Computers & Operations Research 43 (2014) 100–107

Contents lists available at ScienceDirect

Computers & Operations Research

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

A beam search approach to the container loading problem

I. Araya *, M.-C. Riff

Departamento de Informática, Universidad Técnica Federico Santa María, Av. España 1680, Valparaíso, Chile

ARTICLE INFO

Available online 16 September 2013

Keywords:
Beam search
Container loading problem
Load planing
Combinatorial optimization
Constructive algorithms

ABSTRACT

The *single container loading problem* is a three-dimensional packing problem in which a container has to be filled with a set of boxes. The objective is to maximize the space utilization of the container. This problem has wide applications in the logistics industry. In this work, a new constructive approach to this problem is introduced. The approach uses a beam search strategy. This strategy can be viewed as a variant of the branch-and-bound search that only expands the most promising nodes at each level of the search tree. The approach is compared with state-of-the-art algorithms using 16 well-known sets of benchmark instances. Results show that the new approach outperforms all the others for each set of instances.

1. Introduction

The single container loading problem (CLP) is a three-dimensional packing problem in which a large parallelepiped or *container* has to be filled with smaller parallelepipeds or *boxes*. The objective is to maximize the total volume of loaded boxes. Depending on the problem, the rotation of boxes may be freely allowed, restricted to certain orientations, or prohibited. Problem instances are commonly classified in *strongly heterogeneous* (many types of boxes) and *weakly heterogeneous* (few types of boxes) [1]. Under the improved typology proposed by Wascher et al. [2], the weakly heterogeneous CLP is classified as a three-dimensional rectangular single large object placement problem (3D SLOOP), while the strongly heterogeneous CLP is classified as a single knapsack problem (3D SKP).

The CLP is NP-hard [3]. Few exact algorithms have been proposed to solve it. Fekete and Schepers [4] present a general framework for multiple dimensional packing problems. Martello et al. [5] develop an exact branch-and-bound method for the CLP. Nowadays, heuristics and meta-heuristics are the only viable alternative to find near-optimal packings when time is crucial. A compacted and not necessarily disjoint classification of these methods is proposed by Zhu et al. [6]. They group packing approaches, according to the way in which the loading plans are generated, in three categories: constructive, divide-and-conquer and local-search. *Constructive* approaches generate solutions by loading boxes into the container until no more boxes can be loaded. Most of the approaches belong to this group. For instance, the heuristic of Bischoff and Ratcliff [7] is a constructive method that fills the container with stacks arranged on the floor of the container. The tree search approach of Eley [8], the tabu search method of Bortfeldt [9], the hybrid simulated annealing and tabu search of Mack et al. [10] and the tree-search-based method of Zhu et al. [6] fill the container with cuboid blocks of boxes. *Divide-and-conquer* methods divide the container into subcontainers. Then, they recursively solve the resultant smaller problems before recombining them into a complete solution. Examples in this category include the works of Chien and Wu [11], Lins et al. [12] and Morabito and Arenales [13]. *Local search* methods start with a complete loading plan, then apply neighborhood operators to generate new loading plans. The works of Gehring and Bortfeldt [14], Parreño et al. [15] and Mack et al. [10] belong to this category.

Nowadays, the most successful approaches for CLP are the block-building-based ones. They are constructive methods which use blocks, instead of boxes, to construct solutions. A block is a subset of boxes that is placed compactly inside its minimum bounding cuboid. They are classified into two types. *Simple blocks* are composed of only one type of boxes. *General blocks* can be composed of multiple types of boxes and/or a set of boxes of the same dimensions placed using different orientations. Stacks and layers of boxes can be seen as particular kinds of blocks. The most successful block-building approaches use incomplete search tree methods to select, at each step, which block to place [4,6,16]. They use general blocks which are generated by combining the existing boxes and blocks in pairs repeatedly.

Zhu et al. [6] propose an analytic framework to describe block-building-based approaches. The framework is based on six common elements present in most of these approaches: (K1) the representation of the free space in the container; (K2) the mechanism to generate blocks; (K3) the heuristic used to rank free spaces; (K4) the heuristic used to rank boxes; (K5) the heuristic used to decide how a selected block is placed in the

* Corresponding author. Tel.: +56 322944516.
E-mail addresses: iaraya@inf.utfsm.cl (I. Araya), mcriff@inf.utfsm.cl (M.-C. Riff).

0305-0548/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cor.2013.09.003

selected free space and (K6) the overall search strategy. They also proposed G2LA, a greedy-based approach, taking key elements from other approaches and implementing a new search-tree-based method for selecting which block to place at each step. When G2LA has been proposed, it outperformed the best single-threaded approaches.

A similar idea is followed in this work. The proposed approach takes most of the key elements from the G2LA algorithm but replacing the greedy-based search strategy by a beam-search-based one. The complexity of the new approach is similar to G2LA.

The remainder of the paper is organized as follows. In Section 2, the single container loading problem is defined, Section 3 describes the new beam-search-based approach, Section 4 reports the computational experiments, Section 5 discuss the relation of our approach to the state-of-art ones and Section 6 concludes the work and gives ideas about the future work.

2. The single container loading problem

The single container loading problem may be stated as follows. Given a container Γ of dimensions L, W, H and a set of boxes $C=\{c_1,c_2,\ldots,c_N\}$, the goal of the problem is to place boxes into the container in order to maximize the total volume of loaded boxes. Boxes do not overlap with any other box and lie completely inside the container with their faces parallel to the container faces.

Formally, the objective may be stated as follows:

$$\max \sum_{i=1}^{N} p_i * V(c_i), \tag{1}$$

where $V(c_i)$ corresponds to the volume of the box c_i and p_i is a boolean variable indicating if the box c_i is placed into the container.

Let $l_0(c_i)$, $w_0(c_i)$ and $h_0(c_i)$ be respectively the length, width and height of a box c_i in its *original orientation*. A box can be placed into the container in any orientation which maintains its faces parallel to the container faces. Let l_i, w_i and h_i be respectively the auxiliary variables indicating the length, width and height of the box arranged in one of its six orientation variants inside the container. The following constraints must be satisfied for all $i=1..N$:

$$\begin{aligned}
&(l_i,w_i,h_i)=(l_0(c_i),w_0(c_i),h_0(c_i))\vee\\
&(l_i,w_i,h_i)=(l_0(c_i),h_0(c_i),w_0(c_i))\vee\\
&(l_i,w_i,h_i)=(w_0(c_i),l_0(c_i),h_0(c_i))\vee\\
&(l_i,w_i,h_i)=(w_0(c_i),h_0(c_i),l_0(c_i))\vee\\
&(l_i,w_i,h_i)=(h_0(c_i),l_0(c_i),w_0(c_i))\vee\\
&(l_i,w_i,h_i)=(h_0(c_i),w_0(c_i),l_0(c_i))
\end{aligned}$$

Additional constraints, taken from the large number of constraints found in practice (see [7]), are also considered:

- *Orientation constraint:* For each box, the number of allowed orientations is restricted (e.g., the top side of a refrigerator must be always on top).
- *Stability constraint (optional):* To guarantee load stability, the bottom sides of all boxes not placed directly on the container floor must be completely supported by the top sides of one or more boxes. When this constraint is imposed, we call the resultant problem variant: *single container loading problem with full support* (CLP-FS)

Let $(x_i,y_i,z_i)\in\{0,1,\ldots,L-l_i\}\times\{0,1,\ldots,W-w_i\}\times\{0,1,\ldots,H-h_i\}$ be the location of the lowest-leftest-and-deepest corner of the box c_i into the container. The following constraint, indicating that a box c_i does not overlap with any other box c_j in the container, must be satisfied for all $i=1..N$:

$$\neg p_i\vee\neg p_j\vee(x_i+l_i\le x_j)\vee(y_i+w_i\le y_j)\vee(z_i+h_i\le z_j)\quad\forall j=1..N, j\neq i \tag{2}$$

3. BSG-CLP: a beam-search-based approach to the CLP

In this section a new block-building approach is introduced. BSG-CLP is a *B*eam-*S*earch-based algorithm that uses a *G*reedy-based function to evaluate states in the search graph. Beam search can be viewed as an adaptation of the branch-and-bound search that expands only a subset of the most promising nodes at each level of the search graph (see [17]).

3.1. Representation

BSG-CLP works by exploring the search space to find a path from some initial state to some terminal state. Intermediate states (or partial solutions) correspond to partial loading plans and consist of three components: a list R of overlapping cuboids which represents the residual space in the container, a list of the remaining boxes C, and a list of blocks B.

A block $b\in B$ is a cuboid containing a set of boxes of C *efficiently placed*, i.e., these boxes use a higher percentage of the cuboid (e.g., $>98\%$). The boxes do not overlap with any other box, satisfy the orientation constraint and lie completely inside b. The block orientation is fixed.

In the initial state, the list R contains only one cuboid: the container Γ; the list C corresponds to the initial list of boxes given by the problem and B is given by a preprocessing method detailed in Section 3.2. A terminal state (or complete solution) corresponds to a loading plan in which no more boxes can be added. In other words, any box in C fits a cuboid in R.

3.1.1. State transition

Consider a nonterminal state consisting of a list of cuboids $R=\{r_1,r_2,\ldots,r_p\}$, a list of remaining boxes C and a list of blocks $B=\{b_1,b_2,\ldots,b_o\}$. The only way in which BSG-CLP moves from one state to another is by loading a block $b_i\in B$ into a free space cuboid $r_j\in R$. Thus, the search space can be represented by a directed acyclic graph (DAG), where the vertices are the states, and the edges are the transitions.

When a block b_i is loaded into r_j, C is updated by removing the boxes in b_i and B is updated by removing *unfeasible blocks*, i.e., blocks consisting of boxes which are no more in C.

In order to update the residual space (R) a more complex procedure is performed. The procedure is briefly explained in the following section.

3.1.2. Representation of the residual space (K1)

We use a *cover representation* [6] for representing the residual space. That is, the residual space is represented by a list of cuboids $R=\{r_1,r_2,\ldots,r_p\}$ which may be overlapped. This representation was proposed by Lim et al. [18] and has been used in several works [19,15,20,21,6,16].

Using the cover representation, when a block b_i is loaded into r_j, three new overlapping cuboids r_{j1},r_{j2},r_{j3} are generated (see Fig. 1). In a similar way, each free space cuboid in R that overlaps with the placed block may generate up to 6 new cuboids. The cuboid r_j and the free space cuboids overlapping b_i are removed from R. Then, all the new generated cuboids $r_{j1},r_{j2},r_{j3},\ldots$ are added into R. Finally, all the non-maximal cuboids are removed from R. More details related to the procedure for updating R can be found in [18,22].

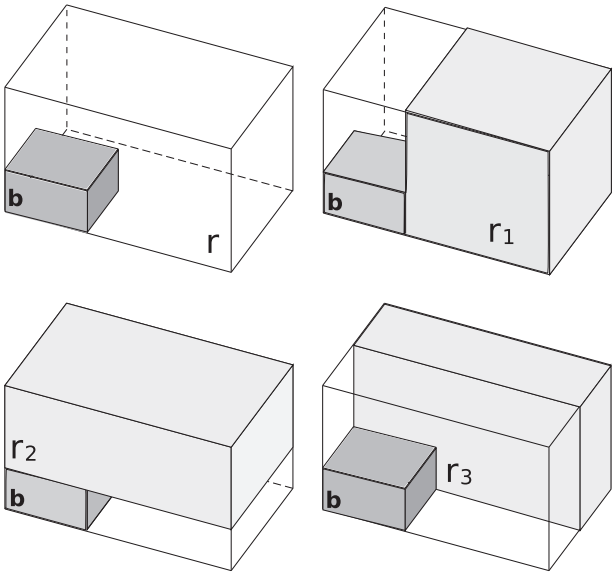

Fig. 1. When a box b is placed into a free space cuboid r, three new overlapping free space cuboids (r_1, r_2 and r_3) are generated.

3.2. The algorithm

Algorithm 1 shows the overall strategy. BSG-CLP gets as input the container Γ, the set of cargo boxes C and two user-defined parameters. These parameters are used by the method `GeneralBlockGeneration` which creates the list of blocks that will be used for constructing solutions. Once the blocks are generated, a beam search is performed several times until a timeout is reached. s_{best} keeps the best complete solution found so far in the search. Besides of the elements Γ, C and B, related to the initial state of the problem, the procedure `BeamSearch` also uses as input the parameter w. This parameter is related with the *search effort*. w limits the number of calls to a greedy procedure which dominates the running time of the beam search: at most, w^2 calls to the greedy are performed at each step. Beginning from 1, w is increased to $\lceil\sqrt{2}w\rceil$ after each execution of the beam search. In this way, an execution of the beam search performs around twice more calls to the greedy than its previous execution. This mechanism is called *double search effort* and has been used in some other constructive approaches [4,6,16].

Algorithm 1. `BSG-CLP` (**in:** Γ, C, *max_fr*, *max_bl*).

$B \leftarrow$ `GeneralBlockGeneration`$(\Gamma, C, max\_fr, max\_bl)$
$w \leftarrow 1$; $s_{best} \leftarrow$ **null**
while ¬`timeout()` **do**
  $s \leftarrow$ `BeamSearch`$(\Gamma, C, B, w, s_{best})$
  $w \leftarrow \lceil\sqrt{2}w\rceil$ // double effort mechanism
end while
return s_{best}

3.2.1. The `GeneralBlockGeneration` procedure (K2)

This procedure constructs *general blocks* by combining the available boxes in C (general blocks consist of one or more types of boxes placed in different orientations). It was proposed by Fanslau and Bortfeldt [4]. First, each box generates at most 6 blocks. These blocks are obtained by all the feasible rotations of the boxes (depending on the orientation constraint). Then, an iterative procedure attempts to combine every block with each other by placing them in contact along the axes. Two blocks are combined only if they use at least *min_fr*% of the volume of the new block. Remark that setting *min_fr* to 100 forces the method to generate almost only simple blocks (i.e., blocks consisting of only one type of boxes).

Blocks with the same dimensions and containing the same boxes are considered just once. The algorithm discards blocks which cannot be constructed using the available boxes and blocks whose dimensions exceed the size of the container Γ. The process stops when *max_bl* blocks are generated or no more different blocks may be generated. A pseudocode of the procedure can be found in [6].

3.2.2. The `BeamSearch` procedure

Algorithm 2 describes our beam search. First, an initial state s_0 is constructed. Note that the list of free space cuboids R has only one element: the container. S keeps the list of states in the current level of the DAG.

At each iteration, the states of the current level of the DAG are expanded. For each state, a subset *succ* of size w of all successors is generated (w^2 if the node corresponds to the root[1]). The successors are put in S', then the w most promising ones, according to an evaluation given by the `greedy` procedure, are preserved for the next level (if $|S| < w$, then all the states are preserved). Roughly speaking, the greedy procedure constructs, for each state s, a temporary complete solution. Then, it returns the total volume reached by this solution. s_{best} keeps the best complete solution found so far by the greedy procedure.

Algorithm 2. `BeamSearch` (**in:** Γ, C, B, w, **in-out:** s_{best}).

$R \leftarrow \{\Gamma\}$
$s_0 \leftarrow$ `initial_state`(R, C, B)
$S \leftarrow \{s_0\}$
while $S \neq \emptyset$ **do**
  List of successors $S' \leftarrow \emptyset$
  for all $s \in S$ **do**
    if $s = s_0$ **then** $succ \leftarrow$ `expand`(s, w^2)
    else $succ \leftarrow$ `expand`(s, w) **end if**
    $S' \leftarrow S' \cup succ$
  end for
  $S \leftarrow w$ states $s \in S'$ maximizing `greedy` (s, s_{best})
end while

The beam search terminates when no more successors can be generated.

3.2.3. Removing similar states

In order to avoid the states converge to similar or symmetrically equivalent solutions a simple mechanism has been implemented. Just before generating the set S of new current states, S' is filtered in order to remove *similar states*. We consider that two states s_1 and s_2 are similar if the solutions constructed by the greedy procedure contain exactly the same boxes (not necessarily in the same position). If this is the case, only the state with the lowest current loaded volume is kept in S'.

In the experiment section we show that this mechanism is crucial for obtaining good results.

3.2.4. The `expand` procedure

The `expand` procedure (Algorithm 3) generates w successors for a given state s. First, a free space cuboid r is chosen. This cuboid minimizes the (*Manhattan*) distance to the corners of the container. Then, each successor of s is obtained by placing one of the w most promising blocks at the corner of r which is closest to a

[1] The first DAG level is forced to have the same number of successor as the rest of the levels of the DAG.

corner of the container (*the anchor corner*). These blocks are selected according to a volume-based heuristic function $f(b,r)$.

Algorithm 3. `Expand` (**in:** $s(R,C,B)$, w).

```
Successors succ←∅
Select the free space r ∈ R that minimizes the Manhattan
    distance
B′← the w blocks in B maximizing f(b,r)
for all b ∈ B′ do
  s′←s
  Update s′ by placing b at the anchor corner of r
  push (succ, s′)
end for
return succ
```

Finally, the `expand` procedure returns the list *succ* with the generated successors of s.

3.2.5. Free space selection: the Manhattan distance (K3, K5)

The Manhattan distance is used for selecting the next free space cuboid to be filled. A cuboid r has eight corners, each of them is associated to the corresponding corner of the container. For instance the lowest, rightest and deepest corner of a cuboid is associated with the lowest, rightest and deepest corner of the container. For each corner of the cuboid at coordinates (x_1,y_1,z_1) and the corresponding container corner at coordinates (x_2,y_2,z_2), the Manhattan distance is $|x_1-x_2|+|y_1-y_2|+|z_1-z_2|$. The corner with the smallest distance corresponds to the *anchor corner* of r. BSG-CLP selects the free space cuboid with the smallest Manhattan distance, with ties broken by greater volume. Eventually, the selected block will be placed at the anchor corner of the selected free space.

3.2.6. Block selection: the function $f(b,r)$ (K4)

The evaluation function $f(b,r)=V(b)-V_{loss}(b,r)$ is used for ranking blocks to be load in a selected free space cuboid r. This function was proposed in [6]. $V(b)$ corresponds to the used volume of a block b and $V_{loss}(b,r)$ is an estimated *wasted volume* in the residual space r of the cuboid. The estimation is based on the fact that the maximum usable space on each axis of a free space cuboid must be a linear combination of the dimension of the boxes. If the block b does not fit the cuboid r, then the function returns $-\infty$.

Consider that the dimensions of b are $l(b)$, $w(b)$ and $h(b)$ and the dimensions of r are $l(r)$, $w(r)$ and $h(r)$. Consider also that l_{max} is the maximum linear combination of the remaining box lengths satisfying the capacity constraint, i.e., $l_{max}<l(r)-l(b)$. Analogously, w_{max} and h_{max} are the maximum linear combination related to the width and height dimensions respectively. Finally, $V_{loss}=V(r)-(l(b)+l_{max})*(w(b)+w_{max})*(h(b)+h_{max})$, where $V(r)$ is the initial volume of the cuboid.

To compute the maximums (l_{max}, w_{max} and h_{max}), the problem is modeled as a knapsack problem. It is solved using the standard knapsack-problem algorithm (KPA) using dynamic programming that runs in pseudo-polynomial time [23]. KPA returns a set of optimal solutions corresponding to all the possible values related to the capacity constraint (i.e., $l_{max}<c$, for all $c=1..L$). KPA is executed only once for each state: the first time that $f(b,r)$ needs to be evaluated. The output is then used to select the w blocks maximizing $f(b,r)$ in the `expand` procedure and also in each iteration of the `greedy` procedure (for selecting the block to place next).

3.2.7. The `greedy` procedure

We introduce the greedy procedure that is used to evaluate states. The procedure computes an underestimation of the maximal total volume of loaded boxes one may reach from a partial solution (or nonterminal state) s. The greedy algorithm is initialized with s and constructs a complete solution s_F by placing iteratively a block b into a free space cuboid r until no more blocks can be loaded. Free space cuboids are selected based on the Manhattan distance while blocks are selected using the evaluation function $f(b,r)$. Finally, the evaluation of s is given by the total volume of the loaded boxes in s_F.

3.3. The key elements of BSG-CLP

Using the analytic framework proposed by Zhu et al. [6], the algorithm may be summarized in terms of the following six key elements:

(K1) the free space is represented using *the cover representation*;
(K2) solutions are constructed using *general blocks*;
(K3) residual spaces are selected based on the *smallest Manhattan distance*;
(K4) blocks are selected using the evaluation function $f(b,r)$;
(K5) blocks are placed at the *anchor corner* of residual spaces;
(K6) the overall search is performed by a *beam-search-based strategy* embedded into a *double search effort* mechanism.

For each element but K6, we have chosen one of the best related procedures according to the work of Zhu et al. [6]. Among them, we believe that K1 and K2 are the most crucial decisions.

The effectiveness of using the cover representation for K1 can be seen by comparing the MS algorithm [15], which uses the cover representation, to the implementation of GRASP using non-overlapped free space cuboids (partition representation) as done by Moura and Oliveira [24]. The main difference between these two approaches is related to the choice of the residual space representation. MS outperformed GRASP in all test cases by significant margins (4.29% on an average). The completeness of the search space provided by the cover representation is likely to be a major contributing factor to the effectiveness of MS. As shown in [4], the construction of general blocks (K2) reduces greatly the search space without losing potential good solutions.

The election of the block evaluation function (K4) is relatively important. Intuitively it is relatively clear that it is preferred to place largest blocks first (ideally the smallest blocks could be placed, at the end, in the small residual spaces between the largest blocks). Zhu et al. [6] showed that using the sophisticated evaluation function $f(b,r)$ the results improve 0.35% on an average w.r.t. the selection of the largest block first. The function $f(b,r)=V(b)-V_{loss}(b,r)$ also depends on the used volume of the block. Finally, elements K3 and K5 were selected by its simplicity and logicality: blocks are placed first near to the corners, then edges, and then faces of the container.

3.4. Handling the full support constraint

In order to adapt BSG-CLP to handle the full support constraint, we must ensure that the following conditions hold during the algorithm execution [16]:

- *F1:* All boxes in blocks are fully supported by boxes or by the base of the block. These blocks are called *fully supported blocks (FSB)*
- *F2:* The bases of all residual space cuboids are covered by the top faces of some placed boxes or the floor of the container. These residual spaces are called *fully supported residual spaces*.
- *F3:* Blocks are placed only on the base of a fully supported residual space.

We implemented the same modifications proposed by Zhu et al. [16] in order to generate fully supported blocks and fully supported residual spaces.

FSBs add a new attribute called *packing area* to the block structure. The packing area of a block is a rectangular region on the top face of the block that is *fully covered* by the top faces of some boxes in the block. Simple blocks are FSBs and its packing area corresponds to the entire top face. Recall that general blocks are generated by combining iteratively two blocks, thus, in order to generate general FSBs only it is needed to know how to create a FSB from two FSBs.

Combining two FSBs along the length or width direction will always result in an FSB. In our approach (as in [4,16]), two FSBs b_1 and b_2 can be combined along the length (resp. width) direction only if:

1. the height of the blocks is the same
2. the length (resp. width) of the packing area spans the entire length (resp. width) of the corresponding block

In this case, the resultant FSB packing area p is given by

$$length(p) = length(p_1) + length(p_2)$$
$$width(p) = min(width(p_1), width(p_2))$$

where p_1 and p_2 are the packing area of blocks b_1 and b_2 respectively.

When two FSBs are combined along the height direction, the block with the smallest base (top block) must fit in the packing area of the bottom block. The packing area of the resultant FSB corresponds to the packing area of the top block.

Related to work with fully supported residual spaces, one additional consideration must be taken. When a block is loaded into a residual space, the new generated residual spaces corresponding to the top face of the placed block must reduce their bases (widths and lengths) by intersecting them with the packing area of the block. Remark that residual spaces corresponding to the other faces (right, left, front and back) are automatically fully supported. Note also that, due to condition F3, no residual spaces corresponding to the bottom face of the block will be generated.

4. Experiments

Experiments were performed on a server PowerEdge T420, with 2 quad-processors Intel Xeon, 2.20 GHz and 8 GB RAM, running Ubuntu Linux. The codes were implemented in C++ and compiled using gcc.

The effectiveness of the approach is analyzed using the 16 classical test cases BR0–BR15 that are commonly employed in literature. BR1–BR7 were generated by Bischoff and Ratcliff [7], while BR0 and BR8–BR15 were generated by Davies and Bischoff [25]. Each set consists of 100 instances. The 16 sets can be classified into three categories: BR0 consists of only one type of boxes (homogeneous); BR1–BR7 consist of a few types of boxes per instance (weakly heterogeneous) and BR8–BR15 consist of up to 100 types of boxes per instance (strongly heterogeneous). All test sets impose a variety of restrictions on the possible orientations for individual boxes.

The two parameters of BSG-CLP are related with the `GeneralBlockGeneration` method. As other approaches [6,16,4], the parameter *max_bl* is set to 10,000. Zhu and Lim [16] observed that using general blocks is more effective for strongly heterogeneous problem instances, whereas using only simple blocks is better for weakly heterogeneous instances. Thus, *max_fr* is set to 98% in strongly heterogeneous instances BR8–BR15 (same as other approaches [6,16,4]) and to 100% for homogeneous and weakly heterogeneous instances BR0–BR7 (this is almost equivalent to only allow the use of simple blocks as in [16]).

Table 1
Comparison between BSG and some variants.

Inst.	BSG	BSG-nr	GG	BSGi	G2LA'	BS-G2LA
BR0	**90.97**	90.34	90.46	90.48	90.91	**90.97**
BR1	95.69	94.50	94.51	94.56	95.45	**95.76**
BR2	96.24	94.94	95.11	95.10	95.70	**96.29**
BR3	96.49	95.12	95.18	95.24	95.90	**96.54**
BR4	96.31	94.94	94.94	95.11	95.64	**96.35**
BR5	96.18	94.73	94.95	94.94	95.46	**96.23**
BR6	96.05	94.62	94.82	94.87	95.32	**96.08**
BR7	95.77	93.99	94.40	94.54	94.98	**95.80**
BR8	**95.33**	93.12	94.09	94.07	95.09	95.19
BR9	**95.07**	92.49	93.79	93.84	94.81	94.98
BR10	**94.97**	91.98	93.63	93.62	94.60	94.78
BR11	**94.80**	91.35	93.42	93.49	94.41	94.59
BR12	**94.64**	90.92	93.30	93.38	94.32	94.44
BR13	**94.59**	90.46	93.22	93.28	94.12	94.37
BR14	**94.49**	89.82	93.08	93.15	94.02	94.31
BR15	**94.37**	89.22	93.09	93.14	93.96	94.25
Average	**95.12**	92.66	93.87	93.93	94.67	95.05

Experiments are mainly related to the CLP without the full support constraint. When the full support constraint is considered it will be indicated.

The experimental results and an executable version of the algorithm can be found online at http://www.inf.utfsm.cl/~iaraya/BSG_CLP2013/.

4.1. Component analysis

The main difference between BSG-CLP and other approaches in the literature corresponds to the use of the beam search strategy. We performed a series of experiments in order to compare beam search with other alternatives and to study the interest of its mechanism for removing similar states.

Table 1 summarizes the experiments. The column BSG corresponds to our BSG-CLP approach. The second column (BSG-nr) corresponds to the algorithm BSG-CLP without the mechanism for removing similar states. The third column (GG) corresponds to our approach restricted to one beam, in other words, it is a Greedy which selects the next state by means of the Greedy-based evaluation. The fourth column (BSGi) is the same as BSG-CLP but using independent beams: in each iteration only the best successor of each state is kept. This is equivalent to perform parallel independent runs of the GG algorithm starting from loading different blocks. The fifth column (G2LA') corresponds to our implementation of the G2LA algorithm proposed in [6]. The only difference w.r.t. BSG-CLP is related to the overall search strategy (K6): G2LA performs a greedy which selects the next state by means of a greedy-based two-step lookahead method (for more details see Section 5 or refer to the original work). The last column (BS-G2LA) corresponds to the algorithm BSG-CLP but using, for selecting states, the greedy-based two-step lookahead method of G2LA instead of the greedy procedure. In all the variants, the double search effort mechanism has been modified accordingly in order to effectively duplicate the effort between one iteration and the next.

The values in the table correspond to the average *volume utilization* obtained by the corresponding strategy for the given set of instances, i.e., the total volume of loaded boxes divided by the volume of the container expressed as a percentage. In all the cases, the execution time was restricted to 500 s per instance.

The removal of similar states is crucial for obtaining competitive results. Note that, on an average, the volume utilization increases almost a 2.5% thanks to this mechanism (compare

Table 2
Comparison between BSG-CLP and the best existing strategies.

Inst.	BSG-CLP			ID-GLTS	G2LA	BRKGA	CLTRS	VNS	FDA
	500 s	150 s	30 s	(2012)	(2012)	(2012)	(2010)	(2010)	(2011)
BR0	**90.97**	90.91	90.88	90.79	90.80	–	89.95	–	–
BR1	**95.69**	95.60	95.39	95.59	95.54	95.28	95.05	94.93	92.92
BR2	**96.24**	96.12	95.88	96.13	95.98	95.90	95.43	95.19	93.93
BR3	**96.49**	96.32	96.08	96.30	96.08	96.13	95.47	94.99	93.71
BR4	**96.31**	96.14	95.84	96.15	95.94	96.01	95.18	94.71	93.68
BR5	**96.18**	96.00	95.73	95.98	95.74	95.84	95.00	94.33	93.73
BR6	**96.05**	95.88	95.56	95.81	95.61	95.72	94.79	94.04	93.63
BR7	**95.77**	95.58	95.24	95.36	95.14	95.29	94.24	93.53	93.14
BR8	**95.33**	95.06	94.53	94.80	94.63	94.76	93.70	92.78	92.92
BR9	**95.07**	94.82	94.37	94.53	94.29	94.34	93.44	92.19	92.49
BR10	**94.97**	94.71	94.14	94.35	94.05	93.86	92.09	91.92	92.24
BR11	**94.80**	94.52	93.99	94.14	93.78	93.60	92.81	91.46	91.91
BR12	**94.64**	94.36	93.86	94.10	93.67	93.22	92.73	91.20	91.83
BR13	**94.59**	94.34	93.77	93.86	93.54	92.99	92.46	91.11	91.56
BR14	**94.49**	94.23	93.60	93.83	93.36	92.68	92.40	90.64	91.30
BR15	**94.37**	94.13	93.58	93.78	93.32	92.46	92.40	90.38	91.02
Av(1–7)	**96.11**	95.95	95.67	95.90	95.72	95.74	95.02	94.53	93.53
Av(8–15)	**94.78**	94.52	93.98	94.17	93.83	93.49	92.88	91.46	91.91
Av(1–15)	**95.40**	95.18	94.77	94.98	94.71	94.54	93.88	92.89	92.67
Time (s)	500	150	30	500	500	147	320	296	633

columns BSG and BSG-nr in the table). We think that the mechanism maintains a minimum level of diversity among the different beams/states that are crucial in this kind of algorithms.

Compared to BSGi and GG, BSG-CLP is better in every instance set. These results highlight the interest of using beam search instead of simply a greedy procedure (GG) or parallel independent runs of a greedy procedure (BSGi). BSG-CLP may be seen as parallel runs of a greedy procedure with information exchange and competition: the best states among *all* the successors survive after each iteration.

Comparing columns BSG and G2LA' we can observe directly the benefits of using the beam search strategy instead of the greedy+ two-step lookahead approach of G2LA. BSG-CLP increases the volume utilization a 0.45% compared to G2LA'.

Finally note in column BS-G2LA that if the beam search uses the selection mechanism of G2LA instead of the greedy procedure, the results are quite similar. In particular, for weakly heterogeneous instances (BR1–BR7) BS-G2LA is slightly better than BSG-CLP, whereas for strongly heterogeneous instances (BR8–BR15), BSG-CLP is slightly better than BS-G2LA. Because of its simplicity, we prefer BSG-CLP.

4.2. Comparison with existing approaches (CLP)

Table 2 summarizes the results of BSG-CLP and the best known existing strategies. The values in the table correspond to the average volume utilization obtained by the corresponding strategy for the given set of instances. Columns 2–4 correspond to the BSG-CLP algorithm with different time limits (500, 150 and 30 s). The results reported in columns 5–10 were extracted from existing literature. Column 5 corresponds to ID-GLTS [16] (Iterative-Doubling Greedy-Lookahead Tree Search Algorithm), the best known approach; column 6 corresponds to G2LA [6] (Greedy 2-Step Lookahead Algorithm). Both, ID-GLTS and G2LA, use simple blocks for instances BR0–BR7 and general blocks for instances BR8–BR15. Column 7 corresponds to BRKGA [19], a parallel Biased Random-Key Genetic Algorithm; column 8 corresponds to CLTRS [4] (Container Loading by Tree Search); column 9 corresponds to VNS [21] (Variable Neighborhood Search) and the last column corresponds to FDA [20], a Fit Degree Algorithm. The "Av" rows show the average performance for some instance sets. Finally, the column Time shows the CPU time taken by each strategy. The best result for each instance set is shown in bold.

Table 3
Statistical comparison between BSG-CLP and ID-GLTS using the Wilcoxon signed-rank test

Wilcoxon signed-rank test	
Positive Ranks	1217
Negative Ranks	285
Z	−26.63
Asymp. Sig.	0.00

Remark that G2LA and ID-GLTS use similar hardware to us [16] (Intel Xeon Quad-Core clocked at 2.27 GHz with 8GB RAM, using CentOS linux).

BSG-CLP has obtained the best overall results. Furthermore, for each of the instance sets, BSG-CLP (500 s) outperforms the results obtained by any other strategy. Compared to the best strategies, note that the average performance of BSG-CLP (limited to 150 s) slightly improves the average performance of ID-GLTS and the average performance of BSG-CLP (limited to only 30 s) is comparable to the average performance of G2LA and better than the average performance of BRKGA.

4.3. Statistical analysis

An statistical comparison between BSG-CLP and ID-GLTS, using the Wilcoxon signed-rank test [26], was performed in order to evaluate the significance of our results.

Both algorithms were run once on each of the 1600 instances and the pair-wise differences were analyzed using the Wilcoxon test. The null hypothesis is that the mean volume utilization reached by both algorithms is equal. The alternative hypothesis is that the mean volume utilization reached by BSG-CLP is greater than the mean volume utilization reached by ID-GLTS.

Table 3 summarizes the results of the test. From the $N=1502$ nonzero differences, in 1217 (81.0%) instances, BSG-CLP outperforms ID-GLTS, while in 285 (19.0%) instances, ID-GLTS outperforms BSG-CLP.

Table 4
Comparison between BSG-CLP and the best existing strategies for the CLP-FS.

Inst.	BSG-CLP		ID-GLTS	BRKGA	CLTRS
	150 s	30 s	(2012)	(2010)	(2010)
BR0	90.26	90.22	**90.29**		89.83
BR1	**94.50**	94.35	94.40	94.34	94.51
BR2	**95.03**	94.84	94.85	94.88	94.73
BR3	**95.17**	94.97	95.10	95.05	94.74
BR4	**94.97**	94.75	94.81	94.75	94.41
BR5	**94.80**	94.57	94.52	94.58	94.13
BR6	**94.65**	94.39	94.33	94.39	93.85
BR7	**94.09**	93.77	93.59	93.75	93.20
BR8	**93.15**	92.71	92.65	92.65	92.26
BR9	**92.53**	92.03	92.11	91.90	91.48
BR10	**92.04**	91.56	91.60	91.28	90.86
BR11	**91.40**	90.89	90.64	90.39	90.11
BR12	**90.92**	90.31	90.35	89.81	89.51
BR13	**90.51**	89.96	89.69	89.27	88.98
BR14	**89.93**	89.34	89.07	88.57	88.26
BR15	**89.33**	88.77	88.36	87.96	87.57
Av(1–7)	**94.74**	94.52	94.51	94.53	94.20
Av(8–15)	**91.22**	90.70	90.56	90.23	89.88
Av(1–15)	**92.87**	92.48	92.40	92.24	91.90
Time (s)	150	30	150	232	320

As the sample is large ($N > 10$), the test statistic is given by $Z = (|W| - N(N+1)/4)/(\sqrt{N(N+1)(2N+1)/24}) = -26.63$, where $W = -116,737$ is computed by adding the negative ranks. The null hypothesis is rejected because $Z < -Z_\alpha = -1.645$ (with a significance level $\alpha = 0.05$). Thus, according to the Wilcoxon test, BSG-CLP outperforms significantly ID-GLTS in the whole set of instances.

Furthermore, from the 10% of the instances with the largest differences in performance, the percentage of positive ranks is 97.5%. Largest differences for BSG-CLP are in the instance sets BR11, BR13, BR14 and BR15 (16–22 cases per instance set).

4.4. Comparison with existing approaches (CLP-FS)

Table 4 summarizes the results of BSG-CLP and the best known existing strategies for the CLP with the full support constraint. The values in the table correspond to the average volume utilization obtained by the corresponding strategy for the given set of instances. Columns 2–3 correspond to the BSG-CLP algorithm with different time limits (150 and 30 s respectively). The results reported in columns 4–6 were extracted from existing literature. Column 4 corresponds to ID-GLTS [16], the best known approach; column 5 corresponds to BRKGA [19] and column 6 corresponds to CLTRS [4]. The "Av" rows show the average performance for some instance sets. Finally, the column Time shows the CPU time taken by each strategy. The best result for each instance set is shown in bold.

BSG-CLP has obtained the best overall results. Furthermore, for each of the instance sets, BSG-CLP (150 s) outperforms the results obtained by any other strategy. Note that the average performance of BSG-CLP limited to 30 s is comparable to the average performance of ID-GLTS and better than the average performance of BRKGA.

5. Related work

Several components of BSG-CLP have been taken from existing approaches. The representation of the free space as a list of overlapping cuboids was first proposed by Lim et al. [18] and has been used in several approaches [19,15,20,21,6,16]. The approaches in [4,6] also use a greedy procedure to evaluate partial solutions. The use of Manhattan distance for selecting free space cuboids was proposed by Zhu and Lim [16]. The heuristic function $f(b,r)$ for ranking blocks is used in [6,16]. The method for generating blocks was proposed by Fanslau and Bortfeldt [4] and was used in [6,16]. The double effort search mechanism is also used in [4,6,16].

G2LA is the most similar approach to BSG-CLP. The main difference is related to the overall search strategy. G2LA executes a sophisticated greedy procedure several times until a time limit is reached. The greedy constructs solutions by iteratively adding a block b into a free space cuboid r. b is selected by using a 2-step-lookahead method. A temporary search tree, limited to two levels, is generated from the current partial solution. Each level of the tree is restricted to w nodes. From the leaves of the tree a standard greedy algorithm is executed generating a set of w^2 complete temporary solutions. The first block of the path leading to the best temporary solution corresponds to b. r is selected using a Manhattan-like heuristic. Once a complete solution is generated by the overall greedy procedure, the value of w is increased in order to duplicate the effort performed by the 2-step-lookahead.

ID-GLTS offers some improvements to the original G2LA algorithm. Basically, it maintains the overall search strategy, but considering two different schemes for selecting free space cuboids. The results reported by ID-GLTS in Tables 2 and 4 correspond to the best solution returned by both schemes, each of them running half of the total CPU time.

Unlike G2LA and ID-GLTS, BSG-CLP is based on beam search. Thus, it is able to explore the space search in several different paths (G2LA and ID-GLTS follows only one main path). BSG-CLP can also be seen as a population-based procedure that maintains in the population the most promising individuals or paths. Note that worse paths are automatically replaced by better ones. G2LA and ID-GLTS are greedy-based algorithms, thus, they have no alternative paths to explore. Only when a solution is completed, they can try a different path.

6. Conclusions

This paper presents a new approach for the single container problem. Beginning with the basic components of the G2LA algorithm [6], the new approach replaces the overall strategy by a beam-search-based one. Limited to only 30 s, BSG-CLP finds comparable solutions to G2LA, executed during 500 s in a similar machine. BSG-CLP also updates the best known results for the problem.

The beam search based strategy maintains a set of different promising paths and not only one as most of the building based approaches. The competition among different states allows our algorithm to quickly discard worse states, opening the search to more promising states. Also a simple mechanism for removing similar states maintains a minimum required level of diversity among the states.

In the experiments we observed that the mechanism for removing similar states is crucial in the beam search. Thus, in a future work we will attempt to improve this mechanism improving, for instance, the way in which two states are compared. We also think that adding randomization to the block/free-space selection may improve the solution quality by increasing the diversity among the states.

Related to the other key elements of block-building-based approaches, we believe that the main improvements could be done to the mechanism to generate blocks. For instance, the fixed parameters (*min_fr* and *max_bl*) could become adaptive. Also it can be interesting to study the construction of more complex general blocks by combining, for instance, three or more sub-blocks.

Acknowledgements

This work was supported by the Fondecyt project 1120781, the Fondecyt project 11121366 and Centro Cientifico Tecnologico de Valparaiso (CCT-Val) FB0821.

References

[1] Bischoff E, Janetz F, Ratcliff M. Loading pallets with non-identical items. European Journal of Operational Research 1995;84(3):681–92.
[2] Wäscher G, Haußner H, Schumann H. An improved typology of cutting and packing problems. European Journal of Operational Research 2007;183(3):1109–30.
[3] Scheithauer G. Algorithms for the container loading problem. Operational Research 1992:445–52.
[4] Fanslau T, Bortfeldt A. A tree search algorithm for solving the container loading problem. INFORMS Journal on Computing 2010;22(2):222–35.
[5] Martello S, Pisinger D, Vigo D. The three-dimensional bin packing problem. Operational Research 2000;48(2):256–67.
[6] Zhu W, Oon W, Lim A, Weng Y. The six elements to block-building approaches for the single container loading problem. Applied Intelligence 2012;37(3):1–15.
[7] Bischoff E, Ratcliff M. Issues in the development of approaches to container loading. Omega - International Journal of Management Science 1995;23(4):377–90.
[8] Eley M. Solving container loading problems by block arrangement. European Journal of Operational Research 2002;141(2):393–409.
[9] Bortfeldt A, Gehring H, Mack D. A parallel tabu search algorithm for solving the container loading problem. Parallel Computing 2003;29(5):641–62.
[10] Mack D, Bortfeldt A, Gehring H. A parallel hybrid local search algorithm for the container loading problem. International Transactions in Operational Research 2004;11(5):511–33.
[11] Chien C, Wu W. A recursive computational procedure for container loading. Computers & Industrial Engineering 1988;35(1):319–22.
[12] Lins L, Lins S, Morabito R. An n-tet graph approach for non-guillotine packings of n-dimensional boxes into an n-container. European Journal of Operational Research 2002;141(2):421–39.
[13] Morabito R, Arenalest M. An and/or-graph approach to the container loading problem. International Transactions in Operational Research 1994;1(1):59–73.
[14] Gehring H, Bortfeldt A. A genetic algorithm for solving the container loading problem. International Transactions in Operational Research 2006;4(5–6):401–418.
[15] Parreño F, Alvarez-Valdes R, Tamarit J, Oliveira J. A maximal-space algorithm for the container loading problem. INFORMS Journal on Computing 2008;20(3):412–22.
[16] Zhu W, Lim A. A new iterative-doubling greedy-lookahead algorithm for the single container loading problem. European Journal of Operational Research 2012;222(3):408–17.
[17] Norvig P. Paradigms of artificial intelligence programming: case studies in Common LISP. Morgan Kaufmann; 1992.
[18] Lim A, Rodrigues B, Wang Y. A multi-faced buildup algorithm for three-dimensional packing problems. Omega - International Journal of Management Science 2003;31(6):471–81.
[19] Gonçalves J, Resende M. A parallel multi-population genetic algorithm for a constrained two-dimensional orthogonal packing problem. Journal of Combinatorial Optimization 2011;22(2):180–201.
[20] He K, Huang W. An efficient placement heuristic for three-dimensional rectangular packing. Computer & Operations Research 2011;38(1):227–33.
[21] Parreño F, Alvarez-Valdes R, Oliveira J, Tamarit J. Neighborhood structures for the container loading problem: a vns implementation. Journal of Heuristics 2010;16(1):1–22.
[22] Neveu B, Trombettoni G, Araya I. Incremental move for 2d strip-packing. In: Proceedings of tools with artificial intelligence, 2007 (ICTAI 2007), vol. 2. IEEE; 2007. p. 489–96.
[23] Martello S, Toth P. Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc; 1990.
[24] Moura A, Oliveira J. A grasp approach to the container-loading problem. IEEE Intelligent Systems 2005;20(4):50–7.
[25] Davies A, Bischoff E. Weight distribution considerations in container loading. European Journal of Operational Research 1999;114(3):509–27.
[26] Wilcoxon F. Individual comparisons by ranking methods. Biometrics Bulletin 1945;1(6):80–3.