. Tables 5 and 6 report the computational results of all the algorithms. All the data in Tables 5 and 6 denote the average filling rate (%) for one class, while Mean denotes the average filling rate (%) for all the classes.

From Table 5, we can observe that no matter whether C2 is considered or not, HBMLS outperforms all the compared algorithms

Table 3
HBMLS (without C2).

class	Box type	*Meann*	Filling rate (AS) (%)	Filling rate (AC) (%)	Filling rate (*Combine*) (%)	*Compositebetter*	Time (s)
BR0	1	206	89.90	89.77	89.95	4	3.38
BR1	3	50	94.87	93.54	94.92	13	14.1
BR2	5	27	95.41	94.47	95.48	26	34.18
BR3	8	17	95.56	95.12	95.69	28	79.43
BR4	10	13	95.38	95.10	95.53	35	115.59
BR5	12	11	95.22	95.08	95.44	52	155.1
BR6	15	9	95.10	95.21	95.38	60	217.57
BR7	20	7	94.69	94.87	95.00	68	327.88
BR8	30	4	94.16	94.60	94.66	71	537.41
BR9	40	3	93.76	94.24	94.30	82	730.33
BR10	50	3	93.38%	94.08	94.11	89	874.59
BR11	60	2	92.87	93.86	93.87	93	1050.7
BR12	70	2	92.59	93.67	93.67	98	1161.61
BR13	80	2	92.25	93.45	93.45	98	1145.13
BR14	90	1	91.84	93.34	93.34	100	1256.03
BR15	100	1	91.53	93.14	93.14	100	1255.71
Mean 1–7	10.43	19.14	95.18	94.77	95.35	40.29	134.84
Mean 8–15	65	2.25	92.80	93.80	93.82	91.38	1001.44
Mean 1–15	39.53	10.13		94.25	94.53	67.53	597.02
			93.91				

Table 4
HBMLS (with C2).

class	Box type	*Meann*	Filling rate (AS) (%)	Filling rate (AC) (%)	Filling rate (*Combine*) (%)	*Compositebetter*	Time (s)
BR0	1	206	89.76	89.69	89.81	6	2.98
BR1	3	50	94.30	93.95	94.43	33	14.71
BR2	5	27	94.74	94.39	94.87	35	36.43
BR3	8	17	94.89	94.67	95.06	48	80.33
BR4	10	13	94.69	94.54	94.89	53	116.13
BR5	12	11	94.53	94.41	94.68	44	153.38
BR6	15	9	94.32	94.25	94.53	56	204.15
BR7	20	7	93.78	93.69	93.96	56	295.69
BR8	30	4	92.88	93.13	93.27	67	454.76
BR9	40	3	92.07	92.54	92.60	75	603.94
BR10	50	3	91.28	92.02	92.05	89	722.46
BR11	60	2	90.48	91.45	91.46	93	842.52
BR12	70	2	89.65	90.91	90.91	95	956.2
BR13	80	2	88.75	90.43	90.43	98	1019.06
BR14	90	1	87.81	89.80	89.80	99	1129.06
BR15	100	1	86.94	89.24	89.24	98	1152.71
Mean 1–7	10.43	19.14	94.47	94.27	94.63	46.43	128.69
Mean 8–15	65	2.25	89.98	91.19	91.22	89.25	860.09
Mean 1–15	39.53	10.13	92.07	92.63	92.81	69.27	518.77

for BR2–BR7. For BR1, HBMLS is a little bit worse than CLTRS. From the average filling rate, HBMLS outperforms all algorithms. In particular, compared to the latest algorithm FDA and the best algorithm CLTRS for BR1–BR7, HBMLS improves performance by 1.82% and 0.34%, respectively, while considering C1. Compared to CLTRS, HBMLS improves performance by 0.44% while considering C1&C2.

From Table 6, we can observe that no matter if C2 is considered or not, HBMLS outperforms all the compared algorithms for BR8–BR15. It shows that HBMLS performs better for strongly heterogeneous loading problems. In particular, compared to the latest algorithm FDA and the best algorithm CLTRS for BR8–BR15, HBMLS improves performance by 1.91% and 0.89%, respectively, while considering C1. Compared to CLTRS, HBMLS improves efficiency by 1.38% while considering C1&C2.

Since different algorithms are run on different computers, it is very difficult to fairly compare the running time of different algorithms. Therefore, we give only a comparison of HBMLS and CLTRS because they are run on the same computer (CPU to 3.16 GHz). For BR1–BR15, while considering C1, the average running time of HBMLS algorithm is 597.02 s, while the average running time of CLTRS is 321.46 s. While considering C1&C2, the average running time of HBMLS is 518.77 s, while CLTRS is 317.55 s.

As the container loading problems are of great practical value, many researchers have conducted their researches on them, and the obtained filling rate is more and more close to the optimum. Therefore, further improving the filling rate has become increasingly difficult. However, when considering C1, C1&C2, compared to the best algorithm CLTRS, HBMLS still obtains the average improvement of 0.63% and 0.94% for BR1–BR15. Experimental results show that HBMLS is very effective for the container loading problem. At the same time, we can also observe that the advantage of the solution, in a sense, can be inherited. This fact is quite logical because a good solution is likely to be extended to another good solution. This feature can be described as follows. Although N optimal solutions in k layer may not be the parent node of N optimal solutions in $k+1$ layer, N optimal solutions in k layer are likely to include all parent nodes of N optimal solutions in $k+1$ layer. This is why HBMLS is better than the best CLTRS. Only selecting one optimal solution may miss the

Table 5
Results of different algorithms for BR1–BR7.

Algorithms	Constraints	BR1	BR2	BR3	BR4	BR5	BR6	BR7	Mean
HGA_BG [14]	C1	87.81	89.4	90.48	90.63	90.73	90.72	90.65	90.06
PGA_GB [15]	C1	88.1	89.56	90.77	91.03	91.23	91.28	91.04	90.43
H_E [25]	C1&C2	88	88.5	89.5	89.3	89	89.2	88	88.79
PHYB [17]	C1	93.41	93.82	94.02	93.68	93.18	92.64	91.68	93.20
MFB [8]	C1	87.4	88.7	89.3	89.7	89.7	89.7	89.4	89.13
RHA [9]	C1	88.4	89.0	89.4	89.5	89.4	89.6	89.5	89.26
H_B [7]	C1&C2	89.39	90.26	91.08	90.90	91.05	90.70	90.44	90.55
SPBBL-CC4 [28]	C1	87.3	88.6	89.4	90.1	89.3	89.7	89.2	89.09
CH [20]	C1	89.94	91.13	92.09	91.94	91.72	91.45	90.94	91.32
GRASP1 [18]	C1	93.52	93.77	93.58	93.05	92.34	91.72	90.55	92.65
GRASP2 [19]	C1	93.85	94.22	94.25	94.09	93.87	93.52	92.94	93.82
HSA [21]	C1&C2	93.81	93.94	93.86	93.57	93.22	92.72	91.99	93.30
VNS [22]	C1	94.93	95.19	94.99	94.71	94.33	94.04	93.53	94.53
FDA [23]	C1	92.92	93.93	93.71	93.68	93.73	93.63	93.14	93.53
CLTRS [29]	C1	**95.05**	95.39	95.45	95.18	94.96	94.80	94.26	95.01
	C1&C2	**94.50**	94.67	94.74	94.41	94.05	93.83	93.15	94.19
HBMLS	C1	94.92	**95.48**	**95.69**	**95.53**	**95.44**	**95.38**	**95.00**	**95.35**
	C1&C2	94.43	**94.87**	**95.06**	**94.89**	**94.68**	**94.53**	**93.96**	**94.63**

Table 6
Results of different algorithms for BR8–BR15.

Algorithms	Constraints	BR8	BR9	BR10	BR11	BR12	BR13	BR14	BR15	Mean
HGA_BG [14]	C1	89.73	89.06	88.4	87.53	86.94	86.25	85.55	85.23	87.34
PGA_GB [15]	C1	90.26	89.5	88.73	87.87	87.18	86.7	85.81	85.48	87.69
GRASP1 [18]	C1	90.26	89.50	88.73	87.87	87.18	86.70	85.81	85.48	87.69
A2 [11]	C1	88.41	88.14	87.9	87.88	87.92	87.92	87.82	87.73	87.97
GRASP2 [19]	C1	91.02	90.46	89.87	89.36	89.03	88.56	88.46	88.36	89.39
HSA [21]	C1&C2	90.56	89.7	89.06	88.18	87.73	86.97	86.16	85.44	87.98
VNS [22]	C1	92.78	92.19	91.92	91.46	91.2	91.11	90.64	90.38	91.46
FDA [23]	C1	92.92	92.49	92.24	91.91	91.83	91.56	91.3	91.02	91.91
CLTRS [29]	C1	93.74	93.51	93.14	92.90	92.79	92.49	92.46	92.42	92.93
	C1&C2	92.27	91.49	90.79	90.02	89.51	88.87	88.19	87.57	89.84
HBMLS	C1	**94.66**	**94.30**	**94.11**	**93.87**	**93.67**	**93.45**	**93.34**	**93.14**	**93.82**
	C1&C2	**93.27**	**92.60**	**92.05**	**91.46**	**90.91**	**90.43**	**89.80**	**89.24**	**91.22**

optimal solution of the next layer, while selecting more than one optimal solution for the next search may include the optimal solution of the next layer with greater probability.

5. Conclusions

This paper proposes a very effective heuristic block-loading algorithm for the container loading problem. This algorithm introduces composite block and gives some limits, and presents an efficient block selection algorithm based on multi-search to evaluate the current state so that more appropriate blocks can be selected for loading in each stage. The multi-layer search algorithm evaluates blocks accuracy. Computational results on 1500 instances show that HBMLS outperforms the current best algorithm for the problems with and without C2. Of course, due to the complexity of loading problems and the inherent defects of heuristic algorithms, HBMLS still has some shortcomings. For example, it needs more computational time for large-scale problems. Therefore, future work is to further optimize it to improve the calculation speed and apply it for different problems with practical applications and additional constraints.

Acknowledgements

The authors would like to thank Prof. Bortfeldt for the CLTRS programs. The authors would like to express their appreciation to the reviewers for making valuable comments and suggestions to the paper. Their comments have improved and enriched the quality of the paper immensely. This work was supported by the National Nature Science Foundation of China (Grant no. 60773126).

References

[1] Dyckhoff H, Finke U. Cutting and packing in production and distribution. Heidelberg: Physica; 1992.
[2] Bischoff EE, Ratcliff BSW. Issues in the development of approaches to container loading. Omega 1995;23:377–90.
[3] Scheithauer G. Algorithms for the container loading problem. In: Operations Research Proceedings 1991, Berlin: Springer,; 1992. pp. 445–452.
[4] George JA, Robinson DF. A heuristic for packing boxes into a container. Computers and Operations Research 1980;7:147–56.
[5] Bischoff EE, Marriott MD. A comparative evaluation of heuristics for container loading. European Journal of Operational Research 1990;44:267–76.
[6] Bischoff EE, Janetz F, Ratcliff MSW. Loading pallets with non-identical items. European Journal of Operational Research 1995;84:681–92.
[7] Bischoff EE. Three-dimensional packing of items with limited load bearing strength. European Journal of Operational Research 2006;168:952–66.
[8] Lim A, Rodrigues B, Yang Y. 3-D container packing heuristics. Applied Intelligence 2005;22:125–34.
[9] Juraitis M, Stonys T, Starinskas A, Jankauskas D, Rubliauskas D. A randomized heuristic for the container loading problem: further investigations. Information Technology and Control 2006;35(1):7–12.
[10] Huang W, He K. A new heuristic algorithm for cuboids packing with no orientation constraints. Computers and Operations Research 2009;36(2):425–32.
[11] Huang W, He K. A caving degree approach for the single container loading problem. European Journal of Operational Research 2009;196:93–101.
[12] Bortfeldt A, Gehring H. A tabu search algorithm for weakly heterogeneous container loading problems. OR Spectrum 1998;20:237–50.

[13] Gehring H, Bortfeldt A. A genetic algorithm for solving the container loading problem. International Transactions in Operational Research 1997;4: 401–18.
[14] Bortfeldt A, Gehring H. A hybrid genetic algorithm for the container loading problem. European Journal of Operational Research 2001;131:143–61.
[15] Gehring H, Bortfeldt A. A parallel genetic algorithm for solving the container loading problem. International Transactions in Operational Research 2002;9:497–511.
[16] Bortfeldt A, Gehring H, Mack D. A parallel tabu search algorithm for solving the container loading problem. Parallel Computing 2003;29:641–62.
[17] Mack D, Bortfeldt A, Gehring H. A parallel hybrid local search algorithm for the container loading problem. International Transactions in Operational Research 2004;11:511–33.
[18] Moura A, Oliveira JF. A GRASP approach to the container-loading problem. IEEE Intelligent Systems 2005;20:50–7.
[19] Parreño F, Alvarez-Valdes R, Oliveira JF, Tamarit JM. A maximal-space algorithm for the container loading problem. INFORMS Journal on Computing 2007;20(3):412–22.
[20] Zhang DF, Wei LJ, Chen QS, Chen HW. A combinational heuristic algorithm for the three-dimensional packing problem. Journal of software 2007;18(9): 2083–9.
[21] Zhang DF, Peng Y, Zhu WX, Chen HW. A hybrid simulated annealing algorithm for the three-dimensional packing problem. Chinese Journal of Computers 2009;32(11):2147–56.
[22] Parreño F, Alvarez-Valdes R, Oliveira JF, Tamarit JM. Neighborhood structures for the container loading problem: a VNS implementation. Journal of Heuristics 2010;16:1–22.
[23] He K, Huang W. An efficient placement heuristic for three-dimensional rectangular packing. Computers & Operations Research 2011;38(1):227–33.
[24] Morabito R, Arenales M. An AND/OR-graph approach to the container loading problem. International Transactions in Operational Research 1994;1:59–73.
[25] Eley M. Solving container loading problems by block arrangement. European Journal of Operational Research 2002;141:393–409.
[26] Hifi M. Approximate algorithms for the container loading problem. International Transactions in Operational Research 2002;9(6):747–74.
[27] Pisinger D. Heuristics for the container loading problem. European Journal of Operational Research 2002;141:143–53.
[28] Bortfeldt A, Mack D. A heuristic for the three-dimensional strip packing problem. European Journal of Operational Research 2007;183(3):1267–79.
[29] Fanslau T, Bortfeldt A. A tree search algorithm for solving the container loading problem. INFORMS Journal on Computing 2010;22:222–35.
[30] OR-Library, 〈http://people.brunel.ac.uk/~mastjjb/jeb/info.html〉.