

A New Method For The Three Dimensional Container Packing Problem

Andrew Lim and Wang Ying

Department of Computer Science, National University of Singapore
3 Science Drive 2, Singapore 117543

Abstract

A new algorithm for solving the three dimensional container packing problem is proposed in this paper. This new algorithm deviates from the traditional approach of wall building and layering. It uses the concept of “building growing” from multiple sides of the container. We tested our method using all 760 test cases from the OR-Library. Experimental results indicate that the new algorithm is able to achieve an average packing utilization of more than 87%. This is better than the results reported in the literature.

1 Introduction

Packing boxes into containers is a common problem faced by the logistics and transportation industries. The three dimensional (3D) packing problem is a difficult extension of the two dimensional (2D) stock cutting problem and the one dimensional (1D) bin packing problem. Even the 1D bin packing problem is known to be NP-Complete. For a survey of packing problems, please refer to [K.A. and W.B., 1992]. While pseudo-polynomial time algorithms have been derived for the 1D packing problem, no such results have been derived for the 3D packing problem.

The 3D packing problem can be divided into two main variants. The first variant is: given a list of boxes of various sizes, pack a subset of these boxes into one container to maximize the volume utilization of the container. The second variant is: given a list of boxes of various sizes, pack all the boxes into as few containers as possible. In addition, constraints, such as orientation restriction, weight distribution, locality restriction, may be imposed on the boxes.

In this paper, we study the packing problem that maximizes the volume utility of a single container with orientation restrictions on the boxes.

Many different heuristics have been developed to “optimize” volume utilization in a single container. [J.A. and B.F., 1980] proposed a primitive “wall-building” method. [Gehring H. and M., 1990] proposed an improved “wall-building” heuristic. [T.H. and Nee, 1992] proposed a layering approach. [Han C.P. and J.P., 1989] provided an algorithm for packing identical boxes, which is a combination of “wall-building” and layering. [B.K. and K., 1993] proposed a data

structure for representing the boxes and free spaces, which is used in their heuristic. [E.E. and M.D., 1990] presented a comparative evaluation of different heuristic rules for freight container loading with the objective of minimizing the container length needed to accommodate a given cargo. [W.B., 1991] examined a number of heuristic rules for the problem. The emphasis on both wall building and layering is to create walls or layers of boxes as flat as possible. At all times, the methods attempt to retain a flat forward packing face. The existing methods most mirror the hand-packing process, which is a natural way to solve the problem. However, they put many restrictions on the location of the boxes which may not be feasible to the actual application. Another problem of the existing approaches is that they may leave a thin layer of non-useful empty spaces near the entrance or the ceiling of the container, which results in space underutilization. Also, the papers in the existing literature used data that are not easily accessible, hence, it is relatively hard to compare the various methods quantitatively.

In this paper, we propose a new method. Boxes are packed into container in “building-growing” fashion instead of “wall building” manner. Furthermore, each wall of the container can be treated as base to locate the boxes, which provides more flexibility. We tested our program on all the 760 test cases in [OR-Lib,]. Our experimental results indicate that we are able to achieve an average packing utilization more than 87%.

2 Problem Specification

We are given a set of rectangular boxes $B = \{b_1, \dots, b_n\}$ and a rectangular container C . Let x_i , y_i and z_i represent the three dimensions of the box b_i , and X , Y , and Z represent the three dimensions of the container.

The objective of the problem is to select a subset $S \subseteq B$ and assign a position to each box, $b_j \in S$, in the 3D space of container C such that $\sum_{b_j \in S} x_j \times y_j \times z_j$ is maximized subjected to the constraints that all boxes must be totally contained in C , no two boxes intersect in 3D space and every edge of the packed box must be parallel or orthogonal to the container walls.

3 Multi-Directional Building-Growing Algo

The idea of this 3D packing algorithm [Wang, 2001] comes from the process of constructing a building. When erecting a building, we construct the basement on the ground first, then the building will “rise up” floor by floor. Here, the box will play the role of the floor, and the wall of the container will act as the ground. Boxes are placed on the wall of the container first as it builds the basement on the ground. After which, other boxes will be placed on top of the basement boxes. Following this process, boxes will be placed one on top of another, just like the construction of a new building. Every wall of the container can be treated as the ground for boxes to stack on. However, the users reserve the right to choose the walls to act as the ground. Note that, this algorithm does not take into consideration whether packed boxes form a flat layer or not, which is the main idea of the “wall building” approach. Figure 1 illustrates the main idea of this algorithm.

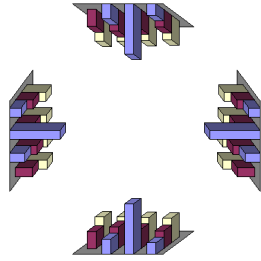


Figure 1: Multi-Directional Building-Growing Approach

3.1 Formulation of the Algorithm

Overall Packing Process

After getting the option from the user for the acting-as-ground walls, this algorithm reads in the container and box information, including length, width, height, orientation and quality information, from the data file. The algorithm then orders the boxes according to certain criteria. Subsequently, actual packing will be performed based on the best matching between base area of boxes and empty spaces. The following pseudo code presents the overall algorithm.

```
main{
  1. get_wall_option()
  2. read in the container information (length,width,height)
  3. for(every box of type b){
    1.read in b's information(length,width,height,orientation,quantity)
    2.bn=create_box_nodes(box b)
    3.insert_box_node(box_node bn)
  }
  4. pack()
}
```

get_wall_option()

Users are free to choose which of the walls of a container to be used as ground/base. Basically, all the empty spaces created from one ground(wall) form an empty space list. There are at most six empty space lists as shown in Figure 2.

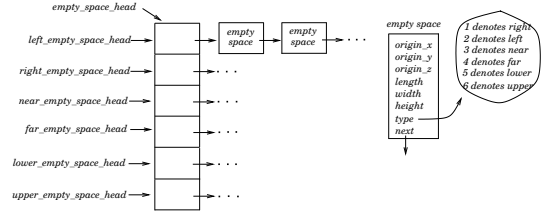


Figure 2: Empty Space List and Empty Space Object Data Structure

An empty space in each empty space list has its own origin and coordinate system as shown in Figure 3. Here the lower coordinate system is the “normal” coordinate system.

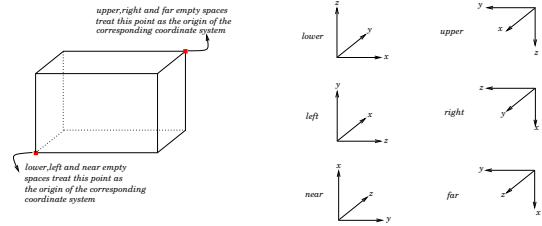


Figure 3: Empty Space Coordinate System Representation

The initial value of each empty space list head is the whole container. If the user does not choose wall i as the ground, then empty space list i 's head should be pointing to null. Note that each empty space list i only takes care of all the empty spaces using wall i as base, without knowing the existence of the other empty space lists. Because of the exact and accurate representation, empty spaces may overlap with each other.

create_box_nodes(box b)

Every box has three attributes namely, length, width and height. Length is measured along the x-axis, so is width along the y-axis and height along the z-axis. And the area surrounded by length and width is called the base area of a box. There is a unique id assigned to each box, yet, there are six orientations associated with it (see Figure 4. Note that a number is used to denote each edge of the box.).

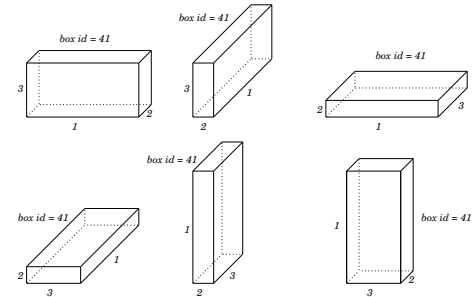


Figure 4: Six Orientations Associated with One Box

For each type of box (same type of box means boxes with same length, width and height), at most six box nodes will

	box1	box2	box3	box4	box5	box6
id	41	41	41	41	41	41
length	1	2	1	3	2	3
width	2	1	3	1	3	2
height	3	3	2	2	1	1

Table 1: *Id* and Attributes Associated with Box in Figure 4

be created for them but with the same *id*. Each box node represents a possible orientation for one type of box. If a certain orientation is not allowed, this algorithm will simply omit the box node of that orientation.

insert_box_node(box_node bn)

Whenever a box information is obtained from the data file, a new box node will be created and inserted to the box_node_list based on the function below. Given a newly created box node bn_2 , this algorithm goes through the box_node_list, and for each existing box node bn_1 , calculates the priority number for bn_1 and bn_2 respectively as follows:

$$priority(bn_1) = vol_per \times \frac{volumn(bn_1)}{volumn(bn_2)} + dim_per \times \frac{big(bn_1)}{big(bn_2)}$$

$$priority(bn_2) = vol_per \times \frac{volumn(bn_2)}{volumn(bn_1)} + dim_per \times \frac{big(bn_2)}{big(bn_1)}$$

where parameters *vol_per* and *dim_per* denote the volume priority percentage and the biggest dimension priority percentage respectively; routines *volume(bn)* and *big(bn)* return the volume and the biggest dimension of box *b* inside box node *bn* respectively. After getting *priority(bn₁)* and *priority(bn₂)*, the algorithm compares these two numbers:

- if $priority(bn_2) > priority(bn_1)$, insert bn_2 to the front of bn_1 .
- otherwise, continue to compare $priority(bn_2)$ with the priority number of the box node behind bn_1 .

According to the above, a box node nearer to the head possesses either larger volume or one bigger dimension, hence, has higher priority to be packed into the container first. Note that boxes with the same *id* but with different orientations are located adjacent to each other, and they have the same priority to be packed into the container, because they are actually the same box.

pack()

In every attempt to pack one box into the container, the algorithm finds a most suitable empty space for each of the first *i* box nodes with the same priority number, in terms of the percentage of the base area of that box occupies the base area of the empty space. Inside all these *i* pairs of box node and empty space, the algorithm finds one pair (bn, oe) with the greatest base area occupation percentage *p*,

- remove bn , if $p > 1$.
- pack bn , if $packing_per \leq p \leq 1$.
- decide to pack or remove bn carefully, if $0 < p < packing_per$

The following pseudo code illustrates the packing process of the algorithm.

```

while(box_node_head!=null){
  for(box node  $bn_i$  with same priority as box_node_head){
    for(every empty space  $oe$  in the 6 empty space lists){
      percentage( $bn_i$ ) =  $\frac{base\_area(bn_i)}{base\_area(oe)}$ 
    }
  }
  find one pair ( $bn, oe$ ) with greatest  $p = percentage(bn)$ 
  if( $p > 1$ ) {
    //bn cannot be packed into any empty space
    remove  $bn$ 
    //because  $bn$  is too big to be packed into any empty space now
    //and empty space will become smaller and smaller, there will
    //be no chance for  $bn$  to be packed into any empty space later
  }
  if( $packing\_per \leq p \leq 1$ ){
    1.create a packed box  $pb$  according to  $oe$ 's coordinate system,
    then convert it to the normal coordinate system and insert it
    into packed_box_list
    2.delete one box id from box node  $bn$ 
    3.update_empty_space(empty_space  $oe$ , packed_box  $pb$ )
  }
  if( $0 < p < packing\_per$ ){
    //in this case, we cannot find a good fit between  $bn$  and  $oe$ ,
    //so we need to decide whether we can drop the first  $i$  box
    //nodes with the same priority number
    if(after removing the first  $i$  box nodes and
     $\sum \frac{volume(the\ rest\ unpacked\ boxes + the\ packed\ boxes)}{volume(container)} \geq moving\_per$ ){
      remove the first  $i$  box nodes
      //because without these few difficult-to-be-packed box nodes,
      //we may still get good utilization
    }
    else{
      1.create a packed box  $pb$  according to  $oe$ 's coordinate system
      then convert it to the normal coordinate system and insert it
      into packed_box_list
      2.delete one box id from box node  $bn$ 
      3.update_empty_space(empty_space  $oe$ , packed_box  $pb$ )
    }
  }
}

```

update_empty_space(empty_space oe , packed_box pb)

After a box is packed into the container, the original set of empty spaces will be modified by the newly packed box pb . Moreover, pb will not only affect the empty space oe it resides in, but also the empty spaces nearby oe , because, in order to represent the empty spaces effectively, the empty spaces overlap. The responsibility of this routine is to update the empty spaces after one box is placed into the container. This process can be performed using the following two steps:

Step 1: Change pb to the coordinate system which oe lies in and absorb the new empty spaces out of oe (Figure 5).

Updating oe :

1. Remove oe from its empty space list
2. If (es_1, es_2, es_3 are not inside the empty_space list oe resides in)

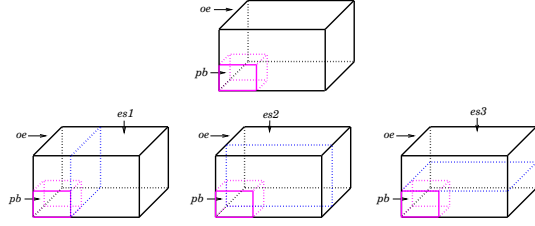


Figure 5: Update Empty Space oe which Provides Empty Space for Packed Box pb

insert it(them) into the empty_space list oe resides in

Step 2: For every empty space e in the six empty space lists,

1. Convert pb to the coordinate system which e is in
2. Test whether there is overlap between e and pb , and update e accordingly

All the possible overlap cases between e and pb will be discussed below and the update method will be given by the end.

case 1: pb intersects one corner of e .

e.g. pb intersects the lower left near corner of e .

case 2: pb intersects one entire edge of e .

e.g. pb intersects the left edge of upper plane of e .

case 3: pb intersects the inner part of one edge of e .

e.g. pb intersects part of lower left edge of e .

case 4: pb gets into e from one plane of it without touching any corner point of e .

e.g. pb gets into e from the left plane of e .

case 5: pb cuts away one entire plane of e .

e.g. pb cuts away the whole left plane of e .

case 6: pb crosses one pair of parallel edges of e without touching any corner point of e

e.g. pb crosses the upper horizontal pair of edges of e .

case 7: pb passes through one pair of parallel planes of e without touching any corner point of e

e.g. pb passes through the near-far planes of e .

case 8: pb cuts e into two new empty_space.

e.g. pb cuts e into two new empty_space from vertical direction.

case 9: the whole pb resides inside e .

case 10: pb covers the whole e .

Updating e in each case above:

1. Remove e from its empty space list
2. If (newly created empty spaces es_i are not inside the empty_space list e resides in)
insert it(them) into the empty_space list e resides in

3.2 Experimental Results

This algorithm was successfully implemented using the Java Programming Language. All the experiments were run on an Alpha 600MHz machine running the Digital Unix operating system. The OR-Library [OR-Lib,] has a collection of test data for a variety of OR problems. There are nine sets of test data with 760 test cases for the 3D packing problem. All the test data in [OR-Lib,] have been tested.

The algorithm was tuned using various parameter settings. The packing result for test file 8 is omitted in all result presentations, because of the fact that we can pack every box into the container in most cases and the size of the container is much larger than the total size of the boxes to be packed into the container.

The best packing result in terms of container volume utilization is shown in Figure 6. The parameter and wall combination used for the best packing result is shown in Table 2.

Parameter	Wall Combination
packing_vol=0.95	left wall
moving_per=1.0	right wall
vol_per=0.9	far wall
dim_per=0.1	lower wall

Table 2: Best Parameter Setting and Wall Combination

In the experimental results shown in Figure 6, the X-axis represents the test case index and the Y-axis represents the container volume utilization.

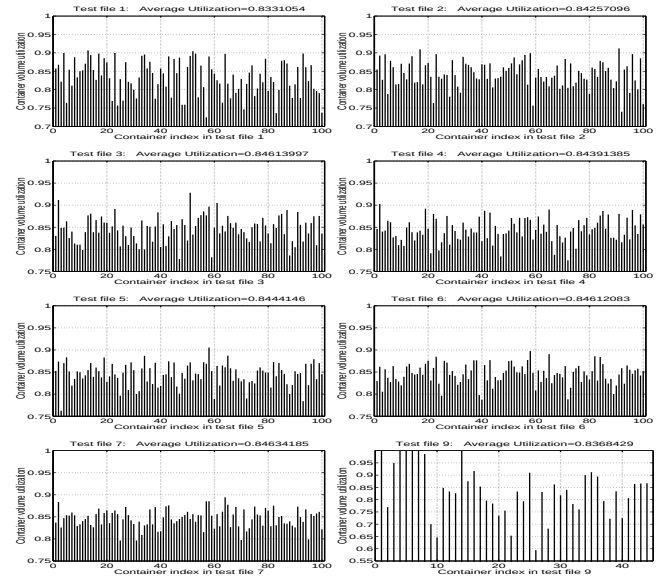


Figure 6: Best Experimental Results for Test File 1-7,9 Using Multi-Directional Building-Growing Approach

Using this parameter setting and wall combination, the average container volume utilization for all the available test cases in the OR-Library (except test file 8) can reach as high as 0.84285194. This utilization is comparable to the published result of the existing literature. Moreover, the special advantage of this algorithm is its speed as it needs about two seconds to pack one container.

On the other hand, the container volume utilization can still be improved for the price of efficiency. In the next section, an improvement phase is introduced.

4 Incorporating a Look-Ahead Mechanism

In the packing process of Multi-Directional Building-Growing (MDBG) approach, the algorithm only chooses one pair of best fit box and empty space in terms of base area occupation percentage among the most difficult-to-be-packed boxes. MDBG Algorithm only takes care of the local information, and the decision made by this algorithm doesn't contain any knowledge for the future. The look-ahead mechanism not only considers the current pairing, but also looks at its potential effect.

4.1 Look-Ahead Improvement Strategy

The strategy will choose k pairs of boxes and empty spaces with the greatest base area occupation percentage. Each one of these k pairs of boxes and empty spaces will be assumed to be the chosen pair for this attempt and be temporarily packed into the container, and the packing process is continued using the MDBG approach from this attempt onwards until all the boxes have been packed into the container or no more boxes can be packed into it. Then based on the final container volume utilization, the improvement strategy will choose one pair of the box and empty space from the initial k pairs, which gives best final container volume utilization. This chosen pair of box and empty space will then be used as the actual packing for this attempt. This pair of box and empty space satisfies both best fit currently and best potential effect in the future. Figure 7 illustrates the look-ahead idea.

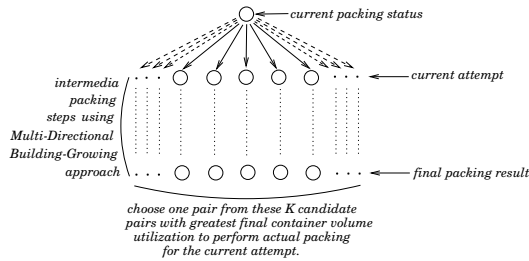


Figure 7: Illustration of the Look-Ahead Improvement Phase

The following is the pseudo-code for the look-ahead improvement strategy.

```
newpack(){
  while(there's box in the box node list){
    //attempt to pack one box into the container
    //1.get K boxes with greatest base area occupation percentage
    for(every box b in the box node list){
      for(every empty space in the 6 empty space lists){
        find a best fit empty space e for b with greatest
        base area occupation percentage p
      }
    }
    if(p > 1){
      //b cannot be packed into any empty space
      remove b
    }
    else if(top K base area occupation percentages < p ≤ 1){
      replace e and b into the top K best fit pairs
    }
  }
}
```

```
}
}
//2. using MDBG approach pack the K candidate pairs
for(i=1..k){
  1.temporarily pack bi to empty space ei at this attempt
  2.continue packing using pack() in section 3
  3.record down the final container volume utilization at ui
}
//3.get one box with the greatest final utilization from the K
// candidates, and use this box in the actual packing of this
// attempt
find one box b from bi and one empty space e from ei with
greatest final container volume utilization u
pack b into e permanently
}
```

4.2 Experimental Results

In the last section, using MDBG approach, we have experimented with all the test data using all possible wall combinations for many possible parameter settings. The best packing result comes from the wall combination: left wall, right wall, far wall, lower wall; and the parameter setting: $packing_vol = 0.95$, $moving_per = 1.0$, $vol_per = 0.9$, $dim_per = 0.1$. Based on this experience, the Look-Ahead mechanism is tested on all data files using this parameter setting and wall combination, and let k go from 1 to 6. The results are summarized in the tables below:

	k=1	k=2	k=3	k=4	k=5	k=6
file1	0.81054	0.83601	0.85984	0.86713	0.87859	0.88356
file2	0.81704	0.83970	0.86335	0.86457	0.87433	0.87576
file3	0.82474	0.83616	0.85437	0.86468	0.86952	0.87036
file4	0.83225	0.84148	0.85376	0.85925	0.87081	0.87068
file5	0.83525	0.84282	0.85540	0.86260	0.86822	0.87072
file6	0.83403	0.84009	0.85665	0.86275	0.86534	0.86679
file7	0.83761	0.84349	0.85048	0.86072	0.86260	0.86575
file9	0.84885	0.87370	0.88711	0.89616	0.90393	0.90656
average	0.82862	0.84196	0.85809	0.86505	0.87193	0.87399

Table 3: Packing Result With Look-Ahead Improvement Phase

For the look-ahead method, the best packing result comes from the wall combination: left wall, right wall, far wall, upper wall and the parameter setting: $packing_vol = 0.95$, $moving_per = 1.0$, $vol_per = 0.9$, $dim_per = 0.1$. The average container volume utilization using this best parameter setting and wall combination can be increased to 0.87813642. The detailed packing result is listed in Figure 8.

Note that in the experimental results shown in Figure 8, the X-axis represents the test case index and the Y-axis represents the container volume utilization, and the average packing utilization and packing time are indicated on the top of each figure.

5 Comparison of Results

To show the effectiveness of the proposed packing algorithm and its coupled improvement strategy, we compare our results run on the test problems from the [OR-Lib,] with

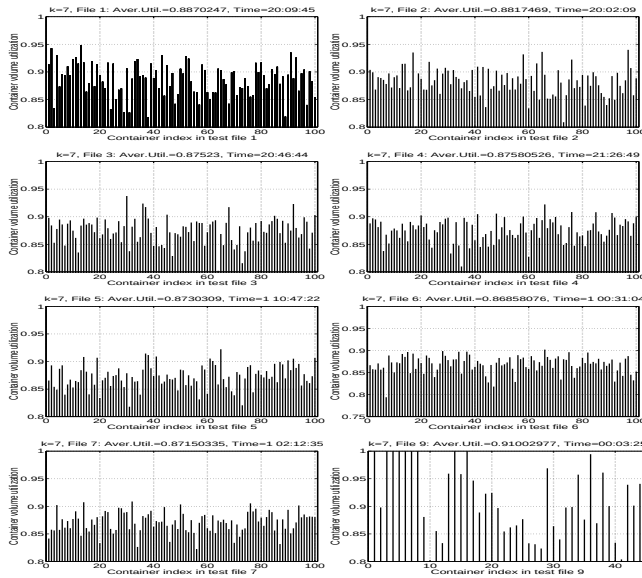


Figure 8: Best Experimental Result for Test File 1-7,9 Using Look-Ahead Improvement Phase

results in [E.E and Ratcliff, 1995] which is the only available work in the existing literature containing packing results on 700 test problems from the OR-Library.

	scheme(a)	scheme(b)	scheme(c)	MDBG Algo	LAM
file1	81.76%	83.79%	85.40%	83.31%	88.70%
file2	81.70%	84.44%	86.25%	84.26%	88.17%
file3	82.98%	83.94%	85.86%	84.61%	87.52%
file4	82.60%	83.71%	85.08%	84.39%	87.58%
file5	82.76%	83.80%	85.21%	84.44%	87.30%
file6	81.50%	82.44%	83.84%	84.61%	86.86%
file7	80.51%	82.01%	82.95%	84.63%	87.15%

Table 4: Performance Comparison Between the Existing Algorithms and the Proposed Algorithms

In Table 4, the scheme(a), scheme(b), scheme(c) are the respective scheme a, b and c in [E.E and Ratcliff, 1995]. MDBG Algo refers to the Multi-Directional Building-Growing Algorithm, and LAM refers to Look-Ahead Mechanism. From the experiment results, it is clear that the container volume utilization using the MDBG Algorithm is slightly better than that of scheme a and b in [E.E and Ratcliff, 1995]. But the MDBG Algorithm coupled with the LAM produces a much higher container volume utilization than scheme c in [E.E and Ratcliff, 1995]. The experimental results provide a strong evidence that the Multi-Directional Building-Growing Algorithm when coupled with the Look-Ahead mechanism gives an significant improvement to the container volume utilization for 3D container packing problem.

6 Conclusion

In this paper, we presented a new approach to solving the 3D container packing problem. Our paper is the first paper to benchmark our results across all the OR-Benchmarks. Our packing utilization of 87% compares very favorably against past works in this area.

References

- [B.K. and K., 1993] Ngoi B.K. and Whybrew K. A fast spatial representation method. *Journal of International Advance Manufacturing Technology*, 8(2), 1993.
- [E.E. and M.D., 1990] Bischoff E.E. and Marriott M.D. A comparative evaluation of heuristics for container loading. *European Journal of Operational Research*, 44:267–276, 1990.
- [E.E and Ratcliff, 1995] Bischoff E.E and MSW Ratcliff. Issues in the development of approaches to container loading. *Omega. Int. J. Mgmt Sci.*, 23(4):337–390, 1995.
- [Gehring H. and M., 1990] Menschner K. Gehring H. and Meryer M. A compute-based heuristic for packing pooled shipment containers. *European Journal of Operational Research*, 1(44):277–288, 1990.
- [Han C.P. and J.P., 1989] Knott K. Han C.P. and Egbelu J.P. A heuristic approach to the three-dimensional cargo-loading problem. *Int. J. Prod. Res.*, 27(5):757–774, 1989.
- [J.A and B.F., 1980] George J.A and Robinson B.F. A heuristic for packing boxes into a container. *Computer and Operational Research*, 7:147–156, 1980.
- [K.A. and W.B., 1992] Dowsland K.A. and Dowsland W.B. Packing Problems. *European Journal of Operational Research*, 56:2–14, 1992.
- [OR-Lib,] OR-Lib. <http://www.ms.ic.ac.uk/library/>.
- [T.H. and Nee, 1992] Loh. T.H. and A.Y.C. Nee. A packing algorithm for hexahedral boxes. In *Proceedings of the Conference of Industrial Automation, Singapore*, pages 115–126, 1992.
- [Wang, 2001] Ying Wang. 3D container packing. Honours thesis, National University of Singapore, 2001.
- [W.B., 1991] Dowsland W.B. Three dimensional packing-solution approaches and heuristic development. *International Journal of Operational Research*, 29:1637–1685, 1991.