Appl Intell (2012) 37:431–445
DOI 10.1007/s10489-012-0337-0

The six elements to block-building approaches for the single container loading problem

Wenbin Zhu · Wee-Chong Oon · Andrew Lim · Yujian Weng

Published online: 31 January 2012

Abstract In the Single Container Loading Problem, the aim is to pack three-dimensional boxes into a three-dimensional container so as to maximize the volume utilization of the container. Many recently successful techniques for this problem share a similar structure involving the use of blocks of boxes. However, each technique comprises several seemingly disparate parts, which makes it difficult to analyze these techniques in a systematic manner. By dissecting block building approaches into 6 common elements, we found that existing techniques only differ in the strategies used for each element. This allows us to better understand these algorithms and identify their effective strategies. We then combine those effective strategies into a greedy heuristic for the SCLP problem. Computational experiments on 1,600 commonly used test cases show that our approach outperforms all other existing single-threaded approaches, and is comparable to the best parallel approach to the SCLP. It demonstrates the usefulness of our component-based analysis in the design of block building algorithms.

Keywords Greedy heuristic · 3D packing · Tree search · Metaheuristics · Block building

W. Zhu (✉)
Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
e-mail: i@zhuwb.com

W.-C. Oon · A. Lim
Department of Management Sciences, City University of Hong Kong, Tat Chee Ave, Kowloon Tong, Hong Kong

Y. Weng
Department of Computer Science, School of Information Science and Technology, Zhong Shan (Sun Yat-Sen) University, Guangzhou, Guangdong, P.R. China

1 Introduction

The Single Container Loading Problem (SCLP) has wide applications in the logistics industry. Whenever a truck or container has to be loaded with boxes of cargo, a loading plan must be formulated with the aim of maximizing the space utilization in the container. Formally, we are given a set of three-dimensional cargo boxes (or simply *boxes*) to be loaded that are rectangular parallelepipeds (henceforth referred to as *cuboids*). 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. Boxes are to be loaded into a single large three-dimensional cuboid called a *container*, whose length, width and height 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 within the container. The objective is to maximize the total volume of loaded boxes (equivalently, the unused space in the container is minimized). The SCLP can be considered a three-dimensional generalization of the knapsack problem.

Depending on the application, the rotation of boxes may be freely allowed; restricted to certain orientations (e.g., storage of refrigerators that must be upright); or completely disallowed (e.g., installation of wood panels with grain patterns). Problem instances may be composed of many types of boxes (*strongly heterogeneous*), which is a common scenario for courier services; a few types of boxes (*weakly heterogeneous*), for instance when batches of goods from a warehouse is sent to retailers; or entirely of one type of box (*homogeneous*), a situation that is prevalent for goods coming off an assembly line.

The SCLP is NP-hard in the strict sense [28]. As expected, exact algorithms can only solve instances with limited size [12]. For many real-world applications, it is therefore necessary to resort to heuristics, metaheuristics and incomplete tree search based methods. The best performing approaches in recent literature [10, 14, 26, 27] share similar structures and can be classified as *block building* approaches. However, each reported technique consists of many seemingly disparate parts. On the surface, it seems as if most of these approaches are completely different, especially since a large proportion of the exposition is spent on explaining complex search procedures for selecting blocks. As a result, it has been difficult to properly analyze these techniques in a structured manner.

In this study, we identify the common structure of all block building approaches to the SCLP, presented as six key elements (of which block selection is one); all techniques of this type differ only in how they resolve these elements. This allows us to recast existing approaches in terms of these elements so that we can compare and analyze these approaches systematically, enabling us to identify which elements are effective and portable, and which are comparatively insignificant or replaceable. We perform in-depth theoretical analysis to explain why certain choices for some of the elements are better than others. We also conduct several experiments to verify our analyses, which include re-implementing the CLTRS [10] and MS [26, 27] algorithms, and then systematically replacing certain components of these algorithms and examining the effects.

Our analysis reveals that some of the elements, namely the type of blocks generated, the representation of free space and the block evaluation function have a significant effect on the effectiveness of the approach. We have also discovered that searching for blocks is, surprisingly, not a major differentiating factor in the performance of such approaches, which is in contrast with most of existing literature. For example, Fanslau and Bortfeldt [10] attributed the success of their CLTRS approach to the use of general blocks and their PCTRS search algorithm. However, we show that the simple 2-step lookahead search works just as well as PCTRS, so the impact of PCTRS may have been overstated. The use of general blocks is therefore probably the main factor in the success of CLTRS. Another example is the MS algorithm by Parreño et al. [26, 27], which differs from previous work mainly by using the maximal space representation proposed by Lim et al. [20] combined with a GRASP search. However, we have found evidence to show that the GRASP component is ineffective.

Consequently, we designed a new algorithm for SCLP by choosing appropriate strategies for each of these elements; our new algorithm outperforms all existing techniques in the standard set of test cases. We would like to emphasize that the main aim of this study is *not* to devise the best approach for solving SCLP. Rather, our intention is to identify the key decision components for all block building approaches to the SCLP and analyze their significance, which would allow researchers to better design and evaluate such algorithms in the future. Note that although all existing literature on block building approaches to SCLP describe how each of the six elements are resolved, these decisions are always presented as parts of a complete approach. This study presents an explicit and structured method for devising and analyzing such approaches, which previous researchers have only implicitly utilized.

The remaining sections are organized as follows. First, we provide a summary of the relevant existing literature in Sect. 2. Next, we present the 6 key elements of block building approaches in Sect. 3, along with examples of how these elements translate to implementation decisions. We proceed to examine two of the leading algorithms in existing literature in terms of these elements: Sect. 4 looks at the Container Loading by Tree Search algorithm [10], while Sect. 5 examines the Maximal Space algorithm [26].

Section 6 provides a deeper analysis of the 6 key elements and how they affect algorithm quality; this includes a set of empirical experiments leading to a 2-step lookahead search strategy. Our examination of the most effective strategies from existing literature allowed us to build a new approach, which we present in Sect. 7. The results of experiments that were conducted to compare the effectiveness of this approach with existing techniques are reported in Sect. 8. Finally, we conclude our study in Sect. 9, where we also suggest some avenues for future research.

2 Literature review

Under an improved typology for cutting and packing problems [31], weakly heterogeneous SCLP is classified as a 3-dimensional rectangular single large object placement problem (3D SLOPP), while the strongly heterogeneous SCLP is classified as a single knapsack problem (3D SKP). The SCLP is one of many variants of cutting and packing problems in operations research; we refer the interested reader to the excellent book by Dyckhoff and Finke [8] for a thorough review of this field.

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 also has the biologically 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 [1, 2, 29]. A useful generalization of the SCLP is to consider multiple containers of different types and costs, where the objective is to load all boxes such that the total

cost of containers is minimized; this can be considered a three-dimensional generalization of the bin packing problem and has been previously studied by Che et al. [5].

It is possible to classify existing algorithms for SCLP 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 Chien and Wu [6], Lins et al. [22]. Finally, *local search* methods start with an existing solution, then repeatedly apply neighbourhood operators to generate new solutions; examples include Gehring and Bortfeldt [13], Parreño et al. [27].

Block building approaches are constructive methods, which are the focus of this study. A *block* is a subset of boxes that is placed compactly inside its minimum bounding cuboid. Each step of a block building approach involves the placement of a block into some free space in the container. This is repeated until no more blocks can fit into the container.

Older block building approaches include the algorithms developed by Bortfeldt et al. [4], Eley [9], Lim et al. [21], Mack et al. [23]. The most successful recent approaches for SCLP over the standard benchmark test cases have been block building approaches. Parreño et al. [26] introduced a Maximal Space (MS) algorithm that uses a two-phase GRASP search, while Parreño et al. [27] proposed a variable neighborhood search (VNS) method. The current best single-threaded approach is the Container Loading by Tree Search algorithm (CLTRS) by Fanslau and Bortfeldt [10] that uses a Partition-Controlled Tree Search (PCTRS) to select the block to load in each iteration, and Gonçalves and Resende [14] devised a parallel Biased Random-Key Genetic Algorithm (BRKGA) for the problem, which produced the best results (using a machine with 6 cores) on benchmark data at the time of writing.

There have also been wall-building approaches [3, 28], where a container is filled by vertical layers (called walls), and layer-building approaches [2, 30], 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. All of these techniques only instantiate blocks when the combination of boxes contained within is considered desirable by some measure (e.g., if the total volume of the boxes take up at least 98% of the entire block volume).

3 The 6 key elements

There are 6 key elements to block building approaches, which we label K1 to K6.

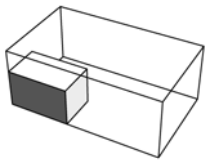


Fig. 1 A single block placed in the container

(K1) how to represent free space in the container;
(K2) how to generate a list of blocks (of boxes);
(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;
(K6) what is the overarching search strategy.

All block building approaches only differ in the way decisions are made for these key elements. Note that the decision made for these elements need not be the same throughout the entire algorithm; it is entirely logical to alter these aspects at different stages of solution construction. Furthermore, all 6 elements are completely independent, and different methods and techniques can be tried for each.

The first key element, K1, involves the representation of free space in the container. In general, free space in the container can be represented by a set of cuboids. Clearly this is true when no boxes have been loaded into the container. Consider the situation when a block is loaded into a free space. Figure 1 shows a single block (the solid rectangular cuboid) that is placed in a corner of the container. The remaining free space (also called *residual space*) is a polyhedron.

The residual space can be represented by three interior-disjoint (i.e., non-overlapping) cuboids, which form a partition of the residual space. There are a total of six possible partitions (see Fig. 2). This representation of residual space is used by the majority of block building approaches; we call this the *partition* representation.

The residual space can also be represented by three overlapping cuboids. Each cuboid is a large rectangular box that is interior-disjoint with the placed block. Figure 3 gives an example. Since the three cuboids overlap with one another, they are illustrated in three separate diagrams (as semi-transparent cuboids) for clarity of presentation. We call this the *cover* representation. It is first proposed by Lim et al. [20], and subsequently employed in the Maximal Space algorithm devised by Parreño et al. [26, 27]. Note that the concept of *action space* defined by He and Huang [16] for the Fit Degree algorithm is exactly the same as the cover representation.

Residual space can also be represented implicitly by computing it when required using the list of boxes already loaded. This is in fact a perfect representation of free space, but the additional time required for computation is significant, and this representation is impractical for instances of

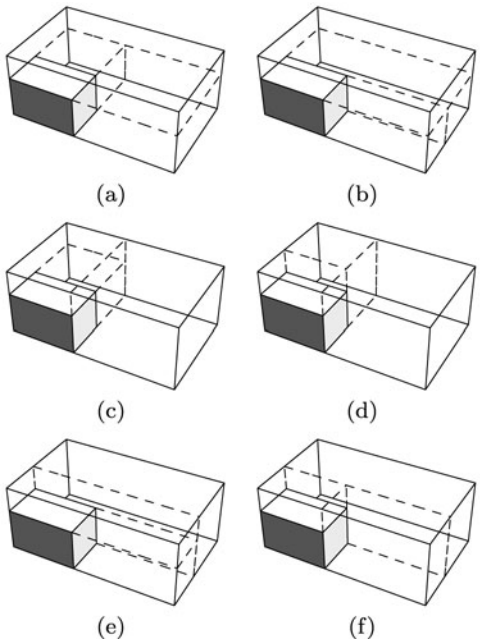


Fig. 2 Six possible partitions of the residual space

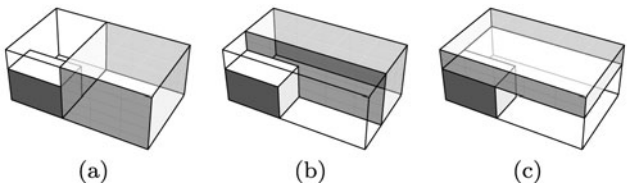


Fig. 3 Three cuboids that form a cover of the residual space

reasonable size. However, small instances of the problem may benefit from the accuracy of this representation. For the rest of this discussion, we will assume that free space is explicitly represented as a list of cuboids.

The second key element, K2, involves the generation of blocks. We classify blocks into two types, both of which have been employed in existing research. A *simple block* consists of only one type of box, where all boxes are placed in the same orientation (Fig. 4(a)). In contrast, a *general block* may consist of multiple types of boxes and/or boxes that are placed in different orientations (Fig. 4(b)). Clearly, a block that consists of only one box is a simple block; any simple block is also a general block. Furthermore, walls and layers are special cases of simple blocks.

The intermediate state of each step in a block building approach can be represented by a list of cuboids S representing the free space in the container, a list of available cargo boxes C that are to be loaded, and a list of blocks B containing boxes in C. The strategies chosen for K1 and K2 determine the states of the search space. The transition between states, however, is determined by the strategies chosen for K3–K5, which we reproduce below using the above notation:

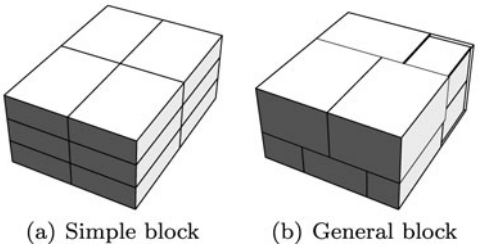


Fig. 4 Classes of blocks

(K3) how to select a cuboid s from the list of free spaces S;
(K4) how to select a block b from the list of blocks B;
(K5) how to place the block b inside the selected cuboid s.

All block building approaches have to devise an oracle for each of the elements K3–K5 that makes each decision. To the best of our knowledge, all existing approaches resolve K3 (choosing a free space cuboid s) and K5 (placing a block into s) using a deterministic rule. However, K4 (selecting the block to load) is often decided using a search strategy; examples include Greedy Randomized Adaptive Search Procedure [25, 26], tree search [9] and Partition Controlled Tree Search [10]. In general, search strategies rely on some evaluation function f that estimates the quality of a given solution. Using f, candidate solutions are ranked in order of quality, which directs the search. Ultimately, these search techniques select the "best" block b to be placed into s.

Once all of the three decisions K3–K5 have been made, the block b is placed into free space s, and s is deleted from the list of available free spaces S. New cuboids are then created to represent the residual space and inserted into S. The set of boxes in b is also removed from the list of available boxes C. As a result of this allocation, there may be some blocks in B that become invalid because they contain types of cargo boxes that exceed their available number; such blocks are removed from B. Hence, a deterministic transition from the current state to the next state occurs once K3–K5 have been decided.

All block building approaches work by exploring a region in the search space to find a path from some initial state to some terminal state. The search space can be represented by a directed acyclic graph (DAG), where the vertices are the states, and the edges are the transitions. The process of finding a loading plan can be described as follows:

(a) start from some initial state, where the list of free space consists of the container, and the list of blocks is generated (explicitly or implicitly) according to some strategy.
(b) move to the next state by selecting a free space cuboid and a block of cargo boxes, followed by placing the selected block into the selected cuboid.
(c) repeat step (b) until we reach a terminal state, where there is no available next state.

Each terminal state in the search space corresponds to a maximal loading plan, where no further boxes can be loaded. Most approaches would generate multiple maximal loading plans, and then select the best one as the final solution. The final key element K6 pertains to how multiple loading plans are generated. In particular, K6 is the set of rules that dictate how the decisions for K1–K5 would change so as to produce different solutions.

Two observations help to explain the success of block building approaches for SCLP. Firstly, the use of blocks introduces shortcuts between states in the search tree. Note that a single box is itself a block. Therefore, as long as all blocks consisting of a single box are included in the list of candidate blocks, and the representation of free space does not exclude possible loading plans, then the search space will include a state for each valid loading plan. However, the addition of blocks containing multiple compactly packed boxes increases the set of primitives such that it includes box configurations that would otherwise require several iterations of search to reach. Block building approaches therefore reduce the average distance between pairs of states that contain blocks with good local structure, which could be a significant factor in the generation of high-quality solutions.

Secondly, observe that for every loading plan, there is an equivalent *gapless* loading plan, where the start coordinates of a box is either 0 or the end coordinates of another box for each axis [11]. The projection of a loading plan with n boxes on an axis is a set of n intervals, which can be represented by an interval graph, where every node represents an interval, and there exists an edge between two nodes if their corresponding intervals overlap. Therefore, every gapless loading plan corresponds to three interval graphs (one for each axis). There are about $n!$ interval graphs with n nodes up to isomorphism [15] (i.e., if we permute the labels of the nodes in a graph to obtain another graph, these two graphs are considered identical). With three dimensions there could be as many as $(n!)^3$ possible interval graph representations, and since there are $n!$ ways to assign boxes to nodes in the interval graphs, there could up to $n! \times (n!)^3 = (n!)^4$ loading plans. However, the number of possible volume utilization values is at most 2^n since every box must either be in the container or not. Hence, the number of possible loading plans dwarfs the number of possible evaluations, which suggests a very flat search landscape. This type of search topology means that hill-climbing or gradient descent methods are unpromising since it is difficult to find clearly superior solutions in a localized search. However, the introduction of blocks with high volume utilization allows search algorithms to examine promising locales at more distant areas of the search landscape in each step, and therefore potentially makes search techniques more effective.

4 The CLTRS algorithm

The Container Loading by Tree Search algorithm (CLTRS) was published by Fanslau and Bortfeldt [10]. It is the first reported SCLP approach to use general blocks, and it makes use of a novel search technique called Partition-Controlled Tree Search (PCTRS) to select the block to load in each iteration. Prior to our work, CLTRS was the best existing single-threaded approach to SCLP based on benchmark tests.

The CLTRS algorithm is carried out in two stages. In the first stage, only simple blocks are considered; in the second stage, general blocks are used. The stages are otherwise identical; the better solution of the two stages is retained.

The implementation details for the formation of general blocks in the second stage were not fully described by the author, but we surmise that it was performed as given in Algorithm 1. Recall that any block that consists of only one box is a general block. There may be more than one block corresponding to each type of box; in fact, there may be a different block for each of the six orthogonal orientations. The generation procedure begins with the set of all blocks B that correspond to single boxes (line 1).

In the first iteration, we attempt to combine all blocks in B with each other by placing them in contact along the x, y and z axes (resulting in at most 3 unique blocks). For each resultant block, we retain it only if the two component blocks take up $\geq min\_fr\%$ of the block. Blocks with the same dimensions containing the same boxes are considered identical even if their internal configurations are different, and all duplicates are discarded. Also discarded are illegal configurations consisting of more boxes of a certain type than is available, or blocks whose dimensions exceed the size of the container. The retained blocks are placed into the set N representing newly constructed blocks (line 9). In subsequent iterations, the blocks in N are first transferred into the set P (line 17), then the blocks in P are combined with all retained blocks generated so far to produce larger blocks, subject to the same duplication and legality conditions (lines 5–12). This continues until the total number of retained blocks is equal to max_bl. The original CLTRS algorithm used the values $min\_fr = 98$ and $max\_bl = 10{,}000$.

The CLTRS algorithm represents free space using the partition representation. Whenever a block is loaded, three free space cuboids are generated; they are termed the *minimal*, *medium* and *maximal* spaces as ordered by increasing size. These cuboids are pushed onto a stack in the following order: minimal, then maximal, then medium; the top free space cuboid on the stack is chosen as the space to be filled for the current iteration. The premise behind this approach is to first fill the larger free spaces (medium and maximal) so that unused space from these cuboids can be transferred to the minimal space cuboid.

Algorithm 1 Generating General Blocks

```
Initialize the set of blocks B corresponding to all permitted orientations of all single boxes;
Blocks generated in previous iteration P ← B;
while |B| < max_bl do
    New blocks N ← ∅;
    for all block b1 in P do
        for all block b2 in B do
            for all axis in {X, Y, Z} do
                Combine b1 and b2 along the axis to obtain b3;
                Insert b3 into N if it is non-duplicate and legal;
            end for
        end for
    end for
    if N = ∅ then
        break;
    end if
    B ← B ∪ N and keep only the first 10,000 blocks;
    P ← N;
end while
Return B;
```

Once a block is chosen, it is placed at the corner of the free space cuboid that is closest to the origin of the coordinate system. The selection of the block to be loaded is decided by the PCTRS method, which is outlined in Appendix.

The 6 key elements for the first phase of the CLTRS algorithm are as follows:

(K1) use the **partition representation** to represent free space;
(K2) construct **simple blocks**;
(K3) select the free space cuboid at the **top of the stack**;
(K4) select a block based on **PCTRS using the *volume* evaluation function**;
(K5) place the block at the **corner closest to the origin**;
(K6) **double search effort** by setting *search_effort* = *search_effort* $\times 2$ (see Appendix).

The second phase is identical to the first phase, except that element K2 becomes:

(K2) construct **general blocks**.

5 The Maximal Space algorithm

The Maximal Space (MS) algorithm was developed by Parreño et al. [26]. It made use of the two-phase GRASP search strategy [25], where a semi-randomised solution is generated in the constructive phase, which is then improved by a local search technique in the improvement phase.

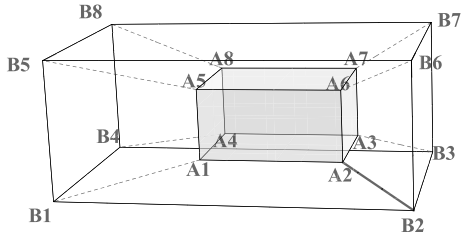


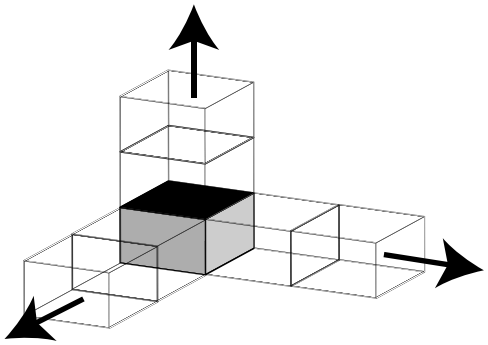
Fig. 5 Corner distance between a free space and the container

During construction, MS chooses a free space cuboid as follows. A free space cuboid s has eight corners, and each corner has a corresponding corner of the container. Figure 5 illustrates eight pairs of corresponding corners: (A_1, B_1), (A_2, B_2), ..., (A_8, B_8). Let (x_1, y_1, z_1) be the coordinates of A_1 and (x_2, y_2, z_2) be the coordinates of B_1; we can compute the triplet $(|x_1 - x_2|, |y_1 - y_2|, |z_1 - z_2|)$. The three components of the triplet are then reordered to (a, b, c) such that $a \leq b \leq c$; the triplet (a, b, c) is called the *corner distance* between A_1 and B_1. All eight pairs of corresponding corners for s and the container can be sorted in ascending lexicographical order of their corner distances. We call the corner in s with the smallest lexicographical order the *anchor corner* of s, and the distance between the anchor corner and its corresponding corner of the container the *anchor distance*; in Fig. 5, A_2 is the anchor corner. The MS algorithm selects the free space cuboid with the smallest anchor distance, with ties broken by greater volume.

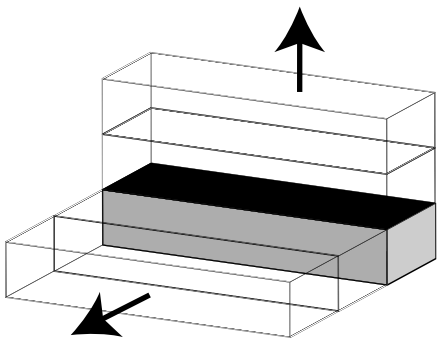
This selection criteria is based on the premise that it is preferable to select a cuboid at a position close to a corner of the container, followed by an edge of the container, followed by a face of the container. As a result of following this strategy, the remaining free space will tend to accumulate in the middle of the container, which helps to keep the free space contiguous and reduce fragmentation. This concept has been termed "*gold corner, silvery side and strawy void*" [16–18, 32].

Having selected a free space cuboid s, the MS algorithm then examines how boxes can be placed into s. It selects a box of type k with q_k such boxes remaining to be loaded, and proceeds to check if multiple boxes of type k can be formed into columns or layers and placed into s at the anchor corner. Figure 6(a) shows the three ways that multiple boxes can be arranged into columns. For each possible column, there are 2 ways that it can be expanded into layers, as shown in Fig. 6(b). Since each box has 6 possible orientations, there are $3 \times 2 \times 6 = 36$ different blocks per box type. In effect, these blocks are a subset of simple blocks since they are composed of only one type of box with identical orientation.

Each of these blocks is ranked by two evaluation functions. The first is *volume*, which is the same measure that was used in the CLTRS algorithm. The second is called *best fit*, which first takes the distances along the three axes dx,



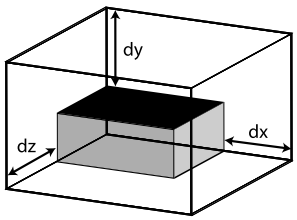
(a) 3 ways to form columns



(b) 2 ways to form layers

Fig. 6 Blocks used by the MS algorithm

Fig. 7 Calculating best fit



dy and dz between the block and the free space cuboid as shown in Fig. 7. These distances are then reordered into a triplet (d_1, d_2, d_3) such that $d_1 \leq d_2 \leq d_3$; the best fit measure selects the smallest value in the lexicographical order of these triplets.

Since the cover representation for free space is employed, the placed block b may intersect with more than one free space cuboid. Hence, the residual space must be computed for each free space cuboid that intersects with block b. For every free space cuboid s that intersects with block b, we replace s with the set of maximal cuboids that form a cover of s after b is placed. After this replacement, there may be some free space cuboids that are fully contained inside other free space cuboids that can be safely removed.

Following the GRASP search strategy, one out of the top $\delta\%$ of these blocks is uniformly randomly selected for each step of the construction phase; the value of δ is dynamically adjusted every 500 iterations based on information gathered from previous iterations (details can be found in [26]).

In the construction phase of the MS algorithm, its 6 key elements are resolved in the following manner:

(K1) use the **cover representation** to represent free space;
(K2) construct **simple blocks by forming columns and layers**;
(K3) select the free space with **smallest anchor distance**;
(K4) select a block from the **top $\delta\%$ with maximal *volume* or with the *best fit***;
(K5) place the block at the **anchor corner**;
(K6) **adjust δ** (every 500 iterations).

In the improvement phase of GRASP, a part of the solution generated is undone, and then redone using a greedy heuristic. Parreño et. al. undid the final 50% of the loaded blocks, and then generated the solution from this point using a deterministic greedy heuristic based on both *volume* and *best fit* measures; their best approach involved running the algorithm once for both measures and retaining the better solution. Therefore, this phase is identical to the construction phase, except that K4 for the two runs now become:

(K4) select a block with **maximal *volume***; and
(K4) select a block with ***best fit***.

Although the MS algorithm generates poorer solutions in terms of space utilization than the newer CLTRS algorithm for the common set of benchmark test cases, its introduction of the cover representation for free space was a significant development at the time of its writing.

6 Analysis of the 6 key elements

When we examine the CLTRS and MS algorithms in terms of the 6 key elements, we find that different decisions were made in all aspects. This section presents a theoretical discussion concerning the impact of each of the 6 elements, using these algorithms and other empirical evidence to support our claims where appropriate.

6.1 Free space representation (K1)

A loading plan P is said to follow the *guillotine-cut* constraint if it can be divided by a plane into two disjoint subsets P_1, P_2 such that P_1 and P_2 are on the opposite sides of the plane; the division can be carried out recursively until both P_1 and P_2 consist of only one box. The partition representation of free space follows the guillotine-cut constraint, but the cover representation does not. However, there are SCLP instances where the optimal loading plans are non-guillotine, so using the partition representation eliminates certain (possibly optimal) configurations from the search

space. In contrast, there is a cover representation for every valid loading plan in the search space for SCLP, i.e., the search space is complete.

The effectiveness of using the cover representation for K1 in the MS algorithm can be seen by comparing it to the implementation of GRASP using the partition representation as done by Moura and Oliveira [25]; even though the primary difference between the two approaches is only the choice of free space representation, MS outperformed GRASP in all test cases by significant margins. The completeness of the search space provided by the cover representation is likely to be a major contributing factor to the effectiveness of MS. However, if the guillotine-cut constraint is imposed (e.g., for certain glass-cutting problems), then the cover representation cannot be used without modification.

6.2 Block generation (K2)

Block building approaches use a block as the primitive for the construction of solutions rather than a box. The introduction of a new set of primitives makes most sense when they represent promising sub-solutions, i.e., the box configuration within the block is likely to belong to good solutions to SCLP when composited. Most existing work has restricted themselves to considering simple blocks, where the boxes are all of the same type and orientation; such approaches include wall- and layer-building techniques that are special cases of simple blocks. In contrast, to the best of our knowledge CLTRS is the first published approach to use general blocks.

The use of general blocks for K2 by CLTRS allows the algorithm to pack dissimilar blocks together rather than limiting the block configurations to collections of similar boxes; multiple blocks can be combined into general blocks directly as long as the space utilization of such a combination is high. Without using general blocks, this combination would take up several iterations of search. General blocks therefore effectively introduce additional shortcuts in the search space that link to promising states consisting of disparate boxes, and compresses the search tree by a greater extent than only using simple blocks.

The use of general blocks is likely to be most effective for strongly heterogeneous test cases, since in those instances there would be many more opportunities to generate general blocks from dissimilar boxes. Conversely, as the heterogeneity of the problem decreases, there would be more boxes of each type in the problem instance, which would increase the number of possible simple blocks. As a result, quality solutions in weakly heterogeneous instances may be composed primarily of simple blocks, reducing the usefulness of general blocks. Therefore, the box composition of the SCLP instance is likely to have a significant effect on the preferable block generation technique for K2.

Note that the technique for generating general blocks given by Algorithm 1 only generates blocks that fulfill the guillotine-cut constraint, since every block is created by combining two blocks. It may be worth investigating the effect of more complex general blocks that do not fulfill this constraint, by combining three or more blocks.

6.3 Free space selection (K3)

All existing block building approaches select free space to be filled (K3) based on a fixed heuristic rule. For example, MS chooses the free space cuboid with the smallest anchor distance, while CLTRS selects the cuboid at the top of the stack constructed using the medium-maximal-minimal rule. Existing approaches also determine how a block is placed into a free space cuboid (K5) using a fixed rule. Consequently, the majority of search effort has gone towards selecting the appropriate block to load into the selected space (K4).

Recall that elements K3, K4 and K5 define the transition rules between states in the search space, and all elements are independent. Therefore, the three elements can be considered an unordered triple (K3, K4, K5) that determine state transition rules within the algorithm. By fixing K3 and K5, two dimensions of the search space are eliminated, allowing the search algorithm to work on the data governed by K4 (i.e., the blocks).

The fact that the triple is unordered is a key observation. Conceptually, we could instead fix K4 and K5, and then search along the dimension represented by K3; this equates to fixing a loading scheme and selecting a block, and then searching for the appropriate space in which to place that box. For example, a technique that selects, say, the largest block in each iteration, and then searches the list of free space cuboids for the best space in which to place the block based on a heuristic-driven search would follow this methodology. The same argument applies to deciding K3 and K4, and then searching based on K5, i.e., picking the free space and the block, and then spending the search effort on how to position the block into the space. It is also logical, for example, to make the decision on K3 and K4 simultaneously, i.e., evaluate the best space-block pair to choose at each stage of the construction.

We believe that this triality of K3, K4 and K5 is an important observation that could potentially lead to new methods of search for SCLP.

6.4 Block selection (K4)

The selection of the block to load is usually decided using a search algorithm. Most search algorithms have two components: the evaluation or fitness function, which estimates the desirability of placing a particular block into the free space

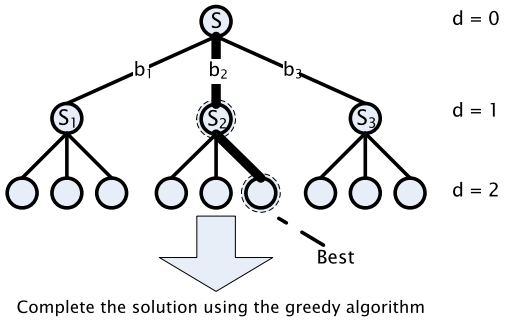


Fig. 8 Two-step look ahead

cuboid; and the search strategy, which examines the search space based on the evaluation function. A search algorithm with no evaluation function is just a series of random walks, while a search algorithm with no search strategy reduces to a greedy heuristic. Ultimately, the aim of all search algorithms for K4 is to select a block to be placed into the free space cuboid.

Arguably the most successful search algorithm to date for this problem is the Partition-Controlled Tree Search (PCTRS) employed by the CLTRS algorithm. For example, the superiority of the results of CLTRS over MS provides strong anecdotal evidence that the PCTRS search strategy is preferable to GRASP for this problem. However, the complexity of the PCTRS strategy makes it difficult to analyze the reasons behind its success.

As a comparison, we examine what happens when we replace the PCTRS search of the CLTRS algorithm with the following simple 2-step lookahead (2LA) technique. Figure 8 illustrates the search tree corresponding to 2-step lookahead. The root node represents the current state. When deciding the block to load, we examine the best w blocks using *volume* as the evaluation function; for each of these w partial solutions, we once again examine the best w blocks to load, resulting in a total of w^2 solutions examined in each iteration. For each of these partial solutions, we find a complete solution using the greedy algorithm based on *volume*. The best choice for the current state is the block that leads to the best completed solution.

The value of w is decided much the same way as the variable *search_effort* for CLTRS. We start with $w = 1$, and increase w to $\lceil \sqrt[d]{2} \times w \rceil$ after every iteration, where $d = 2$, which effectively doubles the number of nodes searched (this decision properly belongs in K6, but we present it here for clarity). Note that the 2-step lookahead can be generalized to d-step lookahead; however, we have computationally determined that $d = 2$ presents the best tradeoff between search accuracy and computation time for this problem. It is reasonable to say that 2LA is a simpler algorithm to implement than PCTRS.

All experiments (including those that will be reported in later sections) were conducted on a rack mounted server with Intel Xeon E5520 Quad-Core CPUs clocked at 2.27 GHz. 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 some of the details of CLTRS were not explained (e.g., tie-breaking rules and block generation order), our implementation was unable to reproduce the original reported result exactly. However, we have carefully calibrated the algorithm so that the performance of our implementation is within $\pm 0.1\%$ of the original published results on average.

Table 1 Volume Utilization of CLTRS using PCTRS and 2LA

Instance	PCTRS	2LA
BR0	89.95	89.99
BR1	95.05	94.89
BR2	95.43	95.13
BR3	95.47	95.34
BR4	95.18	95.19
BR5	95.00	94.91
BR6	94.79	94.72
BR7	94.24	94.28
BR8	93.70	93.84
BR9	93.44	93.44
BR10	93.09	93.15
BR11	92.81	92.98
BR12	92.73	92.91
BR13	92.46	92.64
BR14	92.40	92.61
BR15	92.40	92.43
Avg (1–7)	95.02	94.92
Avg (8–15)	92.88	93.00
Avg (1–15)	93.88	93.90
Time (s)	320 s	324 s

We make use of the same set of 16 test cases (BR0–BR15) used by Franslau and Bortfeldt to analyze their CLTRS algorithm. BR1–BR7 are generated by Bischoff and Ratcliff [2], while BR0 and BR8–BR15 are generated by Davies and Bischoff [7]. 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. No further constraints are imposed.

Table 1 gives the percentage volume utilization of the solutions found by the original CLTRS algorithm and our implementation of CLTRS using 2LA, respectively. For each

set of instances, the figure given in the table is the average volume utilization of 100 instances. We separately list the average of BR1–BR7 and BR8–BR15 in rows *Avg (1–7)* and *Avg (8–15)*. The row *Time* reports the average time taken for each instance. The results show that the original CLTRS outperforms the 2LA implementation for the weakly heterogeneous cases BR1–BR7, but the reverse is true for the strongly heterogeneous cases BR8–BR15. In fact, the overall average performance of 2LA over all test cases is superior to PCTRS by a very small amount, but the difference is insufficient to conclude the superiority of one approach over the other.

These experiments show that the behaviour of the complex PCTRS search method is currently poorly understood, and its effects on solution quality may have been overstated; in any case, the simple 2LA search produces solutions of similar quality.

6.5 Block placement (K5)

At a conceptual level, all of the arguments given for K3 in Sect. 6.3 also apply to finding the appropriate location to place a block for K5. Most current techniques use a fixed heuristic for this purpose, e.g., MS picks the anchor corner, while CLTRS picks the corner closest to the origin. However, given an arbitrary space/block pair, the decision on where to place the block in the space is equally as important as selecting the appropriate block given an arbitrary space/placement pair.

Existing approaches tend to place a block into a corner of the free space cuboid. However, it is unknown whether there is always an optimal packing plan for SCLP has at least one filled corner in the container. If this is in fact not true, then using a K5 rule that chooses a corner of the free space cuboid would be sub-optimal. In any case, this is an equally valid direction from which to approach the problem from a theoretical standpoint.

6.6 Overarching search strategy (K6)

The task of the overarching search strategy is to change the decisions for K1–K5 so as to generate different (and preferably better) solutions. For example, both PCTRS and 2LA change the number of nodes searched after every iteration by doubling the search effort, while GRASP changes the value of δ (and hence the randomness of the search) every 500 iterations based on information from previous iterations. For an algorithm like GRASP with a random element, simply re-running the iteration would introduce a change; but for deterministic algorithms like PCTRS and 2LA, some parameter must change in order to generate different solutions.

While the selection of a proper overarching search strategy is aided by having an understanding of the decisions made for K1–K5, it is important to realize that K6 is a separate decision. Existing literature has tended to encompass both the search used to select a block in K4 and the strategy for generating different solutions in K6 as a single search strategy. However, it is conceptually useful to look at these two components individually.

It is possible to devise a search strategy that changes any or all decisions for K1–K5 across iterations. For example, one could first make use of simple blocks, and then change to general blocks when the number of boxes per type falls below a certain threshold (affecting K2); the K6 decision could determine how this threshold changes. Similar approaches can be applied to the other key elements.

7 A Greedy 2-Step Lookahead algorithm

In this section, we present our use of the 6 key elements directly to develop a new block building algorithm for SCLP, simply by picking strategies for K1–K6. Our resultant algorithm, called the Greedy 2-Step Lookahead (G2LA) algorithm is basically an amalgam of the best aspects of CLTRS and MS, combined with an improved evaluation function for 2LA.

For K1, we follow the lead of the MS algorithm and make use of the *cover representation* for free space cuboids, since it does not restrict our search to guillotine cuts unlike the partition representation.

For K2, we choose to use *general blocks* like CLTRS, which effectively compresses the search space by pre-generating blocks consisting of heterogeneous boxes that have high volume utilization; this decision should provide the most significant effect for heterogeneous test cases. We use the same parameters that were employed for CLTRS, i.e., minimum volume utilization $min\_fr = 98$ and maximum number of blocks generated $max\_bl = 10{,}000$.

For K3, we prefer the method used by the MS algorithm, namely to select the free space cuboid with the smallest *anchor distance*. We feel that this represents a stronger heuristic than the choice made by CLTRS. In particular, since we are using the cover representation for free space, so the motivation for the choice made by CLTRS of allocating unused space to enlarge small free space cuboids does not apply.

For K4, we use the simple *2LA search*. However, the basic *volume* evaluation function does not reflect the amount of unusable space (i.e., space that can never be utilized by any future block) that is created when a block is loaded, which we feel is an important factor in deciding the desirability of a block to be loaded. Therefore, for a free space cuboid s and a given block b, we define an evaluation function $f(s, b) = V + \alpha \cdot V_{\text{loss}}$, where V is the *volume* measure and V_{loss} is the estimated *wasted volume* in the residual space.

Recall that residual space after a block is loaded is represented by three cuboids. Given such a cuboid and a set of 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 that runs in pseudopolynomial time [24] for this purpose. V_{loss} is the total amount of wasted space trimmed from the three cuboids as measured by this technique. For our experiments, we set $\alpha = -1$. If the block is too large to fit into the free space, or the number of boxes of a certain type in the block exceeds the availability in the current state, then $f(s, b) = -\infty$.

For K5, we select the *anchor corner* to place the selected block into the selected free space cuboid (like the MS algorithm), for much the same reasons as for K3.

Finally, for K6, we double the search effort by setting w to $\lceil \sqrt[2]{2} \times w \rceil$.

In summary, the 6 key elements for the G2LA algorithm are as follows:

(K1) use the **cover representation** to represent free space;
(K2) construct **general blocks**;
(K3) select the free space with **smallest anchor distance**;
(K4) select a block based on **2LA using the** $f(s, b)$ **evaluation function**;
(K5) place the block at the **anchor corner**;
(K6) **double search effort** by setting $w = \lceil \sqrt[2]{2} \times w \rceil$.

For completeness, Algorithm 2 lists the pseudocode for the Greedy 2-Step Lookahead algorithm.

Algorithm 2 The G2LA algorithm

```
Generate a block list B with up to max_bl general blocks;
w ← 1;
while there is time remaining do
    Initialize a list S of free space to be the container itself;
    while S ≠ ∅ do
        Select a free space s with smallest anchor distance among S;
        Select a block b with maximum fitness according to 2LA using f(s, b);
        if b exists then
            Place b at the anchor corner of s;
            Update the list of free space S;
        else
            remove s from S;
        end if
    end while
    w ← √2 × w;
end while
```

Assume that our block list B and space list S are implemented as linked lists. We first analyze the time complexity for transitioning from one state to another (i.e., placing a selected block into a selected space) in terms of the number of available blocks and spaces. In each state transition, we check each block in the block list B to remove the invalid blocks. To determine the validity of a block, we check the number of remaining boxes for each of the K box types, so removing all invalid blocks takes $O(|B| \cdot K)$ time. We also scan the space list S for spaces that overlap with the placed block in $O(|S|)$ time; for each overlapped space, up to 6 new spaces may be generated, and each new space is checked to see if it is completely within an existing space (whereupon the new space is discarded) in $O(|S|)$ time. Hence, the time needed to make a state transition in the search is $O(|S|^2 + |B| \cdot K)$.

In the worst case, the number of free spaces in S is at most $O(N^6)$, where N is the number of boxes, since each face of a maximal space coincides with one of the 6 faces of a placed block. Also, the number of free spaces is at least $O(n^3)$ because the corner closest to the origin of each space is a normal position; a box placed at a normal position cannot be pushed towards the origin without also pushing other boxes, and there are at least $O(n^3)$ such positions. However, in our experience these worst case scenarios seldom occur. We have tried to use an R-tree to manage free spaces that results in a better worst case asymptotic time complexity of $O(|S| + |B| \cdot K)$ for a state transition, but our experiments show using an R-tree produced poorer solutions that using a linked list given the same amount of computation time on our test data.

Given a fixed w, our 2LA generates w nodes in the first level. We first select a space in $O(|S|)$ time, then evaluate the fitness of all blocks in $O(|B|)$ time. We can find the best w blocks using a sorted set in $O(w \cdot \log w)$ time. Therefore, the time complexity to produce the first level nodes is $O(|S| + |B| + w \cdot \log w + w \cdot T_{st})$, where $T_{st} = O(|S|^2 + |B| \cdot K)$ is the time required to perform a state transition. We repeat this process for each first level node to generate the second level nodes.

For each second level node, we invoke a greedy heuristic that places one block at a time until no more blocks can be placed. Let P be the number of blocks placed in one invokation of the greedy heuristic; the running time for one invokation is $O(P \cdot T_{st})$. Hence, the total running time of 2LA for a fixed value of w is $O((|S| + |B| + w \cdot \log w + w \cdot T_{st}) \cdot (w + 1) + w^2 \cdot (P \cdot T_{st}))$. We can safely assume that $\log w$ is dominated by $P \cdot |S|^2$, and under this assumption the worst case time complexity of 2LA can be simplified to $O(w^2 \cdot P \cdot |S|^2 + w^2 \cdot P \cdot K \cdot |B|)$. Our G2LA algorithm

Table 2 Algorithm comparison on data sets BR0–BR15

Instance	GRASP	IC	FDA	MS	VNS	CLTRS	BRKGA*	G2LA (200)	G2LA (500)	SB	$\alpha = 0$	G2LA-GRASP
BR0	–	–		–	–	89.95	–	90.57	90.64	90.80	90.49	90.60
BR1	89.07	91.60	92.92	93.85	94.93	95.05	95.28	95.03	95.26	95.54	95.00	94.47
BR2	90.43	91.99	93.93	94.22	95.19	95.43	95.90	95.38	95.60	95.98	95.30	94.31
BR3	90.86	92.30	93.71	94.25	94.99	95.47	96.13	95.67	95.82	96.08	95.41	93.95
BR4	90.42	92.36	93.68	94.09	94.71	95.18	96.01	95.46	95.62	95.94	95.24	93.65
BR5	89.57	91.90	93.73	93.87	94.33	95.00	95.84	95.32	95.47	95.74	95.11	93.36
BR6	89.71	91.51	93.63	93.52	94.04	94.79	95.72	95.28	95.39	95.61	94.97	93.02
BR7	88.05	91.01	93.14	92.94	93.53	94.24	95.29	94.87	95.01	95.14	94.57	92.55
BR8	86.13	–	92.92	91.02	92.78	93.70	94.76	94.52	94.63	94.60	94.25	92.00
BR9	85.08	–	92.49	90.46	92.19	93.44	94.34	94.13	94.29	94.03	93.80	91.62
BR10	84.21	–	92.24	89.87	91.92	93.09	93.86	93.89	94.05	93.51	93.65	91.29
BR11	83.98	–	91.91	89.36	91.46	92.81	93.60	93.63	93.78	92.94	93.46	91.12
BR12	83.64	–	91.83	89.03	91.20	92.73	93.22	93.52	93.67	92.42	93.32	90.96
BR13	83.54	–	91.56	88.56	91.11	92.46	92.99	93.34	93.54	92.02	93.26	90.78
BR14	83.25	–	91.30	88.46	90.64	92.40	92.68	93.18	93.36	91.76	93.13	90.76
BR15	83.21	–	91.02	88.36	90.38	92.40	92.46	93.19	93.32	91.54	92.97	90.64
Avg (1–7)	89.73	91.8	93.53	93.82	94.53	95.02	95.74	95.29	95.45	95.72	95.09	93.62
Avg (8–15)	84.13	–	91.91	89.39	91.46	92.88	93.49	93.67	93.83	92.85	93.48	91.15
Avg (1–15)	86.74	–	92.67	91.46	92.89	93.88	94.54	94.43	94.59	94.19	94.23	92.30
Time	69 s	707 s	633 s	101 s	296 s	320 s	147 s	200 s	500 s	499 s	500 s	500 s

*Parallel algorithm executed on an AMD 2.2 GHz Opteron 6-core CPU running the Linux OS

doubles the search effort in every iteration. If w^2 is the effort used in the last iteration, then the total time taken by G2LA is approximately twice the time taken by the last iteration, so the worst case asymptotic running time bound is the same as 2LA.

8 Computational experiments

The comparisons between G2LA and existing algorithms are summarized in Table 2; once again, we analyze the effectiveness of the algorithms using the 16 test cases BR0–BR15. Column 2 to column 8 are results extracted from existing literature:

GRASP: Greedy Randomized Adaptive Search Procedure [25];
IC: Iterated Construction [19];
FDA: Fit Degree Algorithm [16];
MS: Maximal Space [26];
VNS: Variable Neighborhood Search [27];
CLTRS: Container Loading by Tree Search [10];
BRKGA: A parallel Biased Random-Key Genetic Algorithm [14].

The next two columns give the results of G2LA where each instance is given a time limit of 200 and 500 seconds respectively. The time value shown is the average time taken for each test case.

Except for BRKGA, these algorithms are all single-threaded (non-parallel). The results show that G2LA outperforms all single-threaded approaches when given 200 seconds of CPU time; the improvement reaches as much as 0.71% over the reigning best single-threaded approach CLTRS when 500 seconds is used. Since the benchmarks for these 16 sets of instances have been continuously improved in the past decade, it has become increasingly harder to improve on the results. Hence, a 0.95% improvement for BR8–BR15 and a 0.43% improvement for BR1–BR7 are significant results.

The best current approach is BRKGA, which is a parallel algorithm executed on a machine with six cores. Our G2LA approach performs slightly worse than BRKGA for BR1–7 but slightly better for BR8–15. When given 200 seconds, G2LA is slightly worse than BRKGA on average over all instances, but slightly better after 500 seconds. However, it is difficult to objectively compare our single-threaded approach with the parallel BRKGA.

For a deeper analysis, we made incremental modifications to G2LA and examined the effects. Column *SB* is otherwise the same as *G2LA (500)* except only simple blocks are used. A comparison between the respective results shows that the use of general blocks improved the overall average performance over all test cases by 0.40%. However, the performance for test cases BR0–7, which are homogeneous or weakly heterogeneous, improves slightly when simple blocks are used. This is because good solutions for less heterogeneous instances tend to be composed of mainly simple blocks, so the flexibility provided by general blocks does not justify the added computation time, an observation that concurs with those made by Franslau and Bortfeldt [10]. It may therefore be advantageous to perform G2LA in two phases, one using simple blocks and the other using general blocks, and then take the better solution, particularly if the nature of the problem instance is unknown. However, this is tantamount to running two separate algorithms, so if time is a factor then using general blocks is likely to be the superior choice.

The values in the column labeled $\alpha = 0$ replace the evaluation function with simply the *Volume* measure, i.e., the amount of permanently unusable residual space created is not taken into account. The resultant solution found by the algorithm is inferior to the original formulation for all test cases, which shows convincingly that including the amount of wasted space when evaluating the desirability of loading a particular block improves the accuracy of the evaluation.

Finally, the column labeled *G2LA-GRASP* gives the performance of the algorithm when the 2LA search strategy is replaced by the GRASP search strategy (with δ updated every 500 iterations, and as many iterations are executed as possible within 500 seconds). The results are significantly poorer after this change for all test cases. This suggests that a semi-randomised search technique like GRASP is less appropriate than greedy deterministic search techniques like 2LA or PCTRS for this problem.

The complete experimental results can be found online at http://www.zhuwb.com/sclp-6elements.

9 Conclusion

In this study, we have identified the 6 key elements that are crucial to all block building approaches, namely representing free space (K1); representing blocks (K2); selecting free space (K3); selecting the block to load (K4); loading the selected block into the free space (K5); and the overarching search strategy (K6). We showed how two successful block building algorithms can be analyzed based on their decisions for the 6 key elements, namely the CLTRS algorithm (which was the best existing non-parallel algorithm at the time of this writing) and the MS algorithm. This helped us to decompose both somewhat complex techniques into their component parts, allowing a more focused analysis of their capabilities. In particular, we investigated the effectiveness of the PCTRS search strategy employed by CLTRS, and showed that the comparatively simpler 2-step lookahead strategy produces solutions of similar quality.

We also devised a new block building approach for SCLP by making theoretically sound choices for each of the 6 key elements, i.e., cover representation of free space (K1); general blocks (K2); select the free space with minimal anchor distance (K3); select the block using 2LA under the $f(s, b)$ evaluation function (K4); place the block at the anchor corner (K5); and double search effort (K6). Computational experiments on the 1,600 standard benchmark instances showed that our new approach outperforms all existing single-threaded approaches on each of the individual sets by a significant margin.

The primary aim of this research is not to produce an algorithm that beats the existing benchmarks for SCLP, although there is certainly a tangible practical benefit to devising the strongest possible algorithm. Possible improvements to the performance of G2LA include the 2-phase approach suggested in Sect. 8 and using a metaheuristic like Variable Neighbourhood Search [27] to improve the quality of the solution.

More importantly however, there is much to be learnt from studying the individual components of block building algorithms in terms of the 6 key elements. Each of the individual steps K1–K6 represents an important decision to be made. Innovations such as the cover representation for free space (K1) and general blocks (K2) have already been shown to be important advances in this domain. Furthermore, the observation of the triality of K3, K4 and K5 opens up the possibility of vastly different approaches based on searching the set of free spaces or loading rules rather than the set of blocks (which has been the focus of most existing work).

Appendix: Partition-controlled tree search

In this section, we provide an outline of the partition-controlled tree search, which is used by the CLTRS algorithm. Precise details of this approach can be found in [10].

Assuming at least b successors are generated at each node of the search tree, and the total number of nodes in the search is limited by a value *search_effort*, then the maximal depth of search td is given by:

$$td = \max\{1, \lfloor \log_b search\_effort \rfloor\} \quad (1)$$

The authors set $b = 3$. For example, when *search_effort* $= 32$, then $td = 3$ based on (1). The set of all ordered partitions of the integer td is given by $\pi = \{(3), (2, 1), (1, 2), (1, 1, 1)\}$; for every partition, a separate search is carried out (see Fig. 9(a)).

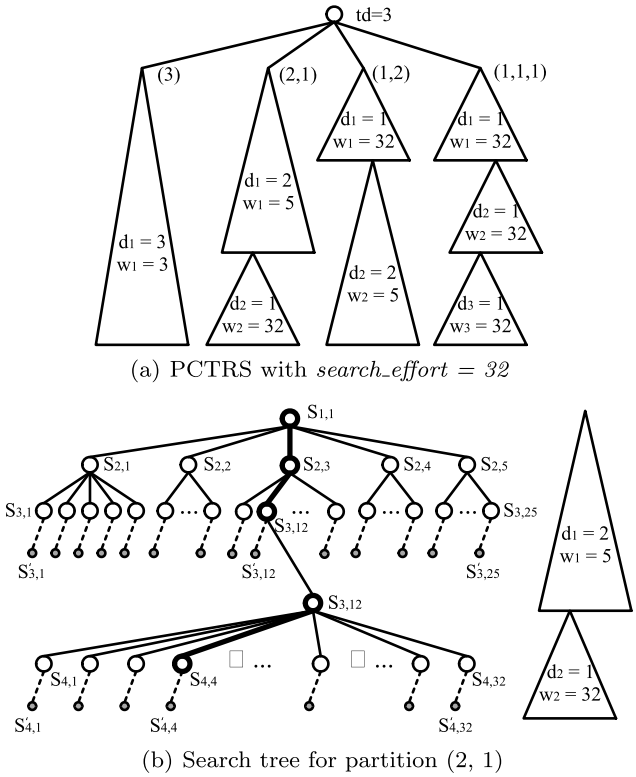


Fig. 9 An example of PCTRS

The search corresponding to the partition (2, 1) is shown in Fig. 9(b). It is divided into two stages, where the first stage can be thought of as a search tree of depth 2, and the second stage is of depth 1. At each stage, w successors are generated for each node. Let d be the depth of the stage; then w is established by:

$$w = \max\{b, \lfloor \sqrt[d]{search\_effort} \rfloor\} \quad (2)$$

For every node, the w blocks with maximal loaded volume are selected to generate the w successors; the values of w_1 and w_2 corresponding to the two stages are calculated to be 5 and 32 respectively. The leaf nodes of the search tree in the first stage are expanded using a greedy heuristic that repeatedly chooses the block with maximum volume that can fit into the top free space cuboid in the stack. Hence in the first stage, 5 successors are generated, and for each of these states a further 5 successors are tried. All of these 25 partial solutions are expanded into complete solutions using the greedy heuristic.

The node leading to the best complete solution is retained as the root node for the second stage; this process is repeated for all the remaining stages. Assume that the partial solution $S_{3,12}$ leads to the best complete solution $S'_{3,12}$. The second stage then starts with $S_{3,12}$ as the root; this search tree has only 1 level, with $w_2 = 32$ branches. Each of these branches are also expanded into complete solutions using the greedy heuristic. The move leading to the best solution (for instance $S'_{4,4}$) is selected as the choice for the branch corresponding to partition (2, 1) (in this case, it is $S_{2,3}$).

The move leading to the best solution out of all of the partitions in π is finally selected, and the block corresponding to that move is loaded into the container. This repeats until no further blocks can be loaded into the container, or until the allotted time runs out. If there is time remaining, then the algorithm restarts with the value of *search_effort* doubled from the previous run. Whenever a complete solution is generated, it is compared to the best solution found so far, and the best solution is updated if the new solution is better.

References

1. Bischoff EE (2006) Three-dimensional packing of items with limited load bearing strength. Eur J Oper Res 168(3):952–966. doi:10.1016/j.ejor.2004.04.037
2. Bischoff EE, Ratcliff MSW (1995) Issues in the development of approaches to container loading. OMEGA Int J Manag Sci 23(4):377–390. doi:10.1016/0305-0483(95)00015-G
3. Bortfeldt A, Gehring H (2001) A hybrid genetic algorithm for the container loading problem. Eur J Oper Res 131(1):143–161. doi:10.1016/S0377-2217(00)00055-2
4. Bortfeldt A, Gehring H, Mack D (2003) A parallel tabu search algorithm for solving the container loading problem. Parallel Comput 29(5):641–662. doi:10.1016/S0167-8191(03)00047-4
5. Che CH, Huang W, Lim A, Zhu W (2011) The multiple container loading cost minimization problem. Eur J Oper Res 214(3):501–511. doi:10.1016/j.ejor.2011.04.017,
6. Chien CF, Wu WT (1998) A recursive computational procedure for container loading. Comput Ind Eng 35(1–2):319–322. doi:10.1016/S0360-8352(98)00084-9
7. Davies AP, Bischoff EE (1999) Weight distribution considerations in container loading. Eur J Oper Res 114(3):509–527. doi:10.1016/S0377-2217(98)00139-8
8. Dyckhoff H, Finke U (1992) Cutting and packing in production and distribution: a typology and bibliography, 1st edn. Contributions to management science. Physica-Verlag, Heidelberg
9. Eley M (2002) Solving container loading problems by block arrangement. Eur J Oper Res 141(2):393–409. doi:10.1016/S0377-2217(02)00133-9
10. Fanslau T, Bortfeldt A (2010) A tree search algorithm for solving the container loading problem. INFORMS J Comput 22(2):222–235. doi:10.1287/ijoc.1090.0338
11. Fekete SP, Schepers J (2004) A combinatorial characterization of higher-dimensional orthogonal packing. Math Oper Res 29(2):353–368. doi:10.1287/moor.1030.0079
12. Fekete SP, Schepers J, van der Veen JC (2007) An exact algorithm for higher-dimensional orthogonal packing. Oper Res 55(3):569–587. doi:10.1287/opre.1060.0369
13. Gehring H, Bortfeldt A (1997) A genetic algorithm for solving the container loading problem. Int Trans Oper Res 4(5–6):401–418. doi:10.1111/j.1475-3995.1997.tb00095.x
14. Gonçalves JF, Resende MGC (2012) A parallel multi-population biased random-key genetic algorithm for a container loading problem. Comput Oper Res 39(2):179–190. doi:10.1016/j.cor.2011.03.009
15. Hanlon P (1982) Counting interval graphs. Trans Am Math Soc 272(2):383–426. doi:10.2307/1998705
16. He K, Huang W (2011) An efficient placement heuristic for three-dimensional rectangular packing. Comput Oper Res 38(1):227–233. doi:10.1016/j.cor.2010.04.015

17. Huang W, He K (2009) A caving degree approach for the single container loading problem. Eur J Oper Res 196(1):93–101. doi:10.1016/j.ejor.2008.02.024
18. Huang W, Chen D, Xu R (2007) A new heuristic algorithm for rectangle packing. Comput Oper Res 34:3270–3280
19. Lim A, Zhang X (2005) The container loading problem. In: SAC'05: Proceedings of the 2005 ACM symposium on applied computing. ACM, New York, pp 913–917. doi:10.1145/1066677.1066888
20. Lim A, Rodrigues B, Wang Y (2003) A multi-faced buildup algorithm for three-dimensional packing problems. OMEGA Int J Manag Sci 31(6):471–481. doi:10.1016/j.omega.2003.08.004
21. Lim A, Rodrigues B, Yang Y (2005) 3-D container packing heuristics. Appl Intell 22(2):125–134. doi:10.1007/s10489-005-5601-0
22. Lins L, Lins S, Morabito R (2002) An n-tet graph approach for non-guillotine packings of n-dimensional boxes into an n-container. Eur J Oper Res 141(2):421–439. doi:10.1016/S0377-2217(02)00135-2
23. Mack D, Bortfeldt A, Gehring H (2004) A parallel hybrid local search algorithm for the container loading problem. Int Trans Oper Res 11(5):511–533. doi:10.1111/j.1475-3995.2004.00474.x
24. Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. Wiley Interscience series in discrete mathematics and optimization. Wiley, Chichester,
25. Moura A, Oliveira JF (2005) A GRASP approach to the container-loading problem. IEEE Intell Syst 20(4):50–57. doi:10.1109/MIS.2005.57
26. Parreño F, Alvarez-Valdes R, Tamarit JM, Oliveira JF (2008) A maximal-space algorithm for the container loading problem. INFORMS J Comput 20(3):412–422. doi:10.1287/ijoc.1070.0254
27. Parreño F, Alvarez-Valdes R, Oliveira JE, Tamarit JM (2010) Neighborhood structures for the container loading problem: a VNS implementation. J Heuristics 16(1):1–22. doi:10.1007/s10732-008-9081-3
28. Pisinger D (2002) Heuristics for the container loading problem. Eur J Oper Res 141(2):382–392. doi:10.1016/S0377-2217(02)00132-7
29. Ratcliff MSW, Bischoff EE (1998) Allowing for weight considerations in container loading. OR Spektrum 20(1):65–71. doi:10.1007/BF01545534
30. Terno J, Scheithauer G, Sommerweiß U, Riehme J (2000) An efficient approach for the multi-pallet loading problem. Eur J Oper Res 123(2):372–381. doi:10.1016/S0377-2217(99)00263-5
31. Wäscher G, Haußner H, Schumann H (2007) An improved typology of cutting and packing problems. Eur J Oper Res 183(3):1109–1130. doi:10.1016/j.ejor.2005.12.047
32. Wu Y (2002) An effective quasi-human based heuristic for solving the rectangle packing problem. Eur J Oper Res 141(2):341–358. doi:10.1016/S0377-2217(02)00129-7

Wenbin Zhu is currently a Ph.D. candidate in the Computer Science and Engineering department at the Hong Kong University of Science and Technology. He received his bachelor's degree from School of Computing at the National University of Singapore in 2002. His research interest is in discrete optimization problems related to operations research.

Wee-Chong Oon recently completed his postdoctoral research fellowship at City University of Hong Kong. He received his B.Sc. (Hons.), M.Sc. and Ph.D. degrees in Computer Science from School of Computing at the National University of Singapore in 1998, 2001 and 2007, respectively. His current research interests are in the fields of algorithms and artificial intelligence, including various topics in operations research, games, timetabling and scheduling.

Andrew Lim is currently a professor in department of Management Sciences, College of Business, City University of Hong Kong. Andrew Lim obtained his Ph.D. in Computer Science in 1992 from the University of Minnesota. The aim of Andrew's research is to help companies compete effectively. This includes deriving customized business value models; identifying key performance indicators, and developing optimized processes that improve these indicators. Andrew's works have been published in key journals in Operations Research and Management Science, and disseminated via international conferences and professional seminars. More importantly these works have contributed to substantial impacts to many renowned multinational companies leading to international awards and company-wide innovative prizes. However, among the number of activities that Andrew has undertaken, he derives greatest satisfaction in education and mentoring which resulted in strong placements of his students at all levels.

Yujian Weng is currently a software engineer in Beijing Yahoo! R&D Center. He received his M.S. degree in Computer Science at the Sun Yat-sen University in 2010. His research areas include intelligent systems and optimization algorithms.