# Space Defragmentation Heuristic for 2D and 3D Bin Packing Problems

**Zhaoyi Zhang** and **Songshan Guo**
Zhong Shan (Sun Yat-Sen) Univ.
Guangzhou, P.R. China
zzy.sysu@gmail.com
issgssh@mail.sysu.edu.cn

**Wenbin Zhu**[*]
HKUST and HK Poly Univ.
Clear Water Bay, HK S.A.R.
i@zhuwb.com

**Wee-Chong Oon** and **Andrew Lim**
City Univ. of HK
Kowloon Tong, HK S.A.R.
{weecoon, lim.andrew}@cityu.edu.hk

## Abstract

One of main difficulties of multi-dimensional packing problems is the fragmentation of free space into several unusable small parts after a few items are packed. This study proposes a defragmentation technique to combine the fragmented space into a continuous usable space, which potentially allows the packing of additional items. We illustrate the effectiveness of this technique on the two- and three-dimensional Bin Packing Problems. In conjunction with a bin shuffling strategy for incremental improvement, our resultant algorithm outperforms all leading meta-heuristic approaches.

## 1 Introduction

Packing and cutting problems have wide practical applications and many variants have been extensively studied, especially the packing and cutting of rectangular two- and three-dimensional items. They model many industrial applications, such as the packing of cargo for shipment and the cutting of sheets of metal, glass, paper or other materials. Although the many variants of packing and cutting problems differ in objectives and have their own unique challenges, the multi-dimensional variants (i.e., greater than 1-D) share one commonality: denser packings tend to lead to higher quality solutions. A common obstacle to obtaining a dense packing is the fragmentation of unutilized (free) space into multiple small parts that are too small to hold any items and are therefore unusable.

In this study, we address the space fragmentation issue by designing two space defragmentation operations. The operations are based on the concept of pushing loaded items along an axis in order to leave sufficient free space for an incoming item. The distance that a loaded item can be pushed along an axis can be computed using a comparability graph representation of the packing pattern for each axis; we show how these operations can be performed in $O(n^2)$ time without explicitly constructing the graphs.

We use the three-dimensional bin packing problem (3D-BPP) as the target problem in order to illustrate the concept of space defragmentation. Our experiments on standard benchmark problems show that by incorporating our space defragmentation operations into the *extreme point insertion* constructive heuristic for 3D-BPP, we can obtain solutions that are comparable to those achieved by the leading existing meta-heuristic approaches for the problem in a fraction of the time. With the addition of a simple incremental improvement procedure using bin shuffling, our algorithm outperforms all existing techniques by a significant amount for both the two-dimensional bin packing problem (2D-BPP) and 3D-BPP.

The 3D-BPP analyzed in this paper is defined as follows. We are given $n$ items that are 3D rectangular boxes, and an unlimited number of identical bins with dimensions $L \times W \times H$. The dimensions of the $i$-th item, $1 \leq i \leq n$, is denoted by $l_i \times w_i \times h_i$, and we assume that the boxes cannot be rotated. The aim is to load all $n$ items into bins such that:

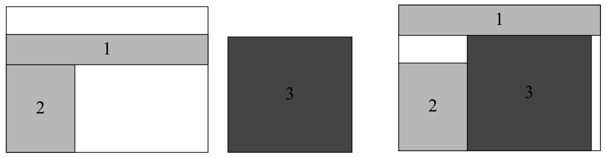
- Placement is orthogonal (i.e., axis-aligned)
- Every item must be completely inside a bin
- Any two items inside the same bin must be interior-disjoint (i.e., non-overlapping)
- The number of bins used is minimized.

Bin packing problems of dimensions other than three can be defined similarly. We assume that bins are placed in the first octant of a Cartesian coordinate system with one corner at the origin. The length $L$ of the bin is parallel to the $X$-axis, the height $H$ is parallel to the $Y$-axis, and the width $W$ is parallel to the $Z$-axis.

## 2 Space Defragmentation

A primary objective for bin packing problems (and packing problems in general) is to identify dense packing patterns so that the utilization of space within each bin is high. When this is achieved, the number of bins required is naturally reduced. The main obstacle against achieving a dense packing of items is the fragmentation of usable space as items are loaded into bins. For example, Figure 1(a) shows the situation inside a bin after two items are loaded. The usable space in the bin is divided into two disconnected parts, and even though the total area of usable space in the bin is sufficient to load item 3, neither part is large enough to accommodate the item. However, by pushing item 1 upwards, we can make enough room to load item 3 (Figure 1(b)).

[*]Corresponding Author



(a) Usable space is fragmented, item 3 cannot be packed

(b) Item 1 is pushed upwards to make room for item 3

Figure 1: Example of space defragmentation

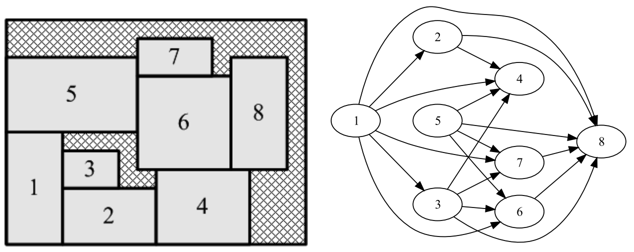


Figure 2: A packing pattern and its $X$-comparability graph

This is the basis for our space defragmentation operations. In particular, when attempting to place an item $i$ at some position $p$ (i.e., the corner of item $i$ that is closest to the origin is at point $p$) and there is not enough space to accommodate the item due to the presence of existing items, we can attempt to push all items that overlap with $i$ away from the origin in order to make room for the new item.

There are two tasks to perform when implementing this concept. Firstly, we must determine if there is sufficient space to accommodate item $i$ after pushing the items away from $p$; secondly, the item $i$ must be loaded at $p$. In this section, we present an $O(n^2)$ algorithm that utilizes the space defragmentation concept by efficiently determining the distance that an item can be pushed along an axis, using the concept of a *comparability graph* for an axis.

For any item $i$ in a bin, its projection onto the $X$-axis is an interval that we denote by $[\underline{x}_i, \overline{x}_i]$. Therefore, the length of an item $i$ is $l_i = \overline{x}_i - \underline{x}_i$. We can construct the *$X$-comparability graph* $G_X = (V_X, E_X)$ for the set of intervals corresponding to the items in a bin as follows:

- Every item $i$ corresponds to a vertex $v_i \in V_X$;
- Each vertex $v_i$ is assigned a weight equal to $l_i$;
- There is a directed edge $(v_i, v_j) \in E_X$ from vertex $v_i$ to $v_j$ if and only if $\overline{x}_i \leq \underline{x}_j$, i.e., item $i$ lies entirely to the left of item $j$.

$G_X$ is a directed acyclic graph; similar graphs can be constructed for the other axes in the same manner. Figure 2 shows an example of a packing in a bin along with the corresponding $X$-comparability graph. We define the *length* of a path as the sum of the weights of all vertices in the path.

Given a packing with $X$-comparability graph $G_X$, we can *right-justify* all items along the $X$-axis such that $[\underline{x}_i^r, \overline{x}_i^r]$ is the projection of item $i$ on the $X$-axis after right-justification, where

$$\overline{x}_i^r = \begin{cases} \min\limits_{(v_i,v_j)\in E_X} \{\underline{x}_j^r\} & : \exists (v_i, v_j) \in E_X \\ L & : \text{otherwise} \end{cases} \quad (1)$$

Note that a feasible packing with $X$-comparability graph $G_X$ remains feasible after all items are right-justified along the $X$-axis. Furthermore, the comparability graphs for the other axes for the resultant packing are unchanged.

For any given point $x$ on the $X$-axis, we call the set of items whose right endpoint lies to the right of $x$ the *right set at* $x$, denoted by $R_x = \{i : \overline{x}_i > x\}$. The set of all other items in the bin is called the *left set at* $x$, denoted by $L_x = \{i : \overline{x}_i \leq x\}$. We define an operator PUSH$(G_X, x)$, which right-justifies every item $i \in R_x$ along the $X$-axis.

For three-dimensional packing problems, we define a composite operator PUSH-OUT$(b, (x, y, z))$, which attempts to collate usable space in bin $b$ around a reference point $(x, y, z)$ by performing PUSH$(G_X, x)$; PUSH$(G_Y, y)$; and PUSH$(G_Z, z)$. The three PUSH operations can be performed in any order, and the resultant packing will be the same.

We now describe how $\overline{x}_i^r$ can be computed in $\Theta(n \log n)$ time for all items without explicitly constructing the $X$-comparability graph $G_X$, where $n$ is the number of items in the bin. This is done by computing the value of $\Delta X_i = \overline{x}_i^r - \overline{x}_i$ for each item $i$. The computation is similar for the other dimensions.

First, sort the $2n$ endpoints of the $n$ intervals corresponding to the projections of the items on the $X$-axis; left endpoints take precedence in a tie. We explain the procedure by imagining a vertical line sweeping from right to left along the $X$-axis. The vertical line serves two purposes:

1) It defines the set of intervals that lie completely to the left of the vertical line, i.e., $L_x = \{i | \overline{x}_i \leq x\}$, where $x$ is the location of the vertical line;
2) It maintains a boundary $x_0$ that marks the greatest value for the right endpoint of any interval in $L_x$; conceptually, we can simultaneously translate all intervals in $L_x$ to the right until the right endpoint of some interval in $L_x$ coincides with the boundary. If $i$ is the interval with largest $\overline{x}_i$ in $L_x$, then $\Delta X_i = x_0 - \overline{x}_i$

At the beginning, we set the boundary $x_0 = L$. When the vertical line encounters a right endpoint $\overline{x}_i$, we update $\Delta X_i = x_0 - \overline{x}_i$. When the vertical line encounters a left endpoint $\underline{x}_i$, we update the boundary $x_0 = \min\{x_0, \underline{x}_i + \Delta X_i\}$. This correctly updates the invariant $x_0$ because for any interval $j \in L_x$, $j$ lies to the left of $[\underline{x}_i, \overline{x}_i]$. By the construction of the comparability graph, there is an edge from $j$ to $i$; hence, $i$ lies on some path passing through $j$. If $i$ in fact lies on the longest path $P(j)$ involving $v_j$, then $\underline{x}_i + \Delta X_i$ is the effective right boundary for $j$. The pseudocode is provided in Algorithm 1.

Finally, we show that once the values of $\overline{x}_i^r$; $\overline{y}_i^r$; and $\overline{z}_i^r$ are computed for all items $i$, then determining if PUSH-OUT$(b, (x, y, z))$ produces sufficient space to accomodate an item can be done in $O(n)$ time.

**Algorithm 1** Compute $\overline{x}_i^r$ for each item $i$ in bin $b$

```
COMPUTE-RJPOS-X($b$)
1   $P$ = list of endpoints $\underline{x}_i$ and $\overline{x}_i$ for all items $i \in b$
2   sort $P$ by $X$-coordinate in descending order;
        left endpoints take precedence in a tie
3   $x_0 = L$
4   for each point $x \in P$
5       if $x$ is a left endpoint $\underline{x}_i$ for some $i$
6           $x_0 = \min(x_0, \Delta X_i + \underline{x}_i)$
7       if $x$ is a right endpoint $\overline{x}_i$ for some $i$
8           $\Delta X_i = x_0 - \overline{x}_i$
9   for all items $i$
10      $\overline{x}_i^r = \overline{x}_i + \Delta X_i$
```

Let $S(p)$ be the set of items that overlap with the current item $i$ when $i$ is placed at a position $p$ (i.e., $p = (\underline{x}_i, \underline{y}_i, \underline{z}_i)$). For every item $j \in S(p)$, we right-justify it along the respective axes. Let $S'(p)$ denote the set of items after translation. The operation PUSH-OUT$(b, p)$ will produce sufficient space to allow the current item $i$ to be placed at $p$ if and only if no items in $S'(p)$ overlap with item $i$ when item $i$ is placed at $p$.

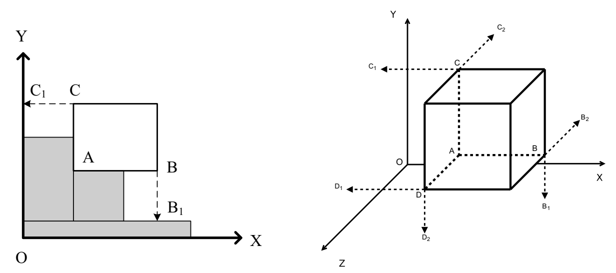
We can therefore determine if item $i$ can be placed at $p$ by checking the resultant position of all items $j$ that overlap with $i$ after right-justification on axes. Since the number of overlapping items is $O(n)$, and assuming there are $O(n)$ possible positions, this procedure takes $O(n^2)$ time per item. In fact, the worst case scenario seldom occurs in practice, and our experiments show that this operation runs in close to linear time on standard 3D-BPP test cases.

## 3 A Constructive Heuristic

The *extreme point insertion* heuristic [Crainic *et al.*, 2008] is a constructive heuristic that loads items one at a time based on a given sequence until all items are loaded, and is the best existing constructive heuristic for the 3D-BPP. It represents the state of a bin by a list of extreme points, where every extreme point is a candidate position to load a new item. An empty bin is represented by one extreme point (0, 0, 0). When a new item is loaded, it occupies one extreme point and introduces a constant number of new extreme points. Figures 3(a) and 3(b) illustrate how the loading of a new item will introduce up to 2 and 6 new points in the 2D and 3D cases, respectively.

Given a sequence of items, the extreme point insertion heuristic attempts to load the current item at an extreme point in an existing bin. If no such point exists, a new empty bin is instantiated and the item is loaded into that bin at (0,0,0). This continues until all items are loaded. We employed the *first fit* strategy when selecting the extreme point for the current item: the current item is loaded into the first bin that can accommodate it at the first feasible extreme point.

We introduce two space defragmentation enhancements to the extreme point insertion algorithm; the resultant algorithm is given in Algorithm 2. The first enhancement is a straightforward application of the technique described in Section 2, i.e., rather than checking if the current item $i$ can be placed



(a) 2 extreme points for 2D (b) 6 extreme points for 3D

Figure 3: Extreme Points

at $p$ into the bin $b$ given the current packing, the algorithm checks if it can be placed at $p$ after the PUSH-OUT$(b, p)$ procedure (line 6). This involves calculating the values of $\overline{x}_i^r$, $\overline{y}_i^r$ and $\overline{z}_i^r$ for the items in the bin using Algorithm 1, and then determining if the insertion at $p$ is feasible in $O(n)$ time. If the insertion is feasible, then PUSH-OUT$(b, p)$ is performed, the item is loaded, and then the bin is normalized (i.e., all items are pushed towards the origin as far as possible).

**Algorithm 2** Extreme Point Insertion with Space Defragmentation

```
EP-INSERT-SD($I, B$)
1   for each item $i \in I$
2       $placed$ = FALSE
3       for each bin $b \in B$
4           if volume of $i$ is less than remaining space in $b$
5               for each point $p \in b.EP\text{-}list$
6                   if $i$ can be placed at $p$ after PUSH-OUT
7                       $placed$ = TRUE
8                       PUSH-OUT$(b, p)$
9                       place item $i$ at $p$
10                      NORMALIZE(b)
11                      update extreme points $b.EP\text{-}list$
12                      update $\overline{x}_i^r, \overline{y}_i^r, \overline{z}_i^r$ for all items in $b$
13                      break and try to load the next item
14      if $placed$ == FALSE
15          for each bin $b \in B$
16              for each item $j \in b$
17                  if INFLATE-REPLACE$(b, i, j)$ == TRUE
18                      $placed$ = TRUE
19                      NORMALIZE(b)
20                      update extreme points $b.EP\text{-}list$
21                      update $\overline{x}_i^r, \overline{y}_i^r, \overline{z}_i^r$ for all items in $b$
22                      insert $j$ to the front of $I$
23                      break and try to load the next item
24      if $placed$ == FALSE
25          add an empty bin $b'$ to the end of $B$
26          place item $i$ into $b'$ at $(0, 0, 0)$
27          append new extreme points to $b'.EP\text{-}list$
```

We describe the second enhancement with the conceptual example given in Figure 4, where three items have previously been loaded into a bin; as a result, item 4 cannot be loaded.

However, suppose we inflate item 3 as much as possible (by pushing other items away from the origin if neccessary), resulting in the inflated item 3'. If item 3' is large enough to encompass item 4, then we could replace item 3' by item 4 (Figure 4(c)). As long as the volume of item 3 is smaller than the volume of item 4, then performing this Inflate-Replace operation will increase the space utilization of the bin.

The maximum amount of inflation for an item $j$ along an axis is the same as the maximum distance that the item can be translated along that axis. Hence, we can inflate an item $j$ along the $X$-; $Y$-; and $Z$-axes such that its right endpoints become $\overline{x}_j^r$; $\overline{y}_j^r$; and $\overline{z}_j^r$, respectively. This procedure is denoted by INFLATE-REPLACE$(b, i, j)$, which attempts to inflate $j$ and replace it by $i$ if $i$ is larger than $j$.

We employ the second enhancement when all existing non-empty bins are unable to accommodate the current item (line 14). At this point, we consider all loaded items $j$ in each existing bin $b$ in turn, and attempt the INFLATE-REPLACE$(b, i, j)$ procedure. A new empty bin is added only if this strategy is unsuccessful (line 24).

The procedure NORMALIZE moves all items as close to the origin as possible, resulting in a normalized packing. We use a standard normalization procedure for this purpose, which translates all items in the bin as far as possible towards the origin along the $X$-axis, then the $Y$-axis, then the $Z$-axis, and repeats until none of the items can be further translated towards the origin along all three axes. After normalization, we recompute all the extreme points in the bin, which takes $O(n^2)$ time where $n$ is the number of items in the bin.

## 4 Bin Shuffling

To further improve the performance of our algorithm, we implemented a simple *bin shuffling* strategy that employs our EP-INSERT-SD procedure as a subroutine. As items are loaded when constructing a solution, their sequence of insertion into each bin is recorded. We then permute the bins while preserving the insertion sequences for each bin to obtain a new sequence of items; this sequence can then be used as a new input for EP-INSERT-SD to produce a new solution.

Given a current solution, we can use Algorithm 3 to search for an improved solution. First, the algorithm identifies the bin with the lowest volume utilization and unloads all items from this bin (line 2); this results in a list of non-empty bins $B$ and a list of items $U$ to be loaded. We shuffle the bins in $B$ to obtain a new sequence of items $I'$ (line 6). Then, we insert the largest item $u \in U$ at a random position in $I'$ and invoke the EP-INSERT-SD procedure on $I'$ to obtain a solution $B'$. If $B'$ has an equal number or fewer bins than $B$ (line 9), then we have successfully loaded $u$ into existing bins. This successful insertion suggests that the current sequence $I'$ produces a good solution using EP-INSERT-SD, so we proceed to attempt to insert all other items in $U$ (in decreasing order of volume) into $B'$ using the same sequence (line 12). However, if $B'$ has more bins than $B$, then we repeat the bin shuffling operation up to $K$ times; this is essentially a local search. After $K$ consecutive unsuccessful attempts, we randomly remove an item from $B$ (line 15), insert it into $U$ and restart the procedure. This continues until $U = \emptyset$, whereupon we have found a solution that uses one fewer bin.

**Algorithm 3** Solution Improvement using Bin Shuffling

```
BIN-SHUFFLE-IMPROVE($B$)
1  $b$ = the bin with lowest volume utilization in $B$
2  $U$ = all items in bin $b$ sorted by decreasing volume
3  remove $b$ from $B$
4  while $U$ is not empty
5      for $k = 1$ to $K$
6          $I' = \text{Shuffle}(B)$
7          randomly insert first item $u$ from $U$ into $I'$
8          EP-INSERT-SD$(I', B')$
9          if $B'$ has equal number or fewer bins than $B$
10             $B = B'$
11             remove $u$ from $U$
12             load as many items from $U$ as possible
                   into $B$ using EP-INSERT-SD
13             break
14     if $k > K$
15         randomly remove an item from $B$ and
               insert it into $U$
16 return $B$
```

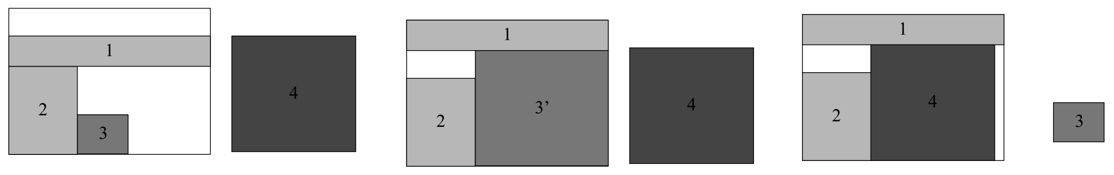
In our implementation, we initially set $K$ to 200. Whenever a new search state is accepted, we update the value of $K$ dynamically: if the number of items in $U$ is the lowest found so far, then we set $K = 200$; otherwise, we set $K = 50$.

Our overall approach is as follows. We first sort all items in descending order of volume, breaking ties by descending order of height (as suggested by [Crainic *et al.*, 2008]), and then invoke the EP-INSERT-SD procedure to construct an initial solution. While the time limit is not exceeded, we iteratively improve the solution using Algorithm 3. We refer to our algorithm as Bin Shuffling using Extreme Point insertion with Space Defragmentation (BS-EPSD). It can be employed on $d$-dimensional bin packing problems for $d \geq 2$.

## 5 Computational Experiments

We compared our BS-EPSD algorithm with the leading algorithms for the 2D and 3D bin packing problems. The experiments were conducted on a rack mounted server with Intel Xeon E5520 Quad-Core CPUs running at 2.26GHz. The operating system is SuSE Linux Enterprise Server 10 SP2. The 64-bit Java Development Kit 1.6.0 from Sun Microsystems was used to implement the algorithm.

For the 2D-BPP, we considered two sets of standard benchmark instances. The first set was generated by [Berkey and Wang, 1987] and consists of 6 classes, which we number 1-6; the second set was generated by [Martello and Vigo, 1998] and consists of 4 classes, which we number 7-10. Each class of instances is further divided into 5 groups, where every group consists of 10 instances with same number of items; the number of items per instance in the five groups are 20, 40, 60, 80 and 100, respectively. All 500 instances and their corresponding best known solution values are available at `http://www.or.deis.unibo.it/research_pages/ORinstances/ORinstances.htm`.



(a) Item 4 cannot be loaded (b) Inflating item 3 (c) Replacing item 3 by item 4

Figure 4: Inflate-Replace Operation

For the 3D-BPP, we used the 320 instances generated by [Martello *et al.*, 2000]. This set of instances consists of 8 classes, and each class is further divided into 4 groups. Every group consists of 10 instances with same number of items; the number of items per instance in the 4 groups are 50, 100, 150 and 200, respectively. This set of instances can be reproduced by the instance generator available at `http://www.diku.dk/~pisinger/codes.html`.

Our computational results are summarized in Table 1. Each entry is the sum of the average number of bins required for each group. A dash ('-') indicates that the results for that class of instances were not reported in the corresponding publication. Columns 3 to 8 correspond to the leading meta-heuristic algorithms in existing literature:

- GRASP: hybrid GRASP/VND [Parreño *et al.*, 2008]
- SCH: Set Cover Heuristic [Monaci and Toth, 2006]
- GLS: Guided Local Search [Faroe *et al.*, 2003]
- TS3: Tabu Search [Lodi *et al.*, 1999] for 2D bin packing; [Lodi *et al.*, 2002] for 3D bin packing
- HBP: HBP heuristic [Boschetti and Mingozzi, 2003]
- TS$^2$Pack: Two-level Tabu Search [Crainic *et al.*, 2009]

The column *Init* gives the statistics of the initial solutions constructed by EP-INSERT-SD, where the items are sorted in descending order of volume, while the column *BS-EPSD* corresponds to the final results produced by our algorithm. The time limit is set to 10 seconds of CPU time for each 2D instance, and 30 seconds of CPU time for each 3D instance, which are considerably stricter time limits than those imposed by existing approaches; this is to address the possibility that the improvement in our results compared to older techniques is due to the increased processing speeds of our modern machines. The row *All classes* gives the sum of the values for all test instances. However, for the 3D instances, classes 2 and 3 have been omitted by many of the existing techniques; hence, we also report the sum of the values over classes 1 and 4-8.

An inspection of the *Init* column shows that the quality of the initial solution constructed by EP-INSERT-SD is comparable to the solutions produced by the leading meta-heuristic algorithms; the gap between our initial solution and the best algorithms for 2D instances are within 2%, and for 3D instances they are within 3%. This is remarkable because EP-INSERT-SD is a simple constructive heuristic, and prior to this work meta-heuristics have outperformed simple heuristics by large margins [Crainic *et al.*, 2008]. With the inclusion of bin shuffling, column *BS-EPSD* shows that our technique is superior to all existing algorithms for both 2D and 3D instances.

Detailed results show that the BS-EPSD algorithm achieved equal or superior solutions compared to the leading meta-heuristic approaches for all 50 groups of 2D instances, and it found equivalent or superior solutions compared to the leading meta-heuristic approaches for all but 7 out of the 32 groups of 3D instances. The detailed results and other supplementary materials (including experiments performed on newly generated test data) can be found at `http://www.zhuwb.com/3d-bpp`.

## 6 Conclusion

Space defragmentation is a natural concept that has not been employed in existing approaches to packing problems. When packing items manually, humans often move existing items in order to allow an extra item to be loaded, and also replace a smaller item with a larger one; this paper presents algorithms that mimic these operations. By incorporating space defragmentation into the extreme point insertion constructive heuristic along with a bin-shuffling improvement strategy, the resultant algorithm outperformed all existing approaches for both the 2- and 3-dimensional bin packing problem.

The concept of space defragmentation introduces a new class of operators that is readily applied to a wide range of $d$-dimensional cutting and packing problems. Other operators that employ the space defragmentation concept can be devised. For instance, a PUSH-AND-REPLACE operator might first perform PUSH-OUT, and then replace the items that overlap with the target item when placed at position $p$ if the total volume utilization is increased as a result.

Note that the comparability graph computation of the maximum translation distance is conservative, i.e., it may be possible for an item $i$ to be translated along the $X$-axis further than $\underline{x}_i^r$. In order to compute the actual maximum translation distance, conceptually we can instead use a *visibility graph*. Unfortunately, we are currently unable to find an efficient visibility graph implementation of space defragmentation better than $O(n^3)$, and experiments show that using the comparability graph implementation is distinctly superior for 3D-BPP. Also note that unlike for comparability graphs, the visibility graph version of the PUSH operation along an axis may alter the visibility graphs for other axes, which must therefore be recomputed after each execution of PUSH. Furthermore,

Table 1: Summarized results for standard test cases

| | Class | GRASP | SCH | GLS | TS3 | HBP | TS$^2$Pack | Init | BS-EPSD |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | **99.7** | **99.7** | 100.2 | 101.5 | 99.9 | - | 100.9 | **99.7** |
| | 2 | **12.4** | **12.4** | **12.4** | 13.0 | **12.4** | - | 12.9 | **12.4** |
| | 3 | 69.6 | 69.6 | 70.2 | 72.6 | 70.3 | - | 72.2 | **69.5** |
| | 4 | **12.3** | 12.4 | 12.5 | 12.8 | 12.5 | - | 12.6 | **12.3** |
| 2D | 5 | 89.3 | 89.3 | 90.2 | 91.3 | 89.9 | - | 91.4 | **89.2** |
| | 6 | 11.2 | 11.2 | 11.4 | 11.5 | 11.3 | - | 11.5 | **11.1** |
| | 7 | 82.8 | **82.7** | 83.4 | 84.0 | 83.2 | - | 85.0 | **82.7** |
| | 8 | **83.4** | 83.6 | 84.1 | 84.4 | 83.9 | - | 85.1 | **83.4** |
| | 9 | **213.0** | **213.0** | **213.0** | 213.1 | **213.0** | - | 213.7 | **213.0** |
| | 10 | 50.4 | 50.4 | 51.0 | 51.8 | 51.1 | - | 51.5 | **50.3** |
| | All classes | 724.1 | 724.3 | 728.4 | 736.0 | 727.5 | - | 736.8 | **723.6** |
| | 1 | **127.3** | - | 128.3 | 127.9 | - | 128.2 | 132.2 | 127.4 |
| | 2 | **125.8** | - | - | 126.8 | - | - | 130.3 | **125.8** |
| | 3 | 126.9 | - | - | 127.5 | - | - | 131.3 | **126.8** |
| 3D | 4 | 294.0 | - | 294.2 | 294.0 | - | **293.9** | 296.0 | 294.0 |
| | 5 | **70.5** | - | 70.8 | 71.4 | - | 71.0 | 73.0 | **70.5** |
| | 6 | **95.4** | - | 96.0 | 96.1 | - | 95.8 | 97.9 | 95.6 |
| | 7 | 59.4 | - | 59.0 | 60.0 | - | 59.0 | 61.1 | **58.5** |
| | 8 | 82.0 | - | 81.9 | 82.6 | - | 81.9 | 84.8 | **81.3** |
| | All classes | 981.3 | - | - | 986.3 | - | - | 1006.6 | **979.9** |
| | Class 1, 4-8 | 728.6 | - | 730.2 | 732.0 | - | 729.8 | 745.0 | **727.3** |

the order of the axes chosen is not independent, e.g., translating along the $X$-axis followed by the $Y$-axis may result in a different packing from translating along the $Y$-axis first.

## Acknowledgments

We would like to thank Ramon Alvarez-Valdes and Francisco Parreño Torres, the co-authors of the GRASP/VND algorithm, for so graciously providing a binary executable of their approach for our additional experiments.

## References

[Berkey and Wang, 1987] J. O. Berkey and P. Y. Wang. Two-Dimensional Finite Bin-Packing Algorithms. *The Journal of the Operational Research Society*, 38(5):423–429, 1987.

[Boschetti and Mingozzi, 2003] Marco A. Boschetti and Aristide Mingozzi. The Two-Dimensional Finite Bin Packing Problem. Part II: New lower and upper bounds. *4OR: A Quarterly Journal of Operations Research*, 1(2):135–147, June 2003.

[Crainic *et al.*, 2008] Teodor G. Crainic, Guido Perboli, and Roberto Tadei. Extreme Point-Based Heuristics for Three-Dimensional Bin Packing. *INFORMS Journal on Computing*, 20(3):368–384, June 2008.

[Crainic *et al.*, 2009] Teodor G. Crainic, Guido Perboli, and Roberto Tadei. TS2PACK: A two-level tabu search for the three-dimensional bin packing problem. *European Journal of Operational Research*, 195(3):744–760, June 2009.

[Faroe *et al.*, 2003] Oluf Faroe, David Pisinger, and Martin Zachariasen. Guided Local Search for the Three-Dimensional Bin-Packing Problem. *INFORMS Journal on Computing*, 15(3):267–283, January 2003.

[Lodi *et al.*, 1999] Andrea Lodi, Silvano Martello, and Daniele Vigo. Heuristic and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems. *INFORMS Journal on Computing*, 11(4):345–357, January 1999.

[Lodi *et al.*, 2002] Andrea Lodi, Silvano Martello, and Daniele Vigo. Heuristic algorithms for the three-dimensional bin packing problem. *European Journal of Operational Research*, 141(2):410–420, September 2002.

[Martello and Vigo, 1998] Silvano Martello and Daniele Vigo. Exact Solution of the Two-Dimensional Finite Bin Packing Problem. *Management Science*, 44(3):388–399, March 1998.

[Martello *et al.*, 2000] Silvano Martello, David Pisinger, and Daniele Vigo. The Three-Dimensional Bin Packing Problem. *Operations Research*, 48(2):256–267, March 2000.

[Monaci and Toth, 2006] Michele Monaci and Paolo Toth. A Set-Covering-Based Heuristic Approach for Bin-Packing Problems. *INFORMS Journal on Computing*, 18(1):71–85, January 2006.

[Parreño *et al.*, 2008] F. Parreño, R. Alvarez-Valdes, J. F. Oliveira, and J. M. Tamarit. Ahybrid GRASP/VND algorithm fortwo-andthree-dimensional bin packing. *Annals of Operations Research*, 179(1):203–220, October 2008.