European Journal of Operational Research 222 (2012) 408–417

Contents lists available at SciVerse ScienceDirect

European Journal of Operational Research

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

Discrete Optimization

A new iterative-doubling Greedy–Lookahead algorithm for the single container loading problem

Wenbin Zhu [a,*], Andrew Lim [b]

[a] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
[b] Department of Management Sciences, City University of Hong Kong, Tat Chee Ave., Kowloon Tong, Hong Kong

ARTICLE INFO

Article history:
Received 14 December 2011
Accepted 25 April 2012
Available online 14 May 2012

Keywords:
Packing
3D packing
Block building
Tree search

ABSTRACT

The aim of the Single Container Loading Problem (SCLP) is to pack three-dimensional boxes into a three-dimensional container so as to maximize the volume utilization of the container. We propose a new block building approach that constructs packings by placing one block (of boxes) at a time until no more boxes can be loaded. The key to obtaining high quality solutions is to select the right block to place into the right free space cuboid (or *residual space*) in the container. We propose a new heuristic for evaluating the fitness of residual spaces, and use a tree search to decide the best residual space-block pair at each step. The resultant algorithm outperforms the best known algorithms based on the 1600 commonly used benchmark instances even when given fewer computational resources. We also adapted our approach to address the full support constraint. The computational results for the full support support variant on the 1600 instances similarly show a significant improvement over existing techniques even when given substantially fewer computational resources.

1. Introduction

In the Single Container Loading Problem (SCLP), the objective is to orthogonally load a set of three-dimensional cargo boxes (or simply *boxes*) into a three-dimensional rectangular container so as to maximize the volume utilization of the container. The boxes can be classified into K types with different dimensions. There are N_k available boxes of each type k ($k = 1,2,\ldots,K$) and the dimensions of boxes of type k are given by length l_k, width w_k, and height h_k. The length, width and height of the container are L, W and H, respectively. Boxes can only be placed with their faces parallel to the faces of the container (often known as *orthogonal packing* in cutting and packing literature), and any pair of boxes cannot overlap. In general, a box can be placed in up to six orientations depending on how its length, width and height are aligned with the length, width and height of the container. In practice, the rotation of boxes may be freely allowed, restricted to certain orientations (e.g., storage of refrigerators that must be upright), or completely disallowed.

Problem instances may be composed of many types of boxes (known as *strongly heterogeneous* instances), which is a common scenario for courier services; a few types of boxes (*weakly heterogeneous*), such as when batches of goods from a warehouse are sent to retailers; or entirely of one type of box (*homogeneous*), a situation that is prevalent for goods coming off an assembly line. Under an improved typology for cutting and packing problems (Wäscher et al., 2007), the weakly heterogeneous SCLP is classified as a three-dimensional rectangular single large object placement problem (3D SLOPP), while the strongly heterogeneous SCLP is classified as a single knapsack problem (3D SKP).

Due to stability considerations, it is often required that every box in the packing must be fully supported from below. When this constraint is imposed, we call the resultant problem variant the Single Container Loading Problem with Full Support (SCLP-FS). This study considers the SCLP both with and without the full support constraint.

Volume utilization is the most common primary objective for many real-world container loading problems. A denser packing may result in the use of fewer containers or vehicles, which has the environmentally beneficial effect of reducing carbon emissions. Other constraints such as the stability of cargo, multi-drop loads, weight distribution and ease of retrieval are also often considered in specific applications (Bischoff and Ratcliff, 1995; Bischoff, 2006; Ratcliff and Bischoff, 1998; Zhu et al., 2011).

The SCLP is NP-hard in the strict sense (Pisinger, 2002). As expected, exact algorithms can only solve instances of limited size (Fekete et al., 2007). For many real world applications, it is therefore necessary to resort to heuristics, metaheuristics and incomplete tree search based methods in order to produce good solutions in reasonable time.

* Corresponding author. Tel.: +852 64067667; fax: +852 34420188.
E-mail addresses: i@zhuwb.com (W. Zhu), lim.andrew@cityu.edu.hk (A. Lim).

0377-2217/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.ejor.2012.04.036

Existing algorithms for the SCLP can be classified into three (not necessarily disjoint) classes. *Constructive* methods generate solutions by repeatedly loading boxes into the container until no further boxes can be loaded. *Divide-and-conquer* methods instead divide the container into sub-containers, and then recursively solve the resultant smaller problems before recombining them into a complete solution; examples include Lins et al. (2002) and Chien and Wu (1998). Finally, *local search* methods start with an existing solution, then repeatedly apply neighbourhood operators to generate new solutions; examples include Gehring and Bortfeldt (1997) and Parreño et al. (2010).

The best performing approaches in recent literature (Fanslau and Bortfeldt, 2010; Parreño et al., 2010, 2008; He and Huang, 2011) share similar algorithm structures and can be classified as *block building* approaches. Block building approaches are constructive methods. A *block* is a subset of boxes that is placed compactly inside its minimum bounding cuboid. Each step of a block building approach involves placing a block into some free space in the container, and this is repeated until no more blocks can fit into the container. Other block building approaches include the algorithms developed by Eley (2002), Bortfeldt et al. (2003), and Mack et al. (2004). There have also been wall-building approaches (Bortfeldt and Gehring, 2001; Pisinger, 2002), where a container is filled using vertical layers (called walls), and layer-building approaches (Bischoff and Ratcliff, 1995; Terno et al., 2000), where the container is filled from the bottom up using horizontal layers. Both a wall and a horizontal layer can be seen as special cases of a block, so these approaches can also be considered block building approaches.

Zhu et al. (2012) proposed an analytical framework that categorizes the common features of all block building approaches into six key elements: (K1) how to represent free space in the container; (K2) how to generate a list of blocks; (K3) how to select a free space; (K4) how to select a block; (K5) how to place the selected block into the selected space and update the list of free space; and (K6) what is the overarching search strategy. Our new approach is also block building in nature and shares many common features with existing approaches. For K1, our approach uses the *maximal space* representation of free space in the container, which is a set of cuboids that form a cover of the free space. This maximal space representation was first proposed by Lim et al. (2003) and subsequently employed in the maximal space algorithm devised by Parreño et al. (2008, 2010). For K2, we employ the technique developed by Fanslau and Bortfeldt (2010) which first combines boxes of the same type into simple blocks, and then simple blocks into guillotine blocks; the same technique was also used by Zhu et al. (2012). For K4–K6, we use the same techniques as Zhu et al. (2012).

There are three main contributions in this study. Firstly, we propose a new space evaluation heuristic for K3 that is simpler and more effective than existing approaches. Secondly, we show that the proper selection of a free space cuboid (which we call a *residual space*) is as important as the proper selection of blocks. All existing approaches process residual spaces in a fixed order, i.e., the residual space is selected using a fixed rule in each step of the construction, which is equivalent to greedily selecting a space according to some measure (e.g., the largest residual space). In contrast, several blocks are considered and evaluated in each step of the construction, usually using a tree search or similar technique. In this sense, existing approaches have placed a heavier emphasis on selecting blocks compared to selecting residual spaces. We show in this paper that the role of residual spaces and blocks are largely symmetric in the search; by allocating some portion of the computational effort to considering various residual spaces at each step with a corresponding reduction in the effort spent on selecting blocks, we obtained better solutions with the same total computational effort. Thirdly, we adapt our new algorithm to handle the SCLP-FS.

The remainder of this paper is organized as follows. In Section 2, we describe our new block building approach for the SCLP, and then show how it can be adapted to handle the SCLP-FS in Section 3. The two main novelties in our approach are a new residual space selection criterion based on Manhattan distance and considering multiple residual spaces during search; we perform computational experiments to analyze the effects of both novelties and summarize the results in Section 4. Section 5 presents a comparison between our approach and the best existing approaches on 1600 commonly used benchmark instances for both the SCLP and SCLP-FS. We conclude our study in Section 6 with some closing remarks.

2. Our block building approach for the SCLP

In a typical constructive method, a packing is obtained by loading one box at a time until no boxes can be loaded into the container. However, in a block building approach, boxes are first arranged into blocks, and in each step a block of boxes is loaded instead of a single box.

Our approach is a block building approach. We start with an empty container and a list of candidate blocks. In the first step, we select a block and place it at one of the corners of the container. The remaining free space in the container can be represented as a list of cuboids, which we call *residual spaces*. In subsequent steps, we select one of the residual spaces and one of the available blocks, and then place the selected block at some corner of the selected space. Once the block is placed, we update the list of residual spaces and available blocks. This process is repeated until the list of residual spaces is empty, whereupon we have produced a complete packing. We use a tree search procedure to generate several complete packings, and return the best packing found when the time limit is reached.

In any step of our solution construction process, the state of the current partial packing can be fully described by:

R: a list of cuboids (residual spaces) representing the free space in the container.
B: a list of candidate blocks.
PB: a list of placed blocks and their locations.

In this section, we first show how we represent free space as a list of residual spaces in Section 2.1, and then briefly describe how boxes are combined into blocks in Section 2.2. We then explain in Section 2.3 how we perform a *d*-step Lookahead tree search that produces several complete packings and returns the best one. Section 2.4 describes how we evaluate the desirability of a placement (i.e., a block, a residual space, and a corner in that space), which is used in our tree search. Finally, our overall approach is outlined in Section 2.5.

2.1. Maximal space representation of free space

We use the *maximal space* concept to represent the free space in the container. When a single block is placed at a corner of the container, the remaining free space can be represented by three overlapping cuboids, where each cuboid is largest rectangular box that is interior-disjoint with the placed block; such a cuboid is called a residual space. Fig. 1 shows the resultant residual spaces from placing such a block; since the three cuboids overlap, they are illustrated in three separate diagrams (as semi-transparent cuboids) for clarity of presentation.

Due to the overlapping nature of the maximal space representation, when a block is placed at some corner of a residual space, it may overlap with other residual spaces. After a block is placed,

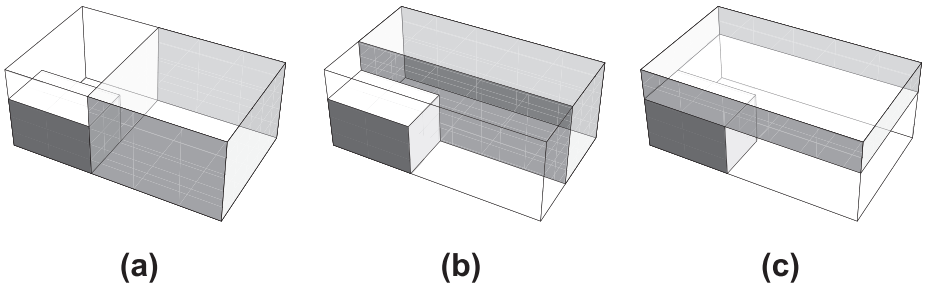


Fig. 1. Three residual spaces that form a cover of the free space.

these overlapped residual spaces must be updated. Fig. 2 illustrates a general case, where a placed block b overlaps with a residual space r. In this case, the remaining free space in r can be represented using the maximal space concept, and the resultant residual spaces can be computed as follows. For each plane that corresponds to a face of the block b, if the plane intersects with r, then it will divide r into two parts, where one part overlaps with b and the other part passes through a residual space. In Fig. 2, the plane corresponding to the right face of b produces the residual space r'. Therefore, for any residual space that overlaps with a placed block b, its remaining free space can be represented by up to six residual spaces corresponding to the six faces of b.

2.2. Block generation

There are two types of blocks that have been employed in existing research. A *simple block* (Fig. 3a) consists of only one type of box, where all boxes are placed in the same orientation. In contrast, a *guillotine block* (Fig. 3b) may consist of multiple types of boxes and/or boxes that are placed in different orientations. A guillotine block is recursively defined: (1) all simple blocks are guillotine blocks and (2) two guillotine blocks that are combined along the length, width or height direction (of the container) is a guillotine block.

It has been observed by Zhu et al. (2012) that using both simple and guillotine blocks is more effective for strongly heterogeneous problem instances, whereas using only simple blocks is better for weakly heterogeneous problem instances. We use the average number of boxes per box type $ht = \sum_k N_k/K$, as an indicator of the heterogeneity of a problem instance. When the average number of boxes per box type is greater than 6, we consider the problem instance weakly heterogeneous, otherwise we consider it strongly heterogeneous.

We employ procedures very similar to those used by Zhu et al. (2012) to generate simple blocks and guillotine blocks. A simple block can be seen as a box replicated nl, nw, and nh times along the length, width and height directions of the container. Hence, simple blocks for a particular box can be generated by enumerating all possible nl, nw, and nh. Initially all guillotine blocks are simple blocks, we then iterative combing two guillotine blocks along length, width and height direction to form larger guillotine blocks. The exact procedures for generating simple block and guillotine blocks are described in Appendices A and B in the online supplements.

For the purpose of finding a solution to the SCLP (without the full support constraint), it is clear that two blocks with the same dimensions that consist of the same set of boxes are equivalent, even if the internal configurations (i.e., relative locations of the boxes) are different. For equivalent blocks, we only keep the first one generated and discard the rest. A block is considered *feasible* only if it can fit into the container and it consists of boxes that are still available for loading, so any block that cannot fit in any residual space or consists of excessive boxes are discarded.

2.3. Greedy d-step Lookahead tree search

Whenever a block is placed into a residual space, it is always placed such that one of its corners coincides with one of the corners of the residual space. Since any block can be potentially placed at any of the eight corners of a residual space, given a state with $|R|$ residual spaces and $|B|$ blocks, there are up to $8|R||B|$ possible placements. A common strategy to determine the best placement for a given state is to use a tree search (Zhu et al., 2012). At the root of the tree, we consider the top m placements according to some fitness measure, resulting in up to m depth one nodes. For each node at depth one, we again try the top m placements, resulting in up to m^2 depth two nodes. We expand the tree up to depth d. For each node at depth d, we use a simple Greedy method to obtain a complete packing (i.e., we repeatedly perform the best placement according to the fitness measure until no more blocks can be loaded). During the tree search, for a given state S, the next state after performing a placement p is obtained by invoking the function PLACE(S, p) (described in Appendix D in the online supplements).

Fig. 4 illustrates a search tree with $m = 3$, $d = 2$. The first placement on the path that leads to the best complete packing (i.e., the volume of the loaded boxes are maximized) is considered to be the best placement for the root node. In Fig. 4, since placement p_2 leads to best complete solution, so p_2 is the best placement for state S. We call such a tree search *d-step Lookahead*.

The tree search described above essentially looks d steps ahead in order to decide the best placement for a given state. We denote

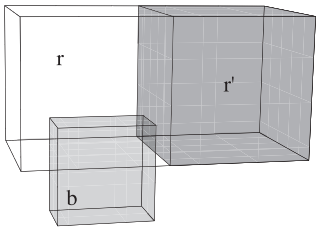


Fig. 2. A placed block b overlaps with residual space r.

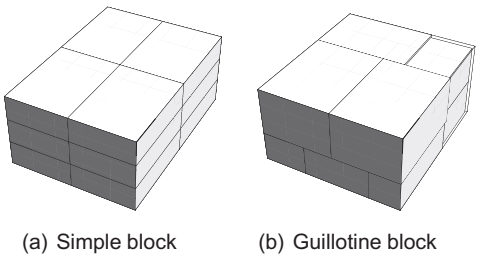


Fig. 3. Classes of blocks.

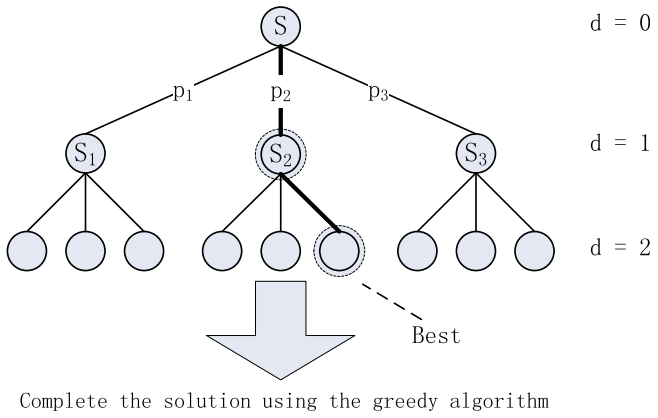


Fig. 4. Two-step Lookahead.

this tree search by the notation LA(S, d, m, Rank), where LA stands for "Lookahead", S is the current state, d is the maximum depth of the tree search, m is the maximum number of subtrees explored at each node of the tree search (also called the *branching factor*), and Rank is a function that ranks all possible placements for the current state S. More precisely, the function Rank (S, m) returns up to m placements out of all possible placements for state S. The tree search procedure LA(S, d, m, Rank) returns a pair (p, CS), where p is the best placement for the current state S, and CS is the best complete solution found during the tree search.

Using the above tree search, we designed a Greedy algorithm called Greedy–Lookahead (Algorithm 1). Starting from the initial state S corresponding to the input SCLP instance, we invoke the tree search procedure LA to find the best placement p. We then move to the next state by performing the selected placement using the function PLACE(S, p), which describe in Appendix D in the online supplements for completeness. We repeat the above process until there are no residual spaces remaining. We record the best solution found using variable *BestSol*; since each invocation of the tree search procedure LA also returns a complete solution CS (line 6), we update the best known solution *BestSol* if CS is superior.

Algorithm 1. Greedy d-Step Lookahead tree search algorithm.

```
   Greedy–Lookahead(B, d, m, Rank)
     // B: a list of blocks
     // d: maximum depth of the tree search
     // m: maximum branching factor of the tree search
     // Rank: a function that ranks all placements of a state
1  BestSol = NULL
2  Create an initial state S, where
     S.R consists of a single free space corresponding to the
   entire container
     S.B consists of all feasible blocks from the list B
3
   while S.R is not empty
4    (p, CS) = LA(S, d, m, Rank)
5    PLACE(S, p)
6    update BestSolif CS is a better solution
7  Update BestSol if S is a better solution
8  return BestSol
```

2.4. Ranking placements

We now describe our placement ranking function, which selects up to m placements out of all possible placements of the state S. Given two integers m_1 and m_2 such that $m_1 \times m_2 \leqslant m$, the function selects up to m_1 residual spaces from $S.R$. For each selected residual space r, up to m_2 blocks that "fit best" into r are chosen, resulting in up to m placements.

We consider two schemes for deciding m_1 and m_2. One simple scheme is to always set $m_1 = 1$ and $m_2 = m$; this is the scheme used by Fanslau and Bortfeldt (2010), which we call the *Single-Best-Space* scheme. An alternative scheme is to set both m_1 and m_2 to about $\sqrt{m}$ for the root node of a search tree (at depth zero), and for all other nodes in the search tree we set $m_1 = 1$ and $m_2 = m$; we call this the *Multi-Best-Space* scheme (given in Algorithm 2).

Algorithm 2. Ranking Placements of a State using Multi-Best-Space Scheme.

```
   Rank-Placement-Multi-Space(S, m)
     // S: a state represent a partial packing
     // m: the maximum number of placements to be
   returned
1  if S corresponds to the root node
2    m1 = max{⌊√m⌋, 1}
3  else m1 = 1
4  m2 = max{⌊m/m1⌋, 1}
5  PlacementList = NULL
6  SpaceList = select the best m1 residual spaces in S.R
7  for each residual space r in SpaceList
8    for the m2 blocks b in S.B that fit best into space r
9      Add placement (r, b, c) to PlacementList
10 return PlacementList
```

It is well known that packing boxes from the corners of the container towards the center tends to lead to better utilization of space, since free space is then collated at the center and tends to be continuous (Parreño et al., 2008; Wu, 2002). Hence, when selecting residual spaces, we prefer to select residual spaces at a corner. Our residual space selection strategy closely resembles that of Zhu et al. (2012) and Parreño et al. (2008) except for two aspects: (1) we may select more than one residual space and (2) we use *Manhattan distance* instead of *corner distance*.

A residual space r has eight corners, and each corner has a corresponding corner of the container. Fig. 5 illustrates eight pairs of corresponding corners: (A_1, B_1), (A_2, B_2),..., (A_8, B_8). For each pair of corners, we compute the Manhattan distance, i.e., if (x_1, y_1, z_1) are the coordinates of A_1 and (x_2, y_2, z_2) are the coordinates of B_1, the Manhattan distance is $|x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$. We call the corner of r with the minimum Manhattan distance the *anchor corner*, and the distance between the anchor corner and its corresponding corner of the container is the *anchor distance*; in Fig. 5, A_2 is the anchor corner. When selecting residual spaces, we prefer the space with the smallest anchor distance, with ties broken by greater volume. If there is still a tie, we prefer the space with smaller lexicographical order of $(y_1, z_1, y_2, z_2, x_1, x_2)$, where (x_1, y_1, z_1) are

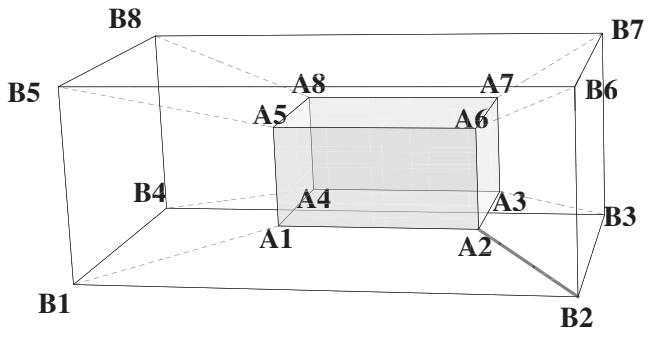


Fig. 5. Corner distances between a residual space and the container.

the coordinates of the corner closest to the origin, and (x_2, y_2, z_2) correspond to the corner farthest from the origin (the container is assumed to be placed with length, width and height aligned with X-, Y-, and Z-axes, respectively).

For each selected residual space, we select up to m_2 blocks from $S.B$ using exactly the same strategy as Zhu et al. (2012). The fitness of a block b with respect to a given residual space r is defined as $f(r, b) = V + \alpha \cdot (V_{\text{loss}} + V_{\text{waste}})$, where V is the total *volume* of the boxes in the block, V_{loss} is the *wasted space volume* after the block b is placed into r, V_{waste} is the *wasted block volume* in block b (i.e., the volume of the block minus the total volume of boxes in the block), and α is a coefficient that controls the relative weight of V and $(V_{\text{loss}} + V_{\text{waste}})$. Following the suggestion made by Zhu et al. (2012), we set $\alpha = -1$.

The value of V_{loss} is computed as follows. Recall that after a block is placed at a corner of a residual space r, the remaining space in r is represented by at most three cuboids (Fig. 1). Given such a cuboid and the set of available boxes, the maximum usable space on each axis must be a linear combination of the dimensions of the boxes. If the magnitude of a dimension of a cuboid is not a valid linear combination of box dimensions, we can "trim" it to the largest linear combination. To compute the largest linear combination subject to the availability of boxes is equivalent to solving a knapsack problem; we make use of the standard knapsack problem algorithm using dynamic programming (DP) that runs in pseudo-polynomial time (Martello and Toth, 1990) for this purpose. We compute V_{loss} as the total amount of wasted space trimmed from the three cuboids as measured by this technique. However, we do not perform the DP during the Greedy procedure that completes a partial solution represented by a node at depth d. The details are explained in Appendix C in the online supplements.

2.5. Overall approach

The parameters d and m in Greedy–Lookahead(B, d, m, Rank) control the size of the search tree explored by the LA(S, d, m, Rank) function, which is proportional to $w = m^d$. The total computational effort by Greedy–Lookahead(B, d, m, Rank) is also approximately proportional to w, although invoking Greedy–Lookahead with a larger w value may not always take more CPU time because the number of tree searches carried out is affected by the selected placements. Hence, our overall approach uses a simple strategy that starts with computational effort $w = 1$, and doubles the effort in subsequent iterations until all allocated CPU time is used.

Algorithm 3. The iterative-doubling Greedy–Lookahead tree search algorithm.

```
ID-GLTS(d, Rank)
  // d: maximum depth of tree search
  // Rank: a function that ranks placements of a state
1  BestSol = NULL
2  avgBoxPerType = ∑_k N_k / K
3  if avgBoxPerType > 6
4    B = Generate-Simple-Blocks(MaxBlockCount)
5  else B = Generate-Guillotine-Blocks(MinUtil,
     MaxBlockCount)
6  w = 1
7  while time limit is not exceeded
8    m = ⌊∛w⌋
9    sol = Greedy–Lookahead(B, d, m, Rank)
10   update BestSol if sol is a better solution
11   w = 2 · w
12 return BestSol
```

Our resultant *Iterative-Doubling Greedy–Lookahead Tree Search* (ID-GLTS) approach is given in Algorithm 3. In our implementation, we set the Lookahead depth $d = 2$. Note that early in the algorithm when the values of w are small, multiple redundant searches using the same value of m would be performed. To avoid this, our actual implementation is slightly more complex, which we explain in Appendix E in the online supplements.

Using the analytic framework proposed by Zhu et al. (2012), the six key elements for our approach are as follows:

(K1) use the **cover representation** to represent free space;
(K2) construct **simple blocks** for weakly heterogeneous problem instances, and both **simple and guillotine blocks** for strongly heterogeneous problem instances
(K3) select the residual space with **smallest Manhattan distance**;
(K4) select the block with highest $f(r, b)$ **evaluation function** value
(K5) place the block at the **anchor corner** of the residual space;
(K6) using the **Greedy** d**-step Lookahead** search algorithm, **double search effort** in each iteration until the time limit exceeded.

3. Handling the full support constraint

In order to adapt our approach to handle the SCLP-FS, we only need to ensure that the following conditions hold throughout the course of our algorithm:

- *FS1*: All boxes in blocks are fully supported from below by boxes or by the base of the block; such blocks are called *fully supported blocks*
- *FS2*: The bases of all residual spaces are fully covered by the top faces of some placed boxes or the floor of the container; such spaces are called *fully supported residual spaces*
- *FS3*: Blocks are placed only on the base of a fully supported residual space (i.e., one of the bottom four corners of the space)

When these conditions hold, the placement of a block will not violate the full support constraint. Since the initial state (the empty container with no boxes) represents a fully supported packing plan, so by induction all states reached from the initial state satisfying these conditions represents a fully supported packing plan.

3.1. Generating fully supported blocks

A *fully supported block* (FSB) is a block where all boxes in the block are fully supported from below by either other boxes or the base of the block. We associate each FSB with a new attribute *packing area*, which is a rectangular region on the top face of the block that is fully covered by the top faces of boxes in that block.

Clearly, any simple block is an FSB and its packing area is the entire top face. FSBs are generated by iteratively combining two FSBs along the length, width or height direction of the container. We use the same technique as Fanslau and Bortfeldt (2010) to combine two FSBs to generate a new FSB. Combining two FSBs along the length or width direction will always result in an FSB since no overhanging boxes will be introduced. Fig. 6 how two FSBs can be combined along the length direction.

In our approach, two FSBs b_1 and b_2 can be combined along the length (resp. width) direction of the container only if:

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

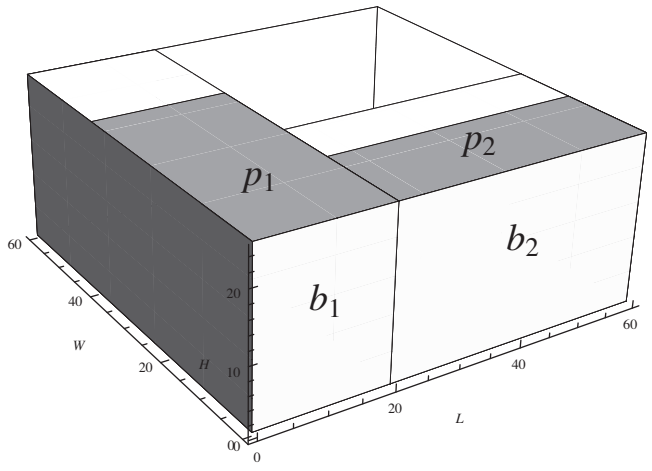


Fig. 6. Combining two fully supported blocks along the length direction.

(3) the resultant block is feasible and has volume utilization no less than *MinUtil*.

When two FSBs b_1 and b_2 with packing areas p_1 and p_2, respectively, are combined along the length direction, the packing area p of the resultant block is given by

$$p.length = p_1.length + p_2.length \quad (1)$$

$$p.width = \min\{p_1.width, p_2.width\} \quad (2)$$

When two FSBs are combined along the height direction, we avoid introducing new overhanging boxes by placing the top block inside the packing area of the bottom block. Hence, two FSBs can be combined along the height direction only if the base area of the top block can contained by the packing area of the bottom block (Fig. 7). The packing area of the resultant block is the packing area of the top block.

3.2. Generating fully supported residual spaces

We extend the concept of a residual space to a *fully supported residual space*. A fully supported residual space is a residual space

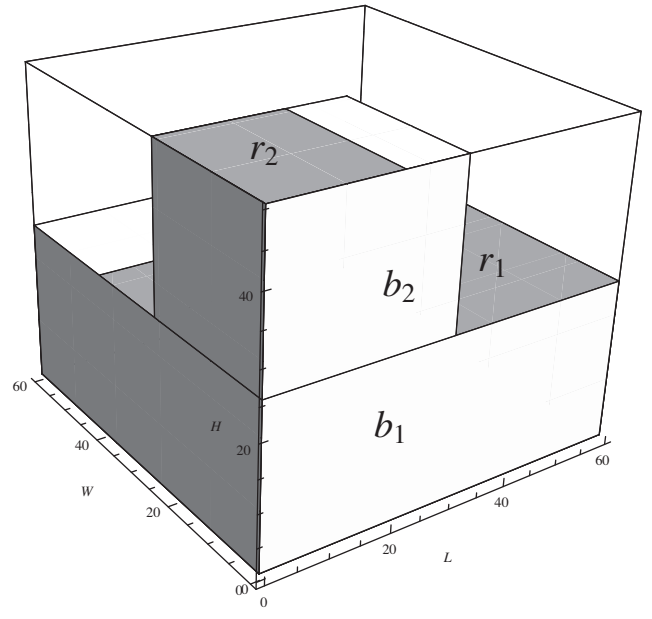


Fig. 7. Combining two fully supported blocks along the height direction.

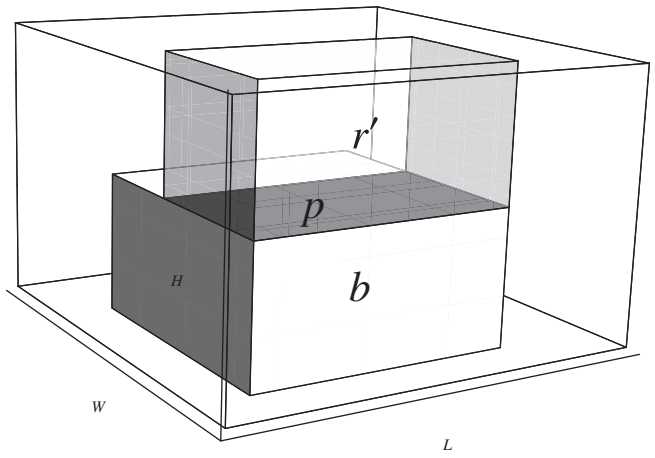


Fig. 8. r' is the residual space corresponding to the top face of block b.

where its base area is fully covered by either the floor of the container or the top faces of some placed boxes.

Recall that for the SCLP without the full support constraint, when a placed block b overlaps with a residual space r, the remaining free space in r can be represented by up to six new residual spaces, each of which corresponds to a face of the placed block. In the case of SCLP-FS, when a placed FSB overlaps with a fully supported residual space, the four spaces corresponding to the left, right, front and back faces of the block are automatically fully supported (we can easily verify that their base areas are fully supported). No residual space corresponding to the bottom face of the block will be generated due to the placement restriction *FS3*. Finally, the residual space corresponding to the top face of the placed block will be fully supported if we use the packing area of the block as its base. Fig. 8 shows a fully supported block b that overlaps with a fully supported residual space. The residual space corresponding to the the top face of block b is r', the base area of r' is exactly the packing area p of block b.

4. Component analysis via computational experiments

There are two main components that differentiate our Iterative-Doubling Greedy–Lookahead Tree Search (ID-GLTS) approach from other approaches in literature. Firstly, we use the Manhattan distance measure to rank residual spaces. Secondly, we may consider several residual spaces when choosing the best placement; all existing techniques select a single residual space and only consider different blocks. We performed a series of experiments to analyze the effects of these two components in our approach, as well as to determine appropriate parameter values.

Our experiments were based on the 16 sets of test cases (BR0–BR15) that are commonly employed by existing SCLP literature. BR1–BR7 were generated by Bischoff and Ratcliff (1995), while BR0 and BR8–BR15 were generated by Davies and Bischoff (1999). Each set consists of 100 instances. The 16 sets of instances can be broadly classified into three categories: BR0 consists of only one type of box (homogeneous); BR1–7 consists of a few types of boxes per instance (weakly heterogeneous); and BR8–BR15 consists of up to 100 types of boxes per instance (strongly heterogeneous). All of the test sets also impose a variety of restrictions on the possible orientations for individual boxes, which may be different for different boxes in the same test instance.

For experiments presented in this section, we selected 160 instances (the first ten instances from each set) to form a smaller test bed. We used this smaller test bed instead of the complete test sets for two reasons. Firstly, this is a common and successful practice in

the artificial intelligence community where smaller test sets are used to train an algorithm, and its performance on larger test sets are used to compare the algorithm with other approaches. This avoids over-fitting the algorithm for the specific test data (an over-fitted algorithm may not work well for other data). Secondly, it allows a broader range of experiments to be conducted since each experiment takes much less time to complete compared to using the complete test sets.

All experiments were conducted on a rack mounted server with Intel Xeon E5520 Quad-Core CPUs clocked at 2.27 GHz with 8G RAM. The operating system is CentOS linux version 5. The 64-bit Java Development Kit 1.6.0 from Sun Microsystems was used to implement the algorithms. Since our algorithm is fully deterministic, we execute our algorithm only once for each problem instance.

The values for the following parameters in our ID-GLTS approach were fixed as follows:

- $ht = 6$: for each instance, if the average box count per type is larger than ht, it is considered weakly heterogeneous; otherwise it is considered strongly heterogeneous.
- *MaxBlockCount* = 10,000: the maximum number of blocks generated.
- *MinUtil* = 98%: the utilization of a block is defined as the total volume of the boxes in the block divided by the total volume of the block; only blocks with utilization not lower than *MinUtil* will be generated.
- $d = 2$: the maximum depth of the search tree in each invocation of the LA function.
- *TimeLimit* = 500*s*: the total time limit in CPU seconds for each instance.

4.1. Manhattan distance vs. corner distance

The first set of experiments aims to evaluate the effectiveness of using Manhattan distance to rank residual spaces, compared to the *corner distance* measure used by Zhu et al. (2012). We implemented two versions of our approach, one using Manhattan distance and the other using corner distance; the two versions are otherwise identical. We chose the Single-Best-Space scheme for ranking placements.

Table 1 summarizes the results for these two algorithms for the SCLP. Columns *Manhattan* and *Corner* give the average volume utilization of the solutions obtained by the corresponding algorithms for the 10 instances for each test set, i.e., the total volume of boxes loaded divided by the volume of the container expressed as a percentage. Column *Impr* is the difference between columns *Manhattan* and *Corner*, where a positive value indicates that using Manhattan distance resulted in better solutions than corner distance. The last three rows of the table provides an overview of the algorithms' performance. Row *Avg BR1–7* shows the average performance for the weakly heterogeneous instances, while row *Avg BR8–15* represents the average performance for the strongly heterogeneous instances.

We see that for all weakly heterogeneous instances, the difference in performance between the Manhattan distance and corner distance measures is small; the average difference is only 0.01%, and the corner distance measure achieved superior solutions on average than Manhattan distance for 3 out of 7 test sets (highlighted in bold). However, the Manhattan distance measure is superior for strongly heterogeneous instances. Although the overall improvement for strongly heterogeneous instances of 0.27% may seem small, there are two reasons to prefer using Manhattan distance: firstly, it is simpler to compute than corner distance; secondly, heterogeneous instances are well known to be much harder to solve than weakly heterogeneous instances.

Table 1
Manhattan distance vs. corner distance on 160 BR instances (SCLP).

Test set	Manhattan	Corner	Impr
BR0	87.43	87.43	0.00
BR1	94.97	94.95	0.01
BR2	96.17	96.22	**−0.05**
BR3	96.26	96.11	0.15
BR4	95.97	96.01	**−0.04**
BR5	95.86	95.88	**−0.02**
BR6	95.66	95.64	0.02
BR7	95.26	95.24	0.02
BR8	94.88	94.51	0.37
BR9	94.37	94.23	0.14
BR10	94.52	93.99	0.53
BR11	94.09	93.93	0.16
BR12	93.90	93.67	0.23
BR13	93.83	93.66	0.18
BR14	93.68	93.31	0.37
BR15	93.66	93.50	0.15
Avg BR1–7	95.73	95.72	0.01
Avg BR8–15	94.12	93.85	0.27
Avg BR1–15	94.87	94.72	0.15

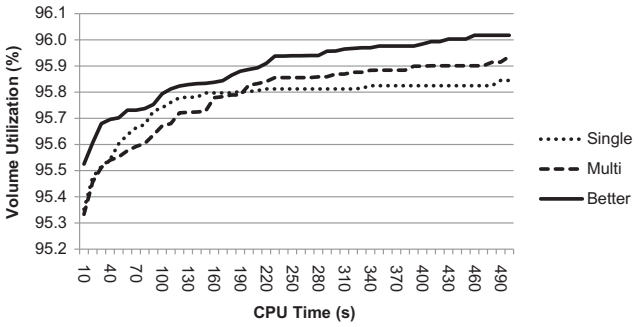
4.2. Single- vs. Multi-Best-Space

The second set of experiments studies the effectiveness of the *Single-Best-Space* and *Multi-Best-Space* strategies for ranking placements. We implemented three versions of our algorithm (all using Manhattan distance to rank residual spaces): the first uses the Single-Best-Space scheme; the second uses the Multi-Best-Space scheme; while for the third we allocate half the CPU time each on separate executions of Single-Best-Space and Multi-Best-Space, and then take the better solution. All other aspects of the three algorithms are the same.

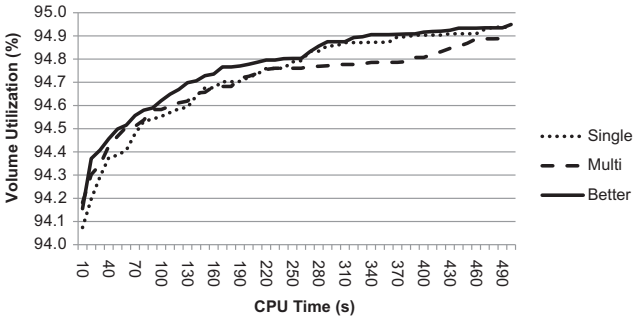
Our experiments showed that on average, Single-Best-Space is slightly better for strongly heterogeneous instances, Multi-Best-Space is slightly better for weakly heterogeneous instances, and the overall performance of the two algorithms are almost the same. We also observed that the solutions found by Single-Best-Space and Multi-Best-Space tend to complement each other for many individual instances, i.e., for some instances Single-Best-Space found solutions that are much better than Multi-Best-Space, whereas for other instances the reverse is true. As a result, allocating half of the computational resources to the Single-Best-Space scheme and the other half to the Mult-Best-Space scheme is more effective than allocating all resources to either scheme alone. The detailed results can be found in Appendix F in the online supplements.

We performed another set of experiments to analyze the convergency behavior of the three space selection strategies. For each instance, we recorded the best solution found by each algorithm every 10 CPU seconds. We grouped the 15 test sets BR1–15 into three categories (BR1–5, BR6–10, and BR11–15) ordered by increasing heterogeneity. Fig. 9 shows the results for these groups in three separate diagrams. In each diagram, the vertical axis is the percentage volume utilization of the solutions, and the horizontal axis is the time taken to produce the solutions in CPU seconds. The diagrams plot the average of the best solutions for the 50 instances over time. The line with label *Single*, *Multi* and *better* represents the ID-G2LA, ID-G2LA-MS and ID-G2LA-S+M respectively.

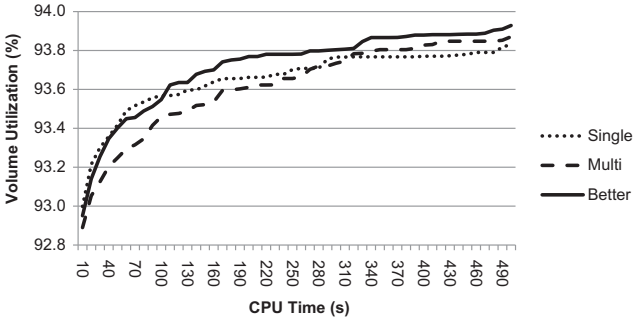
We see in Fig. 9a that the Single-Best-Space scheme found better solutions than the Multi-Best-Space scheme for the weakly heterogeneous instances BR1–5 up to about 200 s, at which point Multi-Best-Space became superior. However, the trend is reversed in Fig. 9b for the BR6–10 instances with middling heterogeneity, where the early solutions by Multi-Best-Space were better than Single-Best-Space, but the later solutions were poorer. Finally, Fig. 9c indicates that Single-Best-Space is better for the strongly



(a) Convergence of space selection strategies for BR1-BR5



(b) Convergence of space selection strategies for BR6-BR10



(c) Convergence of space selection strategies BR11-BR15

Fig. 9. Convergency behavior of the different space selection strategies.

heterogeneous BR11–15 instances up to about 330 s, but Multi-Best-Space is slightly better thereafter. For all three test sets, allocating half the time to each scheme and then taking the better solution of the two is superior to using either scheme alone (although the Single-Best-Space scheme was slightly better for BR11–15 up to about 110 s).

5. Comparison with other algorithms

We compared ID-GLTS with existing approaches using all 1600 instances of the SCLP benchmark test data BR0–15, which is widely used to compare SCLP and SCLP-FS algorithm performance in existing literature. Our ID-GLTS algorithm allocates half the CPU time to the Single-Best-Space scheme and the other half to Multi-Best-Space scheme, and simply takes the better solution when the execution finishes. For the sake of brevity, we only make comparisons with the most recently published and best algorithms since the average results on each individual test set for these latest algorithms dominate those in earlier literature by large margins. In particular, we only compare with complex search or meta-heuristic approaches rather than the simple heuristics proposed in earlier work.

The detailed results for all experiments described in this section and corresponding binary executables are available online at http://www.computational-logistics.org/orlib/sclp-id-glts.

5.1. Comparison on SCLP

Table 2 compares our new ID-GLTS approach with existing algorithms that do not enforce the full support constraint. Columns 2–7 are results extracted from existing literature:

GRASP: Greedy Randomized Adaptive Search Procedure (Moura and Oliveira, 2005).
IC: Iterated Construction (Lim and Zhang, 2005).
FDA: Fit Degree Algorithm (He and Huang, 2011).
VNS: Variable Neighborhood Search (Parreño et al., 2010).
CLTRS: Container Loading by Tree Search (Fanslau and Bortfeldt, 2010).
G2LA: Greedy 2-step Lookahead (Zhu et al., 2012). For BR0–7 the best results were produced by using simple blocks only; for BR8–15 the best results were produced by using guillotine blocks.

The next three columns show the average percentage volume utilization achieved by our ID-GLTS approach when given a search time limit 10, 300, and 500 CPU seconds per instance, respectively. We do not include the time taken to generate blocks in this time limit; this is reported separately in the last column *BG (s)* (for each test set, the average over 100 instances are reported). The last row reports the average CPU time per instance by each approach; for ID-GLTS, this time includes the search time limit as well as the time taken to generate blocks. Due to the differences in computational environments, the CPU time reported cannot be compared directly.

Prior to this work, the best SCLP approach is G2LA by Zhu et al. (2012), whose results dominate all previous approaches for all 16 test sets. Note that the results for G2LA were achieved given 500 CPU seconds per instance and using the same computational environment as our ID-GLTS approach. Column 500 s shows that, given a similar amount of CPU time per instance (503.45s vs. 500s), our approach outperforms G2LA for all test sets. On average, ID-GLTS achieves a percentage volume utilization that is 0.18% higher for weakly heterogeneous instances (BR1–7) and 0.34% higher for strongly heterogeneous instances (BR8–15). Furthermore, Column 300 s shows that, given an average of only 303.45 CPU seconds per instance, our ID-GLTS algorithm dominates all existing approaches for 15 out of the 16 test sets (the only exception is test set BR0, for which we are slightly worse than G2LA).

When we exclude G2LA from our comparison, our ID-GLTS algorithm likewise outperforms all other algorithms for 15 out 16 test sets when given only 13.45 CPU seconds per instance on average (we are only slightly worse than CLTRS for BR1). Given the fact that we only use 1/30 of the CPU time compared to CLTRS (the second-best algorithm prior to this study), it is reasonable to conclude that ID-CLTS is indeed more effective than CLTRS even after our superior computational environment is taken into account.

5.2. Comparison on SCLP-FS

Table 3 compares ID-GLTS with existing algorithms that enforce the full support constraint. Columns 2–6 are results extracted from existing literature:

PGL: Parallel Generalized Layer-wise loading approach by Terno et al. (2000).
PGA: Parallel Genetic Algorithm by Gehring and Bortfeldt (2002).
HB: Heuristics embedded in tree search, proposed by Bischoff (2006).

Table 2
Comparsion on 1600 BR instances (SCLP).

Test set	GRASP (2005)	IC (2005)	FDA (2011)	VNS (2010)	CLTRS (2010)	G2LA (2010)	ID-GLTS			
							10 s	300 s	500 s	BG (s)
BR0	–	–		–	89.95	90.80	90.62	90.79	90.79	0.02
BR1	89.07	91.60	92.92	94.93	95.05	95.54	95.00	95.55	95.59	0.04
BR2	90.43	91.99	93.93	95.19	95.43	95.98	95.53	96.08	96.13	0.05
BR3	90.86	92.30	93.71	94.99	95.47	96.08	95.65	96.24	96.30	0.04
BR4	90.42	92.36	93.68	94.71	95.18	95.94	95.53	96.07	96.15	0.04
BR5	89.57	91.90	93.73	94.33	95.00	95.74	95.24	95.94	95.98	0.04
BR6	89.71	91.51	93.63	94.04	94.79	95.61	95.15	95.72	95.81	0.10
BR7	88.05	91.01	93.14	93.53	94.24	95.14	94.53	95.28	95.36	1.13
BR8	86.13	–	92.92	92.78	93.70	94.63	93.99	94.71	94.80	3.12
BR9	85.08	–	92.49	92.19	93.44	94.29	93.65	94.46	94.53	3.59
BR10	84.21	–	92.24	91.92	93.09	94.05	93.43	94.27	94.35	4.16
BR11	83.98	–	91.91	91.46	92.81	93.78	93.18	94.09	94.14	4.90
BR12	83.64	–	91.83	91.20	92.73	93.67	93.09	94.03	94.10	6.79
BR13	83.54	–	91.56	91.11	92.46	93.54	92.84	93.80	93.86	7.35
BR14	83.25	–	91.30	90.64	92.40	93.36	92.88	93.76	93.83	9.98
BR15	83.21	–	91.02	90.38	92.40	93.32	92.71	93.72	93.78	10.36
Avg (1–7)	89.73	91.8	93.53	94.53	95.02	95.72	95.23	95.84	95.90	0.20
Avg (8–15)	84.13	–	91.91	91.46	92.88	93.83	93.22	94.11	94.17	6.28
Avg (1–15)	86.74	–	92.67	92.89	93.88	94.71	94.16	94.91	94.98	3.45
Time	69 s	707 s	633 s	296 s	320 s	500 s	13.45 s	303.45 s	503.45 s	

Table 3
Comparison on 1600 BR instances (SCLP-FS).

Test set	PGL (2000)	PGA (2002)	HB (2006)	GRASP (2005)	CLTRS (2010)	ID-GLTS			
						30 s	150 s	Impr	BG (s)
BR0	–	–	–	–	89.83	90.25	90.29	0.46	0.00
BR1	89.9	88.10	89.39	89.07	94.51	**94.25**	**94.40**	−0.11	0.00
BR2	89.6	89.56	90.26	90.43	94.73	**94.62**	94.85	0.12	0.00
BR3	89.2	90.77	91.08	90.86	94.74	94.80	95.10	0.36	0.00
BR4	88.9	91.03	90.90	90.42	94.41	94.46	94.81	0.40	0.00
BR5	88.3	91.23	91.05	89.57	94.13	94.24	94.52	0.39	0.00
BR6	87.4	91.28	90.70	89.71	93.85	93.92	94.33	0.48	0.01
BR7	86.3	91.04	90.44	88.05	93.20	93.22	93.59	0.39	0.18
BR8	–	90.26	–	86.13	92.26	92.27	92.65	0.39	1.29
BR9	–	89.50	–	85.08	91.48	91.65	92.11	0.63	1.19
BR10	–	88.73	–	84.21	90.86	91.13	91.60	0.74	1.14
BR11	–	87.87	–	83.98	90.11	90.25	90.64	0.53	1.00
BR12	–	87.18	–	83.64	89.51	89.77	90.35	0.84	1.02
BR13	–	86.70	–	83.54	88.98	89.11	89.69	0.71	0.96
BR14	–	85.81	–	83.25	88.26	88.40	89.07	0.81	0.77
BR15	–	85.48	–	83.21	87.57	87.70	88.36	0.79	0.67
Avg (1–7)	88.5	90.40	90.50	89.70	94.20	94.22	94.51	0.31	0.03
Avg (8–15)	–	87.69	–	84.13	89.88	90.03	90.56	0.68	1.00
Avg (1–15)	–	89.00	–	86.70	91.90	91.99	92.40	0.50	0.55
Time (s)	–	183	–	69	320	30.55	150.55		

GRASP: Greedy Randomized Adaptive Search Procedure (Moura and Oliveira, 2005).
CLTRS: Container Loading by Tree Search (Fanslau and Bortfeldt, 2010).

The next two columns are the results of ID-GLTS under the search time limits of 30 and 150 CPU seconds per instance, respectively. Column *Impr* gives the difference between the volume utilization of solutions produced by ID-GLTS and CLTRS, which is the best algorithm for SCLP-FS prior to this work and whose results dominate all previous approaches for all 16 test cases. Once again, the time taken to generate blocks is reported separately under Column *BG (s)*.

Given 30 CPU seconds per instance, our ID-GLTS approach outperforms CLTRS (the 2nd best existing approaches) for 13 out of the 16 test sets; for BR1, BR2 and BR3, our approach is slightly worse than CLTRS. When given 150 CPU seconds, our approach outperforms G2LA (the best existing approaches) for 15 out 16 test sets (the exception is BR1, where our approach is slightly worse). For weakly heterogeneous instances (BR1–7), ID-GLTS outperforms CLTRS by 0.31% on average, while for strongly heterogeneous instances (BR8–15), it outperforms CLTRS by 0.68%. An inspection of the values in Column *Impr* suggests that as the heterogeneity of the instances increase, the relative improvement of ID-GLTS compared to CLTRS increases.

6. Conclusion

The Iterative-Doubling Greedy–Lookahead Tree Search (ID-GLTS) approach proposed in this paper is currently the best algorithm for both the SCLP and SCLP-FS on standard benchmark

test data. It is a typical block building approach with two main differences compared to existing approaches. Firstly, we use the Manhattan distance measure to evaluate the fitness of a residual space, which is simpler to compute and slightly more effective than the existing corner distance measure. Secondly, rather than greedily selecting a residual space and then selecting a block to place using a tree search (which we call the Single-Best-Space scheme), our algorithm also incorporates a tree search on block-space pairs (which we call the Multi-Best-Space scheme).

Our final implementation of ID-GLTS allocates half the alloted time to Single-Best-Space and half to Multi-Best-Space, and takes the better solution of the two. Our experiments show that the two schemes are often complementary, and so this 50/50 allocation of time is superior to using all the time on either scheme alone. Note that the Single-Best-Space and Multi-Best-Space computations are totally independent and can be trivally parallelized, which enables our ID-GLTS approach to take advantage of the multi-core architecture of the latest CPUs. The total execution time of this parallel version is expected to be a little more than half of the sequential version.

Since our approach is able to obtain high quality solutions in short time, it is a good candidate as a subrouting to solve other loading problems, such as the multiple container loading problem and the 3D strip packing problem (3DSP). In the process of solving the multiple container loading problem (Che et al., 2011), the single container loading problem are solved many times. Our approach may inspire more efficient algorithms to such problems.

Appendix A. Supplementary material

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.ejor.2012.04.036.

References

Bischoff, E.E., 2006. Three-dimensional packing of items with limited load bearing strength. European Journal of Operational Research 168, 952–966.

Bischoff, E.E., Ratcliff, M.S.W., 1995. Issues in the development of approaches to container loading. OMEGA the International Journal of Management Science 23, 377–390.

Bortfeldt, A., Gehring, H., 2001. A hybrid genetic algorithm for the container loading problem. European Journal of Operational Research 131, 143–161.

Bortfeldt, A., Gehring, H., Mack, D., 2003. A parallel tabu search algorithm for solving the container loading problem. Parallel Computing 29, 641–662.

Che, C.H., Huang, W., Lim, A., Zhu, W., 2011. The multiple container loading cost minimization problem. European Journal of Operational Research 214, 501–511.

Chien, C.F., Wu, W.T., 1998. A recursive computational procedure for container loading. Computers and Industrial Engineering 35, 319–322.

Davies, A.P., Bischoff, E.E., 1999. Weight distribution considerations in container loading. European Journal of Operational Research 114, 509–527.

Eley, M., 2002. Solving container loading problems by block arrangement. European Journal of Operational Research 141, 393–409.

Fanslau, T., Bortfeldt, A., 2010. A tree search algorithm for solving the container loading problem. INFORMS Journal on Computing 22, 222–235.

Fekete, S.P., Schepers, J., van der Veen, J.C., 2007. An exact algorithm for higher-dimensional orthogonal packing. Operations Research 55, 569–587.

Gehring, H., Bortfeldt, A., 1997. A genetic algorithm for solving the container loading problem. International Transactions in Operational Research 4, 401–418.

Gehring, H., Bortfeldt, A., 2002. A parallel genetic algorithm for solving the container loading problem. International Transactions in Operational Research 9, 497–511.

He, K., Huang, W., 2011. An efficient placement heuristic for three-dimensional rectangular packing. Computers and Operations Research 38, 227–233.

Lim, A., Rodrigues, B., Wang, Y., 2003. A multi-faced buildup algorithm for three-dimensional packing problems. OMEGA the International Journal of Management Science 31, 471–481.

Lim, A., Zhang, X., 2005. The container loading problem. In: SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 913–917.

Lins, L., Lins, S., Morabito, R., 2002. An *n*-tet graph approach for non-guillotine packings of *n*-dimensional boxes into an *n*-container. European Journal of Operational Research 141, 421–439.

Mack, D., Bortfeldt, A., Gehring, H., 2004. A parallel hybrid local search algorithm for the container loading problem. International Transactions in Operational Research 11, 511–533.

Martello, S., Toth, P., 1990. Knapsack problems: algorithms and computer implementations. In: Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England.

Moura, A., Oliveira, J.F., 2005. A GRASP approach to the container-loading problem. IEEE Intelligent Systems 20, 50–57.

Parreño, F., Alvarez-Valdes, R., Oliveira, J.E., Tamarit, J.M., 2010. Neighborhood structures for the container loading problem: a VNS implementation. Journal of Heuristics 16, 1–22.

Parreño, F., Alvarez-Valdes, R., Tamarit, J.M., Oliveira, J.F., 2008. A maximal-space algorithm for the container loading problem. INFORMS Journal on Computing 20, 412–422.

Pisinger, D., 2002. Heuristics for the container loading problem. European Journal of Operational Research 141, 382–392.

Ratcliff, M.S.W., Bischoff, E.E., 1998. Allowing for weight considerations in container loading. OR Spectrum 20, 65–71.

Terno, J., Scheithauer, G., Sommerweiß, U., Riehme, J., 2000. An efficient approach for the multi-pallet loading problem. European Journal of Operational Research 123, 372–381.

Wäscher, G., Haußner, H., Schumann, H., 2007. An improved typology of cutting and packing problems. European Journal of Operational Research 183, 1109–1130.

Wu, Y., 2002. An effective quasi-human based heuristic for solving the rectangle packing problem. European Journal of Operational Research 141, 341–358.

Zhu, W., Oon, W.-C., Lim, A., Weng, Y., 2012. The six elements to block-building approaches for the single container loading problem. Applied Intelligence, 1–15.

Zhu, W., Qin, H., Lim, A., Wang, L., 2011. A two-stage tabu search algorithm with enhanced packing heuristics for the 3L-CVRP and M3L-CVRP. Computers and Operations Research.