Computers & Operations Research 39 (2012) 2178–2195

Contents lists available at SciVerse ScienceDirect

Computers & Operations Research

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

A two-stage tabu search algorithm with enhanced packing heuristics for the 3L-CVRP and M3L-CVRP

Wenbin Zhu [b], Hu Qin [d,a,*], Andrew Lim [a], Lei Wang [c,a]

[a] Department of Management Sciences, City University of Hong Kong, Tat Chee Ave, Kowloon Tong, Hong Kong
[b] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
[c] Department of Computer Science, School of Information Science and Technology, Zhongshan (Sun Yat-Sen) University, Guangzhou, Guangdong 510275, PR China
[d] School of Management, Huazhong University of Science and Technology, No. 1037, Luoyu Road, Wuhan, China

ARTICLE INFO

Available online 9 November 2011

Keywords:
Vehicle routing problem
3D packing
Deepest-Bottom-Left-Fill
Maximum Touching Area
Tabu search

ABSTRACT

The Three-Dimensional Loading Capacitated Vehicle Routing Problem (3L-CVRP) addresses practical constraints frequently encountered in the freight transportation industry. In this problem, the task is to serve all customers using a homogeneous fleet of vehicles at minimum traveling cost. The constraints imposed by the three-dimensional shape of the goods, the unloading order, item fragility, and the stability of the loading plan of each vehicle are explicitly considered. We improved two well-known packing heuristics, namely the Deepest-Bottom-Left-Fill heuristic and the Maximum Touching Area heuristic, for the three-dimensional loading sub-problem and provided efficient implementations. Based on these two new heuristics, an effective tabu search algorithm is given to address the overall problem. Computational experiments on publicly available test instances show our new approach outperforms the current best algorithms for 20 out of 27 instances. Our approach is also superior to the existing algorithm on benchmark data for the closely related problem variant M3L-CVRP (which uses a slightly different unloading order constraint compared to 3L-CVRP).

1. Introduction

The Three-Dimensional Loading Capacitated Vehicle Routing Problem (3L-CVRP) was first introduced by Gendreau et al. [13] and subsequently studied by Tarantilis et al. [27] and Fuellerer et al. [12]. The task of the 3L-CVRP is to decide the routes traveled by a fleet of homogeneous vehicles that deliver items to customers such that the total distance traveled by all vehicles is minimized. In addition, the three-dimensional loading plan for each vehicle must be formulated while fulfilling a number of constraints that address issues such as the stability of the items, packing requirements for fragile items, and the convenience of loading and unloading. This problem is of great practical interest to the freight distribution industry since the combination of the vehicle routing problem and the 3-D container loading problem with realistic constraints closely models real-life situations, especially when the delivery involves multiple large items of dissimilar dimensions. A few example applications include the delivery of household appliances, kitchen components, mechanical components [12], and furniture [13].

When searching for a solution for 3L-CVRP, we need to repeatedly decide whether all items of the clients visited by a vehicle can be loaded into the vehicle. We handle this Single Vehicle Loading sub-problem by enhancing two well-known heuristics, namely the Deepest-Bottom-Left-Fill (DBLF) heuristic and the Maximum Touching Area (MTA) heuristic, and embed them into a local search. We make use of this local search in a two-phase tabu search algorithm that employs five neighborhood operators to solve the 3L-CVRP. In the first phase, we attempt to find a feasible solution and in the second phase, we attempt to optimize the total travel distance.

We compared our DBLF+MTA Tabu Search (DMTS) algorithm with the Tabu Search (TS) algorithm by Gendreau et al. [13]; the Guided Tabu Search (GTS) algorithm by Tarantilis et al. [27]; and the Ant Colony Optimization (ACO) algorithm by Fuellerer et al. [12] using a standard set of 27 test instances proposed by Gendreau et al. [13]. Our computational experiments show that DMTS gets better results for 20 out of the 27 cases. Additional experiments on the newer set of 12 instances devised by Tarantilis et al. [27] provides further evidence of the superiority of DMTS.

The remainder of this paper is organized as follows. After providing an overview of the relevant existing literature concerning the 3L-CVRP in Section 2, we formally define the problem in Section 3. Solving this problem requires the solution of the single vehicle loading sub-problem, and we provide a procedure to do so in Section 4, which makes use of two heuristics. Section 5 describes the Deepest-Bottom-Left-Fill (DBLF) heuristic and the Maximum

* Corresponding author at: School of Management, Huazhong University of Science and Technology, No. 1037, Luoyu Road, Wuhan, China.
Tel.: +852 64117909; fax: +852 34420189.
E-mail addresses: i@zhuwb.com (W. Zhu), tigerqin@cityu.edu.hk (H. Qin).

0305-0548/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2011.11.001

Touching Area (MTA) heuristic. Next, Section 6 provides the details of a two-phase tabu search approach that makes use of these heuristics to solve the 3L-CVRP. We compare its performance with the current best approaches in Section 7. Finally, we present our conclusions in Section 8.

2. Related work

The vehicle routing problem has many variants [28]. All vehicle routing problems share two common aspects, namely *routing* and *loading*. The goal of routing is to determine a sequence that visits all customers with the minimum total travel cost. The goal of loading is to maximize the space utilization when loading goods onto a vehicle, subject to various loading constraints.

The loading aspect of many variants of the Capacitated Vehicle Routing Problem (CVRP) has usually been simplified. Instead of considering the three dimensional aspect of each item, a scalar value is used to represent the volume of each item; as long as the total volume of items that are to be loaded into the vehicle does not exceed the vehicle's capacity, it is assumed that loading is possible. Furthermore, practical factors such as loading and unloading time are usually not considered. These variants are often viewed as extensions of the Traveling Salesman Problem. A large body of literature exists for the CVRP, including lower bounds, exact algorithms and heuristics [28]. Metaheuristics have often been found to perform well on large instances [7].

The CVRP is a good approximation of the practical environment where goods are relatively small and have similar dimensions (i.e., *homogeneous* items), e.g., shipping goods from factories to warehouses, and from central warehouses to large stores. In such applications, loading can be approximated with high accuracy by considering the volume alone. However, when shipping bulky items, especially if their dimensions differ (i.e., *heterogeneous* items), the shapes of the items play an important role; ignoring the shapes of the items may easily result in a solution that cannot be realized physically. In the most extreme situation, if items have dimensions that are slightly larger than half the length, width, and height of the loading space, only one item can be loaded into a vehicle, resulting in a volume utilization of about 12.5%. As such, using volume can be a very poor measure to approximate whether a set of items can be packed into a container.

The Two-Dimensional Loading Capacitated Vehicle Routing Problem (2L-CVRP) assumes that goods cannot be stacked, and therefore it only considers the base area of each item. It was first studied by Iori et al. [17], who devised a branch and bound algorithm to solve small instances of the problem exactly. Metaheuristic approaches based on Tabu Search [14] and Ant Colony Optimization [29] were subsequently proposed to address this problem for instances of practical size. Malapert et al. [21] considered the two-dimensional loading constraint and the Last-In-First-Out constraint (LIFO) in the context of the Pickup and Delivery Routing Problem and proposed a constraint programming model to solve this problem.

Gendreau et al. [13] were the first to consider the vehicle routing problem with three-dimensional loading constraints. A tabu search approach was proposed to address the 3L-CVRP problem, where the three-dimensional loading sub-problem was also solved by a tabu search metaheuristic. Fuellerer et al. [12] found that using metaheuristics to address the loading sub-problem is not a time efficient approach, and therefore employed a local search to solve the sub-problem. It was found that the time saved on the sub-problem can be better utilized in an ant colony optimization routine to find a higher quality overall solution to the 3L-CVRP.

Moura and Oliveira [24] studied the combination of the Vehicle Routing Problem with Time Window (VRPTW) and the Container Loading Problem. Both a sequential and a hierarchical approach were proposed; the two approaches were compared using a set of instances generated based on the well-known Solomon instances for VRPTW and the Bischoff and Ratcliff test problems for the Container Loading Problem.

The loading aspect of 3L-CVRP is related to container loading problems. Various practical constraints on the supporting surface or the fragility of the items are often considered [2,9,26]. Although many exact algorithms for the variants of the container loading problem exist [8,11,22,23], they are usually too slow for problems of practical size, e.g., the standard 20-foot container with about 200 boxes per container. To date, the best results on problems of reasonable size are held by heuristic-based methods [19,18,25,10].

Gendreau et al. [14], Tarantilis et al. [27], Fuellerer et al. [12] tackled the loading aspect of 3L-CVRP by employing *sequence-based* heuristics that convert loading sequences into packing plans. To generate loading sequences, Gendreau et al. [14] used a tabu search, Fuellerer et al. [12] used a local search, whereas Tarantilis et al. [27] devised three sorting rules to produce three loading sequences. Sequence-based approaches are suitable for 3L-CVRP because they mimic the actual loading process of the items. As a result, the constraints for the problem can be easily handled by checking each item as they are loaded. In contrast, handling these constraints using other types of approaches like block-building [10,25] or divide-and-conquer methods [20,5] is significantly more difficult.

3. Problem description

Let $G=(V,E)$ be an undirected graph, where $V=\{0,\ldots,n\}$ is the set of $n+1$ vertices corresponding to a depot, represented by vertex 0, and n clients, denoted by vertices $1,\ldots,n$; and E is the set of edges. The cost of an edge (i,j) is denoted by c_{ij}. There are v identical vehicles available; each vehicle has a weight capacity D and a three-dimensional rectangular loading space $S=W\times H\times L$ defined by width W, height H and length L. Each client i $(i=1,\ldots,n)$ requires the delivery of a set of m_i three-dimensional items I_{ik} $(k=1,\ldots,m_i)$ having width w_{ik}, height h_{ik} and length l_{ik} with total weight d_i.

In 3L-CVRP, we assume all items are rectangular boxes. The items can only be placed orthogonally inside a vehicle; however, items can be rotated by 90° on the width–length plane. Some items are fragile; only fragile items can be placed on top of other fragile items, whereas any item can be placed on top of a non-fragile item. The stability of the packed items is important; one method to ensure stability is to require items that are placed on top of other items to have sufficient supporting areas. A packing is *feasible* if all items are either placed directly on the floor of the vehicle or on top of other items with total supporting area of at least a percent of their base areas. Another important requirement is to ensure that items can be easily unloaded; this is approximated by a Last-In-First-Out (LIFO) policy. When client i is visited, all of its corresponding items I_{ik} must not be stacked beneath nor be blocked by items of clients that are to be visited later; an item A is beneath B if the interior of the projections of their bases on the vehicle floor intersect, and the top of A is not higher than the bottom of B in the vertical direction. An item is also considered blocked if it will overlap any item of a later client when it is moved along the L axis toward the rear door.

The objective of 3L-CVRP is to find a set of at most v routes (one per vehicle) such that:

(1) Every vehicle starts from the depot, visits a sequence of clients and returns to the depot.

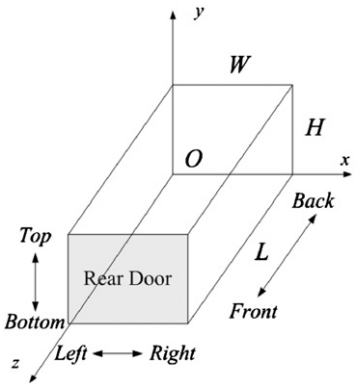

Fig. 1. 3D coordinate system.

(2) All clients are served, and every client is served by exactly one vehicle.
(3) No vehicle carries a total weight that exceeds its capacity.
(4) All items demanded by all the clients served by a vehicle can be orthogonally packed into that vehicle while satisfying the following *loading constraints*:
  (4.a) (*Fragility constraint*) no non-fragile items are placed on top of fragile items.
  (4.b) (*Supporting area constraint*) all items have a supporting area of at least a percent of their base area.
  (4.c) (*LIFO constraint*) all items fulfill the LIFO policy.
(5) The total cost of all edges included in the routes is minimized.

Tarantilis et al. [27] proposed a manual loading variant of 3L-CVRP and named it M3L-CVRP. The only difference between M3L-CVRP and 3L-CVRP is the interpretation of LIFO constraint. We consider both variants in this study:

3L-CVRP: The LIFO constraint is violated if a later item B *stacks over* an earlier item A, i.e., the height of the base of B $\geq$ the height of the top of A and the projections of A and B overlap on the W–L plane. This policy reflects the unloading of items by forklifts, where items are first elevated and then moved toward the unloading door.
M3L-CVRP: The LIFO constraint is violated if a later item B *stacks on top of* an earlier item A, i.e., the height of the base of B = the height of the top of A and the projections of A and B overlap on the W–L plane. This approximates the unloading of items manually.

In this study, we use the following Cartesian coordinate system (Fig. 1). The loading space of the vehicle is in the first octant of the coordinate system where the origin is at the deepest, bottommost, leftmost corner. The width W is parallel to the x-axis, the height H is parallel to the y-axis, and the length L is parallel to the z-axis. The terms *left*, *right*, *top*, *bottom*, *back* and *front* are as illustrated in the diagram.

4. Procedure for the single vehicle loading problem

In the process of solving 3L-CVRP, we need to repeatedly decide whether all items of the clients visited by a vehicle can be loaded into the loading space of the vehicle. In our approach, we treat the loading space of the vehicle as an open-ended bin (i.e., the length can be extended on demand), and devise a local search procedure to load all the items while minimizing the length of the loading space. If the length required does not exceed the length of the vehicle, then all items can be accommodated into the vehicle.

Our local search procedure is given in Algorithm 1. We first generate a loading sequence of the given items, and then use either the Deepest-Bottom-Left-Fill heuristic (DBLF) or the Maximum Touching Area heuristic (MTA) to convert the loading sequence into a loading plan. Both DBLF and MTA will load one item at a time according to the loading sequence and report the minimum length of the loading space required to accommodate all items.

The initial loading sequence is constructed by taking the various loading constraints into account. We sort the items by applying the following sorting rules in order:

SR1: (applies only if the LIFO constraint is to be enforced) Sort the items in reverse order of the clients to be visited. Hence, the items for later customers will be loaded first. Ties are broken using the next rule SR2.
SR2: (applies only if fragility constraint is to be enforced) Sort the items so that non-fragile items are before fragile items. We would like to load the non-fragile items first since fragiles item can be placed on top of non-fragile items, but the reverse is prohibited. Ties are broken using the next rule SR3.
SR3: Sort the items by decreasing order of volume.

Our local search will generate up to K loading sequences for DBLF. If any of the loading sequences leads to a plan that accommodates all items, the local search stops immediately; otherwise, it will generate up to K loading sequences for MTA. Each new loading sequence is generated using the previous loading sequence by swapping a pair of randomly selected items.

We would like to determine the value of K based on the difficulty of the loading instance. For difficult instances where the procedure is unlikely to find a feasible solution, it may be beneficial to reduce K since our computation time may be better spent elsewhere. In our implementation, we estimate the difficulty of the loading instance by the proportion of the total volume of the items to the volume of the loading space. Let r be this value; if $r \leq 0.6$, then we set $K=150$; otherwise, $K = \max\{5, 75 \times (1-r)\}$.

Algorithm 1. Load a list L of items into a vehicle.

```
Load-Vehicle(custList)
    // Input: custList—the list of customers visited by a vehicle
    // Output: the minimum length of the loading space to
    accommodate all items
1   if total volume or weight of all items exceeds capacity of
    vehicle
2     return FAILURE
3   L = the initial loading sequence constructed by sorting
    rules SR1–SR3
4   m = number of items
5   K = value calculated using estimated difficulty of loading
    instance
6   for k=1 to K
7     len = DBLF(L)
8     if len ≤ length of vehicle
9       return SUCCESS
10    Select integers i,j uniformly randomly from [1,m]
11    Swap L_i and L_j
12  for k=1 to K/3
13    len = MTA(L)
14    if len ≤ length of vehicle
15      return SUCCESS
16    Select integers i,j uniformly randomly from [1,m]
17    Swap L_i and L_j
18  return FAILURE
```

5. The enhanced DBLF and MTA heuristics

Both the Deepest-Bottom-Left-Fill (DBLF) heuristic and the Maximum Touching Area (MTA) heuristic load items one at a time according to a given loading sequence. The difference lies in how a position is selected (from a set of candidates) to place the current item. In DBLF, the position with the minimum lexicographical order of z-, y-, and x-coordinates is selected; in MTA, the position that maximizes the contact area of the current item with respect to the placed items and the faces of the loading space is selected. The implementations of DBLF and MTA by Gendreau et al. [13], Fuellerer et al. [12], and Tarantilis et al. [27] maintain only $O(n)$ candidate positions, where n is the number of items already placed. In contrast, our approach enhances the DBLF and MTA heuristics by considering all points in the loading space of the vehicle, which may find feasible loading plans that are not found by existing implementations (see Appendix A). Although there are an uncountably infinite number of points to place an item when the coordinates are real numbers, we show that it is sufficient to consider only $O(n^3)$ points, which can be enumerated in $O(n^3)$ time. Hence our implementation is able to convert a loading sequence that consists of n items into a loading plan in $O(n^4)$ time.

Given n items already placed, our idea is to construct $O(n)$ planes perpendicular to the x-, y-, and z-axis that we call x-, y- and z-planes, respectively. The x-, y- and z-planes divide the loading space of the vehicle into $O(n^3)$ blocks. The intersection of exactly one x-plane, one y-plane and one z-plane is a *grid point*; there are a total of $O(n^3)$ grid points. We will show that the position selected by DBLF is either one of the grid points (when the supporting area constraint is not enforced) or can be associated with one of the grid points (when the supporting area constraint is enforced). In the latter case, the position can be calculated in constant time after the associated grid point is identified. The position selected by MTA is always one of the grid points. We will also show that all $O(n)$ grid points that share the same z- and y-coordinates can be enumerated in $O(n)$ time, which is the key to enumerating the $O(n^3)$ grid points in $O(n^3)$ time.

The following notations that will be used in our discussions. An item with dimensions $w_i \times h_i \times l_i$ placed in a vehicle can be identified by a vector $(x_i,y_i,z_i,\Delta x_i,\Delta y_i,\Delta z_i)$, where (x_i,y_i,z_i) are the coordinates of the corner closest to the origin, and $(\Delta x_i,\Delta y_i,\Delta z_i)$ is a permutation of (w_i,h_i,l_i) denoting the lengths of the item along the x-, y-, and z-axis, respectively. Given this vector, we say that an item i is *placed at* (x_i,y_i,z_i) with *orientation* $(\Delta x_i,\Delta y_i,\Delta z_i)$. For ease of presentation, we may omit some coordinates in our discourse when they are clear from the context, e.g., we may simply say that "an item i is placed at x_i" when the coordinates y_i and z_i and the orientation are obvious. We use the ordered set X to denote (the x-coordinates) of the x-planes; similarly, the ordered sets Y and Z represent the y- and z-planes, respectively.

The four heuristics (i.e., the DBLF and MTA heuristics with and without the supporting area constraint) share a similar structure; we will use the DBLF heuristic without the supporting area constraint as our illustrative example (Algorithm 2). The algorithm maintains the placed items in a set called *placedItems*. The outer for-loop (line 3) attempts to load one item in each iteration. The second for-loop (line 6) enumerates all allowed orientations when placing the current item. For a given orientation $(\Delta x_i,\Delta y_i,\Delta z_i)$, the inner for-loop (line 7) locates the DBL position for the current item i by enumerating the positions in ascending order of their z-, y-coordinates. For a given z-, y-coordinate pair, the Find-Leftmost-Pos procedure will find the position with minimum x-coordinate that can accommodate the item, or return NIL if no such position exists. Since the positions are enumerated in ascending order of z-, y-, x-coordinates, the first position found is the DBL position for the given orientation, whereupon we can skip all subsequent positions (line 11). Once all orientations of item i are tried, we record the DBL position in *bpos* and its corresponding orientation in *ort*.

To eliminate the necessity of handling boundary cases, we create six dummy items (of zero volume) to represent the six faces of the loading space (line 1). For example, the dummy item for right face is of dimensions $0 \times H \times L$ and placed at location $(L, 0, 0)$.

Algorithm 2. DBLF heuristic.

```
DBLF(List)
    // Input: List—the list of m items to be loaded
    // Output: the minimum length of the loading space to
    accommodate all items
1   placedItems = dummy items representing the faces of the
    loading space
2   Initialize X, Y, Z according to placedItems
3   for every item i in List
4     best position bpos = NIL
5     best orientation ort = NIL
6     for each allowed orientation (Δx_i,Δy_i,Δz_i) of item i
7       for each z ∈ Z,y ∈ Y
8         pos = Find-Leftmost-Pos(X,(Δx_i,Δy_i,Δz_i),y,z)
9         if pos ≠ NIL
10          update bpos and ort if pos is better than bpos
11          goto line 3 to try next orientation
      // place item i at bpos using orientation ort
12    append (bpos,ort) to the end of placedItems
13    update X, Y, Z
14  return the largest (z_i + Δz_i)
```

We now proceed to show how we construct the x-, y-, and z-planes, and efficiently implement the Find-Leftmost-Pos procedure for various scenarios.

5.1. DBLF without supporting area constraint

To correctly and efficiently implement the Find-Leftmost-Pos procedure, which finds the leftmost position that can accommodate item i with the specified y- and z-coordinates, we need to construct the x-, y- and z-planes given the placed items and the orientation $(\Delta x_i,\Delta y_i,\Delta z_i)$ of the item to be placed. When the supporting area constraint is not enforced, the y-planes (resp. z-planes) are the top (resp. front) faces of the placed items. The x-planes consists of two types:

- $X_1 = \{x = x_q - \Delta x_i \mid q \in placedItems\}$: the locations where the right face of the current item coincides with the left face of a placed item.
- $X_2 = \{x = x_q \mid q \in placedItems\}$: the left faces of the placed items.

Note that the set of grid points are a superset of the well-known *normal positions* in the existing literature (also known as bottom left stable positions in 2D).

Given an orientation $(\Delta x_i,\Delta y_i,\Delta z_i)$ of the current item, we define the following functions to indicate whether the current item can be placed at some position (x, y, z):

$$o(x,y,z) = \begin{cases} 0 & \text{placing item } i \text{ at } (x,y,z) \text{ violates the non-overlapping constraint} \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

$$f(x,y,z) = \begin{cases} 0 & \text{placing item } i \text{ at } (x,y,z) \text{ violates the fragility constraint} \\ 1 & \text{otherwise} \end{cases} \quad (2)$$

$$g(x,y,z)=\begin{cases}0 & \text{placing item } i \text{ at } (x,y,z) \text{ violates the LIFO constraint}\\ 1 & \text{otherwise}\end{cases} \tag{3}$$

The three indicator functions $o(x, y, z)$, $f(x, y, z)$, and $g(x, y, z)$ share similar properties that allow us to efficiently compute them. We will use $o(x, y, z)$ for illustration.

When the y, z values are fixed, $o(x, y, z)$ is a step function of x. This is because when the current item slides from left to right, the only occasion where $o(x, y, z)$ will switch from 1 to 0 is when the right face of the current item encounters the left face of a placed item q, and the only occasion where $o(x, y, z)$ will switch from 0 to 1 is when the left face of the current item leaves the right face of a placed item q. Since two items are considered overlapping only if their interiors intersect, the set of x values corresponding to $o(x,y,z)=1$ are closed intervals and the set of x values corresponding to $o(x,y,z)=0$ are open intervals. The endpoints of the intervals are $x \in X_1 \cup X_2$.

The values for the indicator function $o(x, y, z)$ can be computed in linear time for fixed y,z and all $x \in X$ (Algorithm 3). Since each x-plane in X is contributed by some placed item, we store the contributing item along with the x-coordinate, i.e., X is implemented as a list of pairs (x,q), where q is the placed item that contributed an x-plane with coordinate x and the list is sorted in ascending order of x.

Throughout the algorithm, the value λ is the right boundary of the enumerated placed items, i.e., when the current item is placed, its x-coordinate must be at least λ to avoid overlapping with the placed items (that have already been enumerated). The value of λ is updated as the placed items are enumerated based on the following fact: if placing the current item slightly to the right of x will overlap with some placed item q, then the current item will continue to overlap with the item q until $x \geq x_q+\Delta x_q$ (line 8).

Algorithm 3. Computation of indicator function $o(x,y,z), \forall x \in X$.

```
Compute-Indicator-O(X,y,z,(Δx_i,Δy_i,Δz_i))
1  λ = −∞
2  Initialize an empty associative array O
3  for each (x,q) in X
4     if x < λ
5        O[x] = 0
6     else O[x] = 1
       // δ is a small positive constant
7     if placing item i at x+δ overlaps with item q
8        λ = max(λ,x_q + Δx_q)
9  return O
Find-Leftmost-Pos(X,(Δx_i,Δy_i,Δz_i),y,z)
1  O = Compute-Indicator-O(X,y,z,(Δx_i,Δy_i,Δz_i))
2  F = Compute-Indicator-F(X,y,z,(Δx_i,Δy_i,Δz_i))
3  G = Compute-Indicator-G(X,y,z,(Δx_i,Δy_i,Δz_i))
4  for each (x,q) in X
5     if O[x]+F[x]+G[x] = 3
6        return (x, y, z)
7  return NIL
```

To compute $f(x, y, z)$ and $g(x, y, z)$, we simply replace line 7 to check the fragility and LIFO constraints; we call the resultant algorithms Compute-Indicator-F and Compute-Indicator-G, respectively.

The Find-Leftmost-Pos procedure finds the leftmost position to place the current item, i.e., the smallest $x \in X$ such that $o(x,y,z)=1, f(x,y,z)=1$ and $g(x,y,z)=1$. Hence, we first compute $o(x,y,z), f(x,y,z)$ and $g(x,y,z)$ for all $x \in X$, which takes $O(n)$ time since $|X|$ is exactly twice the number of the placed items. We use an associative array to store the values of the indicator functions, which allow us to access $O[x]=o(x,y,z)$ in constant time (line 5). Since we only access the associative array in a sequential manner (both writing during the computation of the indicator functions and reading in the Find-Leftmost-Pos procedure are done sequentially), we can implement it as list of pairs $(x,o(x,y,z))$ sorted in ascending order of x. The Find-Leftmost-Pos procedure can therefore report the leftmost position in time linear to the number of placed items.

5.2. DBLF with supporting area constraint

To handle the supporting area constraint, we define a function $s(x, y, z)$ to denote the supporting area of the current item when placed at (x, y, z). We will show that the local maxima of this function are at the grid points. This implies that the positions with enough supporting area can be found "close to" the grid points. The key is to compute this function efficiently for all grid points.

To do so, we require finer grids than when the supporting area constraint is not enforced, which we define as follows. The y-planes are the top faces of the placed items. Let i be the current item that we wish to place in orientation $(\Delta x_i,\Delta y_i,\Delta z_i)$. The x-planes consist of the following four types:

- $X_1=\{x=x_q-\Delta x_i \mid q \in placedItems\}$: the locations where the right face of the current item coincides with the left face of a placed item.
- $X_2=\{x=x_q \mid q \in placedItems\}$: the left faces of placed items.
- $X_3=\{x=x_q+\Delta x_q-\Delta x_i \mid q \in placedItems\}$: the locations where the right face of the current item coincides with the right face of a placed item.
- $X_4=\{x=x_q+\Delta x_q \mid q \in placedItems\}$: the right faces of placed items.

Since the z-axis plays a symmetric role as the x-axis as far as supporting area is concerned, the four types of z-planes are similarly constructed; we simply replace "x" by "z", "right face" by "front face" and "left face" by "back face" in the above definitions.

We now precisely define the notion of *supporting area*. Fig. 2 shows an item i that is placed on top of an item q. The bottom face of item i is in contact with the top face of item q (i.e., the two faces lie on the same plane and overlap on that plane). The area of the overlap is the supporting area contributed by the placed item q. We use $A_q(x,y,z)$ to denote the supporting area contributed by the placed item q when the current item i is placed at (x, y, z). The total supporting area of the current item when placed at (x, y, z) is

$$s(x,y,z)=\sum_{q \in placedItems} A_q(x,y,z) \tag{4}$$

The supporting area $A_q(x,y,z)$ (the shaded area in Fig. 2(b)) can be thought of as the area swept by line segment s_{iq} starting from the left edge of the top face T_q of item q end at the right edge of the bottom face B_i of the current item i. From this perspective, we can devise an efficient algorithm to compute $s(x, y, z)$ for all $x \in X$ given fixed y,z.

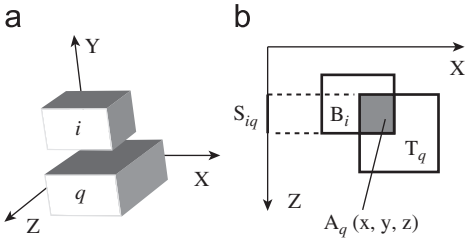

Fig. 2. An example of supporting area. (a) 3D view. (b) Top-down view.

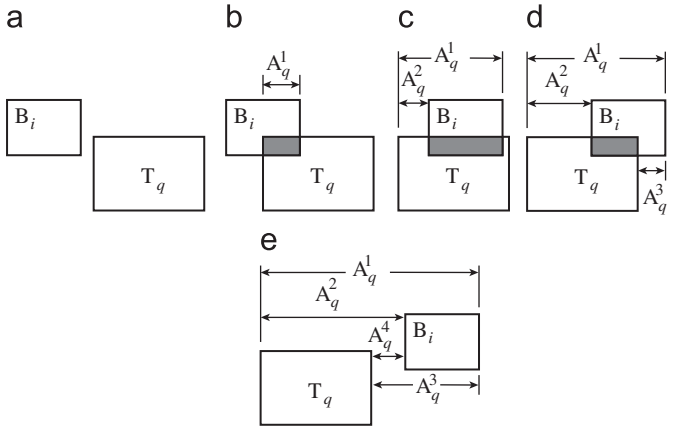

Fig. 3. Calculation of supporting area. (a) $A_q = A_q^1 = A_q^2 = A_q^3 = A_q^4 = 0$. (b) $A_q^2 = A_q^3 = A_q^4 = 0$. (c) $A_q^3 = A_q^4 = 0$. (d) $A_q^4 = 0$.

Assume item q is placed at (x_q, y_q, z_q) with orientation $(\Delta x_q, \Delta y_q, \Delta z_q)$. We can decompose $A_q(x,y,z)$ into four components:

(1) $A_q^1(x,y,z)$ is the area swept by s_{iq} from the left side of T_q to the right side of B_i if $x_q < x + \Delta x_i$; 0 otherwise.
(2) $A_q^2(x,y,z)$ is the area swept by s_{iq} from the left side of T_q to the left side of B_i if $x_q < x$; 0 otherwise.
(3) $A_q^3(x,y,z)$ is the area swept by s_{iq} from the right side of T_q to the right side of B_i if $x_q + \Delta x_q < x + w_i$; 0 otherwise
(4) $A_q^4(x,y,z)$ is the area swept by s_{iq} from the right side of T_q to the left side of B_i if $x_q + \Delta x_q < x$; 0 otherwise.

The following decomposition lemma can be verified by inspecting various scenarios listed in Fig. 3:

Lemma 1. *Given fixed y, z, regardless of the relative position of item i and item q on the x-axis, the following identity holds*:

$$A_q(x,y,z) = A_q^1(x,y,z) - A_q^2(x,y,z) - A_q^3(x,y,z) + A_q^4(x,y,z) \tag{5}$$

Correspondingly, we can decompose the supporting area function $s(x, y, z)$ into four components

$$s(x,y,z) = s^1(x,y,z) - s^2(x,y,z) - s^3(x,y,z) + s^4(x,y,z) \tag{6}$$

$$s^r(x,y,z) = \sum_{q \in placedItems} A_q^r(x,y,z), \quad r = 1,2,3,4 \tag{7}$$

Given fixed values of y and z, the rth component of the supporting area $A_q^r(x,y,z)$ has the following properties that enable us to compute their sum s^r efficiently:

- $A_q^r(x,y,z)$ is a piecewise linear function that consists of exactly two pieces. The special case where item q never contributes to the supporting area of item i can also be treated as two pieces with slope zero.
- Each $A_q^r(x,y,z)$ has a *activation point* such that before this point, the value of $A_q^r(x,y,z)$ is always zero; after this point, $A_q^r(x,y,z)$ is a linear function with constant slope and the slope is the length of line segment s_{iq}. The activation points are in fact the rth type of x-plane we defined in the beginning of this section.

Each of the four components can be computed using the same algorithm. The Compute-Support-Area-Component (Algorithm 4) computes the rth component of the supporting area function $s^r(x,y,z), \forall x \in X_r$ and fixed y, z. We assume all x-planes in X_r have been sorted in ascending order of x-coordinate, and each x-plane is stored as a pair (x, q) where x is the coordinate and q is the placed item that contributed the x-plane. As we slide the current item i from left to right, whenever item i encounters an x-plane contributed by some item q (i.e., the item q may start contributing to the supporting area of item i), we increase the slope of $s^r(x,y,z)$.

Algorithm 4. Computation of $s^r(x,y,z), \forall x \in X_r$.

```
Compute-Support-Area-Component(X_r,(Δx_i,Δy_i,Δz_i),y,z)
1   a = 0 // the slope of the current piece
2   t = 0 // accumulated sum
3   Initialize an empty associative array S^r
4   for each (x,q) ∈ X_r
5     s_iq = [z,z+Δz_i] ∩ [z_q,z_q+Δz_q]
6     if y_q + Δy_q = y and |s_iq| > 0
7       a = a + |s_iq|
      // x^- is the element precedes x in X
8   t = t + a · (x − x^-)
9   S^r[x] = t
10  return S^r
```

Once $s^r(x,y,z)$ has been computed for each $x \in X_r$, we can simply merge the results to obtain $s(x,y,z), \forall x \in X$. Note that in the process of merging, we will require the value of $s^r(x,y,z)$ for some $x \notin X_r$. Since s^r is a linear function, these values can be obtained by interpolation in constant time.

We now proceed to show that the DBL positions that ensure sufficient support for the current item must be "close to" one of the grid points:

Theorem 1. *If (x_i, y_i, z_i) is the best position for the current item i with respect to the DBL rule and satisfies the non-overlapping, fragility, and LIFO constraints, then either $x_i \in X$ or $z_i \in Z$ or both.*

Proof. Assume the contrary, i.e., the best position $a = (x_i, y_i, z_i)$ is in the interior of a grid square (Fig. 4). The position a satisfies the non-overlapping constraint, the fragility constraint, and the LIFO policy. We can easily verify that for the indicator function o, $o(x_1, y, z_1) = o(x_2, y, z_2)$ if (x_1, z_1) are in the same grid square as (x_2, z_2); the same can be said for the indicator function f and g. Hence, any position (in particular a, b, c, d) in the same grid square will also satisfy the non-overlapping constraint, the fragility constraint, and the LIFO policy.

By our construction, $s(x, y, z)$ is a linear function of x when y and z are fixed, and the slope within a grid square remains constant. Similarly, $s(x, y, z)$ is a linear function of z when x and y are fixed, and the slope within a grid square remains constant.

Suppose we slide item i from position a leftwards along the line $z = z_i$ within the grid square. The supporting area $s(x, y_i, z_i)$ is a linear function of x. If the slope is not positive, then position b

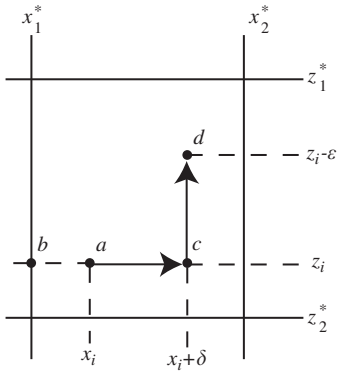

Fig. 4. Illustration for proof of Theorem 1 within a grid defined by two consecutive x-planes x_1^*, x_2^* and two consecutive z-planes z_1^*, z_2^*.

(corresponding to a smaller value for x) will have a supporting area that is larger than or the same as a, which implies that b satisfies the supporting area constraint. Since b is a feasible position to place item i, this contradicts the fact that (x_i,y_i,z_i) is the leftmost position. Hence, the slope must be positive.

Now suppose we slide item i along the line $z=z_i$ by δ toward the right to position c. Since the slope is positive, the supporting area at position c is strictly greater than the required threshold. At this point, we can slide the item i along $x=x_i+\delta$ by a small amount ϵ to reach position d. Since $s(x_i+\delta,y_i,z)$ is a linear function of z, we will be able to find such a position d with supporting area that is greater than or equal to the supporting area at a. Since the position d is deeper than (x_i,y_i,z_i), therefore a does not respect the DBL rule.

Hence, either $x_i \in X$, or $z_i \in Z$. □

Let x^+ be the smallest element in X that is greater than or equal to x, and z^+ be the smallest element in Z that is greater than or equal to z. Using a similar argument as the proof of Theorem 1, the following lemma holds.

Lemma 2. *If (x, y, z) is the best position for the current item i that honors the DBL rule and satisfies all loading constraints, then (x^+,y,z^+) is a feasible position to place item i.*

The direct implication of Theorem 1 and Lemma 2 is that there exists a grid point $(x,z) \in X \times Z$ such that the best position for item i, (x_i,y,z_i) is either the result of pushing item i from location (x, y, z) along the z-axis to its deepest position in the grid square, or pushing item i from location (x, y, z) along the x-axis to its leftmost position in the grid square. Consequently, we only need to search for feasible positions among grid points $(x,y,z),\forall(x,z) \in X \times Z$.

We are now ready to modify the DBLF heuristic to handle the supporting area constraint. Firstly, we update the Find-Leftmost-Pos procedure so that it will try to find the leftmost x that provides enough support for the current item; the updated version is given in Algorithm 5. We will compute $s(x,y,z),\forall x \in X$ immediately after computing the three indicator functions (line 4). In line 7, we also check that the position provides sufficient support. Once a feasible grid point (x, y, z) is found for the current item, if the supporting area at the grid point is larger than required, we will try to push the current item to the left along the x-axis or to the back along the z-axis to obtain the DBL position (line 8). Let z^- be the element immediately preceding z in Z. Since o,f,g are step functions, if $o(x,y,z^-)+f(x,y,z^-)+g(x,y,z^-)=3$, we can push the current item backwards along the z-axis to obtain a deeper feasible position (line 9); if we keep the indicators computed for the preceding z-coordinate, we can access $o(x,x,z^-)$ in constant time. Similarly, if $o(x^-,y,z)+f(x^-,y,z)+g(x^-,y,z)=3$, we can push the current item to the left along the x-axis to obtain a feasible solution (line 11). Note that we must scan all grid points since it is possible that a grid point with larger x may result in a feasible position with smaller z, which is a better position according to the DBL rule.

Algorithm 5. Finding the leftmost position with enough support.

```
Find-Leftmost-Pos(X,(Δx_i,Δy_i,Δz_i),y,z)
1   O = Compute-Indicator-O(X,y,z,(Δx_i,Δy_i,Δz_i))
2   F = Compute-Indicator-F(X,y,z,(Δx_i,Δy_i,Δz_i))
3   G = Compute-Indicator-G(X,y,z,(Δx_i,Δy_i,Δz_i))
4   S = Compute-Support-Area(X,y,z,(Δx_i,Δy_i,Δz_i))
5   bpos = NIL
6   for each (x,q) in X // sorted in ascending order of x
      // a · Δx_i · Δz_i: required supporting area
7     if O[x]+F[x]+G[x] = 3 and S[x] ≥ a · Δx_i · Δz_i
        // a feasible grid point (x, y, z) has been found for item i
8       if S[x] > a · Δx_i · Δz_i
          // z⁻: the element immediately preceding z in Z
9         if o(x,y,z⁻)+f(x,y,z⁻)+g(x,y,z⁻) = 3
10          Push item i backwards along the z-axis
          // x⁻: the element immediately preceding x in X
11        if O[x⁻]+F[x⁻]+G[x⁻] = 3
12          Push item i to the left along the x-axis
13        update bpos if a better position is found
14  return bpos
```

5.3. Maximum Touching Area heuristic

The MTA heuristic (Algorithm 6) is largely similar to DBLF without support constraint. The two main differences are: (1) for the given z- and y-coordinates, we need to find a position that maximizes the touching area (by invoking the Find-MaxTouchingArea-Pos procedure instead of the Find-Leftmost-Pos procedure (line 8)); and (2) we have to scan all grid points before we can determine which grid point is best. We will use exactly the same sets of x-, y- and z-planes as DBLF.

Algorithm 6. The Maximum Touching Area heuristic.

```
MTA(List)
    // Input: List—the list of m items to be loaded
    // Output: the minimum length of the loading space to
    accommodate all items
1   placedItems = dummy items representing the faces of the
    loading space
2   Initialize X, Y, Z according to placedItems
3   for every item i in List
4     best position bpos = NIL
5     best orientation ort = NIL
6     for each allowed orientation (Δx_i,Δy_i,Δz_i) of item i
7       for each z ∈ Z,y ∈ Y
8         pos = Find-MaxTouchingArea-Pos(X,(Δx_i,Δy_i,Δz_i),y,z)
9         if pos ≠ NIL
10          update bpos and ort if pos is better position
      // place item i at bpos using orientation ort
11    append (bpos,ort) to the end of placedItems
12    update X, Y, Z
13  return the largest (z_i+Δz_i)
```

The correctness and efficiency of the MTA heuristic relies on the fact that the position that maximizes the touching area must coincide with a grid point. When the current item is placed into the vehicle, all six faces may be in contact with some of the placed items. We have shown that the touching area of the bottom face (which is the supporting area) $s(x,y,z),\forall x \in X$ can be computed in linear time when y, z are fixed in the previous subsection by decomposing the function into four components. The same technique can be used to compute the touching areas of the top, front and back faces of the current item, so these touching areas are all piecewise linear functions with local optimal achieved at grid points. For the left face, the contact area is non-zero only when the current item is placed with its left face touching right faces of the placed items, i.e., $x \in X_4$. Similarly, the contact area of the right face is non-zero only when the current item is placed at some $x \in X_1$. Therefore, the position with maximum touching area for the current item must be at some grid point.

Algorithm 7. Finding the position with maximum touching area and enough support.

```
Find-MaxTouchingArea-Pos(X,(Δx_i,Δy_i,Δz_i),y,z)
```

O = Compute-Indicator-O(X,y,z,(Δx_i,Δy_i,Δz_i))
F = Compute-Indicator-F(X,y,z,(Δx_i,Δy_i,Δz_i))
G = Compute-Indicator-G(X,y,z,(Δx_i,Δy_i,Δz_i))
S = Compute-Support-Area(X,y,z,(Δx_i,Δy_i,Δz_i))
// compute touching area for all $x \in X$ with fixed y, z
T = Compute-Touching-Area(X,y,z,(Δx_i,Δy_i,Δz_i))
$bpos$ = NIL
for each (x,q) in X // sorted in ascending order of x
  // $a \cdot \Delta x_i \cdot \Delta z_i$: required supporting area
  if $O[x]+F[x]+G[x]=3$ and $S[x] \geq a \cdot \Delta x_i \cdot \Delta z_i$
    // a feasible grid point (x, y, z) has been found for item i
    update $bpos$ using (x, y, z) if the touching area $T[x]$ is larger
return $bpos$

5.4. Additional remarks

There are approaches for 2D packing that can potentially be adapted to 3D packing to form algorithms that better the running time of our straightforward approach, but it is unclear how such approaches can handle practical constraints. For example, Chazelle [4] developed an algorithm that can place all n items in the 2D case in $O(n^2)$ time, resulting in an amortized cost to place one item of $O(n)$. Unfortunately, the algorithm relies on the special structure of the unoccupied space in the container; this property does not hold beyond the 2D case, and therefore it cannot be generalized to higher dimensions. Another example is the technique devised by Healy et al. [15] that can place one item in $O(n \log n)$ time using a segment tree and a balanced binary search tree in the 2D case. However, a direct adaptation of this approach to 3D can only handle the basic non-overlapping constraint. Additionally, Hopper [16] implemented the Bottom-Left Fill (BLF) heuristic for the 2D case by first placing the current item such that its bottom left corner is at the top left or bottom right corner of some loaded item, and then slides the current item as far to the bottom and to the left as possible. This algorithm does not always honor the bottom-left criterion, which we show in Appendix B.

6. A two-phase tabu search for the 3L-CVRP

The 3L-CVRP is a combination of two hard combinatorial optimization problems, where even finding a feasible solution is non-trivial. To effectively handle the complexity of the problem, we address the feasibility and optimality of the solutions separately. We begin with an initial solution that is generated using a greedy heuristic. Should this initial solution be infeasible, the first phase of the tabu search focuses on finding a feasible solution. Once a feasible solution is located, the second phase focuses on minimizing the total traveling cost by exploring the neighborhood of the solution while maintaining feasibility.

6.1. Generation of initial solution

We adapted the savings algorithm for the CVRP [6] to construct our initial solution. This algorithm starts with one client per route and iteratively merges two routes into one until no more merging can be done. In each iteration, two routes that will result in the largest savings are merged, where the savings is computed as the difference between the total length of the merged route and the total length of the two routes before merging. We will accept a merging only if the resulting route is feasible, i.e., the total weight of all items does not exceed the weight capacity of the vehicle, and all items can be loaded into the vehicle without violating any loading constraints.

If after the first round of merging the number of routes exceeds the number of available vehicles, then we initiate another round of merging. This time, we allow the total weight of items in a route to exceed the total weight capacity of a vehicle, and the length of the loading space to exceed the actual length of the vehicle. In each iteration, the two routes that when merged will result in the largest saving are identified and merged. The second round of merging terminates when the number of routes equals the number of available vehicles.

6.2. Neighborhood operators

We employed five operators to explore the neighborhood of a current solution in our tabu search algorithm. The first operator is *2-opt*, which is frequently used in TSP and VRP literature. The 2-opt operator can be applied to a route with at least three clients. A pair of clients (i, j) is selected, whereupon the visiting order of all clients between i and j inclusive are reversed. Every pair (i, j), $i \neq j$ where 2-opt can be applied has an equal chance of being selected. In the TSP and the VRP, 2-opt can often eliminate the self-intersection of a route, thereby reducing the total length of the route. However, the probability of improvement is relatively small when the loading constraints are present, especially when the LIFO policy must be considered. This is because reversing the visiting order of a set of clients dramatically changes the loading plan, which often results in an infeasible solution.

The second operator is *2-swap*. Similar to 2-opt, 2-swap will select a pair of clients from a route with at least three clients. The visiting order of the selected pair is swapped, while the other clients on the route remain unchanged. Every pair (i,j), $i \neq j$ where 2-swap is applicable has the same probability of being selected. In symmetric TSP and VRP instances, the 2-swap operator introduces more dramatic changes than the 2-opt operator since it introduces four edges and deletes four edges whereas 2-opt only introduces two edges and deletes two edges. As a result, much existing literature regards 2-swap as a less effective neighborhood operator compared to 2-opt. However, in the 3L-CVRP problem with the LIFO constraint, 2-swap usually introduces milder changes to the solution compared to 2-opt in relation to the difference in the vehicle packing plan.

The third operator is *move*, which transfers a client from one route to another. We uniformly randomly pick a route R_i and a client from R_i to be moved. Next, we uniformly randomly select a route R_j,$j \neq i$, and an insertion point in the route. The selected client is deleted from R_i and inserted into R_j at the selected insertion point. This operator allows an extra route with only one client to be created. This operator may also move the only client in a route and hence reduce the number of vehicles required by one.

The fourth operator is *crossover*. Two routes R_i and R_j (with at least two clients) are uniformly randomly selected along with a splitting point from each route, such that both routes are split into a prefix sequence and postfix sequence (both non-empty). The prefix sequences are exchanged to form two new routes.

The fifth operator is *splitting* a route into two. A route R_i with at least two clients is selected along with a splitting point that divides the route into a non-empty prefix and a non-empty postfix sequence, whereupon these two sequences become separate routes. Although splitting usually produces a worse solution, it proved to be effective in escaping local optima, particularly when other operators cannot make further progress.

The five neighborhood operators 2-opt, 2-swap, move, crossover, and splitting are assigned a weight of 1000, 1000, 3000, 4500 and 500 respectively, which determines the relative probability that the neighborhood operator will be applied. These values were first estimated from our understanding of the relative

effects of the operators, and then adjusted empirically. Appendix C describes some sensitivity analysis experiments we performed on these weights, which showed that our final values produce relatively consistent and stable results.

6.3. Tabu list and tenure

The purpose of the tabu list is to prevent the search from cycling through a small set of solutions without making any progress. The simplest strategy is to record enough information when a neighborhood operator is applied so that the effect will not be undone in the following T iterations. The value T is called *tabu tenure*; we set T to 30 for both phases of our two-phase tabu search implementation.

For the neighborhood operators 2-opt, 2-swap and crossover, it is sufficient to record the clients (i,j) involved and the iteration in which the operator is applied. For reasons of efficiency, every neighborhood operator has its own tabu list. We used a two-dimensional array to implement each tabu list, where an element $a[i][j]$ with value k denotes that the operator was applied to client pair (i,j) in the kth iteration. For the move operator, we recorded the selected client i and the destination route R_j in the tabu list so that after a client is inserted into R_j, it will remain in route R_j for at least the tabu tenure period. No tabu list is needed for the splitting operator, since there is no direct way to undo its effect.

6.4. Phase one of tabu search

This phase will be invoked only if the initial solution is infeasible. In this phase, the search explores infeasible solutions with the primary goal of finding a feasible solution. Items assigned to a route in an infeasible solution may exceed the total weight capacity of the vehicle, and/or the length of the loading space required to load all items may exceed the length of the vehicle. We define the following objective function to capture the excess weight and excess length:

$$\text{objective value} = (\text{route length}) + \alpha(\text{excess weight}) + \beta(\text{excess length}) \tag{8}$$

$$\alpha = 20\bar{c}/D \tag{9}$$

$$\beta = 20\bar{c}/L \tag{10}$$

The objective value of a route is the sum of the total length, the excess weight penalized by α, and the excess length penalized by β. The quality of a solution is the sum of the objective value of all routes in the solution. A solution S_1 is considered better than a solution S_2 if either S_1 is feasible and S_2 is infeasible, or both S_1 and S_2 are infeasible with objective value $f(S_1) < f(S_2)$. The two parameters α and β are initialized by the characteristics of the input: $\bar{c}$ is the average cost of the edges, D is the weight capacity of a vehicle, and L is the vehicle length. The values of α and β are increased if no progress is made in 10 consecutive iterations: α will be increased by 50% if some route in the current solution exceeds the weight capacity D, and β will be increased by 50% if some route in the current solution requires a vehicle with length greater than L.

In this phase, only the 2-swap, move, and crossover operators are used to generate the neighborhood. A total of $N=500$ neighbors from the neighborhood of the current solution are randomly sampled in each iteration (the details are provided in Algorithm 8). An operator is selected with probability proportional to its weight, and applied to the current solution S to create a new solution S'. If S' is not prohibited by a tabu list, it is added to the candidate list NS. However, if S' is prohibited by the tabu list, then S' is first compared with S, and it is added to NS only if it is superior to S (this technique is known as aspiration).

The best neighbor is selected as the current solution for the next iteration. Phase one will repeat until a feasible solution is found, the time limit of 5000 CPU seconds has been exceeded, or a maximum of $3000 \times n$ iterations are reached, where n is the number of clients. If a feasible solution is found or $3000 \times n$ iterations have been reached, we continue to phase two. However, if a feasible solution cannot be found within the time limit, then the algorithm fails without invoking phase two.

Algorithm 8. Neighborhood generation.

```
GENERATE-NEIGHBORS(N, S)
    // Inputs:
    // N—number of neighbors to be generated
    // S—current solution
1   initialize solution list NS = ∅
2   while size of NS < N
3     select an operator based on relative weight
4     randomly generate a neighbor S' of S using the selected
    operator
5     if S' is not tabu or S' is better than S
6       insert S' into NS
7       update tabu list of the selected operator
8   return NS
```

6.5. Phase two of tabu search

This phase focuses on minimizing the total traveling cost by exploring only feasible solutions. All five operators are used to generate the neighborhood of the current solution using Algorithm 8; however, only feasible solutions are generated. In each iteration, a total of $N=500$ neighbors are generated, and the one with minimum total traveling cost is selected as the current solution for the next iteration. The best solution found after a total of $1000 \times n$ iterations is retained.

7. Computational experiments

The two-phase tabu search algorithm was coded in C++ and compiled using the g++ compiler. We call our approach the DBLF+MTA Tabu Search algorithm (DMTS). The algorithm was tested on a Dell server with an Intel Xeon E5520 2.26 GHz CPU, 8 GB RAM and running Linux using the 27 instances proposed by Gendreau et al. [13] (which we call G27) and the 12 instances generated by Tarantilis et al. [27] (which we call T12); the methods used for the generation of these instances are summarized in Appendix D. These two sets of benchmark instances are applicable to both the 3L-CVRP and M3L-CVRP; hence, we tested our DMTS algorithm on both sets of data and on both problem variants.

We compared the results of DMTS against the Tabu Search (TS) approach of Gendreau et al. [13], the Guided Tabu Search approach of Tarantilis et al. [27], and the Ant Colony Optimization (ACO) technique used by Fuellerer et al. [12]. The computational environments of these algorithms are summarized in Appendix E. All of these algorithms have reported results on G27 on 3L-CVRP, but only Tarantilis et al. [27] has reported results for T12 and for the M3L-CVRP variant. The detailed results are given in Appendix F.

7.1. Results on 3L-CVRP

The comparison with these approaches on G27 on 3L-CVRP is given in Table 1. The first column *Instance* is the instance name. For TS and GTS, only one execution was performed for each instance, so the column z reports the total traveling cost of the

Table 1
Comparison on Gendreau's 27 instances (3L-CVRP).

Instance	TS			GTS			ACO			DMTS			
	z	*t* (s)	*tt* (s)	*z*	*t* (s)	*tt* (s)	Avg *z*	Avg *t* (s)	Avg *tt* (s)	Avg *z*	Avg *t* (s)	Avg *tt* (s)	Impr (%)
E016-03m	316.32	129.5	1800.0	321.47	7.8	13.2	305.35	11.2	12.0	302.23	85.1	371.7	1.02
E016-05m	350.58	5.3	1800.0	334.96	7.2	11.5	334.96	0.1	0.6	334.96	3.5	36.7	0.00
E021-04m	447.73	461.1	1800.0	430.95	352.6	540.6	409.79	88.5	121.8	409.44	450.1	1050.1	0.09
E021-06m	448.48	181.1	1800.0	458.04	204.0	323.5	440.68	3.9	5.4	439.98	51.2	126.3	0.16
E022-04g	464.24	75.8	1800.0	465.79	61.3	99.6	453.19	22.7	30.9	447.36	287.8	713.5	1.29
E022-06m	504.46	1167.9	1800.0	507.96	768.8	1212.4	501.47	17.5	18.4	499.99	130.6	311.9	0.30
E023-03g	831.66	181.1	1800.0	796.61	241.5	364.8	797.47	51.4	67.4	773.31	421.0	810.9	2.92
E023-05s	871.77	156.1	1800.0	880.93	140.0	230.0	820.67	56.2	78.6	807.59	548.2	1081.9	1.59
E026-08m	666.10	1468.5	1800.0	642.22	604.7	982.2	635.50	15.3	16.3	630.13	95.5	518.8	0.85
E030-03g	911.16	714.0	3600.0	884.74	803.1	1308.4	841.12	241.2	246.7	839.75	601.5	1252.3	0.16
E030-04s	819.36	396.4	3600.0	873.43	308.5	522.5	821.04	172.4	199.8	790.47	434.4	1152.3	3.53
E031-09h	651.58	268.1	3600.0	624.24	180.8	294.6	629.07	46.2	48.2	615.05	224.8	463.6	1.47
E033-03n	2928.34	1639.1	3600.0	2799.74	1309.5	2193.1	2739.80	235.4	308.8	2732.85	654.6	1608.4	0.25
E033-04g	1559.64	3451.6	3600.0	1504.44	2678.1	4581.3	1472.26	623.8	642.8	1460.34	2659.3	3863.4	0.81
E033-05s	1452.34	2327.4	3600.0	1415.42	1466.3	2528.3	1405.48	621.0	656.8	1386.75	984.6	1608.0	1.33
E036-11h	707.85	2550.3	3600.0	698.61	2803.2	4256.5	698.92	12.8	14.8	698.69	50.2	131.2	**−0.01**
E041-14h	920.87	2142.5	3600.0	872.79	1208.6	2096.0	870.33	11.8	14.9	869.96	177.2	308.3	0.04
E045-04f	1400.52	1452.9	3600.0	1296.59	1300.9	2275.2	1261.07	2122.2	2209.8	1252.67	2258.6	3495.7	0.67
E051-05e	871.29	1822.3	7200.0	818.68	1438.4	2509.0	781.29	614.3	623.6	777.96	1407.2	2285.2	0.43
E072-04f	732.12	790.0	7200.0	641.57	1284.8	1940.9	611.26	3762.3	3901.0	600.82	7466.0	8511.0	1.71
E076-07s	1275.20	2370.3	7200.0	1159.72	1704.8	2823.4	1124.55	5140.0	5180.6	1140.11	2848.6	5061.7	**−1.38**
E076-08s	1277.94	1611.3	7200.0	1245.35	1663.5	2685.6	1197.43	2233.6	2290.3	1199.14	1890.0	3311.7	**−0.14**
E076-10e	1258.16	6725.6	7200.0	1231.92	3048.2	4659.1	1171.77	3693.4	3727.6	1176.07	2829.5	4853.5	**−0.37**
E076-14s	1307.09	6619.3	7200.0	1201.96	2876.8	4854.1	1148.70	1762.8	1791.5	1161.87	2391.6	3813.1	**−1.15**
E101-08e	1570.72	5630.9	7200.0	1457.46	3432.0	5725.8	1436.32	8619.7	8817.1	1442.62	3580.3	5001.8	**−0.44**
E101-10c	1847.95	4123.7	7200.0	1711.93	3974.8	6283.1	1616.99	6651.2	6904.3	1614.56	2968.7	4100.4	0.15
E101-14s	1747.52	7127.2	7200.0	1646.44	5864.2	9915.7	1573.50	10,325.8	10,483.9	1571.38	2837.8	4966.7	0.13
Avg *z*	1042.26			997.18			966.67			962.08			0.57
Avg time		2058.9	4200.0		1471.6	2415.9		1746.5	1793.1		1419.9	2252.2	

Table 2
Comparison on Tarantilis' 12 instances (3L-CVRP).

Instance	GTS			DMTS			
	z	*t* (s)	*tt* (s)	Avg *z*	Avg *t* (s)	Avg *tt* (s)	Impr (%)
3l_cvrp_50_1	1457.78	1387.6	2308.7	1451.136	1404.364	2778.286	0.46
3l_cvrp_50_2	2257.60	975.8	1600.4	2222.496	389.993	619.848	1.55
3l_cvrp_50_3	1838.40	1065.8	1719.9	1817.521	3356.394	4644.881	1.14
3l_cvrp_75_1	2059.32	2286.0	3776.2	2078.002	2564.972	3723.478	**−0.91**
3l_cvrp_75_2	3279.16	1460.1	2495.7	3166.69	858.169	1124.281	3.43
3l_cvrp_75_3	2508.17	1768.3	3507.1	2540.582	5019.729	6329.303	**−1.29**
3l_cvrp_100_1	2690.23	4895.5	8457.5	2613.041	7448.454	12,691.21	2.87
3l_cvrp_100_2	4342.64	3281.8	5967.9	4237.801	835.251	1091.138	2.41
3l_cvrp_100_3	4190.92	4682.5	9301.9	4263.589	9639.495	10,941.97	**−1.73**
3l_cvrp_125_1	3298.22	6850.2	12,590.0	3260.232	8416.266	12,021.29	1.15
3l_cvrp_125_2	5788.12	5281.7	9727.3	5450.576	1854.556	2198.872	5.83
3l_cvrp_125_3	5177.97	5704.4	11,191.9	5078.295	14,219.32	14,952.24	1.92
Avg *z*	3240.71			3181.66			1.82
Avg time		3303.3	6053.7		4667.2	6093.1	

best solution found, the columns *t (s)* give the time taken to find the best solution, and the columns *tt (s)* give the total execution time in CPU seconds. For ACO and DMTS, the algorithm was executed 10 times for each instance, and the average values for the 10 executions are reported in columns *avg z*, *avg t (s)* and *avg tt (s)*. Finally, the last column *Impr* gives the improvement of the results of DMTS over the best result of TS, GTS and ACO. Our experiments show that DMTS is better than the best of TS, GTS and ACO in 20 out of 27 instances. The average improvement is about 0.57%.

Note that in the original publication by Gendreau et al. [13] proposing the G27 instances, in Table 1 specifying the attributes of each instance, the values for the number of vehicles are incorrect. The correct values are given in the data files, which are smaller than or equal to the corresponding values in the table; this error has been verified via e-mail correspondence with the authors. We have listed both sets of values for the number of vehicles in Table D2 in Appendix D. The results for the corresponding experiments with the "incorrect" values are given in Appendix G, which see DMTS outperform all existing algorithms for 26 out of the 27 instances and by about 1.11% on average.

The comparison with GTS on the T12 instances proposed by Tarantilis et al. [27] is given in Table 2. The values show that DMTS finds better solutions than GTS for 9 out of the 12 instances using comparable amounts of computation time. The average improvement is about 1.82%.

7.2. Results on M3L-CVRP

We also performed experiments comparing DMTS against GTS on both the G27 test set and the T12 test set on the M3L-CVRP variant of the problem proposed by Tarantilis et al. [27]. These results are presented in Tables 3 and 4, respectively. For the G27 test set, DMTS is superior to GTS for 24 out of 27 instance (and worse for only one instance), with an average improvement of 3.17%. For the T12 instances, DMTS outperforms GTS for 8 out of 12 instances and by 1.28% on average.

7.3. Impact of constraints

Finally, we analyzed the impact of various loading constraints on the performance of the DMTS algorithm on the 3L-CVRP variant using the G27 test set. We examined the effects of running DMTS in five scenarios: (1) all constraints imposed, (2) without the fragility constraint, (3) without the supporting area constraint, (4) without the LIFO policy, and (5) with only the 3D loading constraint (i.e., the items must fit inside the vehicles). For every instance and scenario, DMTS was executed 10 times with

Table 3
Comparison on Gendreau's 27 instances (M3L-CVRP).

Instance	GTS			DMTS			
	z	t (s)	tt (s)	Avg z	Avg time	Avg tt (s)	Impr (%)
E016-03m	322.47	6.7	12.1	301.99	58.2	360.7	6.35
E016-05m	334.96	9.9	18.2	334.96	3.5	36.9	0.00
E021-04m	403.51	286.8	517.0	404.94	465.3	1113.7	**−0.35**
E021-06m	458.04	207.6	325.1	439.98	21.1	120.8	3.94
E022-04g	465.79	53.0	86.0	447.27	426.0	807.2	3.98
E022-06m	507.96	683.7	1254.0	499.30	84.2	310.9	1.71
E023-03g	796.61	256.1	449.5	772.11	444.1	761.1	3.08
E023-05s	843.76	124.2	245.3	806.22	494.7	859.8	4.45
E026-08m	642.22	563.8	957.1	630.13	147.5	554.1	1.88
E030-03g	879.79	738.5	1209.1	837.24	747.2	1290.4	4.84
E030-04s	873.43	308.1	511.9	786.40	482.1	1135.3	9.96
E031-09h	624.24	164.7	263.2	615.15	181.2	464.8	1.46
E033-03n	2782.89	1272.3	2051.0	2728.97	1019.0	1906.8	1.94
E033-04g	1508.83	2777.9	5303.4	1470.59	3494.7	4784.5	2.53
E033-05s	1456.97	1542.8	2753.3	1381.56	869.4	1635.1	5.18
E036-11h	698.61	2640.2	4311.9	698.61	40.2	121.5	0.00
E041-14h	872.79	1054.6	1987.2	870.55	223.9	324.4	0.26
E045-04f	1299.91	1106.0	1893.0	1253.61	2287.7	3531.7	3.56
E051-05e	817.19	1503.1	2519.1	782.31	1433.8	2358.6	4.27
E072-04f	635.90	1126.4	2197.1	601.54	6600.5	8155.6	5.40
E076-07s	1163.96	1704.8	3206.6	1133.76	3460.0	5079.6	2.59
E076-08s	1224.25	1450.3	2416.8	1197.71	2088.3	3265.8	2.17
E076-10e	1199.37	2975.2	4888.5	1170.37	2379.4	5119.8	2.42
E076-14s	1183.11	2932.7	5527.6	1145.09	2596.3	3924.7	3.21
E101-08e	1482.85	3387.6	6748.4	1443.68	2819.4	4915.1	2.64
E101-10c	1695.39	4106.0	6282.0	1615.75	2904.7	4292.9	4.70
E101-14s	1613.32	5594.2	9591.4	1568.32	2571.5	4769.6	2.79
Avg z	992.15			960.67			3.17
Avg time		1428.8	2501.0		1420.1	2296.4	

Table 5
Impact of constraints on performance of DMTS based on Gendreau's 27 instances (3L-CVRP).

ID	Instance	Avg z over 10 executions for each instance				
	Name	All constraints	No fragility	No LIFO	No support	3D loading only
1	E016-03m	302.23	301.99	298.39	298.85	297.65
2	E016-05m	334.96	334.96	334.96	334.96	334.96
3	E021-04m	409.44	388.58	367.61	370.80	362.27
4	E021-06m	439.98	434.52	430.89	430.89	430.89
5	E022-04g	447.36	440.29	417.51	424.74	379.43
6	E022-06m	499.99	498.13	495.89	497.70	495.85
7	E023-03g	773.31	763.22	732.52	743.10	720.30
8	E023-05s	807.59	803.98	744.88	790.02	730.66
9	E026-08m	630.13	630.13	630.13	630.13	630.13
10	E030-03g	839.75	822.04	749.31	753.31	688.64
11	E030-04s	790.47	779.21	722.36	754.37	717.32
12	E031-09h	615.05	614.16	610.23	611.13	610.00
13	E033-03n	2732.85	2695.74	2466.52	2581.23	2313.44
14	E033-04g	1460.34	1404.99	1318.02	1355.31	1200.10
15	E033-05s	1386.75	1365.28	1222.22	1338.02	1141.59
16	E036-11h	698.69	698.61	698.61	698.61	698.61
17	E041-14h	869.96	866.92	869.58	864.70	861.87
18	E045-04f	1252.67	1198.21	1166.28	1188.82	1103.35
19	E051-05e	777.96	745.16	693.17	733.77	671.81
20	E072-04f	600.82	581.37	557.03	563.36	510.57
21	E076-07s	1140.11	1099.34	1001.42	1057.04	941.96
22	E076-08s	1199.14	1172.73	1103.73	1122.96	1029.91
23	E076-10e	1176.07	1132.65	1048.57	1084.65	984.07
24	E076-14s	1161.87	1130.68	1078.92	1089.17	1055.22
25	E101-08e	1442.62	1401.13	1281.69	1354.67	1206.69
26	E101-10c	1614.56	1575.04	1520.04	1530.78	1418.20
27	E101-14s	1571.38	1530.04	1407.77	1459.58	1318.42
Avg z		962.08	941.08	887.71	913.43	846.44
Impr (%)			2.18	7.73	5.06	12.02

Table 4
Comparison on Tarantilis' 12 instances (M3L-CVRP).

Instance	GTS			DMTS			
	z	t (s)	tt (s)	Avg z	Avg t (s)	Avg tt (s)	Impr (%)
3l_cvrp_50_1	1457.78	1270.6	2314.7	1458.714	1492.014	2928.47	**−0.06**
3l_cvrp_50_2	2242.18	903.8	1404.7	2232.968	324.782	520.88	0.41
3l_cvrp_50_3	1853.43	990.1	1746.7	1817.644	3763.533	4912.891	1.93
3l_cvrp_75_1	2061.56	2134.0	3410.1	2071.234	2354.449	3595.874	**−0.47**
3l_cvrp_75_2	3226.62	1526.5	2690.4	3164.051	1312.263	1458.042	1.94
3l_cvrp_75_3	2477.61	1672.3	2686.6	2486.014	5492.42	6475.141	**−0.34**
3l_cvrp_100_1	2653.91	4562.5	8958.4	2641.281	10,066.58	13,961.16	0.48
3l_cvrp_100_2	4325.19	3319.0	5674.8	4232.687	742.619	1008.464	2.14
3l_cvrp_100_3	4069.59	4798.1	7923.9	4060.725	9476.273	10,555.4	0.22
3l_cvrp_125_1	3273.95	6472.2	10,784.9	3269.218	7887.612	11,532.13	0.14
3l_cvrp_125_2	5759.24	5384.7	9171.3	5446.886	2153.902	2531.33	5.42
3l_cvrp_125_3	5110.85	5524.9	8559.0	5135.951	15,887.68	16,831.85	**−0.49**
Avg z	3209.33			3168.11			1.28
Avg time		3213.2	5443.8		5079.5	6359.3	

Table 6
Overall performance of algorithms under various scenarios (G27, 3L-CVRP).

Scenario	TS		GTS		ACO		DMTS	
	Avg *z*	Impr (%)	Avg *z*	Impr (%)	Avg *z*	Impr (%)	Avg *z*	Impr (%)
All constraints	1042.26	–	997.18	–	966.66	–	962.08	–
No fragility	1014.49	2.66	965.14	3.21	945.04	2.24	941.08	2.18
No LIFO	951.19	8.74	950.59	4.67	916.25	5.21	887.71	7.73
No support	939.53	9.86	934.96	6.24	919.69	4.86	913.43	5.06
3D loading only	876.31	15.92	876.39	12.11	856.67	11.38	846.44	12.02

different random seeds; the values given in Table 5 are the total traveling costs of the best solution found by DMTS averaged over the 10 runs. The second last row reports the average traveling cost for each scenario, and the last row reports the average reduction in traveling cost for each scenario using the *All constraints* scenario as the basis. The results show that the LIFO constraint has largest impact out of all the other individual constraints on the performance of DMTS, followed by the supporting area constraint, and finally the fragility constraint.

Previous researchers have performed similar experiments on the 3L-CVRP using the G27 test set; their results are summarized in Table 6. Each column *avg z* provides the average costs of the solutions found by the corresponding approach for the given scenarios (for ACO and DMTS, the values are the averages of the averages over 10 executions). For the latter four rows, the values under the column *Impr* are the percentage differences in the average solution cost for the approach compared to the complete 3L-CVRP with all constraints. These values verify the observation that although all three constraints have a significant effect on the cost of the final solution, the LIFO and supporting area constraints have a more significant effect than the fragility constraint.

For previous approaches, the supporting area constraint seems to be a difficult constraint to handle. However, we see that for DMTS the supporting area constraint has a less significant effect on the solution quality. This suggests that our efficient implementation of the DBLF and MTS heuristics to solve the single vehicle loading sub-problem has more successfully handled the supporting area constraint than previous approaches. Also note that DMTS outperforms all the existing algorithms on all test sets under all five scenarios on average.

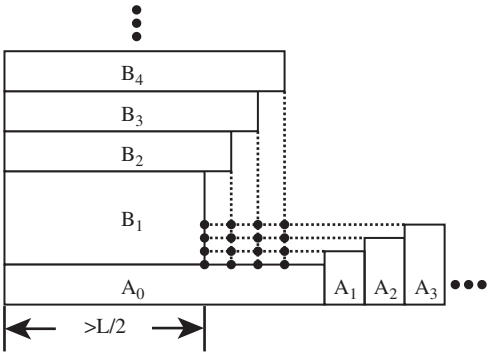

Fig. A1. Normal positions in 2D.

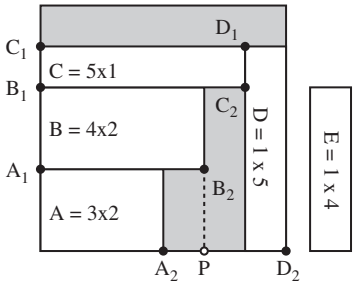

Fig. B1. A counterexample of BLF by Hopper [16].

8. Conclusions

In this study, we examined the Three-Dimensional Loading Capacitated Vehicle Routing Problem, which is highly relevant to the freight transportation industry and addresses practical loading constraints. It is the combination of two hard combinatorial optimization problems, and is therefore a complex problem to solve. We addressed the complexity of the problem by dividing it into two parts: finding a good solution to the loading sub-problem, and developing an effective overall algorithm to handle the routing problem using the loading sub-problem as a subroutine.

We improved two sequence-based loading heuristics for the loading sub-problem, namely the Deepest-Bottom-Left-Fill (DBLF) and the Maximum Touching Area (MTA) heuristics. For the overall algorithm, we adapted the tabu search framework, which has proven to be an effective approach for vehicle routing problems. We defined five neighborhood operators to explore the solution space. To improve the effectiveness of the algorithm, we divided the search into two phases: the first phase focuses on finding a feasible solution, while the second phase attempts to improve the quality of the solution while maintaining feasibility. Computational experiments showed that our resultant DMTS algorithm outperforms the best known algorithms in 20 out of the 27 instances of the G27 test set for the 3L-CVRP variant. For the M3L-CVRP variant, our DMTS algorithm outperforms the GTS algorithm for 24 out of 27 instances for the G27 test set, and 9 out of 12 instances for the T12 test set.

Acknowledgment

We would like to thank Mr. Shi Chen for his coding efforts in the preliminary version of the Tabu Search algorithm.

Appendix A. Number of normal positions using DBL rule

Healy et al. [15] has shown in the 2D case that if n boxes are already placed in general positions (i.e., not necessarily bottom-left), then there are potentially $N = O(n^2)$ normal positions to place a new item if its size is not known in advance. We now show this statement to be true even if the n boxes are placed following the bottom-left rule.

Assume that there are n boxes, $A_0, A_1, A_2, \ldots, B_1, B_2, \ldots$, and the boxes are placed in the sequence $\langle A_0, A_1, A_2, \ldots, B_1, B_2, \ldots \rangle$.

Assuming that the lengths of $A_0,B_1,B_2,\ldots$, are all greater than half the length of the container, the realization of the BLF method is illustrated in Fig. A1. The dots in the figure are the intersections of the downward extensions of the right edges of the boxes of $B_2,B_3,\ldots$ and the leftward extensions of the top edges of the boxes $A_1,A_2,\ldots$. These dots are in fact the set of normal positions for new items, and there are $\Theta(n^2)$ such dots. A similar argument can be extended to three-dimensional packing, which results in $\Theta(n^3)$ normal positions.

Note that Healy et al. [15] also showed that if the size of each new item is known beforehand, then only $\Theta(n)$ normal positions are applicable for that item in the 2D case; this result can be extended to the 3D case, which equates to $N=\Theta(n^2)$ applicable normal positions. Therefore, any algorithm that checks all normal positions for all n items using the straightforward approach requires at least $\Omega(n\cdot n\cdot n)=\Omega(n^3)$ time to load n2D boxes, and $\Omega(n\cdot n^2\cdot n)=\Omega(n^4)$ time to load n3D boxes. This observation contradicts the findings by Gendreau et al. [13], who conjectured that there are at most $O(n)$ normal positions in the 3D case, resulting in an $O(n^3)$ time algorithm that inspects all normal positions; we have shown this conjecture to be false.

Table C1
Sensitivity analysis for *2-opt* weights.

Id	Avg z			Std dev		
	2-opt=500	Base	2-opt=1500	2-opt=500	Base	2-opt=1500
6	502.06	498.86	499.06	2.10	1.23	1.67
11	785.78	790.28	792.22	3.61	2.82	4.53
16	698.61	698.61	698.61	0.00	0.00	0.00
21	1135.57	1139.16	1135.15	12.66	5.69	9.16
26	1609.17	1615.49	1612.99	9.12	4.86	9.64

Table C2
Sensitivity analysis for *2-swap* weights.

Id	Avg z			Std dev		
	2-swap=500	Base	2-swap=1500	2-swap=500	Base	2-swap=1500
6	501.77	498.86	499.41	2.86	1.23	1.50
11	787.96	790.28	792.60	3.06	2.82	4.24
16	698.61	698.61	699.55	0.00	0.00	2.12
21	1137.40	1139.16	1139.28	9.13	5.69	14.71
26	1614.01	1615.49	1613.91	5.65	4.86	19.91

Table C3
Sensitivity analysis for *move* weights.

Id	Avg z			Std dev		
	Move=2000	Base	Move=4000	Move=2000	Base	Move=4000
6	499.96	498.86	499.06	2.05	1.23	1.67
11	792.24	790.28	789.07	3.82	2.82	1.93
16	698.77	698.61	698.61	0.36	0.00	0.00
21	1141.37	1139.16	1146.90	8.89	5.69	12.16
26	1618.52	1615.49	1613.40	14.86	4.86	13.89

Appendix B. Discussion on BLF heuristic implementation by Hopper [16]

Hopper [16] implemented the BLF heuristic for the 2D case by first placing the current item such that its bottom left corner is at the top left or bottom right corner of some loaded item. There are at most $2n$ such positions; the bottom-leftmost such position where the current item can be placed is chosen. Then, the algorithm slides the current item as far to the bottom and to the left of the container as possible. Since the sliding can be implemented in $O(n)$ time, the overall time complexity for this algorithm to place all n items is $O(n^2)$; a 3D adaptation of this approach would run in $O(n^3)$ time (Fig. B1).

However, this implementation does not actually honor the BLF criteria, as illustrated by the following counterexample. Four items $A=3\times 2$, $B=4\times 2$, $C=5\times 1$, and $D=1\times 5$ are placed into the container (6×6) in that order. According to this procedure, $\{A_1,A_2,B_1,B_2,C_1,C_2,D_1,D_2\}$ is the set of candidate positions to place the fifth piece $E=1\times 4$, but none of the candidates provides a feasible position for E. Nevertheless, the position P is a feasible position.

Table C4
Sensitivity analysis for *crossover* weights.

Id	Avg z			Std dev		
	Crossover=3500	Base	Crossover=5500	Crossover=3500	Base	Crossover=5500
6	500.08	498.86	501.02	2.68	1.23	2.65
11	790.07	790.28	787.40	1.46	2.82	2.32
16	698.98	698.61	698.61	0.84	0.00	0.00
21	1134.69	1139.16	1138.15	10.44	5.69	10.33
26	1615.03	1615.49	1616.34	20.34	4.86	5.77

Table C5
Sensitivity analysis for *splitting* weights.

Id	Avg z			Std dev		
	Splitting=0	Base	Splitting=1000	Splitting=0	Base	Splitting=1000
6	498.86	498.86	499.50	1.22	1.23	1.63
11	788.33	790.28	788.69	2.64	2.82	3.22
16	698.61	698.61	698.61	0.00	0.00	0.00
21	1139.98	1139.16	1130.21	4.81	5.69	13.57
26	1602.86	1615.49	1605.09	7.55	4.86	6.51

Appendix C. Sensitivity analysis on operator weights

We performed some sensitivity analyses on the operator weights in our tabu search that determines the relative probability that the neighborhood operator will be applied. The values used in our implementation is considered the base setting, i.e., the five neighborhood operators 2-opt, 2-swap, move, crossover, and splitting are assigned a weight of 1000, 1000, 3000, 4500 and 500, respectively. We then consider each operator individually and varied their weights up and down by a small amount. We tested each resulting configuration on five of the G27 instances (see Appendix D for details), running each instance five times.

The results for 2-opt, 2-swap, move, crossover, and splitting are given in Tables C1–C5, respectively. The values under the heading *avg z* are the average traveling cost for the five executions; the column *base* gives the results for the base setting, and the two columns to its left and right show the results when the given operator weight is set to the stated value (the weights for the other operators remain unchanged from the base setting). The values under the heading *std dev* give the standard deviation of the results.

We see that although the base setting did not produce the highest quality solutions on average on these instances (e.g., assigning a weight of 500 or 1500 for 2-opt both produced better solutions on average than the base setting), its performance is most stable among the alternative settings (i.e., the fluctuation of solution quality due to randomness is smallest as measured by the standard deviation). Especially for instances 21 and 26, the standard deviations of the alternative settings are much larger. Since the differences in solution quality between the different settings are minimal, and no single setting outperformed the base setting on all five instances, we prefer our base settings.

Appendix D. Benchmark instances

There are two sets of test instances:

G27: 27 instances generated by Gendreau et al. [13] available at http://www.or.deis.unibo.it/research_pages/ORinstances/ORinstances.htm, and

T12: 12 instances generated by Tarantilis et al. [27] available at http://www.msl.aueb.gr/management_science/3L-CVRP/3L-CVRP.html

The G27 instances were created based on 27 Euclidean CVRP instances (detailed descriptions can be found in Toth and Vigo [28]). The cost of an edge is the Euclidean distance between the two clients. The dimensions of the vehicles are $W=25$, $H=30$ and $L=60$. The number of vehicles was increased to ensure all items can be loaded into the vehicles. The input threshold a for the minimum supporting area was set to be 0.75. The number of requested items for each client is a random integer uniformly distributed in [1, 3], and the dimensions of each item are uniformly randomly generated in the interval between 20% and 60% of the corresponding vehicle dimensions.

The T12 instances were created in the following manner. Four graphs were generated containing 50–125 clients. The central depot lies at (0, 0). Each client is placed at $(r\cos\theta, r\sin\theta)$, where r and θ are random numbers uniformly distributed in [10, 100] and

Table D1
Item demand characteristics—range of parameters.

Class	m_i	w	h	l
0	[1,3]	[0.2W, 0.6W]	[0.2H, 0.6H]	[0.2L, 0.6L]
1	[2,4]	[0.2W, 0.4W]	[0.2H, 0.4H]	[0.2L, 0.4L]
2	[1,2]	[0.4W, 0.6W]	[0.4H, 0.6H]	[0.4L, 0.6L]
3	[1,3]	[0.1W, 0.7W]	[0.1H, 0.7H]	[0.1L, 0.7L]

Table D2
Benchmark instances by Gendreau et al. [13] and Tarantilis et al. [27].

Id	G27 Gendreau et al. [13]						Id	T12 Tarantilis et al. [27]				
	Name	Class	n	M	v	v^*		Name	Class	n	M	v
1	E016-03m	0	15	32	4	5	28	3l_cvrp_50_1	1	50	143	9
2	E016-05m	0	15	26	5	5	29	3l_cvrp_50_2	2	50	73	14
3	E021-04m	0	20	37	4	5	30	3l_cvrp_50_3	3	50	97	10
4	E021-06m	0	20	36	6	6	31	3l_cvrp_75_1	1	75	227	14
5	E022-04g	0	21	45	6	7	32	3l_cvrp_75_2	2	75	114	20
6	E022-06m	0	21	40	6	6	33	3l_cvrp_75_3	3	75	142	15
7	E023-03g	0	22	46	6	6	34	3l_cvrp_100_1	1	100	309	17
8	E023-05s	0	22	43	6	8	35	3l_cvrp_100_2	2	100	149	29
9	E026-08m	0	25	50	8	8	36	3l_cvrp_100_3	3	100	189	20
10	E030-03g	0	29	62	8	10	37	3l_cvrp_125_1	1	125	309	17
11	E030-04s	0	29	58	8	9	38	3l_cvrp_125_2	2	125	149	29
12	E031-09h	0	30	63	9	9	39	3l_cvrp_125_3	3	125	189	20
13	E033-03n	0	32	61	8	9						
14	E033-04g	0	32	72	9	11						
15	E033-05s	0	32	68	9	10						
16	E036-11h	0	35	63	11	11						
17	E041-14h	0	40	79	14	14						
18	E045-04f	0	44	94	11	14						
19	E051-05e	0	50	99	12	13						
20	E072-04f	0	71	147	18	20						
21	E076-07s	0	75	155	17	18						
22	E076-08s	0	75	146	18	19						
23	E076-10e	0	75	150	17	18						
24	E076-14s	0	75	143	16	18						
25	E101-08e	0	100	193	22	24						
26	E101-10c	0	100	199	26	28						
27	E101-14s	0	100	198	23	25						

$[0, 2\pi]$, respectively. The distances between each pair of vertices is Euclidean. The dimensions of the vehicles are $W=25$, $H=30$ and $L=60$. Three classes of item demand characteristics were introduced: Class 1 contains small items, Class 2 contains large items, and Class 3 involves diverse item dimensions. For each graph and class, one instance is generated, resulting in a total of 12 instances. The number of items per client is randomly taken from [2,4], [1,2], and [1, 3] for classes 1, 2, and 3, respectively. The dimensions h, w, l of each item are uniformly randomly generated from the intervals [0.2H, 0.4H], [0.2W, 0.4W], [0.2L, 0.4L] for class 1 (resp. [0.4H, 0.6H], [0.4W, 0.6W], [0.4L, 0.6L] and [0.1H, 0.7H], [0.1W, 0.7W], [0.1L, 0.7L] for Classes 2 and 3).

For consistency we denote the item demand characteristics of the G27 instances by Class 0. All four classes of item demand characteristics are summarized in Table D1.

Table D2 summarizes the characteristics of all 39 instances. The column *class* gives the characteristics of the item demand, column n is the number of clients (excluding the depot), column M is total number of items demanded, and column v is the number of vehicles. For the G27 instances, the values listed in column v were taken from the individual data files, while the column v^* gives the number of vehicles listed in Table 1 in the original publication [13]; we have verified with the authors via email correspondence that the intended values for these instances are those in column v.

Appendix E. Comparison of computational environments

Table E1 summarizes the computational environments used to generate the results reported by the respective publications. Note that even though the machine used to perform the experiments for our DMTS algorithm has more RAM, our approach is not memory-intensive. Furthermore, our algorithm is sequential and does not take advantage of the parallel processing capabilities of the CPU. We believe that our test machine is comparable to the machines used for the previous approaches. A dash '–' in a cell indicates the relevant information was not reported.

Appendix F. DMTS results

This section contains the detailed results for DMTS on the 27 instances proposed by Gendreau et al. [13] and the 12 instances generated by Tarantilis et al. [27] on both the 3L-CVRP and M3L-CVRP, resulting in four tables. Since DMTS contains random

Table E1
Comparison of computational environments.

Computational environment	TS	GTS	ACO	DMTS
Language	C	C#	C++	C
Compiler	–	Visual C#	G++	GCC 4.1.2
CPU	Pentium IV 3 GHz	Pentium IV 2.8 GHz	Pentium IV 3.2 GHz	Xeon E5520 2.26 GHz
RAM	512 MB	1 GB	2 GB	8 GB
O/S	–	–	Linux	Linux

Table F1
DMTS results on Gendreau's 27 instances (3L-CVRP).

Id	Instance		Init solution		Best solution					
	Name	#exe	Init	Avg t (s)	Min	Max	Std	Avg	Avg t (s)	Avg tt (s)
1	E016-03m	10	349.35	30.1	302.02	304.13	0.67	302.23	85.1	371.7
2	E016-05m	10	349.11	2.7	334.96	334.96	0.00	334.96	3.5	36.7
3	E021-04m	10	546.18	89.8	394.00	429.93	13.18	409.44	450.1	1050.1
4	E021-06m	10	482.86	8.3	437.19	440.68	1.47	439.98	51.2	126.3
5	E022-04g	10	516.14	158.4	443.61	449.49	2.33	447.36	287.8	713.5
6	E022-06m	10	563.18	46.0	498.32	504.39	2.33	499.99	130.6	311.9
7	E023-03g	10	849.45	174.2	768.85	777.67	2.95	773.31	421.0	810.9
8	E023-05s	10	959.25	197.9	805.35	812.81	2.89	807.59	548.2	1081.9
9	E026-08m	10	751.01	51.8	630.13	630.13	0.00	630.13	95.5	518.8
10	E030-03g	10	952.72	169.6	834.80	842.29	2.30	839.75	601.5	1252.3
11	E030-04s	10	898.66	0.1	786.19	795.40	3.18	790.47	434.4	1152.3
12	E031-09h	10	738.01	90.3	612.25	622.62	2.77	615.05	224.8	463.6
13	E033-03n	7	3212.57	119.1	2719.65	2744.24	7.94	2732.85	654.6	1608.4
14	E033-04g	9	1793.70	1209.1	1403.75	1489.25	25.57	1460.34	2659.3	3863.4
15	E033-05s	9	1647.22	0.3	1371.58	1405.25	11.95	1386.75	984.6	1608.0
16	E036-11h	10	807.44	19.9	698.61	699.42	0.26	698.69	50.2	131.2
17	E041-14h	10	996.97	94.0	866.40	874.58	3.46	869.96	177.2	308.3
18	E045-04f	10	1380.82	429.1	1231.80	1262.22	8.50	1252.67	2258.6	3495.7
19	E051-05e	10	927.68	42.0	767.19	786.70	6.71	777.96	1407.2	2285.2
20	E072-04f	5	1014.71	5520.1	597.75	604.64	2.98	600.82	7466.0	8511.0
21	E076-07s	10	1362.60	452.7	1133.29	1147.28	5.68	1140.11	2848.6	5061.7
22	E076-08s	10	1416.61	319.1	1193.82	1208.18	4.06	1199.14	1890.0	3311.7
23	E076-10e	10	1482.29	559.0	1163.10	1188.34	9.48	1176.07	2829.5	4853.5
24	E076-14s	10	1491.55	382.1	1151.13	1180.82	7.92	1161.87	2391.6	3813.1
25	E101-08e	10	1702.23	372.5	1414.85	1459.73	13.11	1442.62	3580.3	5001.8
26	E101-10c	10	1959.42	442.0	1607.85	1625.21	5.77	1614.56	2968.7	4100.4
27	E101-14s	10	1944.76	744.2	1553.77	1591.51	12.27	1571.38	2837.8	4966.7
Avg			1151.72	434.2	952.67	970.81	5.92		1419.9	2252.2

components, we ran the algorithm 10 times on each instance. For each run, we record the initial solution found at the end of phase one and the time taken, the best solution found by the end of phase two, the time required to find this best solution, and the total execution time.

For each table, the columns under the *Init solution* heading concerns the initial solution. The column *avg z* gives the average of the objective value for the initial solution, while the column *avg t(s)* gives the average amount of time required for phase one in CPU seconds. For some of the larger instances, not all of the 10 executions of DMTS were able to produce a feasible solution after the first phase of our tabu search; the column *#exe* gives the total number of executions out of 10 where DMTS found feasible solutions.

Under the *Best solution* heading, we report the minimum, maximum, standard deviation, and average of the objective value of the best solution found in the columns *min*, *max*, *std*, and *avg*, respectively. The column *avg t (s)* gives the average time required to obtain the best solution, while *avg tt (s)* reports the average total execution time in CPU seconds (Tables F1–F4).

Appendix G. Comparison on 3L-CVRP using G27 with incorrect fleet size

The G27 instances were originally proposed by Gendreau et al. [13]. The number of vehicles given in their Table 1 do not match the values given in the data files. We have verified with the authors via e-mail correspondence that the values in the data files should be used. The values given in their table are generally larger than those in the data files, which would have an effect on the problem solutions. Since this error has not been discovered until year 2011, there may be researchers who have used these values for their experiments. Hence, we present our results using the

Table F2
DMTS results on Tarantilis' 12 instances (3L-CVRP).

Id	Instance		Init solution		Best solution					
	Name	#exe	Init	Avg *t* (s)	Min	Max	Std	Avg	Avg *t* (s)	Avg *tt* (s)
28	3l_cvrp_50_1	10	1567.94	11.8	1433.08	1467.93	13.42	1451.14	1404.4	2778.3
29	3l_cvrp_50_2	10	2456.44	174.6	2201.30	2248.81	15.94	2222.50	390.0	619.8
30	3l_cvrp_50_3	10	2330.27	459.6	1799.78	1851.37	17.60	1817.52	3356.4	4644.9
31	3l_cvrp_75_1	10	2453.40	194.5	2044.53	2120.02	25.36	2078.00	2565.0	3723.5
32	3l_cvrp_75_2	10	3894.68	369.5	3099.78	3223.73	41.26	3166.69	858.2	1124.3
33	3l_cvrp_75_3	10	3421.59	1764.1	2479.09	2581.94	38.02	2540.58	5019.7	6329.3
34	3l_cvrp_100_1	10	3116.71	305.3	2588.31	2633.55	15.32	2613.04	7448.5	12691.2
35	3l_cvrp_100_2	10	4928.10	292.7	4205.44	4275.69	20.22	4237.80	835.3	1091.1
36	3l_cvrp_100_3	4	5611.45	4622.2	4078.21	4653.71	263.01	4263.59	9639.5	10942.0
37	3l_cvrp_125_1	10	4131.25	625.9	3240.40	3286.41	14.94	3260.23	8416.3	12021.3
38	3l_cvrp_125_2	10	6443.65	1162.8	5379.70	5550.31	50.37	5450.58	1854.6	2198.9
39	3l_cvrp_125_3	2	7513.76	6467.2	4800.27	5356.32	393.19	5078.29	14219.3	14952.2
Avg *z*			3989.10	1370.9	3112.49	3270.82	75.72		4667.2	6093.1

Table F3
DMTS results on Gendreau's 27 instances (M3L-CVRP).

Id	Instance		Init solution		Best solution					
	Name	#exe	Init	Avg *t* (s)	Min	Max	Std	Avg	Avg *t* (s)	Avg *tt* (s)
1	E016-03m	10	351.96	17.4	301.74	302.02	0.09	301.99	58.2	360.7
2	E016-05m	10	350.59	2.6	334.96	334.96	0.00	334.96	3.5	36.9
3	E021-04m	10	549.80	159.0	394.00	419.12	7.48	404.94	465.3	1113.7
4	E021-06m	10	480.30	4.6	437.19	440.68	1.47	439.98	21.1	120.8
5	E022-04g	10	513.57	234.4	443.61	448.54	1.55	447.27	426.0	807.2
6	E022-06m	10	580.80	38.6	498.32	504.39	2.14	499.30	84.2	310.9
7	E023-03g	10	847.93	131.5	771.07	774.58	1.33	772.11	444.1	761.1
8	E023-05s	10	920.63	56.3	803.98	807.75	1.75	806.22	494.7	859.8
9	E026-08m	10	738.86	78.7	630.13	630.13	0.00	630.13	147.5	554.1
10	E030-03g	10	933.98	213.0	828.14	841.97	3.96	837.24	747.2	1290.4
11	E030-04s	10	860.83	0.3	777.96	794.75	5.01	786.40	482.1	1135.3
12	E031-09h	10	724.57	87.2	614.59	618.85	1.35	615.15	181.2	464.8
13	E033-03n	9	3759.54	407.3	2709.55	2736.93	9.00	2728.97	1019.0	1906.8
14	E033-04g	9	1886.89	2039.1	1429.34	1490.77	21.55	1470.59	3494.7	4784.5
15	E033-05s	10	1633.52	0.3	1366.25	1401.87	12.61	1381.56	869.4	1635.1
16	E036-11h	10	809.08	15.6	698.61	698.61	0.00	698.61	40.2	121.5
17	E041-14h	10	984.10	103.8	866.40	875.85	3.28	870.55	223.9	324.4
18	E045-04f	10	1409.29	493.0	1235.30	1273.01	10.18	1253.61	2287.7	3531.7
19	E051-05e	10	950.77	35.2	775.50	789.84	4.36	782.31	1433.8	2358.6
20	E072-04f	5	988.84	4954.4	600.00	604.75	1.89	601.54	6600.5	8155.6
21	E076-07s	10	1332.95	488.3	1118.91	1146.57	7.45	1133.76	3460.0	5079.6
22	E076-08s	10	1409.29	331.8	1182.37	1210.98	7.58	1197.71	2088.3	3265.8
23	E076-10e	10	1481.58	770.7	1151.72	1189.30	11.85	1170.37	2379.4	5119.8
24	E076-14s	10	1511.10	425.6	1125.12	1164.95	12.79	1145.09	2596.3	3924.7
25	E101-08e	10	1725.27	335.3	1427.92	1458.39	8.53	1443.68	2819.4	4915.1
26	E101-10c	10	1944.06	325.9	1590.35	1668.32	21.31	1615.75	2904.7	4292.9
27	E101-14s	10	2019.83	516.3	1555.34	1582.03	7.96	1568.32	2571.5	4769.6
Avg *z*			1174.07	454.3	950.68	970.74	6.17	960.67	1420.1	2296.4

Table F4
DMTS results on Tarantilis' 12 instances (M3L-CVRP).

Id	Instance		Init solution		Best solution					
	Name	#exe	Init	Avg t (s)	Min	Max	Std	Avg	Avg t (s)	Avg tt (s)
28	3l_cvrp_50_1	10	1568.36	0.3	1439.21	1465.60	7.95	1458.71	1492.0	2928.5
29	3l_cvrp_50_2	10	2525.48	90.9	2203.17	2251.67	16.60	2232.97	324.8	520.9
30	3l_cvrp_50_3	10	2434.38	608.9	1794.43	1841.76	14.68	1817.64	3763.5	4912.9
31	3l_cvrp_75_1	10	2383.67	123.6	2034.03	2093.53	18.29	2071.23	2354.4	3595.9
32	3l_cvrp_75_2	10	3711.82	728.5	3110.08	3221.89	36.03	3164.05	1312.3	1458.0
33	3l_cvrp_75_3	9	3514.72	2133.4	2437.17	2523.45	31.11	2486.01	5492.4	6475.1
34	3l_cvrp_100_1	10	3267.25	649.8	2587.67	2705.09	37.25	2641.28	10,066.6	13,961.2
35	3l_cvrp_100_2	10	4947.35	209.0	4202.26	4252.01	18.22	4232.69	742.6	1008.5
36	3l_cvrp_100_3	9	5746.43	5023.5	3935.52	4447.07	159.01	4060.73	9476.3	10,555.4
37	3l_cvrp_125_1	10	3917.27	257.3	3240.09	3337.93	27.67	3269.22	7887.6	11,532.1
38	3l_cvrp_125_2	10	6364.87	1516.4	5409.81	5509.75	34.50	5446.89	2153.9	2531.3
39	3l_cvrp_125_3	3	7361.66	8016.0	4914.65	5263.68	192.41	5135.95	15,887.7	16,831.9
Avg z			3978.60	1613.1	3109.01	3242.79	49.48	3168.11	5079.5	6359.3

Table G1
Comparison on Gendreau's 27 instances (3L-CVRP) with incorrect fleet sizes.

Instance	TS			GTS			ACO			DMTS			
	z	t (s)	tt (s)	z	t(s)	tt (s)	Avg	Avg t (s)	Avg tt (s)	Avg	Avg t (s)	Avg tt (s)	Impr (%)
E016-03m	316.32	129.5	1800.0	321.47	7.8	13.2	305.35	11.2	12.0	302.44	39.4	157.5	0.95
E016-05m	350.58	5.3	1800.0	334.96	7.2	11.5	334.96	0.1	0.6	334.96	3.5	36.7	0.00
E021-04m	447.73	461.1	1800.0	430.95	352.6	540.6	409.79	88.5	121.8	388.35	91.0	237.6	5.23
E021-06m	448.48	181.1	1800.0	458.04	204.0	323.5	440.68	3.9	5.4	439.98	51.2	126.3	0.16
E022-04g	464.24	75.8	1800.0	465.79	61.3	99.6	453.19	22.7	30.9	446.77	173.5	447.1	1.42
E022-06m	504.46	1167.9	1800.0	507.96	768.8	1212.4	501.47	17.5	18.4	499.28	130.6	311.9	0.44
E023-03g	831.66	181.1	1800.0	796.61	241.5	364.8	797.47	51.4	67.4	772.24	421.2	810.9	3.06
E023-05s	871.77	156.1	1800.0	880.93	140.0	230.0	820.67	56.2	78.6	806.93	462.7	711.1	1.67
E026-08m	666.10	1468.5	1800.0	642.22	604.7	982.2	635.50	15.3	16.3	630.13	95.5	518.8	0.85
E030-03g	911.16	714.0	3600.0	884.74	803.1	1308.4	841.12	241.2	246.7	836.44	597.4	1022.8	0.56
E030-04s	819.36	396.4	3600.0	873.43	308.5	522.5	821.04	172.4	199.8	786.35	484.0	1072.8	4.03
E031-09h	651.58	268.1	3600.0	624.24	180.8	294.6	629.07	46.2	48.2	614.94	224.8	463.6	1.49
E033-03n	2928.34	1639.1	3600.0	2799.74	1309.5	2193.1	2739.80	235.4	308.8	2724.80	558.9	1444.0	0.55
E033-04g	1559.64	3451.6	3600.0	1504.44	2678.1	4581.3	1472.26	623.8	642.8	1459.76	1030.3	2560.1	0.85
E033-05s	1452.34	2327.4	3600.0	1415.42	1466.3	2528.3	1405.48	621.0	656.8	1386.78	751.4	1613.4	1.33
E036-11h	707.85	2550.3	3600.0	698.61	2803.2	4256.5	698.92	12.8	14.8	698.61	50.2	131.2	0.00
E041-14h	920.87	2142.5	3600.0	872.79	1208.6	2096.0	870.33	11.8	14.9	869.96	177.2	308.3	0.04
E045-04f	1400.52	1452.9	3600.0	1296.59	1300.9	2275.2	1261.07	2122.2	2209.8	1248.23	1449.7	2225.5	1.02
E051-05e	871.29	1822.3	7200.0	818.68	1438.4	2509.0	781.29	614.3	623.6	776.06	1031.7	2135.0	0.67
E072-04f	732.12	790.0	7200.0	641.57	1284.8	1940.9	611.26	3762.3	3901.0	598.87	1996.3	3300.9	2.03
E076-07s	1275.20	2370.3	7200.0	1159.72	1704.8	2823.4	1124.55	5140.0	5180.6	1121.54	2332.9	4010.8	0.27
E076-08s	1277.94	1611.3	7200.0	1245.35	1663.5	2685.6	1197.43	2233.6	2290.3	1189.41	1348.3	2679.2	0.67
E076-10e	1258.16	6725.6	7200.0	1231.92	3048.2	4659.1	1171.77	3693.4	3727.6	1158.46	1797.8	3644.6	1.14
E076-14s	1307.09	6619.3	7200.0	1201.96	2876.8	4854.1	1148.70	1762.8	1791.5	1150.16	1687.8	2805.0	**−0.13**
E101-08e	1570.72	5630.9	7200.0	1457.46	3432.0	5725.8	1436.32	8619.7	8817.1	1432.13	3580.5	4422.1	0.29
E101-10c	1847.95	4123.7	7200.0	1711.93	3974.8	6283.1	1616.99	6651.2	6904.3	1606.93	2695.3	3733.7	0.62
E101-14s	1747.52	7127.2	7200.0	1646.44	5864.2	9915.7	1573.50	10,325.8	10,483.9	1559.81	2057.4	3889.8	0.87
Avg z	1042.26			997.18			966.67			957.05			1.11
Avg time		2058.9	4200.0		1471.6	2415.9		1746.5	1793.1		937.8	1660.0	

values given in their table for reference in Table G1. For convenience, we also reproduce in this table the published results for TS, GTS and ACO.

Appendix H. Comparison on 3D strip packing instances

We compared our packing algorithm Load-Vehicle (Algorithm 1) with the best known algorithms for the 3D strip packing algorithm by Bortfeldt and MacK [3,1]. Table H1 summarizes the result on the data sets SP-BR1 to SP-BR10, which are commonly used as benchmark instances for the 3D strip packing problem. Each data set consists of 10 instances, and for each instance our Load-Vehicle is executed 10 times using different random seeds, the average is reported under the column *Load-Vehicle (K=150)*. Only 3D loading

Table H1
Comparison on 3D Strip packing instances.

Data set	Bortfeldt and Mack [3]	Atten et al. [1]	Load-vehicle (K=150)
SP-BR1	87.3	90.0	81.2
SP-BR2	88.6	89.6	81.3
SP-BR3	89.4	89.0	79.5
SP-BR4	90.1	88.8	76.8
SP-BR5	89.3	88.5	76.3
SP-BR6	89.7	88.6	75.8
SP-BR7	89.2	88.7	73.7
SP-BR8	87.9	88.3	72.7
SP-BR9	87.3	87.9	71.6
SP-BR10	87.6	87.9	71.4
Avg	88.6	88.7	76.0

constraint is considered, all other considerations such as LIFO, support area and fragility are ignored.

References

[1] Allen SD, Burke EK, Kendall G. A hybrid placement strategy for the three-dimensional strip packing problem. European Journal of Operational Research 2011;209:219–27.
[2] Bortfeldt A, Gehring H. A hybrid genetic algorithm for the container loading problem. European Journal of Operational Research 2001;131:143–61.
[3] Bortfeldt A, MacK D. A heuristic for the three-dimensional strip packing problem. European Journal of Operational Research 2007;183:1267–79.
[4] Chazelle B. The Bottom-left bin-packing heuristic: an efficient implementation. IEEE Transactions on Computers 1983;C-32:697–707.
[5] Chien CF, Wu WT. A recursive computational procedure for container loading. Computers & Industrial Engineering 1998;35:319–22.
[6] Clarke G, Wright JW. Scheduling of vehicles from a central depot to a number of delivery points. Operations Research 1964;12:568–81.
[7] Cordeau J-F, Gendreau M, Hertz A, Laporte G, Sormany J-S. New heuristics for the vehicle routing problem. in: Langevin A, Riopel D, editors. Logistics systems: design and optimization. New York: Springer-Verlag; 2005. p. 279–97 [Chapter 9].
[8] Dowsland KA. An exact algorithm for the pallet loading problem. European Journal of Operational Research 1987;31:78–84.
[9] Eley M. Solving container loading problems by block arrangement. European Journal of Operational Research 2002;141:393–409.
[10] Fanslau T, Bortfeldt A. A tree search algorithm for solving the container loading problem. INFORMS Journal on Computing 2010;22:222–35.
[11] Fekete SP, Schepers J, van der Veen JC. An exact algorithm for higher-dimensional orthogonal packing. Operations Research 2007;55:569–87.
[12] Fuellerer G, Doerner KF, Hartl RF, Iori M. Metaheuristics for vehicle routing problems with three-dimensional loading constraints. European Journal of Operational Research 2010;201:751–9.
[13] Gendreau M, Iori M, Laporte G, Martello S. A tabu search algorithm for a routing and container loading problem. Transportation Science 2006;40:342–50.
[14] Gendreau M, Iori M, Laporte G, Martello S. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. Networks 2008;51:4–18.
[15] Healy P, Creavin M, Kuusik A. An optimal algorithm for rectangle placement. Operations Research Letters 1999;24:73–80.
[16] Hopper E. Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods. PhD thesis. School of Engineering, University of Wales University of Wales, Cardiff; 2000.
[17] Iori M, Salazar-González J-J, Vigo D. An exact approach for the vehicle routing problem with two-dimensional loading constraints. Transportation Science 2007;41:253–64.
[18] Lim A, Rodrigues B, Yang Y. 3-D container packing heuristics. Applied Intelligence 2005;22:125–34.
[19] Lim A, Zhang X. The container loading problem. in: SAC '05: proceedings of the 2005 ACM symposium on applied computing. New York, NY, USA: ACM; 2005. p. 913–7.
[20] Lins L, Lins S, Morabito R. An n-tet graph approach for non-guillotine packings of n-dimensional boxes into an n-container. European Journal of Operational Research 2002;141:421–39.
[21] Malapert A, Guéret C, Jussien N, Langevin A, Rousseau L-M. Two-dimensional pickup and delivery routing problem with loading constraints. In: CPAIOR'08 first workshop on bin packing and placement constraints (BPPC'08). Paris, France; 2008.
[22] Martello S, Pisinger D, Vigo D. The three-dimensional bin packing problem. Operations Research 2000;48:256–67.
[23] Martello S, Pisinger D, Vigo D, Den Boef E, Korst J. Algorithm 864: general and robot-packable variants of the three-dimensional bin packing problem. ACM Transactions on Mathematical Software 2007;33. 7+.
[24] Moura A, Oliveira JF. An integrated approach to the vehicle routing and container loading problems. OR Spectrum 2009;31:775–800.
[25] Parreño F, Alvarez-Valdes R, Tamarit JM, Oliveira JF. A maximal-space algorithm for the container loading problem. INFORMS Journal on Computing 2008;20:412–22.
[26] Pisinger D. Heuristics for the container loading problem. European Journal of Operational Research 2002;141:382–92.
[27] Tarantilis CD, Zachariadis EE, Kiranoudis CT. A hybrid metaheuristic algorithm for the integrated vehicle routing and three-dimensional container-loading problem. IEEE Transactions on Intelligent Transportation Systems 2009;10:255–71.
[28] Toth P, Vigo D, editors. The vehicle routing problem. Monographs on discrete mathematics and applications. SIAM; 2001.
[29] Zachariadis EE, Tarantilis CD, Kiranoudis CT. A guided tabu search for the vehicle routing problem with two-dimensional loading constraints. European Journal of Operational Research 2009;195:729–43.