Natural Computing 11-08 2014

Universiteit Leiden
Computer Science

Combinatorial Optimization

for

3D Container Loading Problem

Xiaolong Mu

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Two characteristic, in NP-hardness and in widely practical application, of the 3D bin packing problem continue to attract the attention of academia and industry. For academia, they are always interested in solving the NP problem. For the industry, the 3-Dimensional bin packing problem is a problem faced in modern industrial processes such as container ship loading, pallet loading, plane cargo management, and warehouse management. The solution of this problem is represented by a packing sequence. To generate this sequence the packing items are selected from several item groups, and each of them does not have any connection, which make the 3-D packing problem a completely discrete optimization problem. The range of this search space is directly decided by the total number of items. Furthermore, the difficulty of solving this problem will be dramatically increased as well when the practical constraints are taken into account, (e.g., Orientation constraints, Loading stability, and Handling constraints). To solve this problem the most efficient approaches are meta-heuristics, such as Greedy Randomized Adaptive Search Procedure, Ant Colony Optimization, and Evolutionary Algorithms because they have a learning component that can help the optimizer find a search path in the search space. Currently, the best result was produced by the Biased Random Key Genetic Algorithm (BRKGA)[1]. This algorithm was first proposed by Bean [2] to tackle the sequencing problems, and was first used to tackle the 3D-Container Loading Problem by Goncalves JF [1]. By implementing the BRKGA, it is found that the initialization component and mutation operator are slightly weak. Thus, in this project, a new initialization method and an ES type mutation operator are used to replace the original ones within the BRKGA in order to make it an improvement to produce a better result.

Acknowledgements

During my two yeas master study, I would like to thank Prof. Thomas Bäck and zhiwei Yang. I own a deep gratitude to them, for their excellent inspiration and supervision all along my study.

Last but not least, I would like to thank my beloved family, especially my wife Xiao. It is impossible for me to finish this thesis without their endless love and support.

Contents

Abstract iii

Acknowledgements iv

Contents v

List of Figures vii

List of Tables viii

List of Algorithm ix

Abbreviations x

Preface xi

1 Design Optimization of 3-Dimensional Bin Packing Problem 1
- 1.1 Problem Definition . . . 2
- 1.2 Mathematical Formulations . . . 5
- 1.3 Placement Strategy . . . 8
  - 1.3.1 Maximal-Empty-Space Selection . . . 9
  - 1.3.2 Empty-Spaces Update . . . 9
  - 1.3.3 The Deep-Bottom-Left Procedure . . . 15
  - 1.3.4 Practical Implementation . . . 16
- 1.4 Additional Consideration . . . 16
- 1.5 Summary . . . 18

2 Optimization Approach 20
- 2.1 Evolutionary Algorithm . . . 20
- 2.2 Introduction of Standard Genetic Algorithm . . . 21
  - 2.2.1 Representation and Fitness Evaluation . . . 24
  - 2.2.2 Selection . . . 24

2.2.3 Recombination . . . 27
2.2.4 Mutation . . . 29
2.2.5 Theory . . . 30
2.3 Random Key Representation . . . 30
2.3.1 The Concept of Random Key . . . 31
2.3.2 Hypercube Sampling . . . 32
2.3.3 Advantage of Random Key Representation . . . 33
2.3.4 Disadvantage for The Random Key Representation . . . 33
2.4 Divide and Conquar . . . 33
2.5 Summary . . . 36

3 Biased Random Key Genetic Algorithm 37
3.1 Representation and Decoding . . . 38
3.2 Evolutionary Process . . . 40
3.2.1 Initialization . . . 41
3.2.2 Fitness Function . . . 42
3.2.3 Selection Operator . . . 45
3.2.4 Crossover operator . . . 45
3.2.5 Mutation Operator . . . 46
3.3 Modification . . . 47
3.3.1 Initialization Component Modification . . . 47
3.3.2 Mutation Operator Modification . . . 50
3.4 Summary . . . 51

4 Experiments 53
4.1 Benchmark Description . . . 54
4.2 Optimizer Parameter Configuration . . . 55
4.3 Different Components Setting . . . 56
4.4 Experiment Implementation . . . 57
4.4.1 The Components Modification . . . 57
4.4.2 Parameter Tuning . . . 59
4.4.3 The Overall Comparison Experiments . . . 64

5 Conclusion 67

A The Practical Packing Solution and Pattern 69

Bibliography 80

List of Figures

1.1 Example of Placing a box of type i . . . 7
1.2 Example of Difference Process . . . 10
1.3 Example of 3-Dimensional Difference Process . . . 11
1.4 Example of Elimination Process . . . 13

2.1 General Crossover model . . . 28
2.2 Random Mapping Procedure . . . 31
2.3 HyperCube sample . . . 32
2.4 Feasible Layer Type . . . 34
2.5 General-purpose Metaheuristic Framework . . . 36

3.1 First M Gene decoding . . . 39
3.2 Architecture of BRKGA . . . 40
3.3 Population Structure . . . 41
3.4 Example of Crossover Operator . . . 46
3.5 Original Position Pattern . . . 48
3.6 New Position Pattern . . . 49
3.7 Population Structure . . . 52

4.1 New Position Pattern vs Old Position Pattern . . . 58
4.2 New Mutation Operator vs Old Mutation Operator . . . 58
4.3 Crossover Rate Analysis . . . 60
4.4 BOT Size Analysis . . . 61
4.5 Elites Size Analysis . . . 63
4.6 3 Tuning Step Comparison . . . 64

A.1 The Benchmark One . . . 73
A.2 The Benchmark Two . . . 74
A.3 The Benchmark Three . . . 75
A.4 The Benchmark Four . . . 76
A.5 The Benchmark Five . . . 77
A.6 The Benchmark Seven . . . 79

List of Tables

2.1 Decoding and Encoding 24
2.2 Example of Roulette-wheel Selection 25

4.1 Parameters Setting of Problem Generator 55
4.2 The Algorithm Parameter Setting 56
4.3 The Approaches in Comparison 57
4.4 The Final Comparison Result 64
4.5 Average Utility Rate (B1-B7) Result 65

A.1 The Best Result in Experiment 4.3 70
A.2 The Best Result in Experiment 4.4 71
A.3 The Best Result in Experiment 4.4 72
A.4 The Best Result in Benchmark One 73
A.5 The Best Result in Benchmark Two 74
A.6 The Best Result in Benchmark Three 75
A.7 The Best Result in Benchmark Four 76
A.8 The Best Result in Benchmark Five 77
A.9 The Best Result in Benchmark Six 78

List of Algorithms

1 Empty Spaces Update(box_j,S) 14
2 Deepest-Bottom-Left Sorting (S) 15
3 Placement Strategy($BTPS$,VLT) 17
4 ItemRotation(box_i,$rotateplag$) 18
5 General Schema of an evolutionary Algorithm 21
6 Roulette Wheel Selection [3] 26
7 Tournament Selection[3] 27
8 LayerGenerator(box_i, ems_i) 35
9 BRKGA_Initialization($problem$) 42
10 Space Utility Rate($solution$) 43
11 Revert($solution$,l,r) 44
12 FitnessFunction($solutionset$) 45
13 CrossoverFunction($parent1$,$parent2$) 47
14 PositionPatternGenerator($Problem$) 49
15 ES_Mutation 51
16 BenchmarkGenerator 54

Abbreviations

NP	**N**on-deterministic **P**olynomial-time
3D-CLP	**3D**imensional **C**ontainer **L**oading
EA	**E A**lgorithms**v**olutionary **A**lgorithms
EP	**G**enetic **P**rogramming
TS	**T**abu **Search**
SA	**S**imulated **A**nnealing
GA	**G**enetic **A**lgorithms
GRASP	**G**reedy **R**andomized **A**daptive **S**earch **P**rocedure
BRKGA	**B**iased **R**andom **K**ey **G**enetic **A**lgorithms
SPP	**S**trip **P**acking **P**roblem
BBP	**B**in **P**acking **P**roblem
MCCP	**M**ulti **C**ontainer **L**oading **P**roblem
KLP	**K**napsack **L**oading **P**roblem
MPLP	**D**istributor's **P**allet **L**oading **P**roblem
MPLP	**M**anufacutrer's **P**allet **L**oading **P**roblem
EMS	**E**mpty **M**aximal **S**pace
DP	**D**ifference **P**rocess
DBLP	**D**eep **B**ottom **L**eft **P**rocedure
BTPS	**B**ox **T**ype **P**acking **S**equence
VLT	**V**ector of **L**ayer **T**ype

Preface

The 3-Dimensional Bin Packing Problem can be divided into a number of branches by adding different constraints that are required in practical applications. There is no any single algorithm which can cover all the problems, so the problem that is chosen in this project is one branch of them, namely 3 Dimensional Container Loading Problem (3D-CLP). The details of this problem will be introduced in the chapter 1. There are two reasons to choose this branch. The first reason is, there is a widely used benchmark, produced by Bischoff and Ratcliff [4], which offers a standard to evaluate the performance of an optimizer. The second one is that this problem can extend to other branches easily by changing or adding necessary constraints.

The 3D-CLP is NP-hard [5]. Using an exact method to solve this problem is unrealistic, especially, when there are too many items needed to be packed in which the search space is too large to search. To date, only a few exact methods have been suggested in the literature. Fekete and schepers [6] presented a general framework for an exact solution of multi-dimensional packing problem. Martello et al.[7] developed an exact branch-and bound method for the 3D-CLP.

To fill the above gaps, the heuristics have been the only viable alternative to find optimal or near-optimal solution. Many heuristic procedures have been proposed for solving the 3D-CLP. Fanslau and Bortfeldt [8] classified approaches for the 3D-CLP according to packing heuristics and method type. The packing heuristics are grouped as the below list.

- The wall building approaches fill the container with vertical layers. This approach has been used by Loh and Nee [9], Borteldt and Gehring [10], George and Bobinson [11] and Pisinger [12] as well.

- Stack-building approaches arrange the packing items into different stacks first. After that, these stacks will be loaded into a container along with the floor of the container, in a way, that saves the most space. The examples of the use of this method that can be found in Bischoff and Ratcliff [4], Gehring and Bortfeldt [13].

- Horizontal layer-build approaches build horizontal layers that are intended to cover the largest possible part of the load surface underneath, then fill them into a container from the bottom to top. These approaches have been implemented by Bischoff et al. [14] and Terno et al. [15].

- Block-building approaches fill the container with cuboid blocks of boxes. The tree-search method of Eley [16], the tabu search method of Bortfeldt et al. [17], and the hybrid simulated annealing and tabu search method of Mack et al. [18] are examples.

- Guillotin-cutting approaches are based on a slicing tree representation of a packing plan. Each slicing tree corresponds to a successive segmentation of the container into smaller pieces by guillotine cuts, whereby the leaves correspond to the boxes to be packed. The graph-search method of Morabito and Arenales [19] is based on this approaches.

In the below list, Fanslau and Bortfeldt [8] categorized solution methods as meta-heuristics, tree search methods, and conventional heuristics.

- Meta-heuristics search strategies have been the preferred methods in the last ten years, which includes the tabu search approaches (TS) of Bortfeldt et al. [17], the simulated annealing methods (SA) of Mack et al. [18], the genetic algorithms (GA) of Gehring and Bortfeldt [13, 20], and Bortfeldt and Gehring [10], the method of Bischoff [21], based on the Nelder and Mead algorithm, and the greedy randomized adaptive search procedures (GRASP) of Moura and Oliveira [22] and Parreno et al. [23].

- Tree-search methods or graph-search methods have been successfully applied to the 3D-CLP by Morabito and Arenales [19], Eley [16], Hifi [24]. Pisinger [12], and Fanslau and Bortfeldt [8].
- Conventional heuristics incorporates construction methods and improvement methods. Examples can be seen in papers by Bischoff et al. [14], Bischoff and Ratcliff [4], and Lim et al [25].

Additional practical constraints have been considered by other authors. For instance, the weight distribution of cargo within a container was taken into account by Davies and Bischoff [26], Eley [16], and Gehring and Bortfeldt [13]. The impact of varying the load-bearing strength was examined by Bischoff [21]. Bortfeldt and Gehring [10], Bortfeldt et al. [17], and terno et al. [15] considered loading stability in their research. Other container-related factors, such as orientation constraints [20] and the grouping of boxes [4, 27], have also been considered.

Currently, the most efficient method to solve the 3D-CLP is the Biased Random Key Genetic Algorithm (BRKGA)[1]. This project aims at improving the BRKGA by modifying its mutation operator and initialization component in order to produce a better result. The relevant background information and the details of the implementation will be presented in chapters 2 and 3. This document is structured as follows:

- **Chapter 1**: The 3-Dimensional container loading problem description.
- **Chapter 2**: Background information about BRKGA.
- **Chapter 3**: The details of implementation.
- **Chapter 4**: Experiment.
- **Chapter 5**: Concludes and indicates future research directions.

Chapter 1

Design Optimization of 3-Dimensional Bin Packing Problem

The main goal in the basic form of the three-dimensional container loading problem is to find a best three-dimensional packing pattern for loading a set of rectangular boxes into a container so that the total volume of the boxes loaded is maximized, and the boxes do not overlap. The bin packing problem initially appears fairly simple. However, scholars have found its behaviour rather complex.

In practical applications, there are several issues in the production and the transportation planning directly modelled by the 3D bin packing problem, which include pallet loading, warehouse management, container ship loading, plane cargo management and pallet loading. For these enterprises, possessing an efficient utilization of materials and the transportation capacity is a significant competitive advantage. Therefore, the requirement to improve the efficiency of maximizing the utilization is strongly necessary.

The section 1.1 introduces the overall detail of the 3D-CLP. In the section 1.2, we present the relevant mathematical models. In the section 1.3, we briefly describe the placement strategy. In the section 1.4, two additional considerations will be presented.

1.1 Problem Definition

The problem of loading boxes into containers can be classified into four variants [12]: the Strip Packing Problem (SPP), the Bin packing Problem (BBP), the Multi-Container Loading Problem (MCLP), and the Knapsack Loading Problem (KLP or CLP). The SPP considers a container of which two dimensions are fixed (e.g., width and height), and the third dimension (e.g., length) is a variable. The problem is to decide how to pack all boxes of different sizes inside the container, so that the variable dimension (length) is minimized (e.g., [28–30]). For the BBP, the aim is to find a minimal number of container (Bin) to load all boxes of different sizes (e.g., [15, 31–33]). Unlike the BPP, the containers in the MCLP do not necessarily have the same sizes and costs, and the problem is to decide how to load all the boxes so that the total cost of the chosen subset of containers to be loaded is minimized (e.g.,[34, 35]).

This project mainly focus on the Container Loading Problem (CLP). The number of the container in this problem is only one with fixed size. There are more than one groups of packing items, each item in the same group is assigned a same size. The objective of this problem is to maximize the space utility of the container.

There are many researches on the container loading problem (e.g., [4, 7, 11, 16, 19, 26, 36]). It can be seen as a three dimensional problem of loading rectangular boxes onto pallets. The problem of carrying boxes on pallets can be divided into two cases [37]: the Manufacutrer's pallet loading problem (MPLP) and the Distributor's pallet loading problem (DPLP). In the first case, there is only one type of box (all boxes has a same size) while in the DPLP the types are two or more. Both problems can be solved in their two- or three- dimensional version, although the first one is more common in practice. The difference between them is that in the two dimensional case, the loading pattern is built in horizontal layers on the pallet, while in the three-dimensional one, the loading pattern can be generic. The last case can also be seen as a KLP with only one type of box.

Practical Requirements

There are twelve practical considerations in [4], which can be used to model more realistic container loading problems. It is perhaps necessary to emphasize that no claim is made that the factors described are of importance in every case. What is claimed, however, is that there are many cases in which some of the factors listed below play an important role.

- **Orientation Constraints:**
  The instruction is usually seen, 'This way up', on cardboard boxes. It is a simple example of this kind of restriction. However, it may not only be the vertical orientation which is fixed, but also the horizontal orientation is restricted. For instance, a two-way entry pallet is loaded by forklift truck.

- **Load Bearing Strength of Items:**
  'Stack no more than x items high' is another instruction seen on many boxes in many situations. This constraint can be considered as a straightforward figure for the maximum weight per unit of area on which a box can support depended on its construction and also its contents. Usually the side walls of a cardboard box provides the bearing strength, so that it might be acceptable to stack an identical box directly on top, whereas placing an item of half the size and weight in the centre of the top face causes damage. The load bearing ability of an item may, of course, also depend on its vertical orientation.

- **Handling Constraints:**
  The items of positioning within a container is usually determined by its size, or weight and the loading equipment. For instance, the large items should be placed on the container floor, or to fix its position below a certain height. It may also be desirable from the viewpoint of easy/safe materials handling to place certain item near the door of the container.

- **Load Stability:**
  If the cargo is easily damaged, to ensure that the load cannot move significantly during transport is necessary. Also, during loading and (especially)

unloading operations, an unstable load can have important safety implications. For handling this constraint, some devices will be used to restrict or prevent cargo movement, such as Straps, Airbags. However, the cost can be considerable, especially in terms of time and effort spent.

- **Grouping of Items:**
  A load might be easy to operate when the items belong to the same group. For instance, several items that are defined by a common recipient or the item type are positioned in close proximity. It may also have advantages in terms of the efficiency of loading operations.

- **Multi-drop Situations:**
  In order to avoid unloading and reloading a large part of the cargo several times when the container is to send consignments for a number of different destinations, the items have to be loaded within the same consignment closely and to order the consignments within the container.

- **Separation of Items within a Container:**
  To ensure that cargo which may adversely affect some of the other goods separate to load is necessary. For instance, if they include both foodstuffs and perfumery articles, or different chemicals, this constraint has to be taken into account.

- **Complete Shipment of Certain Item Groups:**
  Functional entities may include a subset of the cargo. For instance, components for assembly into a piece of machinery, or may need to be treated as a single entity for administrative reasons. To guarantee all the relevant items complete packed is necessary.

- **Shipment Priorities:**
  In the real world, the shipment of some items will be given more priorities to delivery when these items are more important than the others, for instance, delivery deadlines or the shelf life of the product concerned. More specifically, the item might have a priority rating. Depending on the practical context, this rating may represent an absolute priority. In the sense that no item in a lower priority class should be shipped if this causes items with higher rating

to be left behind, or it may have a relative character, reflecting the value placed on inclusion in the shipment without debarring trade-offs between priority classes merely.

- **Weight Distribution within A Container:**
  From the viewpoint of transporting and handling the loaded container- such as lifting it onto a ship-, it is desirable that its centre of gravity be close to the geometrical mid-point of the container floor. If the weight is distributed very unevenly, certain handling operations may be impossible to carry out. In cases where a container is transported by road at some stage of its journey, the implications of its internal weight distribution for the axle loading of a vehicle can be an important consideration. The same, of course, applies if the 'container' is a truck or trailer.

- **Container Weight Limit:**
  If the cargo to be loaded is fairly heavy, the weight limit of a container may represent a more stringent constraint than the loading space available.

Although the aforementioned studies consider the practical issues described, in general, mathematical formulation is rarely presented. Some papers, such as [38–41], present formulations for two-dimensional cutting and packing problems that can be easily extended to the three dimensional container loading problem.

1.2 Mathematical Formulations

The definition of the mathematical model of the 3 dimensional container loading problem in this project is to referred the literature [42]. In this definition, the item can be presented as, the item can be presented as different types of boxes with given length l_i, width w_i, height h_i, value v_i, and a maximum quantity b_i, $i = 1, \cdots, m$, which can be loaded inside the object (container, truck, rail-road car or pallet) with given length L, width W, and height H (when considering a pallet, H is the maximum allowed height of the cargo loading). The dimensions of the boxes are integer, and they can only be placed orthogonally into the container.

This last assumption can be easily relaxed in the models presented and here it is considered only to simplify the presentation of the formulations.

The back-bottom-left corner of the container can bee seen as the origin of the Cartesian coordinate system, and the possible coordinate where the back-bottom-left corner of a box can be placed is represented by (x, y, z). These possible positions along axes L, W, and H of the container belong to the sets: $X = \{0, 1, 2, \cdots, L - \min_i(l_i)\}$, $Y = \{0, 1, 2, \cdots, W - \min_i(w_i)\}$, $Z = \{0, 1, 2, \cdots, H - \min_i(h_i)\}$, respectively. As the view of [38, 43], for a given packing pattern, each packed box could be moved back and/or down and/or the left, until its back, bottom and left-hand face are adjacent to other boxes or the container. Due to each type box can rarely be packed completely, the number of practical packed items can be represented by ε_i, $\varepsilon_i \leq b_i$. Thus, without loss of generality, the sets X, Y and Z can be expressed as below:

$$X = \{x | x = \sum_{i=1}^{m} \varepsilon_i l_i, 0 \leq x \leq L - \min_i(l_i), 0 \leq \varepsilon_i \leq b_i, i = 1, \cdots, m\}. \quad (1.1)$$

$$Y = \{y | y = \sum_{i=1}^{m} \varepsilon_i w_i, 0 \leq y \leq W - \min_i(w_i), 0 \leq \varepsilon_i \leq b_i, i = 1, \cdots, m\} \quad (1.2)$$

$$Z = \{z | z = \sum_{i=1}^{m} \varepsilon_i h_i, 0 \leq z \leq H - \min_i(h_i), 0 \leq \varepsilon_i \leq b_i, i = 1, \cdots, m\}. \quad (1.3)$$

A possible placement of a box of type i inside the container is depicted by figure 1.1. To describe the constraints that avoid overlapping of boxes inside the container we define $c_{ixyzx'y'z'}, (i = 1, \cdots, m), (x, x' \in X), (y, y' \in Y), (z, z' \in Z)$ as

$$c_{ixyzx'y'z'} = \begin{cases} 1 \text{ if a box of type } i \text{ placed with its back-bottom-left corner at} \\ \quad (x, y, z), \text{ occupied point}(x', y', z'); \\ 0 \text{ otherwise.} \end{cases}$$

The mapping $c_{ixyzx'y'z'}$ is not a decision variable and it is computed a priori as below:

$$c_{ixyzx'y'z'} = \begin{cases} 1 \text{ if } & 0 \le x \le x' \le x + l_i - 1 \le L - 1; \\ & 0 \le y \le y' \le y + w_i - 1 \le W - 1; \\ & 0 \le z \le z' \le z + h_i \le H - 1; \\ 0 & \text{otherwise.} \end{cases}$$

Before packing the item in the position (x, y, z), the algorithm can check the feasibility of this placement based on the above formula. Here, the (x', y', z') represents the position of packed item. a feasible packing solution should hold all $c_{ixyzx'y'z'}$ to be zero.

Let $X_i = \{x \in X | 0 \le x \le L - l_i\}$, $Y_i = \{y \in Y | 0 \le y \le W - w_i\}$ and $Z_i = \{z \in Z | 0 \le z \le H - h_i\}, i = 1, \cdots, m$. The decision variables $a_{ixyz}, i = 1, \cdots, m, x \in X_i, y \in Y_i, z \in Z_i$, of the model are defined as

$$a_{ixyzx'y'z'} = \begin{cases} 1 & \text{if a box of type } i \text{ is placed} \\ & \text{with its back-bottom-left corner at the postion}(x, y, z) \\ & \text{so that } 0 \le x \le L - l_i, 0 \le y \le W - w_i \text{ and } 0 \le z \le H - h_i; \\ 0 & \text{otherwise.} \end{cases}$$

The single container loading problem that is without additional considerations

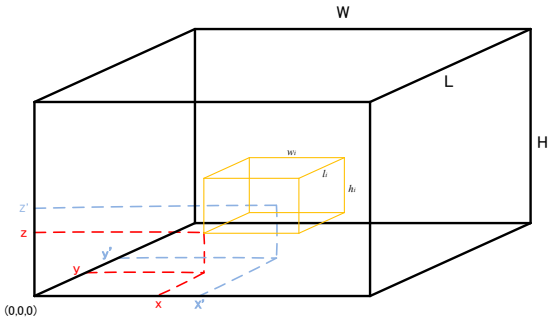


FIGURE 1.1: Example of placement of a box of type i inside a container.

can be written as a direct extension of a $0 - 1$ integer linear programming model

proposed in [38] for the two dimensional non-guillotine cutting problems:

$$\max \sum_{i=1}^{m} \sum_{x \in X_i} \sum_{y \in Y_i} \sum_{z \in Z_i} v_i \cdot a_{ixyz} \tag{1.4}$$

$$100\% \frac{\sum_{k=1}^{K} v_k NP_k}{L \times W \times H} \rightarrow \max \tag{1.5}$$

$$\sum_{i=1}^{m} \sum_{x \in X_i} \sum_{y \in Y_i} \sum_{z \in Z_i} c_{ixyzx'y'z'} \cdot a_{ixyz} \leq 1, \tag{1.6}$$
$$x' \in X, y' \in Y, z' \in Z,$$

$$\sum_{x \in X_i} \sum_{y \in Y_i} \sum_{z \in Z_i} a_{ixyz} \leq b_i, i = 1, \cdots, m. \tag{1.7}$$

$$\begin{aligned} &a_{ixyz} \in \{0, 1\},\ i = 1, \cdots, m. \\ &x \in X_i, y \in Y_i, z \in Z_i. \end{aligned} \tag{1.8}$$

For formulations (1.4-1.8), the objective function (1.4) aims to maximize the total value of the boxes packed inside the container (if $v_i = (l_i \cdot w_i \cdot h_i)$, (1.4) maximizes the total volume of the boxes). The objective function (1.5) is extended from (1.4) in order to calculate the maximal container utility rate, in which NP_k is the number of the type k box packed in a solution, v_k is the volume of a box of the type k and the denominator represents the volume of the container. Constraints (1.6) avoid the overlapping of the boxes packed, constraints (1.7) limit the maximum number of boxes packed, and constraints (1.8) define the domain of the decision variables.

1.3 Placement Strategy

In order to calculate the fitness, the space utility rate, of a solution in bin packing problem, the evaluation component in the relevant algorithm has to simulate the packing process that packs the item into the container by following particular rule

when the packing pattern is given. In this project, the deep bottom left method is selected, which is widely used in many literatures. In this placement strategy, it is divided into three main components: the Maximal-Empty-Spaces Selection, the Deep Bottom Left Packing Procedure, and the Empty-Spaces Update.

1.3.1 Maximal-Empty-Space Selection

There is a list S, which saves all empty maximal-spaces (EMSs) that are largest empty parallelepiped spaces available for filling with boxes. The EMS_s are represented by their vertices with the minimum and the maximum coordinates (x_i, y_i, z_i and X_i, Y_i, Z_i). When an EMS has been searching in the list S to pack a box, only the EMS with minimum vertices coordinate (x_i, y_i, z_i) is considered. The initial stage of list S only has one element, the size of which is equal to the size of the container. After each time packing a box, this list is updated and reordered by the Difference Process (DP), shown in the section 1.3.2, and the Deep-Bottom Left Procedure (DBLP), described in the section 1.3.3.

1.3.2 Empty-Spaces Update

In order to keep track of the EMSs, the Difference Process (DP) is introduced, which is developed by Lai and Chan [44]. To demonstrate this method we are assisted by an example of the application of the DP process in a 2D packing instance as shown in the Fig.1.2.

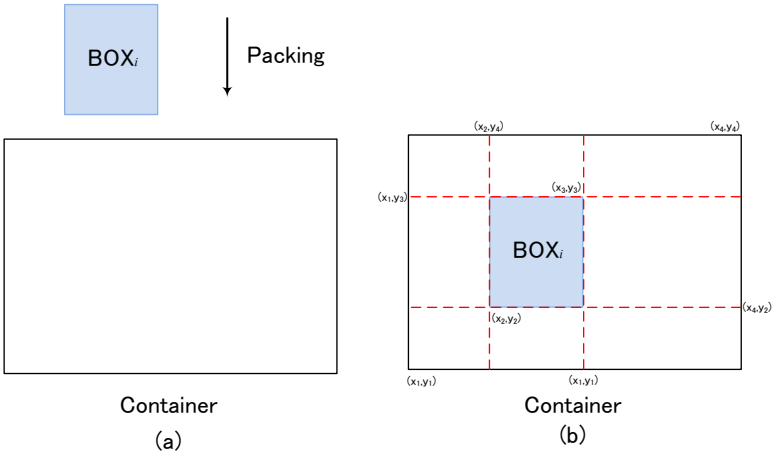


FIGURE 1.2: The Example of Using The Difference Process.

- **New Empty Space Generation**

  As shown in the Fig.1.2(a), the Box$_i$ has to be packed into the container. Suppose that the bottom left corner of the Box$_i$ can be located in the position of the container as shown in the Fig.1.2(b), the bottom-left and the top right corners of the container and the Box$_i$ are represented by four points, $\{(x_1, y_1),(x_4, y_4)\}$, $\{(x_2, y_2), (x_3, y_3)\}$. After packing Box$_i$, the current empty space that was used to load the Box$_i$ was divided into at most four new EMS$_s$. Each of these EMS$_s$ is represented by two points that are calculated by following rules:

$$\text{Difference: } [(x_1, y_1), (x_4, y_4)] - [(x_2, y_2), (x_3, y_3)]$$
$$=$$

$$\text{NewEMS}_1 = [(x_1, y_1), (x_2, y_4)]; \tag{1.9}$$
$$\text{NewEMS}_2 = [(x_1, y_1), (x_4, y_2)]; \tag{1.10}$$
$$\text{NewEMS}_3 = [(x_1, y_3), (x_4, y_4)]; \tag{1.11}$$
$$\text{NewEMS}_4 = [(x_3, y_1), (x_4, y_4)]; \tag{1.12}$$

  In the 2D problem, the maximal empty spaces are the area that is between

two sides which are parallel and belong to the container and box_i, respectively. Thus, to extend this rule to the 3D problem a maximal empty space can be seen as the space between two surfaces which are parallel and belong to the container and the box_i as depicted by the Fig.1.3.

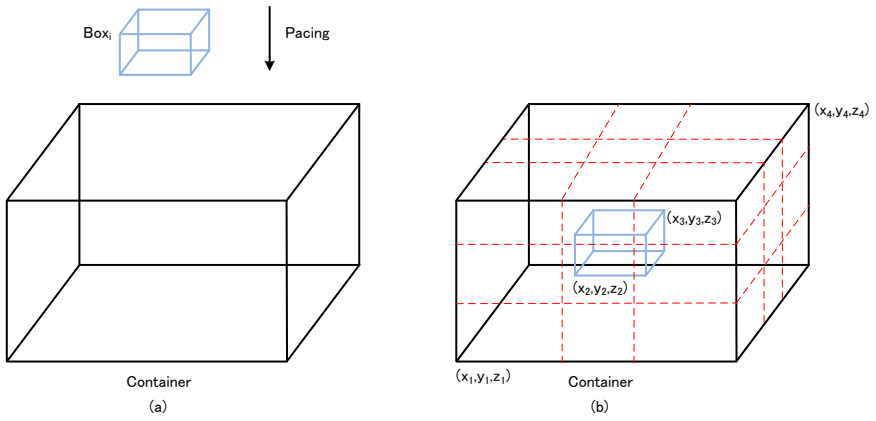


FIGURE 1.3: The Example of Using The 3-Dimensional Difference Process.

In the 3D case, after packing a box, there are at most six EMS_s generated. To implement the 3D DP, the equation (1.3.2) can be rewritten as the following model:

$$\begin{aligned}\text{Difference: } & [(x_1,y_1,z_1),(x_4,y_4,z_4)] - [(x_2,y_2,z_2),(x_3,y_3,z_3)] \\ & =\end{aligned}$$

$$\text{NewEMS}_1 = [(x_1,y_1,z_1),(x_2,y_4,z_4)]; \tag{1.13}$$
$$\text{NewEMS}_2 = [(x_1,y_1,z_1),(x_4,y_2,z_4)]; \tag{1.14}$$
$$\text{NewEMS}_3 = [(x_1,y_1,z_1),(x_4,y_2,z_4)]; \tag{1.15}$$
$$\text{NewEMS}_4 = [(x_1,y_1,z_3),(x_4,y_4,z_4)]; \tag{1.16}$$
$$\text{NewEMS}_5 = [(x_1,y_3,z_1),(x_4,y_4,z_4)]; \tag{1.17}$$
$$\text{NewEMS}_6 = [(x_3,y_1,z_1),(x_4,y_4,z_4)]; \tag{1.18}$$

- **Elimination Process**

  After generating the new empty spaces, or called intervals, some of them will be eliminated because they are unavailable to pack any items, and to eliminate them can save the computer memory storage space. The intervals with an infinite thinness or those are totally inscribed by the other intervals will be removed from the list. To implement this process the new empty spaces have to undergo two types checking:

  - **Cross-Checking:**

    Compare each interval with each other in order to check whether it is totally inscribed by other interval. Suppose that we have two intervals: $[(x_1, y_1, z_1), (x_4, y_4, z_4)]$, $[(x_2, y_2, z_2), (x_3, y_3, z_3)]$.

    Eliminate $[(x_2, y_2, z_2), (x_3, y_3, z_3)]$ from $[(x_1, y_1, z_1), (x_4, y_4, z_4)]$,
    If $x_2 \geq x_1$ and $y_2 \geq y_1$ and $z_2 \geq z_1$ and $x_3 \leq x_4$ and $y_3 \leq y_4$ and $z_3 \leq z_4$

  - **Self-Elimination:**

    Check whether the interval is infinitely thin or not. Suppose we have an interval $[(x_2, y_2, z_2), (x_3, y_3, z_3)]$.

    Eliminate $[(x_2, y_2, z_2), (x_3, y_3, z_3)]$ from maximal empty spaces list S.
    If $x_2 = x_3$ or $y_2 = y_3$ or $z_2 = z_3$

    The above checking processes can easily be adapted to the two-dimensional problem by setting $z_1 = z_2 = z_3 = 0.0$.

  There is a 2-Dimensional instance, depicted by Fig.1.4, used to introduce this process. For convenience purposes, the coordinates in the Fig 1.4 is represented by the capital letters. By following the bottom left rule, we put the box$_i$ into the bottom-left corner of the container as shown in the Fig 1.4(a). By using the two dimensional DP, the four empty spaces are generated, which can be presented as below:

$$[\mathrm{A}, \mathrm{I}] - [\mathrm{A}, \mathrm{E}] = \{[\mathrm{A}, \mathrm{G}], [\mathrm{A}, \mathrm{C}], [\mathrm{D}, \mathrm{I}], [\mathrm{B}, \mathrm{I}]\}$$

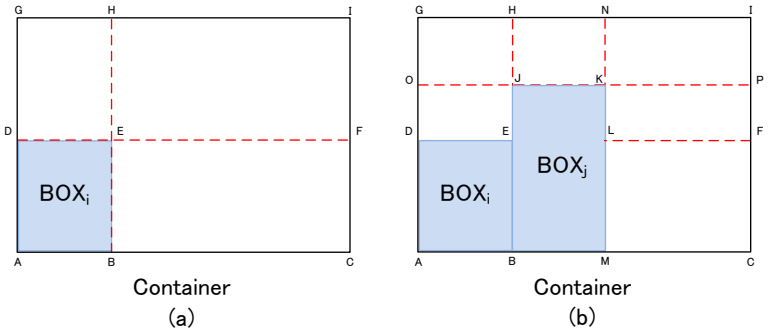


FIGURE 1.4: Example of Elimination Process.

Where [A,G] and [A,C] will be eliminated by following the Self-Elimination process. Thus [D,I] and [B,I] will be saved into the list S. After that, we assume that the Box$_j$ is the next packing item, which is packed in the space [B,I] by following the Deep-Bottom-Left rule as illustrated by the Fig.1.4(b). The space [B,I] is divided into four subspaces as well by following same rules:

$$[\mathrm{B,I}] - [\mathrm{B,K}] = \{[\mathrm{B,H}], [\mathrm{B,C}], [\mathrm{J,I}], [\mathrm{M,I}]\}$$

However, the difference from packing the Box$_i$ is that when the Box$_j$ has been packed into space [B,I], the space [D,I] was no longer available because the space [D,I] was intersected by the Box$_j$. To detect which space will be intersected by the Box$_j$, we use the following criteria: suppose there is an empty space s, which can be represented by their bottom left point and top-right point:(x_1, y_1),(x_2, y_2). Also, a box can be represented by (x_3, y_3) and (x_4, y_4).

Judgement : the space s does not intersect with the current box,
If and only if : $x_1 \geq x_4$ or $x_2 \leq x_3$ or $y_1 \geq y_4$ or $y_2 \leq y_3$
Otherwise : the space will intersect with the box.

To extend this criteria to the 3 dimensional problem we only need to add coordinate z by the same way. After finding the intersected empty spaces, it has to be updated by using the DP as well. Thus, there are four more empty spaces generated.

$$[\mathrm{D}, \mathrm{I}] - [\mathrm{B}, \mathrm{K}] = \{[\mathrm{D}, \mathrm{H}], [\mathrm{D}, \mathrm{F}], [\mathrm{O}, \mathrm{I}], [\mathrm{L}, \mathrm{I}]\}$$

To summarize current empty spaces there are eight successors waiting to save into the list S. However, before saving them, their availability has to be checked by using the Elimination Process. In this case, [B,H], [B,C], and [D,F] will be eliminated by implementing the Self-Elimination process. The interval [J,I] inscribes within [O,I], so it was eliminated by using the Cross-Checking process. After elimination, four empty spaces, [M,I], [D,H], [O,I], and [L,I], were saved into the list S.

To implement the Empty-Spaces updating all above steps are integrated into one function. During the whole packing procedure, this function will be called at each packing operation. The algorithm 1 shows the pseudo-code of the Empty-Spaces update method.

Algorithm 1 Empty Spaces Update(box_j,S)

Initialization:

Define a empty list TS and NS,;
Let (x_b, y_b, z_b) and (x'_b, y'_b, z'_b) represent the deepest-bottom-left corner and the front-top-right corner of the box$_j$.

Iteration:

for each element i in the list S **do**
  if $x_b \geq x'_i$ **or** $x'_b \leq x_i$ **or** $y_b \geq y'_i$ **or** $y'_b \leq y_i$ **or** $z_b \geq z'_i$ **or** $z'_b \leq z_i$ **or then**
    //Separate the spaces that are not intersected
    //by element i from the old list;
    Save the element i into list TS
  else
    Implement 3-Dimensional **DP** process to element i.
    Save the new empty spaces into list NS.
  end if
end for
Implement **Elimination Process** to list NS and TS.
Clear all element in S.
$S = NS + TS$. // Update the list S;

Output: The new list S.

1.3.3 The Deep-Bottom-Left Procedure

Recall from the section 1.3.1 that the fist step in the placement strategy is to select an EMS with the minimum coordinates from the list S, all the element in the list S will be arranged into the lexicographical order. Thus, the Deep-Bottom-Left procedure is actually a sorting algorithm, which is used to sort the EMS_s into the lexicographical order: $\text{EMS}_i < \text{EMS}_j$ if $y_i < y_j$, or if $y_i = y_j$ and $z_i < z_j$, or if $y_i = y_j, z_i = z_j$ and $x_i < x_j$. After each time of the packing, the list S will be sorted again by using this method. The algorithm 2 shows the pseudo-code of the Deep Bottom Left (DBL) procedure.

Algorithm 2 Deepest-Bottom-Left Sorting (S)

Initialization:
Set sw =**false**;
Let $rindex$ be the maximal index of list S;
Iteration:

```
for i = 0 to rindex − 1 do
  for j = rindex ; j > i ; j + + do
    if y_j < y_{j−1} then
      swap(S_j, S_{j−1});//First check the length order;
      sw=true;
    else if y_j = y_{j−1} then
      if z_j < z_{j−1} then
        swap(S_j, S_{j−1});//Second check the height order;
        sw=true;
      else if z_j = z_{j−1} then
        swap(S_j, S_{j−1}); //Third check the width;
        if x_j < x_{j−1} then
          sw=true;
        end if
      end if
    end if
  end for
  if not sw then
    return
  end if
end for
```

The available loading space will be divided into several subspaces after packing a layer. The algorithm 2 will be called during each packing procedure, in which the subspaces will be ranged into a lexicographical order and saved into the list S.

1.3.4 Practical Implementation

According to the above sections, the placement strategy can be considered as an important part in the fitness function. In the 3-dimensional packing problem, there does not exit a normal object function as in the standard linear problem. Thus, the whole packing process has to be simulated in order to calculate the space utility rate. The functions that were mentioned in the above sections are integrated together, called the Placement Strategy as shown in the algorithm 3, which is used to simulate the whole packing process. The layer selection and packing part in the algorithm 3 are used to group the boxes with the same size together to produce a new block. In such way, the problem search space will be reduced. This part details will be introduced in the chapter 2.

1.4 Additional Consideration

The 3D packing problem in this project focuses on the general terms. The practical constraints are not added because the test sets used are general benchmarks, rather than practical instances. There are only two points that we want to mention:

- **Another Objective Function**
  The container utility rate is used as the main objective function that is given in the formulation (1.5). This objective function is widely used in most literatures. However, in [9], it introduced the packing density function as shown in the equation 1.19, to evaluate the packing solution, and claims that this function can improve the final space utility.

$$100\% \frac{\sum_{k=1}^{K} NP_k}{TNB} \rightarrow \max \tag{1.19}$$

  Where NP_k is the number of the type k box packed in a solution, and the TNB is the total number of the boxes that can be packed.
  Unfortunately, during the experiments in many literatures, even in this project, this objective function does not have a good performance, and it

Algorithm 3 Placement Strategy($BTPS$,VLT)

Initialization:

Let Placed$_i$ be a flag that indicates whether the box type given by $BTPS_i$ has already been used to pack a box type or not;
Let S be the list of available empty EMS_s;
Let $QtRemain_k$ be the remaining quantity of unpacked boxes of type k;
Let $Skip_k$ be a flag that indicates whether the box type k should be or not when searching for the next box type to pack; $S \leftarrow$ Container size; $QtRemain_k \leftarrow N_k$, $Skip_k \leftarrow$**false** for all k. //N_k is the number of box in type k

Iteration:

while $\exists k : Skip_k =$**false do**
  /*Box type selection.*/
  Let i^* be the first index in $BTPS$ for which $Placed_i =$**false** and $Skip_{BTPS(i)} =$**false**;
  /* Maximal space selection.*/
  $EMS^* \leftarrow 0$;
  EMS^* be the EMS in S in which a box of type k^* is packed after applied Deepest-Bottom-Left Process(DBLP);
  if $EMS^* = 0$ **then**
    $Skip_{k^*} =$**true**;
  else
    /* Layer selection */
    According to $QTRemain_{k^*}$ fill the vector $Layers$ with all the layer-types packable into maximal-space EMS^*;
    Let $MaxLayers$ be number of layer in vector $Layers$;
    $Layer^* = Layers(\lfloor VLT(i^*) \times MaxLayer \rfloor)$ be the layer type selected for placing the box type k^*.
    /* Layer packing */
    Pack $Layer^*$ at the origin of maximal space EMS^*;
    /* Information Update */
    Let $nBox$ be the number of boxes of type k^* contained in $Layer^*$;
    $QtRemain_{k^*} = QtRemain_{k^*} - nBox$;
    if $QtRemain_{k^*} = 0$ **then**
      $Skip_{k^*} =$**true**;
    end if
    $Placed_{i^*} =$**true**;
    EmptySpaceUpdate(S) //By using Algorithm 1;
    DeepBottomLeft(S) //By using algorithm 2
  end if
end while

Output: All packed box will be assigned a location in the container.

usually will overstate the volume utilization achieved. Thus, the formulation 1.5 are used as the objective function.

- **Rotation Consideration**

  In order to do the fair comparisons with other approaches, the box are allowed to rotate when it simulates the packing procedure. In the 3-Dimensional

container loading problem, each box can generate at most six variants by rotating around its axis (x_i, y_i, z_i). This Rotation method can be implemented as the algorithm 4.

Algorithm 4 ItemRotation(box_i,$rotateplag$)

Initialization:

//$rotateplag$ is a flag that indicate if the rotation function is open or not;
Set $sw = rotateplag$;
// $variantset$ uses to space the variant items;
Let $variantset$ be empty;
Let w_i, l_i, h_i be width, length, and high of the item$_i$;
Let w_r, l_r, h_r be the variant box$_i$;
Let $tempswap$ be the save buffer;

Iteration:

1: //This function is used to generate rotation variants of the items;
2: **if** rotateplag=**true then**
3: //original type;
4: save the x_i, y_i, z_i into $variantset$;
5: //Rotate around length;
6: $x_r = z_i$, $z_r = x_i$,$y_r = y_i$;
7: save x_r, y_r, z_r into $variantset$;
8: //Rotate around high;
9: $tempswap = y_r$, $y_r = x_r$, $x_r = tempswap$;
10: save x_r, y_r, z_r into $variantset$;
11: //Rotate around Length;
12: $tempswap = z_r$, $z_r = x_r$, $x_r = tempswap$;
13: save x_r, y_r, z_r into $variantset$;
14: //Rotate around width;
15: $tempswap = z_r$, $z_r = y_r$, $y_r = tempswap$;
16: save x_r, y_r, z_r into $variantset$;
17: //Rotate around high;
18: $tempswap = y_r$, $y_r = x_r$, $x_r = tempswap$;
19: save x_r, y_r, z_r into $variantset$;
20: **else**
21: //There is only one element in $variantset$ when not support rotation;
22: save the x_i, y_i, z_i into $variantset$;
23: **end if**

Output: $variantset$

1.5 Summary

In this chapter, all relevant knowledges of the container loading problem that were involved in this project were presented. In order to make more wide comparison, the constraints in this problem are set by following the most literatures.

Meanwhile, the algorithms that were presented in the chapter 1 have already been tested, and can produce a reliable result. All of them compose the fitness part in our optimizer. As mentioned at the beginning of this chapter, we tend to implement the evolutionary algorithm to tackle this problem. Thus, a reliable fitness function has to be guaranteed. The detail about this optimizer will be introduced in later chapters (2,3).

Chapter 2

Optimization Approach

The 3-Dimensional Container Loading Problem is always a challenge in many fields because of its practical application and the *NP-hard* property. Until now, there is still not an efficient approach to tackle them. In the past, design of packing solution was based on experience. However, in the last three decades, a significant number of computerized methods have been applied to bin packing problem, ranging from linear programming, dynamic programming, and enumeration techniques in early years, to more recently the Evolutionary Algorithm [10, 13], the Tabu Search Algorithm [17], the Tree Search Algorithm [8]. In this work, we tend to choose the GA to be our main algorithm framework.
This chapter is divided into four sections. In the section 2.1, the general structure of the EA will be introduced. The section 2.2 introduces more detail about the standard GA. In the section 2.3, the Random Key concept will be presented . In the section 2.4, the concept of Layer will be added. In the section 2.5, a short summary will be given.

2.1 Evolutionary Algorithm

The GA is one important branch of EA_s, and other two branches are the Evolutionary Strategy (ES) and the Genetic Programming (GP). The idea of EA_s is inspired

from the Darwin's theory of the survival of the fittest and mimic the process of the organic evolution by using operators "*population*", "*mutation*", "*recombination*" and "*selection*" [45]. The better an individual performs, the bigger chance it can be chosen to survival and generate offspring. Over the course of evolution, this leads to a penetration of the population with the genetic information of individuals of the above-average fitness [46]. There is a high level abstraction for evolutionary algorithms given in the algorithm 5, which include all the essential components of standard implementations. For the different evolutionary computation models, such as the GA, the ES and the GP, the reader can obtain more details about them in books [3, 47].

Algorithm 5 General Schema of an evolutionary Algorithm

Initialization:

Set $t := 0$
Initialize population with random candidate solutions
Evaluate each candidate solution

Iteration:

1: **while** terminate condition is **not** satisfied **do**
2: Select parents;
3: Recombine pairs of parents
4: Mutate the resulting offspring
5: Evaluate new candidate solution
6: Select individuals for the next generation
7: $t := t + 1$
8: **end while**

Output: *variantset*

According to this abstraction, we can see that there are several notable characters included in the EA: Population based, and new candidate solutions are generated by the recombination or the mutation, and the stochastic method.

2.2 Introduction of Standard Genetic Algorithm

Genetic algorithms (GAs) are search methods based on principles of the natural selection and genetics (Fraser [48]; Holland [49]). In this section, all relevant information about the GA and associated terminology will be introduced.

The decision variables of the problem will be encoded into a finite-length strings of alphabets by GAs. The term *chromosomes* in the GA is seen as a string which is a candidate solutions to the search problem, the alphabets of it are represented as *genes* and the values of genes are called *alleles*. For example, in the problem such as bin packing problem, a chromosome could represent a packing pattern, and a gene may represent a box. Comparing with traditional optimization techniques, instead of working with parameters themselves, GAs work with coding of parameters.

In order to evolve good solutions and implement a natural selection, a measure that can distinguish qualified solutions from the solution set have to be provided. The measure could be an objective function that is a mathematical model or a computer simulation, or it can be a subjective function where humans choose better solutions over worse ones (Vishwanath [50]). Essentially, a candidate solution's relative fitness is determined by the fitness measure, which will subsequently be used by GAs to guide the evolution of qualified solutions.

The population based is an important feature for the GA. Comparing with traditional search methods, canonical genetic algorithms search a set of candidate solutions (population) simultaneously. The size of the population is one of important factors of the genetic algorithm, which is assigned by user and can affect the scalability and the performance of the GA. For instance, the premature convergence and suboptimal solutions can be caused by a small population size. However, unsuitable large population sizes will consume a great deal of valuable computational resource for nothing.

After encoding the problem into a chromosome and determining the fitness function, the solutions start to evolve in the search space by following the below steps:

- **Initialization**

  In order to cover the domain knowledge or the correspond information, the population will be sampled randomly across the search space in the initial state.

- **Evaluation**

  Once a population is generated, the objective function will be used to evaluate each individual in this population by assigning each of them a fitness

value.

- **Selection**
  After fitness evaluation, the individuals with high fitness value are selected selected to survive and generate a new population. To implement this operator, two selection method can be used, such as the roulette-wheel selection and the tournament selection. By using these methods, the bad individual also have an opportunity to survive to generate an offspring. This property can help the algorithm to escape the local optimal.

- **Recombination**
  In the recombination method, two or more parent individuals will be combined in order to generate new offsprings. These offsprings will inherit a part of genes from the parent individuals. By following this principle, the new individual could be a better solution than the parent individual, but this is not guaranteed. There are several recombination mechanisms which can be adapted in different type problem. They will be introduced more detail in the later section.

- **Mutation**
  The mutation operator is used to generate a new individual by changing several genes of a candidate solution with a small probability. This mutation operator guides the candidate solution randomly walk in its neighbourhood search space.

- **Replacement**
  After creating the offspring population , the old population will be replaced by a new population. To implement this procedure there are many replacement techniques used, such as the elitist replacement, the generation-wise replacement and the steady-state replacement method.

In the GA, the crossover operators play very important role [51]. This operator guides the solution to find the optimal position in the search space by following the informations that are given by the parent individuals. By using this operator, the algorithm can learn the landscape of the search space, but this is the ideal state.

In fact, there does not exist any single crossover operator that can cover all the problems. Thus, in order to make the GA work on different problems efficiently, a suitable crossover operator has to be specified. There are several crossover operators that can be used to tackle different problems, which will be introduced in the section 2.2.3.

2.2.1 Representation and Fitness Evaluation

Normally, the genetic algorithms work on binary strings with a fixed length l, rather than on the real value space as the ES. Thus, before entering evaluation part, there are two different processes that could be implemented for each individual based on the fitness function type. For the boolean objective function, the binary type representation can be evaluated directly. However, the fitness function usually represents a continuous parameter optimization problem, the binary type representation has to be interpreted as a vector of real value within a specified range. There is a simple example for the encoding and the decoding as shown in the table 2.1

TABLE 2.1: Decoding and Encoding

(a) Decode

Binary Strings		Real Value
10101	$\rightarrow$	21
11100	$\rightarrow$	28
11101	$\rightarrow$	29

(b) Encode

Binary Strings		Real Value
01011	$\leftarrow$	11
10011	$\leftarrow$	19
10100	$\leftarrow$	20

2.2.2 Selection

For the selection operator, it can be generally classified into two classes:

- **Fitness Proportionate Selection**

  The fitness proportionate selection chooses an individual based on the proportion of the individual fitness value comparing with the sum of the total fitness. Two selection methods are included in this class, namely the roulette-wheel selection and the stochastic universal selection. Since the roulette-wheel is widely used, it can be used as an example to explain more detail about this type selection, as seen in the following list.

  1 Evaluate the individuals in the current population by using an objective function f_i.

  2 Calculate the proportion (slot size), p_i, of choosing each member of the population: $p_i = f_i / \sum_{j=1}^{n} f_j$, where n presents the population size.

  3 Compute q_i, the cumulative probability, for each individual: $q_i = \sum_{j=1}^{i} p_j$.

  4 Generate a random number r by using $U(0, 1)$

  5 If $r < q_1$, then the first individual, x_1, is selected. Otherwise, the individual x_i such that $q_{i-1} < r \leq q_i$ is selected.

  6 Select n candidates and put into the mating pool by repeating $(4 - 5)$ steps n times.

  There is a practical instance in table 2.2 that can be used to illustrate the above procedure.

Chromosome #	1	2	3	4	5
Fitness :f	35	43	12	9	4
Probability: p_i	0.3398	0.4174	0.1165	0.0873	0.0388
Cumulative Probability: q_i	0.3398	0.7572	0.8737	0.9161	1.000

TABLE 2.2: Example of Roulette-wheel Selection

  The total fitness, $\sum_{j=1}^{n} f_j = 35 + 43 + 12 + 9 + 4 = 103$. The corresponding selection probability and cumulative probability can be found on the above table. Suppose that a random number r is 0.64, then the second chromosome is chosen because of $q_1 = 0.3398 < 0.64 \leq q_2 = 0.7572$. To implement this selection method the algorithm 6 can be referred.

Algorithm 6 Roulette Wheel Selection [3]

Initialization:
Set $current\_member = 1$;
Let the a_i be the cumulative probability of individual i
1: **while** $current\_memeber \leq \mu$ **do**
2: Pack a random value r uniformly from $[0, 1]$;
3: Set i=1;
4: **while** $r < a_i$ **do**
5: Set $i = i + 1$;
6: Set $mating\_pool[current\_member]$= parents$[i]$;
7: Set $current\_member = current\_member + 1$;
8: **end while**
9: **end while**
Output: *parent population*

- **Ordinal Selection**

  Two selection schemas are involved in this class, namely the tournament selection and the truncation selection. In the tournament selection, the individual s are selected randomly and put into a tournament set with size k, in which all the candidate solutions will compete based on their fitness value. The individual with the best fitness value in the tournament set will be selected and put into the mating_pool. Normally, this tournament set will be much smaller than the population size. This tournament procedure will implement μ times in order to fill a mating_pool with size μ. For the truncation selection, it is almost same as the tournament selection, but it chooses the top $(1/s)$th of the individuals to copy them s times into the mating pool. The implementation of the tournament selection can refer the algorithm 7

Algorithm 7 Tournament Selection[3]

Initialization:

Set $current\_member = 1$;

Iteration:

1: **while** $current\_member \leq \mu$ **do**
2: Pick k individuals randomly, with or without replacement;
3: Select the best of these k comparing their fitness values;
4: Denote this individual as i;
5: Set mating_pool[$current\_member$]=i;
6: Set $current\_member = current\_member + 1$
7: **end while**

Output: *parentpopulation*

2.2.3 Recombination

After selection, individuals in the mating pool will be put into the recombination, or called crossover, procedure in order to generate a new population. This new population will own part of informations that inherits from the parent population. Without adding this operator, the GA can not confirm the requirement of the general EA. Thus, the crossover plays an important role by constructing competitive Genetic Algorithms (GA_s) [51] . In the later of this section, several crossover operators will be introduced. For more crossover methods, they can be referred by the literature [3].

For the most of crossover operators, to recombine the individuals they follow a similar process, in which two chromosomes are selected randomly from the mating-pool and recombined based on a probability p_c, named the crossover rate. During this process, an uniform random number r will be generated, and if $r \leq p_c$, the two selected individuals implement the recombination method to produce two offsprings. Otherwise, generating two offsprings is simply copying this two selected individuals. A suitable crossover rate p_c can either be found by the experiments, or the schema-theorem principles (Goldberg [52]).

- **k-point Crossover**

  Normally, the simplest and widely used methods are one-point and two-point crossover models. The one-point crossover is to select a gene site at random over the chromosome length, the alleles of genes on both sides of the

individuals are exchanged. In the two-point crossover model, two gene sites are randomly selected. The alleles between the two sites of two mating parent individual are exchanged. The idea of one point model can be modified to k-point model, in which k crossover points are used, instead of one or two. Above crossover models were illustrated in the Fig.2.1.

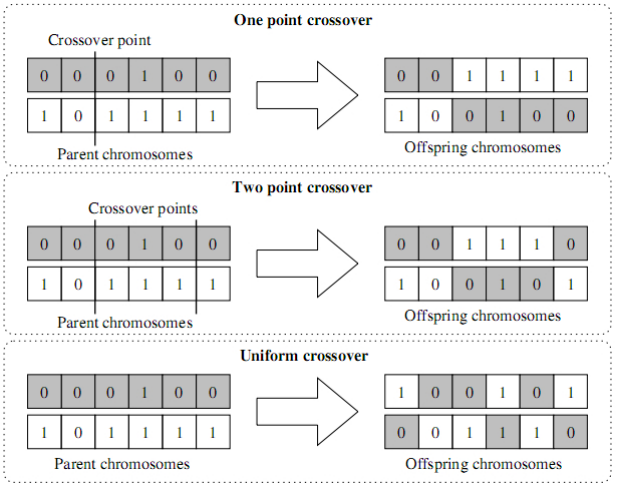


FIGURE 2.1: k-Point Crossover, Uniform Crossover [53]

- **Uniform Crossover**

  Another extension of the k-point Crossover is the uniform crossover, in which every allele in the pair of mating parent individuals is exchanged with the specified crossover rate p_c, as seen in the Fig.2.1.

- **Order-Based Crossover**

  Normally, the k-point Crossover is not suitable to handle the permutation problem, such as the travelling salesman problem, because it usually produces an infeasible solution. Therefore, to make the Crossover component adapt these type problems, Davis [54] introduced the order-based crossover operator, in which two mating parent individual are selected at random, after that, two crossover sites in the two individuals are randomly selected. The genes between two sites are copied to the offsprings. Starting from the second cut point copies the genes that do not appear in the offspring yet from the alternative mating parent by following the original order they appeared.

- **Partially Matched Crossover (PMX)**
  The PMX operator(Goldberg and Lingle [55]) is also an order based crossover approach that can be used to tackle the permutation search problem. In the first step of this crossover method, it will select two mating parent individuals and two crossover sites at random as usual. In the second step, it selects an allele a_i between the two crossover sites of parent individual A, and finds the allele b_i in the parent individual B, which has the same position as the a_i stayed in A. Instead of exchanging allele a_i and b_i as the typical crossover done, it exchanges the allele a_i with a_j that is same as the allele b_i, but is in parent A. After all exchanges, we can obtain two offsprings that include the order information from the parent individuals and also are feasible solutions.

2.2.4 Mutation

According to the crossover procedure, the solution can hold the desirable information that inherits from the parent individuals. However, what the problem is that the crossover component will reduce the population diversity, in the other word, this operator will guide the solution into a local optima. To overcome this disadvantage, the mutation operator is introduced. After crossover, certain genes of each individual in the new population will be mutated based on a small probability, called the mutation rate p_m. By using this mechanism, the algorithm is able to add the diversity into the population and explore the entire search space. The practical implementation for the mutation operators of the GA in the different search problems is quite similar, in which the gene will randomly choose a value with respect the domain of this allele. The standard representation of the GA, for instance, is a binary string. Each allele of a chromosome can either be 0 or 1. Thus, the mutation operator in the standard GA is to flip the certain bit based on the mutation rate p_m.

2.2.5 Theory

To analyse the behaviour of Genetic Algorithms, the most efforts idea is the *schemata*, which is a template that determines a subset of strings with similarities at certain string positions. John Holland proposed this theorem in the 1970_s . To learn more detail the reader can refer to the literature [49].

2.3 Random Key Representation

To address a wide variety of sequencing and optimization problems, such as scheduling, resource allocation, and quadratic assignment problem, a solution can naturally be represented by a permutation. This representation is widely used in Evolutionary Algorithms. However, by using this representation, the solutions of the algorithm will involve an infeasible issue when they undergo the offspring generating process. For instance, in the GA, two permutations, $x = (1, 2, 3, 4, 5)$ and $x' = (5, 3, 4, 2, 1)$, can directly be used as two chromosomes. By using the one-point crossover operation, the permutation would be cut at some point. during the crossover process, these are two offsprings generated by exchange leading segments, such as $(5, 3, 3, 4, 5)$ and $(1, 2, 4, 2, 1)$. For these two individuals, the GA can not accept them because they are not valid tours, or infeasible solutions.
In order to maintain a feasible solution after undergoing this process, the algorithm needs a problem specified operator to overcome the offspring feasibility difficulty, such as the PMX crossover [55], the subsequence-swap operator [56], the subsequence-chunk operator [57], edge recombination [58], and the ARGOT strategy [59]. Although these operators can tackle the infeasible issue, they will be more time consuming. Furthermore, there is an issue for using the permutation in the container loading problem, in which the item in the permutation can not be packed completely. In this case, the order relation ship in the permutation can not be used efficiently. In addition, the need for specialized representations for the different problem variation has always been a problem as well.
Therefore, To overcome this disadvantage a robust representation technique, called *random keys*, was proposed by James C Bean [2]. It can be used to represent the

solutions of many sequencing and optimization problem, and guarantees the feasibility of all offspring without creating additional overhead.

2.3.1 The Concept of Random Key

The solution of a combinatorial problem can be seen as a sequence. By using the traditional binary representation to encode this solution is quite complicated, and the decoding part will become very time consuming. The most critical problem is that the decoding part can not guarantee to produce a feasible solution. In the random key representation, each bit of a solution will be encoded by a random numbers. The solution will be decoded by sorting these random numbers into specified order. Because all of the operations just occur on the random numbers, all solutions are feasible after decoding.
Therefore, the random key encoding can be considered as tags, which can be used to map the literal space into a surrogate space, depicted by the Fig.2.2. This space normally respect with the domain $[0, 1]^n$. The search space usually is more than

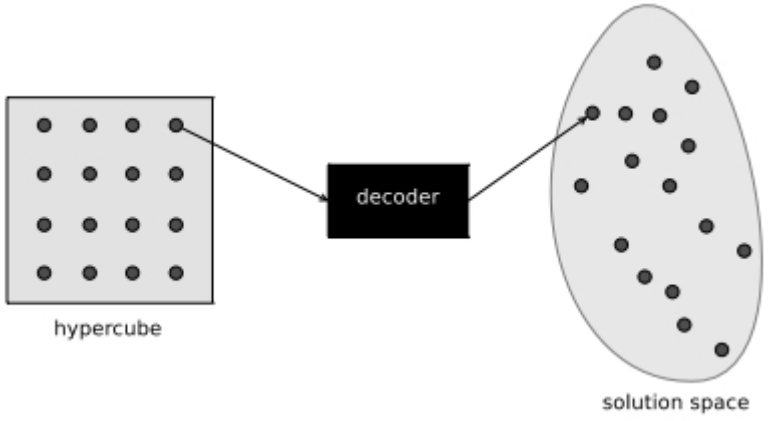


FIGURE 2.2: Decoder used to map solution in the hypercube to solutions in the solution space where fitness is computed [60]

two dimensions. By using this type encoding, each dimension will be tagged by a random key. For this reason, the direct search space of the GA is a continuous n-dimensional unit hypercube as seen in the Fig.2.2. Thus, this encoding can simply be considered as a hypercube sampling procedure.

2.3.2 Hypercube Sampling

the hypercube sampling is a method of sampling that can be used to produce input values for estimation of expectation of functions of output variables [61]. For the population based algorithms, in each iteration it will generate a set of solution, which is a sampling procedure. The distributional situation of the population in the search space determines whether the algorithm can learn the information of the search space or not. A qualified sampling method should has a good space filling properties, which means that the solutions are more uniformly distributed over the domain, otherwise, it is not. There are two examples shown in the Fig.2.3. Where the Fig.2.3(a) shows that the random number in each dimension is generated by normal distribution, and in the Fig.2.3(b), they are produced by the uniform distribution.

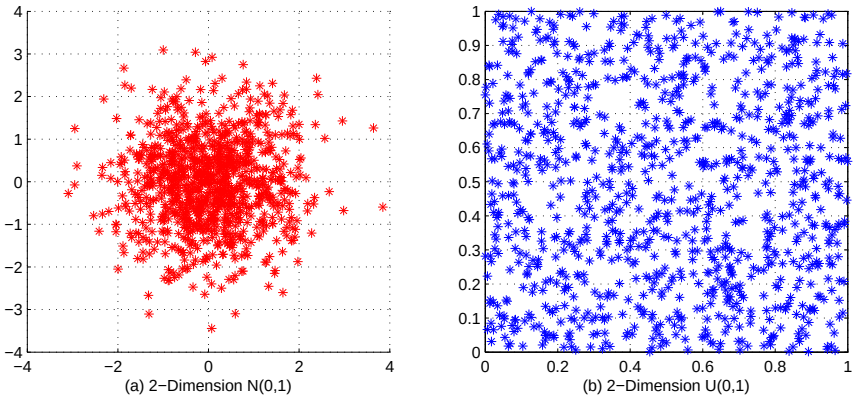


FIGURE 2.3: HyperCube Sample

As mentioned before the solution of the combinatorial problem is a sequence, the solutions in this type problem space are incomparable. The difference between two solutions is the permutation, it is unable to say that one permutation is better than others. Thus, in order to enable them to grasp comprehensive information of search space of combinatorial problem as much as possible, we have to select a sample method, in which the solution will uniformly be distributed, seen as in the Fig.2.3(b).

2.3.3 Advantage of Random Key Representation

There are two main advantage by using this representation. The first one is that the random key encoding is robustness to problem structure. Variations in problem structure are captured in the mapping and the objective function value passed back. Mappings are the problem specific, but generally involve sorting the random keys [2]. The second advantage is offered by the concept of the hypercube, which provides a flexibility to the search space. For instance, if there are a few dimensions dropped out, the search space still belong to a hypercube space. Although it could cause a sub-optimal, it keeps the hypercube design at least. Thus, if one cannot afford another set of data properly designed for the smaller domain, the existing data can be reused without reduction in the number of sampled points [62].

2.3.4 Disadvantage for The Random Key Representation

High dimensions are one of characters for the combinatorial problem. In the 3-Dimensional container loading problem, each item could be considered as a dimension, in such case, the solution in this problem can have more than 100 dimensions. Thus, by using the hypercube sampling, it will suffer from the curse of dimensionality. While uniformity in each dimension is preserved, the space filling properties become questionable. As the number of variables increase, it becomes harder to fill the design space. When the optimization pushes points further apart, the sample tends to create a vacuum in the center of the design space. For such case, it not only appear in the hypercube sampling, but also is a normal phenomenon in the high dimensions problem.

2.4 Divide and Conquar

According to above sections, the central defect of the random key representation in the 3-dimensional container loading problem was found. In order to overcome this shortage, we try to reduce the problem dimensions by following the idea of

the Divide and Conquer.

Firstly, the packing item with the same size will be assigned to the same group. During each packing procedure, a set of items with the same size will be packed into the available space, rather than packing each of them.

Secondly, due to the packing process occur in the 3-dimension space, each available packing space has at most six different packing patterns, called Layers. The layer will be used by a constructive heuristic based loading approach. In the packing procedure, after selecting a packing item and an empty maximal-space, this item could generate six rectangular arrangements by filling each side of this space in rows and columns. This rectangular arrangement is called *Layer*. In this project, items are allowed to rotate when it is packed, so each item can have six different variants, and each variant will generate at most six layers. Therefore, there are at most thirty six layers that can be used to pack the selected item. the layers generated process can be illustrated by the Fig. 2.4.

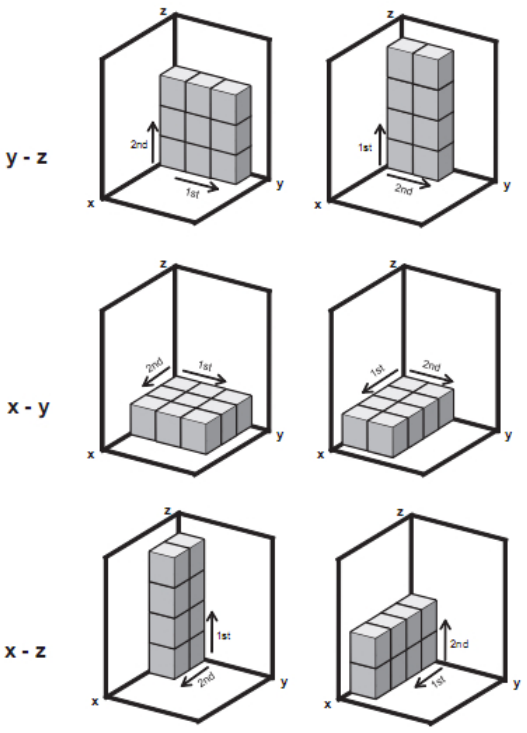


FIGURE 2.4: Six different layer types for an box type rotation variant [1]

Finally, the Placement Strategy will select a layer to pack a set of items with the

same size into the container. After that, this group of items will be seen as an entirety to process. The placement function will assign a location to the packing layer, rather than each item. To implement this function, the algorithm 8 can be used.

Algorithm 8 LayerGenerator(box_i, ems_i)

Initialization:

Let *layerset* be a empty set for saving layers;
Let *Wnub, Lnub, Hnub* be the counters to save the number of packed box in ems_i.
Let CW, CL, CH be the width, length, and high of the ems_i;
Let (x_l, y_l, z_l) and (x'_l, y'_l, z'_l) be the DLB and the FRT point of layer;
Let *boxnub* be the number of unpacked box_i, bw, bl, bc be the width, length, and high
Let $(x_{ems}, y_{ems}, z_{ems})$ and $(x'_{ems}, y'_{ems}, z'_{ems})$ be the DLB and the FRT point of ems_i;

Iteration:

1: //Layer ranged from the left to right, and move the front;
2: $Wnub = (\lfloor CW/bw \rfloor > boxnub)?boxnub : \lfloor CW/bw \rfloor$;
3: $Lnub = (\lfloor boxnub/Wnub \rfloor == 0)?1 : \lfloor boxnub/Wnub \rfloor$;
4: **if** $Lnub * bl \leq CL$ **then**
5: $x_l = x_{ems}$; $y_l = y_{ems}$; $z_l = z_{ems}$;
6: $x'_l = x_l + bw * Wnub$; $y'_l = y_l + Lnub * bl$; $z'_l = z_l + bh$;
7: the two points builds up a layer and saved into *layerset*;
8: **else**
9: $Lnub = \lfloor CL/bl \rfloor$;
10: $x_l = x_{ems}$, $y_l = y_{ems}$, $z_l = z_{ems}$;
11: $x'_l = x_l + bw * Wnub$, $y'_l = y_l + Lnub * bl$, $z'_l = z_l + bh$;
12: The two points builds up a layer and saved into *layerset*;
13: **end if**
14: //Layer ranged from deep to front and move to right direction;;
15: $Lnub = (\lfloor CL/bl \rfloor > boxnub)?boxnub : \lfloor CL/bl \rfloor$;
16: $Wnub = (\lfloor boxnub/Lnub \rfloor == 0)?1 : \lfloor boxnub/Lnub \rfloor$;
17: **if** $Wnub * bw \leq CW$ **then**
18: $x_l = x_{ems}$, $y_l = y_{ems}$, $z_l = z_{ems}$;
19: $x'_l = x_l + bw * Wnub$, $y'_l = y_l + Lnub * bl$, $z'_l = z_l + bh$;
20: The two points builds up a layer and saved into *layerset*;
21: **else**
22: $Wnub = \lfloor CW/bw \rfloor$;
23: $x_l = x_{ems}$, $y_l = y_{ems}$, $z_l = z_{ems}$;
24: $x'_l = x_l + bw * Wnub$, $y'_l = y_l + Lnub * bl$, $z'_l = z_l + bh$;
25: the two points builds up a layer and saved into *layerset*;
26: **end if**
27: //Using the same procedure to generate the rest layers
28: //Layer ranged from left to right and move to top direction;
29: //Layer ranged from bottom to top and move to right direction;
30: //Layer ranged from deep to front and move to top direction;
31: //Layer ranged from bottom to top and move to front direction;

Output: *Layerset*

In the algorithm 8, the box_i makes certain number copies along with the axis (x, y, z) of the ems_i to generate six different layers respectively. For instance, in line 1-12, the number of copies of the box_i are ranged from left to right, then moving front. The $Wnub$ is the maximal number of the box_i which can be packed along with the x axis of the ems_i. The $Lnub$ is the maximal number of the box_i in the y axis. in the other world, the $Wnub$, $Lnub$, and $Hnub$ hold the dimension constraint of the ems_i and quantity constraint of the box_i. For the questions about how to select the layers, more explanation will be given in the chapter 3.

2.5 Summary

In this chapter, the general framework of the evolutionary algorithm and all relevant components are introduced. In order to make the GA adapt the 3-Dimensional container loading problem efficiently, instead of using the traditional binary representation, a specified representation was used to model this problem, called random key representation. Furthermore, according to analysing the shortage of this representation, we proposed a corresponding solution. To combine the concepts in above sections, the general-purpose metaheuristic framework can be constructed as depicted in the Fig.2.5

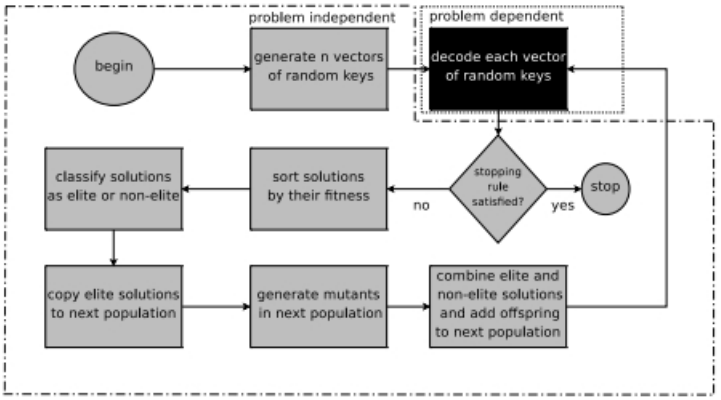


FIGURE 2.5: General-purpose Metaheuristic Framework[60]

In the next chapter, the standard GA will be modified based on the above structure to solve the 3-Dimensional Container Loading Problem (3D-CLP).

Chapter 3

Biased Random Key Genetic Algorithm

Genetic Algorithms (GA_s) are adaptive methods that are used to solve optimization problems [63, 64] by associating solutions of the optimization problem with individuals of the population. Over many generations, natural populations evolve according to the Charles Darwin's principle of natural selection, called survival of the fittest [65]. According to simulating this process, solutions to an optimization problem are able to be evolved by the genetic algorithm if there is a suitable encoded. Thus, to define a genetic algorithm an encoding (or representation) for the problem have to be specified first. Furthermore, a reliably objective function is also important for the whole evolutionary process, which will guide the solution into the desirable region.

Currently, the most efficient approach to solve the 3D-CLP is the Biased Random Key Genetic algorithm (BRKGA)[1]. In this project, in order to improve the BRKGA, the original initialization component and mutation operator will be modified by using a different position pattern generator and an ES mutation operator respectively.

This chapter will introduce two main aspects: In the section 3.1, the encoding and decoding procedure will be explained. In the section 3.2, the whole evolutionary process will be introduced.

3.1 Representation and Decoding

The general framework of our optimizer is referred as the BRKGA [60], the representation of which is a string of random real numbers with respect the uniform random distribution between 0 and 1. The evolve strategy involved is same as the one proposed by Bean [2], except for the method crossover. The important factor of this GA is that all the offsprings produced by crossover are feasible solutions, this is achieved by replacing the feasibility issues into the fitness function evaluation as much as possible. Suppose a feasible solution can be generated according to decode any random key vector, and any solution after crossover is also a feasible solution because it can be seen as a random solution as well. By means of the dynamic of the genetic algorithm, the connection between the random key vectors and corresponding solution will be comprehended by the system.

- **Representation**

  In this GA, the representation will divided into two steps of random real number vectors. The first step is used to encode the box type packing sequence (BTPS). The second step is used to encode the vector of layer types (VLT). These two steps will be used in the placement procedure. The structure of this chromosome can be seen below.

$$Chromosome = \left(\underbrace{gene_1, \cdots, gene_M}_{\text{BTPS}} \ , \ \underbrace{gene_{M+1}, \cdots, gene_{2M}}_{\text{VLT}} \right)$$

  Where the first M genes are used to encode the box type packing sequence, and the vector of layer types can be encoded by the second part M genes. The second part of the chromosome is used as the index indicator. Recalled the $layerset$ generated in the algorithm 8, it will generate at most 36 layers and save in the $layerset$ in each packing method. Each random number in the second part can indicate which layer should be selected from this $layerset$ during current packing process.

- **Decoding**

  When the chromosome has been put into the fitness function, the first M

genes will be decoded into the box type packing sequence (BTPS) by simply sorting the random value with increasing or decreasing order (seen in Fig.3.1). The placement function will decide which item should be packing first by using the BTPS.

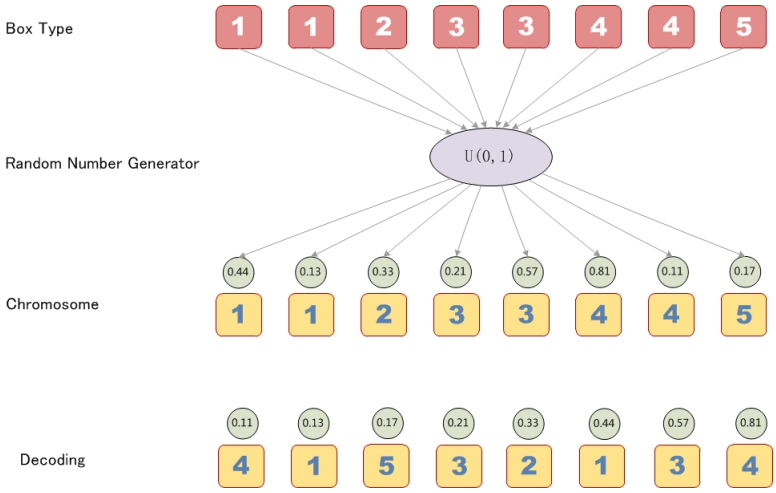


FIGURE 3.1: First M Gene Decoding

The second M genes will be saved in the vector of the layer type (VLT), which will directly be used by layer selection part without any manipulation. These values will be applied in the line 13 of the algorithm 3 as shown below.

$$Layer^* = Layers(\lfloor VLT(i^*) \times MaxLayer \rfloor)$$

Where the $Maxlayer$ will be a number that equal to the size of the set of the $Layers$ ($Layerset$). the $VLT(i^*)$ is the i element of the second part of chromosome. Since the value of $gene_{M+i}$ is random real number between 0 and 1, the $\lfloor VLT(i^*) * Maxlayer \rfloor$ is the index of the $layerset$.

It is should be noted that, during the whole evolutionary process, all relevant operation only work on a random key representation. After each time evaluation, the packing sequence will be sorted back to its original permutation in order to keep the random key always fixed in the same position.

3.2 Evolutionary Process

In this project, the optimizer follows the evolutionary process of the standard GA, so there are five components involved. The first one is initialization part, in which the population is initialized with random-key vector, the elements of this vector are random real numbers uniformly sampled for the interval $[0, 1]$. The second one is the fitness component, in which each individual in the population will be evaluated and assigned a merit value. After that the population will be put into selection part, a fixed number individuals are selected based on their fitness value. In this project, we use the *elitist strategy* [63], so a fixed number of best individuals will be copied into the next generation directly without change. The fourth part is the crossover section, in which the selected individuals will be used to produce new offsprings. In the final part of this optimizer, it is called mutation operator, in which the several random selected genes of the offspring in the new population will be mutated by a small probability. However, The mutation operator (Mutant) in the BRKGA is quite different from one in the standard GA, which generates a fixed number individuals completely at random by using same way as in the initialization component. The purpose of using this mutation operator is also to prevent premature convergence. The Fig.3.2 illustrates the architecture of this algorithm and the evolutionary process.

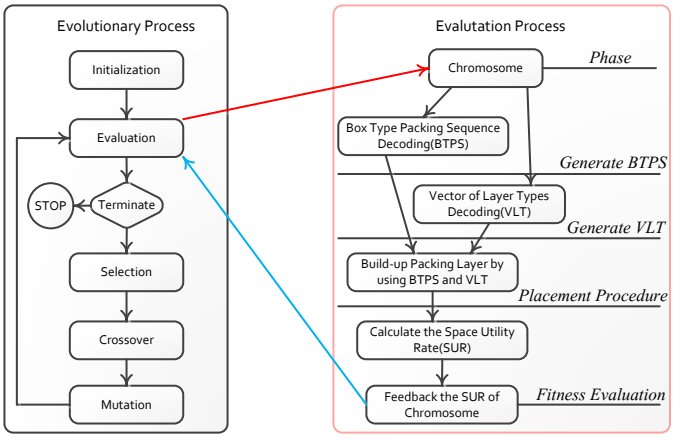


FIGURE 3.2: The Architecture of The BRKGA Algorithm

3.2.1 Initialization

Normally, there are two main tasks in the initialization component. The first task is to construct the representation based on the rule that have already been defined. The second task is to build up the initial population by using the representation model. However, one point should be noted that, in the 3-Dimensional CLP, before assigning each item a random key, the item in the problem has to be arranged into a specified pattern, called position pattern. Each individual will hold this pattern and never change during the whole evolutionary process, except in the decoding procedure this pattern will be reordered by sorting the random key. However, after evaluation, this pattern will be restored. Thus, as mentioned before, the random key representation is just a position tag. There is an example shown the construction of the individual and population in the Fig.3.3.

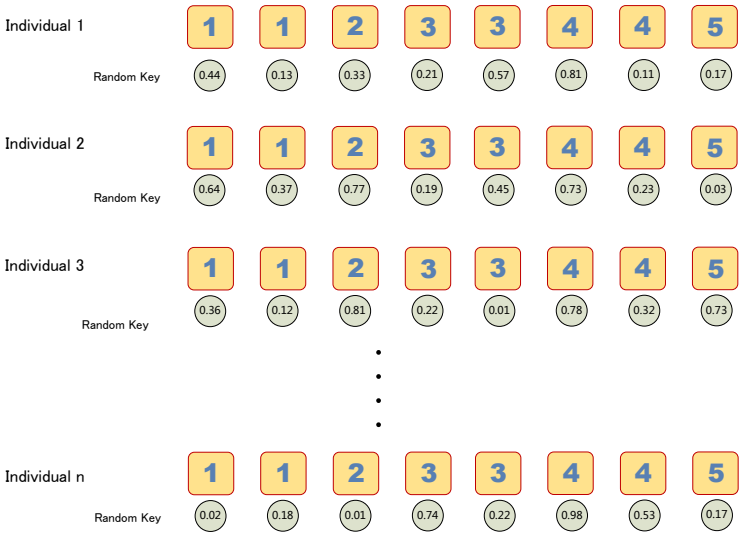


FIGURE 3.3: Population Structure

According to the example, it is clearly seen that the initial patterns in the every individual are same, the random key is just a tag for each position. To implement this part function the algorithm 9 can be referred.

Algorithm 9 BRKGA_Initialization(*problem*)

Initialization:

Define $gene_i$ be a structure which can save a box item;
Define *chromosome* be a vector of *gene*.
Let *populationset* be the set to save the individuals;
Let box_i be the i type box in the *problem*;
Set *nbcounter* be 0;
Let λ be the size of the initial population;
Set a uniform random number generator $U(0, 1)$;
Set VID be 1, which is ID number generator used to give each item an ID;

Iteration:

while size of the *populationset*! $= \lambda$ **do**
  repeat
    Let *numberbox* be the total number of box_i should be packed;
    while *nbcounter* $\leq$ *numberbox* **do**
      Generate a random key by using $U(0, 1)$.
      Save this random key, box_i,and VID into $gene_i$;
      nbcounter=*nbcounter*+1, $VID = VID + 1$;
      Save the $gene_i$ into the *chromosome*;
    end while
    nbcounter $= 0$;
  until box_i be the last item in the *problem*
  Save the *chromosome* into *populationset*;
  Initialize the *chromosome*;
end while

Output: *populationset*

3.2.2 Fitness Function

The operation procedure of the fitness function has already been illustrated in the Figure 3.2. The kernel function, the Placement Strategy, was also introduced by the algorithm 3 in the chapter 2. However, there are still several auxiliary functions included in the fitness component that have not been mentioned yet.

- **Space Utility Rate Calculate**

  After undergoing the Placement Strategy function, the items will be packed into a different layer, and the layer will be assigned with the corresponding position in the container. For the items cannot be packed, they will be given an unavailable coordinate in order to make it be ignored in the utility rate calculation function. The Space Utility Rate function calculates the space utility rate (SUR) by adding volume of each layer together and divide

the container volume, which can be represented by the equation 1.5. To implement this function the algorithm 10 can be used.

Algorithm 10 Space Utility Rate(*solution*)

Initialization:
Let $Vlayer_i$ will be the volume of i layer element in the *solutions*;
Set *packedvolume* to save the total packed volume;
Let SUR be the current space utility rate;
Let $TotalVolume$ be the container volume;

Iteration:
1: **repeat**
2: $packedvolume = packedvolume + Vlayer_i$;
3: **until**
4: $SUR = packedvolume/TotalVolume$;

Output: SUR

- **Pattern Revert Function**

  In this function, the solution will be sorted to the original order as explained in the section 3.2.1. By using this function, a random key will always tag the fixed position of the sequence, which can help the algorithm to learn the connection between the random key and the corresponding item. To implement this function is quite simple, it is a sorting algorithm, which reorder the solution by using the VID of the gene. It can be implemented as the algorithm 11. To implement this function we use the concept of quick sorting, proposed by Tony Hoare. This concept can significantly increase running efficiency of the fitness function because the complexity of this sorting algorithm, on average, is $O(n \log n)$. It is faster than other sorting algorithm, such as the bubble sort ($O(n^2)$).

Algorithm 11 Revert(*solution*,*l*,*r*)

Initialization:

//The l, r represent the most left index and the most right index of the *solution*;
//The *solution* is a individual which have already been evaluated;

Iteration:

```
1: if l < r then
2:    Let temp_VID be the VID of the most left item in the solution;
3:    i = l,j = r, temp_VID = VID_j
4:    while i < j do
5:       while i < j and VID_j > temp_VID do
6:          j++;
7:       end while
8:       if i < j then
9:          Swap(solution_{i++},solution_j);
10:      end if
11:      while i < j and VID_i ≤ temp_VID do
12:         i++;
13:      end while
14:      if i < j then
15:         Swap(solution_{j--},solution_i);
16:      end if
17:   end while
18:   Set solution_i to be the element with temp_VID.
19:   Revert(solution,l,i-1);
20:   Revert(solution,i+1,r); //Recursive call;
21: end if
```

Output: *solution* with original order;

All functions mentioned are included in the fitness component. The stability of this fitness function is guaranteed through running many experiments. The implementation of this function is described by the algorithm 12. Through the practical implementation, we found that this fitness function is still very time consuming, even though we improved some sub components of it by modified their calculation method. Thus, in order to increase its running speed, it is compiled as parallelization, but this modification is only limited in the fitness component. The GA logic is not paralleled. For the paralleled version fitness function, its computational core is same as the algorithm 12. We just assign several threads to run it.

Algorithm 12 FitnessFunction(*solutionset*)

Initialization:
Let $solution_i$ be one solution in *solutionset*;
Iteration:
1: **repeat**
2: Decode $solution_i$ to generate $BTPS$ and VLT;
3: PlacementStrategy($BTPS$,VLT); //Algorithm 3;
4: SpaceUtilityRate($solution_i$); //Algorithm 10;
5: Revert($solution_i$,l,r) //Algorithm 11;
6: **until** $solution_i$ to be the last solution in *solutionset*
Output: SUR

3.2.3 Selection Operator

After the evaluation, each individual in the population will be assigned with a fitness value, SUR. In this function, the candidate with the qualified fitness value will be selected to be the parent of the next generation. The different selection methods have already been introduced in the section 2.2.2 of the chapter 2. To implement the selection component in this project, we chose the tournament selection. Again, the running efficiency of the sorting method in this selection is an issue. After each evaluation process, the individual in the population has to be sorted in a specified sequence in order to be selected as the best candidates. Suppose that a population with 2000 individuals has to be sorted into the decreasing order, and each reorder operation in C++ is $10^{-6}s$. If we use the bubble sort, one sort process will consume almost $4s$ on average. Otherwise, if the concept of quick sort is used, as seen in the algorithm 11, one sorting process will stay with $0.006s$ on average. It is a big improvement. Thus, the sorting method in this selection component also referred the idea of quicksort.

3.2.4 Crossover operator

The parametrized uniform crossover [66] was applied into this algorithm. During each crossover process, two parent individuals will randomly be selected from the TOP of population and the old part of population. The construction of the population will be illustrated in the later section. After selecting two parent individuals, at each gene we toss a biased coin to select which parent will contribute

the allele. An example of the crossover operator is presented by the Fig.3.4.

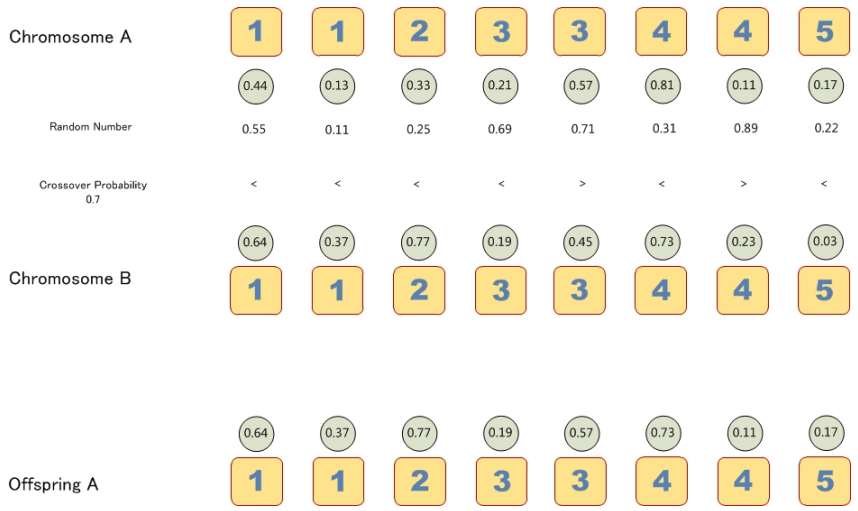


FIGURE 3.4: Example of Crossover Operator

It is assumed that a coin toss of head selects the gene from the first parent, a tail chooses the gene from the second parent, and that the probability of tossing a heads is 0.7, or say the crossover probability is 0.7.
Recall in the begin of this chapter, the random key is considered as a tag for each packing item, every operator in this algorithm just work on a random key vector, rather than on the position pattern. It is quite clear shown in this example that the crossover operator just manipulate the random key between two parent individuals, but the offspring that is generated by the crossover operator can be produced a different packing pattern by sorting these random values. This function can be implemented by means of the algorithm 13.

3.2.5 Mutation Operator

The mutation operator in this project does not work as the traditional mutation operator. Instead of mutating the gene one by one, this mutation operator simply creates a set of individuals completely at random, the process of which is same as the initialization component done. Although the operation procedure of the mutation operator used in here is changed, the purpose of using this mutation

Algorithm 13 CrossoverFunction($parent1$,$parent2$)

Initialization:
Let $gene1_i$ be the i gene of the $parent1$;
Let $gene2_i$ be the i gene of the $parent2$;
Let $offspring1, offspring2$ be the new individual;
Let $U(0,1)$ be a uniform random number generator; // Use to toss the coin to select.
Set Crossover Rate(pc) be the 0.7.

Iteration:
1: **repeat**
2: **if** $U(0,1) \leq pc$ **then**
3: Save the random key of $gene1_i$ into $offspring1$;
4: Save the random key of $gene2_i$ into $offspring2$;
5: **else**
6: Save the random key of $gene1_i$ into $offspring2$;
7: Save the random key of$gene2_i$ into $offspring1$;
8: **end if**
9: **until** $gene_i$ be the last element of $parent1$

Output: $offspring1, offspring2$

operator does not change. It is also used to hold the diversity of the population in order to prevent premature convergence. During each evolutionary process, The BOT of population will be filled by using this mutation operator (seen as the algorithm 9).

3.3 Modification

In order to make the original version BRKGA_CLP an improvement, in this project, we try to modify certain component of it. According to a series of attempts of modification, we found that by using the different initial position pattern of packing items, and a different mutation operator, there is a slightly improvement in some benchmarks.

3.3.1 Initialization Component Modification

As mentioned in the section 3.2.1, every chromosome maintains a specified pattern, which indicates the initial position of the packing item. Each random key is just a tag for each position. To build up this pattern by using the original rule, each items

within the same group will be selected repeatedly until its quantity constraint was met, and the position order of the each item will be given based on its box type number. For example, there are three box types, namely BT1, BT2, and BT3. Each box types has different quantity, suppose they are 3, 2 and 3. The original position pattern is arranged as depicted by the Fig.3.5.

FIGURE 3.5: Original Position Pattern

After this pattern has been constructed, every chromosome will be built up by assigning two random numbers to each position of this pattern. The two random numbers are BTPS and VLT, which have already been introduced at beginning of this chapter.

By observing the above example, it is found that when the problem become more weakly heterogeneous, the position pattern are divided by several segments, each of which are occupied by a groups of the items with the same type. Suppose each box type in the above example has more than forty items, each segment will include four same items. Thus, using the random key to tag the position in this position patter is useless because the BTPS is generated by sorting these random key, but the random key is assigned by following this position pattern, the BTPS is always same. by using this initial position pattern, the optimizer will easily be guided into a local optima.

To overcome this defect, the item in each groups has to be randomly selected to construct the position pattern. To achieve this aim, the concept of roulette-wheel selection was introduced. Based on the quantity of each box type, the slot size will be calculated by using the formula: $p_i = q_i / \sum_{i=1}^{n} q_i$, where the q_i is the quantity of the box type i. It should be noted that after each time of selecting a box, each slot size has to be calculated again because the total number of the packing items has been changed. By using this method the position pattern in the above example can be modified as shown in the Fig.3.6.

FIGURE 3.6: New Position Pattern

The function to generate this pattern is called only once in the initialization component. This pattern will be hold by every individual and never be changed. To implement this function the algorithm 14 can be used.

Algorithm 14 PositionPatternGenerator($Problem$)

Initialization:

Let $totalnub$ be the total number of the items in $Problem$;
Let box_i be the box type i in $Problem$; Let qb_i be the set to save the quantity of each box type
Let $typenub$ be the total number of box type in $Problem$;
Let pro_i be the proportion of the box type i in total number items;
Let $culpro$ be the cumulative probability; Let the $positionpatter$ be a set to save the selected box.
Let $U(0,1)$ be an uniform random number generator
Set $PID = 1$, which is a position label;

Iteration:

```
1: //main loop for position pattern construction;
2: repeat
3:    for i = 1 to typenub do
4:       pro_i = qb_i/tnub;
5:       culpro+ = pro_i;
6:       if U(0,1) <= culpro then
7:          Label box_i with VID;
8:          Save box_i into positionpatter;
9:          qb_i = qb_i − 1;
10:         totalnub = totalnub − 1;
11:         Break this loop;
12:      end if
13:   end for
14:   VID = VID + 1;
15: until all boxes are saved into positionpatter
```

Output: *positionpatter*

After generating the position pattern, the individual in the population set will be initialized by assigning two random keys to each position of this pattern.

3.3.2 Mutation Operator Modification

In this project, the original mutation operator in the BRKGA was replaced by the mutation operator of the Evolutionary Strategies. The initial reason of using the ES mutation operator is that we desire to introduce the idea of the solutions similarity into our mutation operator in order to control the range of the mutation. Thus, the ES mutation operator is a suitable candidate. However, through the practical implementation, the results of this modification were unsatisfactory because we cannot find a suitable mathematical model to add the similarity factor into the ES mutation operator. In spite of the fact that to add similarity factor was unsuccessful, there was an unexpectedly result on the weakly heterogeneous benchmark when the original mutation operator was only replaced by the standard mutation operator of the ES.

This ES mutation operator is an uncorrelated mutation with n step size. In this type mutation, each dimension of individual can own an independent mutation step size, so during the mutation process, all dimensions can be mutated differently. This is because that the fitness landscape can have a different slope in one direction than in another direction. This mutation mechanism can be described as follows:

$$\sigma_i' = \sigma_i \cdot e^{\tau \cdot N(0,1) + \tau' \cdot N_i(0,1)} \tag{3.1}$$

$$x_i' = x_i + \sigma_i' \cdot N_i(0,1) \tag{3.2}$$

where τ and τ' are named global and local learning rate respectively. The common base mutation $e^{\tau \cdot N(0,1)}$ allows an overall change of the mutability, while the $e^{\tau' \cdot N_i(0,1)}$ offers the flexibility to use different mutation strategies in different directions.

In the standard ES mutation operator, the new individual is generated by adding a normal distributed random perturbation, to the old value of the vector (as seen algorithm 15). The corresponding standard deviation are also subject to the evolution process and are thus multiplied in each step by a logarithmic distributed random number. Schwefel [67] termed the resulting process as self-adaptive because the adaptation of the mutation parameters is governed by an evolutionary

process itself. The general idea behind self-adaptation is that, if a set of different individuals is generated, each with a different probability distribution, the individual with the best object variables is also likely to be the one with the best probability distribution that lead to the generation of these objective variables. Thus the parameters of this probability distribution are also inherited by the offspring individual.

Algorithm 15 ES_Mutation

Initialization:

Input: $r_1, \cdots, r_n, \sigma_i, \cdots, \sigma_n$;

Iteration:

1: $N_c \leftarrow N(0,1)$ //Generate and store a normally distributed random number;
2: $\tau \leftarrow \frac{1}{\sqrt{2n}}; \tau' \leftarrow \frac{1}{\sqrt{2\sqrt{n}}}$ //Initialize global and local learning rate;
3: **for all** $i \in \{1, \cdots, n\}$ **do**
4: $\quad \sigma_i' \leftarrow \sigma_i \exp(\tau N_c + \tau' N(0,1))$
5: $\quad r_i' \leftarrow r_i + \sigma' N(0,1)$;
6: **end for**
7: //Interval boundary treatment, it is between 0 and 1 in this project;
8: **for all** $i \in \{1, \cdots, n\}$ **do**
9: $\quad r_i' \leftarrow T[r_i^{\min}, r_i^{\max}](r_i')$;
10: **end for**

Output: $r_1', \cdots, r_n', \sigma', \cdots, \sigma_n'$;

3.4 Summary

The relevant components in the BRKGA were explained through the above sections. The whole running procedure follows the standard genetic algorithm. The differences with the original BRKGA were that a new position pattern generator was added into the initialization component, while the original mutation operator was replaced by the mutation operator of the ES. Before closed this chapter, the construction of the population has to be mentioned that there are three blocks built up the whole population, namely the TOP, the MID, and the BOT (seen as the Fig.3.7).

According to the above figure, it can be seen that the entire population will be sorted after evaluation, the best individuals will be arranged into the TOP block. Since the elitist strategy is used in this GA, the individuals in the TOP block will

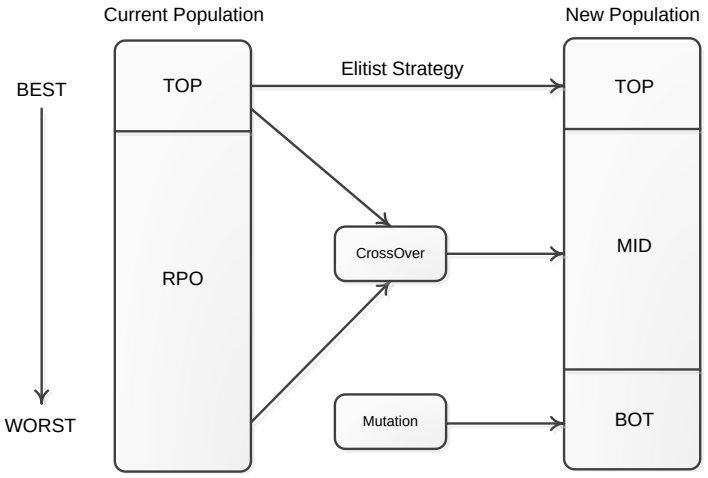


FIGURE 3.7: Population Structure

be copied to the next generation directly without any change. The MID block is filled by new offspring in the crossover process. The crossover operator selects two parents, one is always selected from the TOP block, and the other is selected from the rest of the population, they will mate and generate two new individuals. The individuals in the BOT block are generated by the mutation operator.

Chapter 4

Experiments

In order to evaluate the performance of our algorithm, a set of experiments were implemented based on the famous benchmarks, proposed by Bischoff E and Ratcliff M [4]. This new version BRKGA will run on several small experiments at first in order to find a relativity good parameter setting, and observe the performance of the new components.

In the end of this section, an overall test will be arranged to our algorithm, and the results were produced by this algorithm were compared with several optimizers presented in table 4.3, they are the most efficient in the literatures to date.

In the section 4.1, The detail of the test benchmark will be introduced. The section 4.2 will present the different parameter configuration. For the section 4.4, the final experiment result will be shown.

In order to observe a practical packing situation of the solution, the relevant location data of each layer is organized into a table, and a 3D figure is used to illustrate the packing pattern. The reader can find these tables and figures in the Appendix A.

4.1 Benchmark Description

Currently, There are seven benchmarks used in this project (BR1-BR7), which totally include 700 problem instances. This test set was proposed by Bischoff E and Ratcliff M. They are widely used to evaluate the container loading optimization algorithm in many literatures. The benchmarks can either be downloaded in http://paginas.fe.up.pt/~esicup/tiki-index.php or generated by a problem generator which can be implemented by following the algorithm 16.

Algorithm 16 BenchmarkGenerator

Initialization:

Set Cargo target volume T_c;
Set Number of different Box type n;
Set Lower and upper limits on box dimensions $a_j, b_j, j \in [1,3]$;
Set Box stability limit L;
Set Seed number s; //using in the random number generator;
Set box type index $i = 1$;
Initialized random number generator and discard first 10 random number;
Let $InstanceNub$ be the total number of the instance for each problem.

Iteration:

// the main loop used to generate different instances for each problem;
for j=1 to $InstanceNub$ **do**
  // Generate n number type box;
  for i=1 to n **do**
    repeat
      Generate 3 random number $r_j, j \in [1,3]$
      Determine box dimensions using:
      $d_{ij} = a_j + \lfloor r_j \times (b_j - a_j + 1) \rfloor, j \in [1,3]$;
    until (All $[d_{ij} / \min(d_{ij})] < L$)
    Initialize box quantity m_i for box type $i : m_i = 1$;
    Let the box volume $v_i = \prod_{j=1}^{3} d_{ij}$;
  end for
  //Give the quantity to each box type i;
  repeat
    Calculate cargo volume:$C = \sum_{i=1}^{n} m_i v_i$;
    Generate the next random number r
    and set box type indicator $k = 1 + \lfloor r \times n \rfloor$;
    $m_k = m_k + 1$;
  until $(T_c > C + v_k)$
end for

Output: Problem set;

In order to generate this seven benchmarks, the parameters in the algorithm 16 should be set by following the table 4.1, in which the T_c is defined as the container volume, which can be represented by $L \times W \times H$. In addition, there are 100

instances generated for each problem. Thus, there are seven hundreds instances included in the test set. The random number generator in this algorithm is a special variant, suggested in [68], of the multiplicative congruential method and, as pointed out in [69], can be implemented in most programming languages on almost any computer. The seed number s is used in this random number generator in order to reproduce the individual problems without generating the complete set. To calculate the seed s we use the formula $s = 2502505 + 100(p-1)$, p is the problem number.

TABLE 4.1: Parameters Setting of Problem Generator

PID	Container			L	Box Type				Instance Number
	TC				Dimension			Quantity	
	L	W	H		(a_1, b_1)	(a_2, b_2)	(a_3, b_3)		
1	586	233	220	2	$(30, 120)$	$(25, 100)$	$(20, 80)$	3	100
2								5	
3								8	
4								10	
5								12	
6								15	
7								20	

4.2 Optimizer Parameter Configuration

To find the best parameter setting for the GA is also an optimization problem. Normally, the Evolutionary Strategy (ES) can be used to tackle the parameter tuning optimization problem, this is another story in the optimization problem. In this work, we just arrange a set experiments by tuning the parameter in our algorithm in order to find a relatively good parameter settings. The corresponding parameter settings in this algorithm are presented in the table 4.2.

TABLE 4.2: The Algorithm Parameter Setting

Prarameter	Interval
POS	20
TOP	0.1 - 0.7
BOT	0.1 - 0.5
PC	0.1 - 0.5
ROTA	YES - NO

The POS represents the population size, the PC is the crossover rate, and the ROTA is a flag that indicates whether the box will be allowed to rotate in this algorithm. According to the above table, we can see that there are totally 192 possible configurations for each problem, and each problem also has 100 instances. By following the above setting, there are an enormous number of experiments needed. Thus, in order to reduce the running time of total experiments, two computers with Intel 8-core CPU running the Win 8 operating system were used to carry out these experiments. For each computer, three independent runs of the algorithm were made.

4.3 Different Components Setting

In this project, the original initialization component and mutation operator of the BRKGA were modified. For the original initialization, the position pattern generator was modified by a new method, which can generate a quite random position pattern. For the mutation operator, the original one will be replaced by a standard ES type mutation operator. In order to observe the effect of these modifications, the experiments in this section will be built up by running the BRKGA with the new components and old version. The results generated are compared between the new version and the old version. The details of this part experiments will be seen in the later section.

4.4 Experiment Implementation

The average utility rate of 100 instances of each problem that are produced by this optimizer will be compared with eight efficient methods in the table 4.3. Because we cannot obtain the code of these algorithms, the results that are produced by these approaches are directly collected from its corresponding literatures.

TABLE 4.3: The Approaches in Comparison

Algorithm	Source		Method
GH_L	Lim	[25]	Greedy Heuristic
PTS_BO	Bortfeldt	[17]	Parallel Tabu Search
PST_M	Mack	[18]	Parallel SA/TS
GRA(200000)_P	Parreno	[23]	GRASP
GRA(5000)_P	Parreno	[23]	GRASP
VNS_P	Parreno	[70]	VNS
TRS_FB	Fanslau and Bortfeldt	[8]	TRS
HBS_HH	He and Huang	[71]	Heuristic Beam Search

4.4.1 The Components Modification

The experiments in this section are divided into two subsection. The tests in the first subsections were organized by implementing the new position pattern generator into the BRKGA while other components were set with the same type, and its result will be compared with the old version. In experiments of the second subsection, the ES type mutation operator are used to replace the original one. All mentioned experiments were implemented based on the seven benchmarks. The parameters fixed in this algorithm were the TOP with 0.15, the BOT with 0.15, the POS with 500, and the number of generation with 500.

New Position Pattern Generator

According to analysing the result shown in the Fig.4.1, it can be sees that the results produced by using the new pattern generator on benchmarks 1, 2, 3, and 4 were better than using the original one, while this four benchmarks are the

weakly heterogeneous problem. Thus, it is considered that a more random position pattern could improve the final results on the weakly heterogeneous benchmarks. However, for the strongly heterogeneous benchmarks, this pattern generator can not produce a satisfactory result.

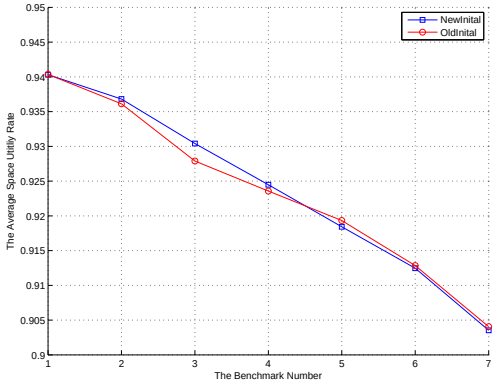


FIGURE 4.1: New Position Pattern vs Old Position Pattern

ES Type Mutation operator

In this part experiments, we fixed the initialization component with using the new pattern generator, while the two different mutation operators were implemented respectively on all benchmarks. The final comparison results were shown in the Fig.4.2.

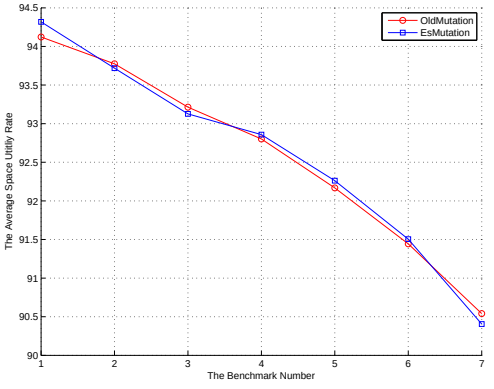


FIGURE 4.2: New Mutation Operator vs Old Mutation Operator

According to above experiment results, it is consider that the ES type mutation operator provided a positive effect for the BRKGA because the algorithm with the ES type mutation operator can generate a good result on most benchmarks, especially in the benchmark 1. Thus, in experiments of later sections, The original mutation operator will be replaced by the ES type mutation operator.

4.4.2 Parameter Tuning

In order to find a relatively good parameter setting before running the overall tests, we started with implementing several small experiments, which were organized by tuning the parameters in the table 4.1, but the population size was fixed with 500 in order to reduce the experiment time. For each parameter portfolio in each benchmark, the algorithm runs five times, and each time had 500 iterations.

Crossover Rate Tuning

The first group experiments were produced by applying nine different crossover rates from 0.1 to 0.7 with 0.05 interval. Here the population size was set to be 500, the TOP block was 0.15%, the BOT was 0.15%, and the ROTA was allowed.

The two groups experiments were included in the Fig.4.3, which summarised the result that were produced by implementing this algorithm with different crossover rates and benchmarks. In order to find a suitable crossover rate, There were nine experiments illustrated in the Fig.4.3(a), which were generated by applying this algorithm with tuning crossover rate from 0.1 to 0.7, to process the benchmark 5. According to the Fig.4.3(a), there was a slightly increasing trend from C01 to C65. However, after that the results dropped below the average level. Thus, the crossover rate was set to be 0.65% and applied it to process all benchmarks in order to observe the overall performance on all benchmarks.

To implement the tests in the Fig.4.3(b), two other crossover settings which are close to 0.65% were introduced, in order to see the comparison result by running on all benchmarks. Although there was not any significant difference between the results that were generated by using these crossover rates, the performance of the algorithm with the

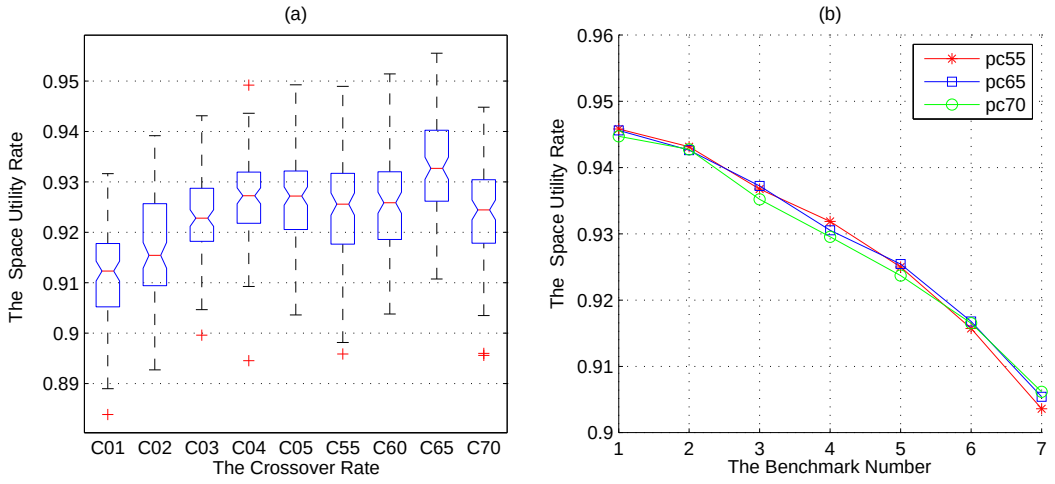


FIGURE 4.3: Different Crossover Comparison

(a): The labels on x axis represent the different crossover rates. For example, C01 means that the crossover rate is 0.1. Here, all experiments run on benchmarks 5.

(b): The labels on x axis represent the different benchmark. Here, all experiments are implemented by applying different benchmarks.

crossover rate 0.65% on all benchmarks was slightly better than others. For the experiments in the Fig.4.3(b), there is one more point, and I should touch on, that to handle the weakly heterogeneous benchmarks a smaller crossover rate would be appreciated, as mentioned before the heterogeneous strength of benchmarks was raised by increasing the number of box type in the problem. As an example of on first five benchmarks, the results of crossover rate with 0.7 are almost always lower than two others because the heterogeneous strength of rest two benchmarks is significantly stronger than the first five benchmarks. For a number of box type in first five benchmarks, they are from 3 to 12. For the rest of two benchmark (6,7), the number of box type are 15 and 20 respectively. Otherwise, for the strongly heterogeneous problems, a higher crossover rate can be a good choice.

BOT Size Tuning

The individuals in the BOT block are only generated by mutation operator. Thus, the size of the BOT set can be considered as a control parameter of this genetic algorithm. By tuning this parameter could build up the experiments in this section. Here, the range of this parameter was set from 0.1 to 0.5 while the rest of the parameters were fixed based on the result of previous experiments.

This part of experiments were divided into two steps. The first step of experiments was to implement this algorithm on a fixed benchmark with tuning the size of the BOT in order to find a reasonable setting. Here, the benchmark five was still used for generating the tests in the first step because the best results of different tuning steps were compared in later of this section. For the second step, the algorithm with three fixed BOT sizes, one of them was the best setting which was found through the first step experiments, was applied on all benchmarks in order to observe the overall performance of this best setting on seven benchmarks.

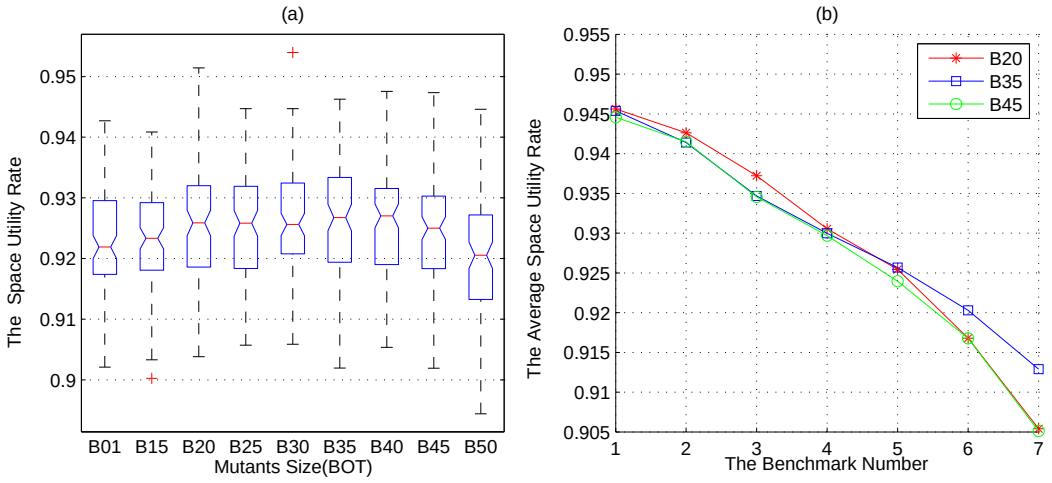


FIGURE 4.4: Different BOT Size Comparison

(a): The labels on x axis represent the different BOT Size. For example, B01 means that the mutants size is 10% of whole population size. Here, all experiments run on benchmark 5.

(b): The labels on x axis represent the different benchmarks. Here, all experiments with different BOT size are implemented by applying different benchmarks.

The results of the two steps tests were demonstrated by the Fig.4.4. For the Fig.4.4(a), it illustrated the nine experiments which were arranged according to applying this algorithm on benchmark 5 by tuning the size of the BOT from 0.1 to 0.5. Based on observing the median and the main body of the box in the box-plot, there was a slightly increasing from B01 to B35 while the point B35 can be considered as a divide, the data after it illustrated a decreasing trend. Thus, the size of the BOT with be set to 0.35 was a reasonable setting.

For the second step in the Fig.4.4, three different BOT settings, 0.2, 0.35 and 0.45, were used into this algorithm to process the seven benchmarks. The reason to choose the size with 0.2 was that the performance of this setting in the benchmark 5 can be seen as a second favourable setting, it could have a big opportunity to produce a good result on different problems,. The size of the BOT with 0.45 was selected because this setting is close to the best one, and the solution that were generated by this setting can cover a more wide range. The result, shown in the Fig.4.4(b), was the average space utility rate of each benchmark. We can see that the setting with 0.35 had reasonably good performance on overall views, and mainly it can produce a better result than two others when the benchmark become more strongly heterogeneous such as the problems from 5 to 7. For the weakly heterogeneous problem such as 2 and 3, it worked unsatisfactorily.

Therefore, depending on the result analysis, it was found that setting with low value is rather appreciated by more weakly heterogeneous problem. However, for more strongly heterogeneous problem, the setting does not have such discipline.

The Size of Elites Set Tuning (TOP)

To organize the experiments in this portion the algorithm was applied on the benchmark 5 by setting up with two fixed parameters and a variable one. The two fixed parameters were the crossover rate and the size of the BOT, which were set with 0.65 and 0.35 respectively by referred the previous experiments. The size of the TOP was selected as the variable one, which was tuned from 10% of the whole population size to 50%. After finding the best TOP size in the first phase, the setting was used by this algorithm to process the seven benchmarks. The final results were used to make a comparison with the results that were generated by applying two different settings within the same type parameter.

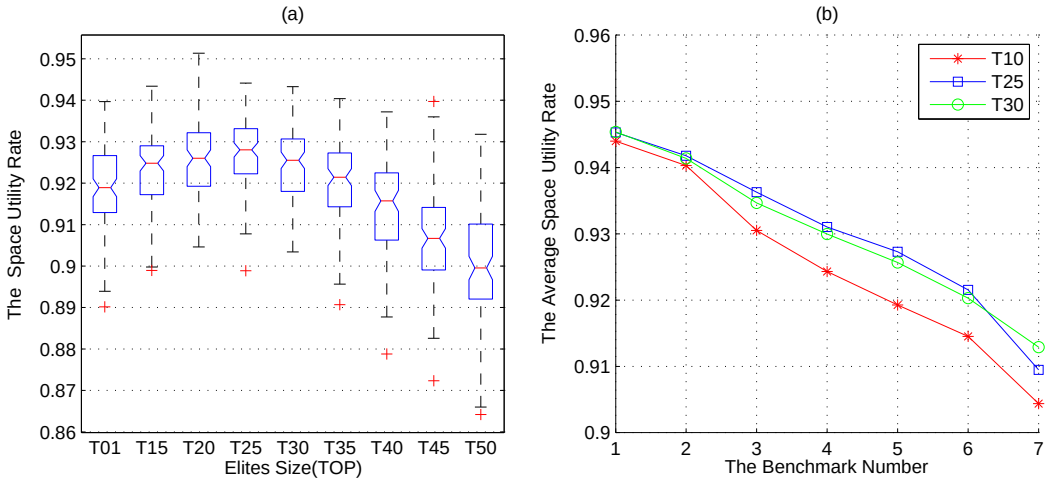


FIGURE 4.5: Different Elites Size Comparison

(a): The labels on x axis represent the different Elites Size. For example, the T01 means that the Elites Size is 10% of whole population size. Here, all experiments run on benchmark 5.

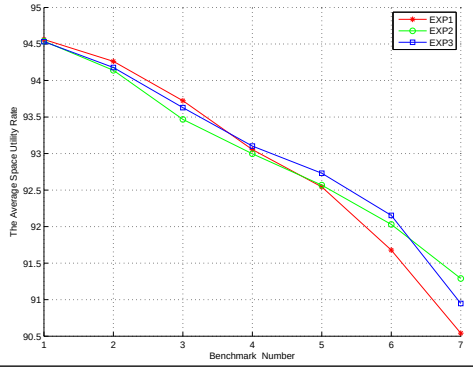
(b): The labels on x axis represent the different benchmarks. Here, all experiments with different Elites size are implemented by applying different benchmarks.

The Fig.4.5 manifested the result of the two aspects experiments. In the Fig.4.5(a), the overall trend of the distribution of relevant result was slightly similar with a parabola. Although the best solution was obtained by implementing the setting with 20%, the reasonable peak of this parabola should be a setting with 25%. Since the average space utility of the benchmark five that was generated by the setting with 25% was the biggest, the TOP size with 25% of population size should be a reasonable setting. For the Fig.4.5(b), it illustrated three groups of results that were generated by three different TOP sizes, which were $10\%, 25\%$ and 30%, on seven benchmarks. It was clearly seen that the result produced by the setting with 25% was almost flowing above the two others on every benchmarks, except the benchmark 7, so to fix the parameter with this value could be a desirable choice. According to the result of two other settings, it can also be proved that the settings that are higher or lower than 25% cannot give a valuable help to our algorithm on these benchmarks.

Parameter Overall Comparison

According to the above experiments, the best parameter settings had already been found as seen in the below table. In order make an overall view about the performance of this setting, TOP with 25%, the mutants size with 35%, and the crossover rate with 65%, the best results in each step were illustrated by the Fig.4.6. Through this figure, we can see that the parameter settings that were found by the experiments 4.5 had more stability than two others. Thus, we will implement this parameter setting in the later final experiments to generate the overall final result.

FIGURE 4.6: Three Tuning Steps Comparison



The Best Parameter Setting

Parameter	Interval
POS	20
TOP	0.25
BOT	0.35
PC	0.65
ROTA	YES

4.4.3 The Overall Comparison Experiments

To implement the experiments in this section, this algorithm will be set with the parameter found in above sections to solve the seven benchmarks. The overall results in the table 4.4 were the average space utility rate of 100 instances of each problem, which are referred from [1].

TABLE 4.4: The Final Comparison Result

BID	GH	PTS	PST	GRA(a)	GRA(b)	VNS	TRS	HBS	BRKGA	NBRKGA
1	88.70	93.52	93.70	93.85	93.27	94.93	95.05	87.57	95.28	**95.38**
2	88.17	93.77	94.30	94.22	93.38	95.19	95.43	89.12	95.90	95.69
3	87.52	93.58	94.54	94.25	93.39	94.99	95.47	90.32	96.13	95.61
4	87.58	93.05	94.27	94.09	93.16	94.71	95.18	90.57	96.01	95.38
5	87.30	92.34	93.83	93.87	92.89	94.33	95.00	90.78	95.84	95.03
6	86.86	91.72	93.34	93.52	92.62	94.04	94.79	90.91	95.72	94.79
7	87.15	90.55	92.50	92.94	91.86	93.53	94.24	90.88	95.29	94.26

According to observing the results in the table 4.4, the new version of BRKGA shows the best result on BID 1. For rest of test cases, without considering the old version BRKGA, NBRKGA defected all other algorithms on overall view. However, for the benchmark 6, result of our optimizer was same as the TRS.

TABLE 4.5: Average Utility Rate (B1-B7) Result

	GH	HBS	PTS	PST	GRA(a)	VNS	TRS	NBRKGA	BRKGA
Average (1-7)	87.61	90.02	92.65	93.78	93.82	94.53	95.02	95.16	95.74

In order to analyse the performance of each algorithm in all problems, the overall average utility rate were calculated, and the algorithms will be ranked into an increasing order by using this result, as shown in table 4.5. Two worst results were produced by the GH and the HBS, they are much less than rest algorithms. by analysing the two algorithms, it is found that the GH and the HBS packs each item independently. In GH, boxes are placed on the wall of the container first so as to construct a base, after that other boxes will be placed on top of these boxes. For the HBS, in each packing iteration, a packing item will be put into the packing space first in order to detect a packing layer, after that the algorithm will based on this layer to pack the different type items. This kind of packing heuristic will produce too much piecemeal spaces which cannot use to pack the item, so this two algorithms cannot produce a good result. to compare the results of this two algorithms, HBS had a better result than the GH because the HBS will pack the item into layer first, which can reduce the number of piecemeal spaces. For the rest of algorithms, they will put same items into a same block or layer first, which won not generate too much piecemeal spaces. Thus, it is considered that the block and layer concept are more appropriate to handle the 3 dimensional container loading problem.

However, according to observing the PTS, it it found that to put the same items into a same block can reduce the dimension of the search space, but it will reduce the diversity of the search space as well. Thus, if the algorithm only depend on the block or layer concept, the result will easily be trapped into a local optima. From the PST to BRKGA, all of them have a mechanism to add the diversity into the search space. For instance, the PST maintain multiple search paths simultaneously. and the GRA will add a randomization strategy to obtain different solution at each iteration. For the VNS and the TRS, they can combine two blocks or layers to produce new type blocks so that to add the diversity. In fact, in these algorithms, they try to find a way to balance this two

aspect, low dimensional of search space and diversity of search space. The more suitable balance point it finds, The better result it obtains.

Therefore, for the BRKGA, it is considered that there are three reasons to make it a better result in all benchmarks. The first one is that its representation includes two parts, namely BTPS and VLT. The BTPS can be seen as a packing plan. The item in here can be treated independently, which can offer the diversity of search space. For the VLT, the block and layer concept can be operated in this part, by which the dimension of search space will be reduced. The second one is that the GA is a population based algorithm. It can maintain multiple search paths simultaneously, which means that this kind of algorithms can have bigger chance to find a better result. The final one is the random key. The normal representation of solution for the combinatorial problem is the permutation. the algorithms in the table 4.4 used this kind of representation, except the BRKGA and NBRKGA. To use the permutation in the 3 dimensional container loading problem will involve an issue, in which the items in the permutation can not be packed completely. In this case, the representation can not provide a complete order relationship to the algorithm. To use the random key representation can avoid this defect. As mentioned, the items in the BRKGA and the NBRKGA will ranged into a position pattern in the initialization component. This pattern never change. The random key just tag the each position in this pattern. All the operations in this kind of algorithms are just on the random keys. During the evolutionary process, the algorithm will find the relationship between the random key and corresponding position while the balance point between low dimensional and diversity of search space will be found as well. In this case, even though certain items can not be packed, the representation still can provide the useful information.

Chapter 5

Conclusion

In this project, the single container loading problem was tackled, in which several rectangular boxes of different sizes are to be loaded into a single rectangular container. The approach used to solve this problem was to refer the BRKGA-CLP, which is used a novel multi-population biased random-key genetic algorithm. The representation of it involved two parts, which are BTPS and VLT respectively. The BRKGA is used to evolve the order (BTPS) in which the box type are loaded into the container and the corresponding type of layer (VLT) used in the placement procedure. To test this optimizer, the complete set of benchmark problems of Bischoff and Ratcliff [4] was used. The benchmark set is made up of 700 instance which range from weakly to strongly heterogeneous cargo. The final results of this algorithm were compared with 9 other solution techniques.

According to implement the original version BRKGA, it was found that the position pattern generated by the original initialization component easily guide the solution into a local optima when the benchmark is a more weakly heterogeneous problem. To tackle this problem, a new position pattern generator based on the idea of the roulette-wheel was used to replace the original one in order to generate a new random position pattern.

In addition, the mutation operator in the old version BRKGA just offer the diversity of the population. For the neighbourhood search, it is just based on the crossover operator. However, in the container loading problem, the crossover operator usually will miss to search some quite close neighbourhood solutions, especially in the random key representation, which means that we need to find a mutation operator, the mutation

range of which can be controlled. In order to achieve this purpose, the idea of population similarity was introduced into this algorithm. Initially, the original mutation operator was replaced by mutation operator of the ES because the mutation range of this mutation operator can be controlled by its mutation step size. Thus, we intend to use the similarity factor to guide the mutation step size. When the population similarity is small, the mutation step size will be reduced in order to make the algorithm more opportunity to search in the neighbourhood. Otherwise, the step size will be increased.

However, according to the practical experiments, it is found that to add the similarity factor into the algorithm could not reach the expected aim because a suitable mathematical model can not be found to implement the similarity factor into this mutation operator. Although the similarity factor cannot offer a good effect to this algorithm, by only implementing the standard ES mutation operator, this algorithm can produce an unexpected good result on first benchmark. Thus, it is considered that a sophisticated mutation operator can provide a positive effect for the BRKGA. For the bad results in final experiment, it is considered that the standard mutation operator cannot offer enough population diversity, which causes the algorithm to trap into a local optima.

Therefore, Future work should involve to develop a suitable mathematical model for adding the similarity into the mutation operator in order to make this algorithm enable to perceive the similarity change so that the mutation step size can be modified into a suitable range.

Appendix A

The Practical Packing Solution and Pattern

The best results produced in the experiments section for each benchmarks are organized within a table, and the final packing pattern will be demonstrated by a 3D figure. There are two figure for each table. The life one shows the result in current table, and the right one is the worst case in same benchmark, which is used to do the comparison with the best one.

Best Packing Pattern for Benchmark 5 in Experiment 4.3												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 103)$	$(0, 267)$	$(0, 219)$	6	9	89	103	73	1	3	3	
2	$(103, 201)$	$(0, 580)$	$(0, 80)$	8	10	116	98	40	1	5	2	
3	$(103, 195)$	$(0, 570)$	$(80, 174)$	2	12	95	92	47	1	6	2	
4	$(195, 233)$	$(0, 585)$	$(80, 124)$	5	5	117	38	44	1	5	1	
5	$(0, 55)$	$(267, 559)$	$(0, 162)$	7	12	73	55	54	1	4	3	
6	$(201, 233)$	$(0, 500)$	$(0, 80)$	11	10	100	32	40	1	5	2	
7	$(55, 102)$	$(267, 457)$	$(0, 184)$	2	4	95	47	92	1	2	2	
8	$(195, 227)$	$(0, 400)$	$(124, 164)$	11	4	100	32	40	1	4	1	
9	$(103, 147)$	$(0, 561)$	$(174, 220)$	1	11	51	44	46	1	11	1	
10	$(147, 191)$	$(0, 51)$	$(174, 220)$	1	1	51	44	46	1	1	1	
11	$(0, 55)$	$(267, 340)$	$(162, 216)$	7	1	73	55	54	1	1	1	
12	$(195, 233)$	$(0, 530)$	$(164, 194)$	10	5	106	38	30	1	5	1	
13	$(147, 193)$	$(51, 102)$	$(174, 218)$	1	1	51	46	44	1	1	1	
14	$(147, 193)$	$(102, 459)$	$(174, 218)$	1	7	51	46	44	1	7	1	
15	$(55, 103)$	$(267, 329)$	$(184, 206)$	9	1	62	48	22	1	1	1	
16	$(0, 53)$	$(340, 514)$	$(162, 188)$	12	2	87	53	26	1	2	1	
17	$(55, 92)$	$(329, 584)$	$(184, 216)$	4	5	51	37	32	1	5	1	
18	$(55, 99)$	$(457, 574)$	$(0, 114)$	5	3	117	44	38	1	1	3	
19	$(0, 38)$	$(340, 446)$	$(188, 218)$	10	1	106	38	30	1	1	1	
20	$(195, 233)$	$(400, 506)$	$(124, 154)$	10	1	106	38	30	1	1	1	
21	$(0, 53)$	$(446, 533)$	$(188, 214)$	12	1	87	53	26	1	1	1	
22	$(193, 232)$	$(0, 580)$	$(194, 217)$	3	5	116	39	23	1	5	1	
23	$(55, 103)$	$(457, 519)$	$(114, 180)$	9	3	62	48	22	1	1	3	
24	$(147, 186)$	$(459, 575)$	$(174, 197)$	3	1	116	39	23	1	1	1	
25	$(201, 227)$	$(500, 587)$	$(0, 53)$	12	1	87	26	53	1	1	1	
26	$(0, 53)$	$(559, 585)$	$(0, 87)$	12	1	26	53	87	1	1	1	
27	$(147, 186)$	$(459, 575)$	$(197, 220)$	3	1	116	39	23	1	1	1	
28	$(0, 48)$	$(514, 576)$	$(162, 184)$	9	1	62	48	22	1	1	1	
29	$(195, 227)$	$(506, 557)$	$(124, 161)$	4	1	51	32	37	1	1	1	
30	$(55, 99)$	$(519, 581)$	$(114, 162)$	9	2	62	22	48	2	1	1	
31	$(48, 96)$	$(519, 581)$	$(162, 184)$	9	1	62	48	22	1	1	1	
32	$(195, 232)$	$(530, 581)$	$(161, 193)$	4	1	51	37	32	1	1	1	
33	$(0, 37)$	$(533, 584)$	$(184, 216)$	4	1	51	37	32	1	1	1	
34	$(0, 48)$	$(559, 581)$	$(87, 149)$	9	1	22	48	62	1	1	1	
TPB	TUPB	UPBT					Instance ID			SUR		
126	9	(3,5), (10,1), (12,3)					98			0.951438		

TABLE A.1: The Best Result in Experiment 4.3

- **BT**: Box type ID.
- **BN**: Packed box number.
- **TPB**: Total number of Packed box.
- **TUPB**: Total number of unpacked box.
- **UPBT**: Unpacked box type and corresponding number.
- **SUR**: Space Utility Rate.

Best Packing Pattern for Benchmark 5 in Experiment 4.4												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 182)$	$(0, 246)$	$(0, 104)$	4	12	41	91	104	2	6	1	
2	$(0, 210)$	$(0, 567)$	$(104, 178)$	2	14	81	105	74	2	7	1	
3	$(182, 232)$	$(0, 532)$	$(0, 73)$	3	7	76	50	73	1	7	1	
4	$(0, 72)$	$(0, 456)$	$(178, 219)$	11	12	38	72	41	1	12	1	
5	$(72, 226)$	$(0, 456)$	$(178, 220)$	1	8	114	77	42	2	4	1	
6	$(182, 232)$	$(0, 576)$	$(73, 102)$	10	8	72	50	29	1	8	1	
7	$(0, 178)$	$(246, 421)$	$(0, 54)$	7	14	25	89	54	2	7	1	
8	$(0, 73)$	$(246, 550)$	$(54, 104)$	3	4	76	73	50	1	4	1	
9	$(73, 146)$	$(246, 474)$	$(54, 104)$	3	3	76	73	50	1	3	1	
10	$(0, 89)$	$(421, 446)$	$(0, 54)$	7	1	25	89	54	1	1	1	
11	$(210, 230)$	$(0, 93)$	$(102, 169)$	6	1	93	20	67	1	1	1	
12	$(210, 233)$	$(93, 543)$	$(102, 172)$	5	6	75	23	70	1	6	1	
13	$(146, 178)$	$(246, 582)$	$(54, 102)$	12	8	84	32	24	1	4	2	
14	$(89, 156)$	$(421, 514)$	$(0, 40)$	6	2	93	67	20	1	1	2	
15	$(0, 91)$	$(456, 560)$	$(178, 219)$	4	1	104	91	41	1	1	1	
16	$(0, 89)$	$(446, 586)$	$(0, 46)$	8	5	28	89	46	1	5	1	
17	$(156, 180)$	$(421, 505)$	$(0, 32)$	12	1	84	24	32	1	1	1	
18	$(91, 225)$	$(456, 549)$	$(178, 198)$	6	2	93	67	20	2	1	1	
19	$(73, 143)$	$(474, 549)$	$(46, 92)$	5	2	75	70	23	1	1	2	
20	$(89, 164)$	$(514, 584)$	$(0, 46)$	5	2	70	75	23	1	1	2	
21	$(91, 158)$	$(456, 549)$	$(198, 218)$	6	1	93	67	20	1	1	1	
22	$(158, 225)$	$(456, 549)$	$(198, 218)$	6	1	93	67	20	1	1	1	
23	$(178, 228)$	$(532, 561)$	$(0, 72)$	10	1	29	50	72	1	1	1	
24	$(0, 93)$	$(567, 587)$	$(46, 180)$	6	2	20	93	67	1	1	2	
25	$(93, 227)$	$(567, 587)$	$(102, 195)$	6	2	20	67	93	2	1	1	
26	$(93, 129)$	$(549, 584)$	$(46, 94)$	9	2	35	36	24	1	1	2	
27	$(91, 231)$	$(549, 585)$	$(195, 219)$	9	4	36	35	24	4	1	1	
28	$(0, 84)$	$(560, 584)$	$(180, 212)$	12	1	24	84	32	1	1	1	
29	$(164, 200)$	$(561, 585)$	$(0, 35)$	9	1	24	36	35	1	1	1	
30	$(178, 213)$	$(561, 585)$	$(35, 71)$	9	1	24	35	36	1	1	1	
TPB	TUPB	UPBT					Instance ID			SUR		
129	9	(5,4), (8,4), (10,1)					83			0.957639		

TABLE A.2: The Best Result in Experiment 4.4

- **BT**: Box type ID.
- **BN**: Packed box number.
- **TPB**: Total number of Packed box.
- **TUPB**: Total number of unpacked box.
- **UPBT**: Unpacked box type and corresponding number.
- **SUR**: Space Utility Rate.

Best Packing Pattern for Benchmark 5 in Experiment 4.5												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 83)$	$(0, 560)$	$(0, 70)$	5	14	40	83	70	1	14	1	
2	$(83, 233)$	$(0, 520)$	$(0, 74)$	4	15	104	50	74	3	5	1	
3	$(0, 69)$	$(0, 540)$	$(70, 217)$	7	15	108	69	49	1	5	3	
4	$(69, 109)$	$(0, 83)$	$(74, 214)$	5	2	83	40	70	1	1	2	
5	$(109, 233)$	$(0, 77)$	$(74, 218)$	9	12	77	62	24	2	1	6	
6	$(109, 181)$	$(77, 545)$	$(74, 220)$	8	12	78	72	73	1	6	2	
7	$(181, 230)$	$(77, 509)$	$(74, 143)$	7	4	108	49	69	1	4	1	
8	$(181, 233)$	$(77, 448)$	$(143, 219)$	6	14	53	52	38	1	7	2	
9	$(69, 109)$	$(83, 557)$	$(74, 124)$	12	6	79	40	50	1	6	1	
10	$(69, 107)$	$(83, 564)$	$(124, 173)$	11	13	37	38	49	1	13	1	
11	$(181, 231)$	$(448, 527)$	$(143, 183)$	12	1	79	50	40	1	1	1	
12	$(181, 231)$	$(448, 503)$	$(183, 209)$	1	1	55	50	26	1	1	1	
13	$(181, 233)$	$(509, 562)$	$(74, 112)$	6	1	53	52	38	1	1	1	
14	$(83, 162)$	$(520, 560)$	$(0, 50)$	12	1	40	79	50	1	1	1	
15	$(181, 231)$	$(503, 558)$	$(183, 209)$	1	1	55	50	26	1	1	1	
16	$(181, 231)$	$(509, 564)$	$(112, 138)$	1	1	55	50	26	1	1	1	
17	$(69, 106)$	$(83, 563)$	$(173, 215)$	2	10	48	37	42	1	10	1	
18	$(162, 210)$	$(520, 562)$	$(0, 74)$	2	2	42	48	37	1	1	2	
19	$(181, 228)$	$(527, 579)$	$(138, 172)$	3	2	26	47	34	1	2	1	
20	$(0, 42)$	$(540, 577)$	$(70, 166)$	2	2	37	42	48	1	1	2	
21	$(83, 157)$	$(520, 554)$	$(50, 71)$	10	1	34	74	21	1	1	1	
22	$(109, 159)$	$(545, 585)$	$(71, 150)$	12	1	40	50	79	1	1	1	
23	$(107, 181)$	$(545, 566)$	$(150, 218)$	10	2	21	74	34	1	1	2	
24	$(42, 68)$	$(540, 587)$	$(70, 206)$	3	4	47	26	34	1	1	4	
25	$(0, 148)$	$(560, 581)$	$(0, 68)$	10	4	21	74	34	2	1	2	
26	$(0, 38)$	$(540, 577)$	$(166, 215)$	11	1	37	38	49	1	1	1	
27	$(68, 102)$	$(557, 583)$	$(70, 117)$	3	1	26	34	47	1	1	1	
28	$(148, 222)$	$(562, 583)$	$(0, 68)$	10	2	21	74	34	1	1	2	
29	$(181, 215)$	$(558, 584)$	$(172, 219)$	3	1	26	34	47	1	1	1	
30	$(159, 233)$	$(562, 583)$	$(68, 102)$	10	1	21	74	34	1	1	1	
31	$(159, 233)$	$(564, 585)$	$(102, 136)$	10	1	21	74	34	1	1	1	
32	$(68, 102)$	$(564, 585)$	$(117, 191)$	10	1	21	34	74	1	1	1	
33	$(102, 176)$	$(566, 587)$	$(150, 184)$	10	1	21	74	34	1	1	1	
34	$(102, 176)$	$(566, 587)$	$(184, 218)$	10	1	21	74	34	1	1	1	
TPB	TUPB	UPBT					Instance ID			SUR		
151	9	(1,4), (3,1), (8,1),(11,1),(12,1)					64			0.964155		

- **BT**: Box type ID.
- **BN**: Packed box number.
- **TPB**: Total number of Packed box.
- **TUPB**: Total number of unpacked box.
- **UPBT**: Unpacked box type and corresponding number.
- **SUR**: Space Utility Rate.

TABLE A.3: The Best Result in Experiment 4.4

Best Packing Pattern for Benchmark 1 in Overall Experiment												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 231)$	$(0, 378)$	$(0, 37)$	2	66	63	21	37	11	6	1	
2	$(0, 208)$	$(0, 25)$	$(37, 217)$	1	20	25	52	36	4	1	5	
3	$(0, 100)$	$(25, 565)$	$(37, 220)$	3	45	36	100	61	1	15	3	
4	$(100, 136)$	$(25, 525)$	$(37, 220)$	3	15	100	36	61	1	5	3	
5	$(136, 172)$	$(25, 525)$	$(37, 220)$	3	15	100	36	61	1	5	3	
6	$(172, 208)$	$(25, 525)$	$(37, 220)$	3	15	100	36	61	1	5	3	
7	$(208, 233)$	$(0, 572)$	$(37, 73)$	1	11	52	25	36	1	11	1	
8	$(0, 61)$	$(378, 578)$	$(0, 36)$	3	2	100	61	36	1	2	1	
9	$(208, 233)$	$(0, 572)$	$(73, 217)$	1	44	52	25	36	1	11	4	
10	$(61, 211)$	$(378, 586)$	$(0, 36)$	1	24	52	25	36	6	4	1	
11	$(100, 204)$	$(525, 550)$	$(36, 216)$	1	10	25	52	36	2	1	5	
12	$(100, 200)$	$(550, 586)$	$(36, 219)$	3	3	36	100	61	1	1	3	
13	$(211, 232)$	$(378, 567)$	$(0, 37)$	2	3	63	21	37	1	3	1	
14	$(0, 63)$	$(565, 586)$	$(36, 73)$	2	1	21	63	37	1	1	1	
15	$(63, 100)$	$(565, 586)$	$(36, 99)$	2	1	21	37	63	1	1	1	
16	$(0, 37)$	$(565, 586)$	$(73, 136)$	2	1	21	37	63	1	1	1	
17	$(37, 100)$	$(565, 586)$	$(99, 210)$	2	3	21	63	37	1	1	3	
18	$(0, 37)$	$(565, 586)$	$(136, 199)$	2	1	21	37	63	1	1	1	
TPB	TUPB	UPBT					Instance ID			SUR		
280	4	(1,3), (3,1)					12			0.986502		

TABLE A.4: The Best Result in Benchmark One

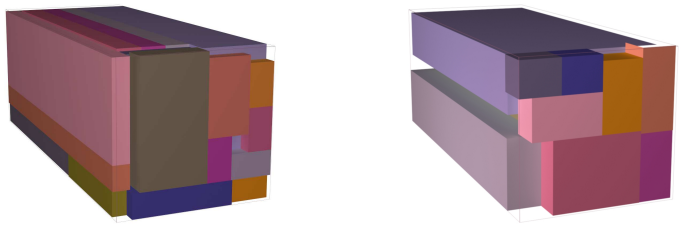


FIGURE A.1: The Benchmark One

Best Packing Pattern for Benchmark 2 in Overall Experiment												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 72)$	$(0, 304)$	$(0, 220)$	4	40	76	72	22	1	4	10	
2	$(72, 128)$	$(0, 366)$	$(0, 216)$	3	36	61	56	36	1	6	6	
3	$(128, 200)$	$(0, 31)$	$(0, 220)$	2	4	31	72	55	1	1	4	
4	$(128, 202)$	$(31, 580)$	$(0, 198)$	5	27	61	74	66	1	9	3	
5	$(0, 66)$	$(304, 526)$	$(0, 183)$	5	9	74	66	61	1	3	3	
6	$(202, 233)$	$(0, 72)$	$(0, 220)$	2	4	72	31	55	1	1	4	
7	$(202, 233)$	$(72, 576)$	$(0, 55)$	2	7	72	31	55	1	7	1	
8	$(202, 233)$	$(72, 576)$	$(55, 110)$	2	7	72	31	55	1	7	1	
9	$(202, 233)$	$(72, 576)$	$(110, 220)$	2	14	72	31	55	1	7	2	
10	$(128, 202)$	$(31, 581)$	$(198, 220)$	1	20	55	37	22	2	10	1	
11	$(0, 66)$	$(304, 579)$	$(183, 220)$	1	15	55	22	37	3	5	1	
12	$(66, 128)$	$(366, 438)$	$(0, 220)$	2	8	72	31	55	2	1	4	
13	$(66, 127)$	$(438, 586)$	$(0, 198)$	5	6	74	61	66	1	2	3	
14	$(0, 66)$	$(526, 587)$	$(0, 148)$	5	2	61	66	74	1	1	2	
15	$(66, 121)$	$(438, 586)$	$(198, 220)$	1	4	37	55	22	1	4	1	
16	$(0, 55)$	$(526, 563)$	$(148, 170)$	1	1	37	55	22	1	1	1	
TPB	TUPB	UPBT					Instance ID		SUR			
204	5	(1,2), (2,1), (3,1), (4,1)					84		0.981847			

TABLE A.5: The Best Result in Benchmark Two

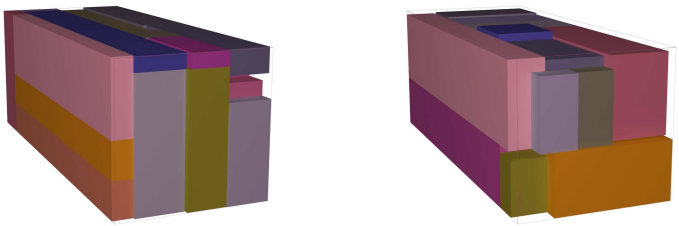


FIGURE A.2: The Benchmark Two

Best Packing Pattern for Benchmark 3 in Overall Experiment												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 195)$	$(0, 33)$	$(0, 219)$	4	15	33	39	73	5	1	3	
2	$(0, 54)$	$(33, 529)$	$(0, 177)$	7	24	62	54	59	1	8	3	
3	$(54, 143)$	$(33, 537)$	$(0, 180)$	1	36	56	89	45	1	9	4	
4	$(143, 196)$	$(33, 523)$	$(0, 220)$	8	25	98	53	44	1	5	5	
5	$(143, 196)$	$(523, 567)$	$(0, 98)$	8	1	44	53	98	1	1	1	
6	$(143, 197)$	$(523, 585)$	$(98, 157)$	7	1	62	54	59	1	1	1	
7	$(196, 233)$	$(0, 583)$	$(0, 34)$	2	11	53	37	34	1	11	1	
8	$(196, 233)$	$(0, 578)$	$(34, 87)$	2	17	34	37	53	1	17	1	
9	$(197, 232)$	$(0, 575)$	$(87, 141)$	3	23	25	35	54	1	23	1	
10	$(197, 230)$	$(0, 584)$	$(141, 219)$	4	16	73	33	39	1	8	2	
11	$(0, 29)$	$(33, 583)$	$(177, 219)$	6	11	50	29	42	1	11	1	
12	$(29, 105)$	$(33, 544)$	$(180, 219)$	5	14	73	38	39	2	7	1	
13	$(105, 143)$	$(33, 471)$	$(180, 219)$	5	6	73	38	39	1	6	1	
14	$(105, 138)$	$(471, 544)$	$(180, 219)$	4	1	73	33	39	1	1	1	
15	$(143, 197)$	$(523, 548)$	$(157, 192)$	3	1	25	54	35	1	1	1	
16	$(0, 50)$	$(529, 587)$	$(0, 168)$	6	8	29	50	42	1	2	4	
17	$(50, 137)$	$(537, 587)$	$(0, 168)$	6	12	50	29	42	3	1	4	
18	$(138, 192)$	$(523, 558)$	$(192, 217)$	3	1	35	54	25	1	1	1	
19	$(29, 137)$	$(544, 579)$	$(168, 218)$	3	4	35	54	25	2	1	2	
20	$(137, 191)$	$(548, 583)$	$(157, 182)$	3	1	35	54	25	1	1	1	
21	$(137, 191)$	$(558, 583)$	$(182, 217)$	3	1	25	54	35	1	1	1	
TPB	TUPB	UPBT				Instance ID			SUR			
229	3	(4,1),(8,2)				38			0.97522			

TABLE A.6: The Best Result in Benchmark Three

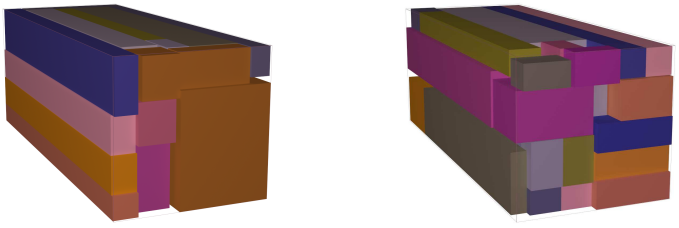


FIGURE A.3: The Benchmark Three

Best Packing Pattern for Benchmark 4 in Overall Experiment												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 192)$	$(0, 79)$	$(0, 220)$	10	24	79	32	55	6	1	4	
2	$(0, 68)$	$(79, 503)$	$(0, 189)$	5	24	53	68	63	1	8	3	
3	$(68, 158)$	$(79, 361)$	$(0, 220)$	4	15	94	90	44	1	3	5	
4	$(192, 223)$	$(0, 42)$	$(0, 217)$	3	7	42	31	31	1	1	7	
5	$(192, 233)$	$(42, 586)$	$(0, 180)$	1	32	34	41	90	1	16	2	
6	$(192, 233)$	$(42, 222)$	$(180, 214)$	1	2	90	41	34	1	2	1	
7	$(158, 189)$	$(79, 439)$	$(0, 192)$	7	15	72	31	64	1	5	3	
8	$(0, 64)$	$(79, 151)$	$(189, 220)$	7	1	72	64	31	1	1	1	
9	$(68, 124)$	$(361, 565)$	$(0, 150)$	8	12	34	56	75	1	6	2	
10	$(124, 158)$	$(361, 586)$	$(0, 168)$	8	9	75	34	56	1	3	3	
11	$(68, 124)$	$(361, 586)$	$(150, 218)$	8	6	75	56	34	1	3	2	
12	$(158, 192)$	$(439, 514)$	$(0, 168)$	8	3	75	34	56	1	1	3	
13	$(0, 68)$	$(151, 585)$	$(189, 220)$	9	14	62	34	31	2	7	1	
14	$(158, 190)$	$(79, 527)$	$(192, 220)$	2	7	64	32	28	1	7	1	
15	$(190, 232)$	$(222, 277)$	$(180, 210)$	6	2	55	21	30	2	1	1	
16	$(190, 232)$	$(277, 587)$	$(180, 211)$	3	10	31	42	31	1	10	1	
17	$(124, 158)$	$(361, 547)$	$(168, 199)$	9	3	62	34	31	1	3	1	
18	$(124, 154)$	$(361, 581)$	$(199, 220)$	6	4	55	30	21	1	4	1	
19	$(158, 188)$	$(439, 549)$	$(168, 189)$	6	2	55	30	21	1	2	1	
20	$(0, 64)$	$(503, 587)$	$(0, 96)$	2	9	28	64	32	1	3	3	
21	$(0, 68)$	$(503, 565)$	$(96, 189)$	9	6	62	34	31	2	1	3	
22	$(158, 188)$	$(514, 535)$	$(0, 165)$	6	3	21	30	55	1	1	3	
23	$(158, 192)$	$(535, 566)$	$(0, 62)$	9	1	31	34	62	1	1	1	
24	$(158, 192)$	$(535, 566)$	$(62, 124)$	9	1	31	34	62	1	1	1	
25	$(124, 186)$	$(549, 583)$	$(168, 199)$	9	1	34	62	31	1	1	1	
26	$(154, 184)$	$(527, 582)$	$(199, 220)$	6	1	55	30	21	1	1	1	
27	$(64, 94)$	$(565, 586)$	$(0, 110)$	6	2	21	30	55	1	1	2	
28	$(94, 124)$	$(565, 586)$	$(0, 110)$	6	2	21	30	55	1	1	2	
29	$(158, 189)$	$(535, 566)$	$(124, 166)$	3	1	31	31	42	1	1	1	
30	$(0, 55)$	$(565, 586)$	$(96, 186)$	6	3	21	55	30	1	1	3	
31	$(55, 110)$	$(565, 586)$	$(110, 140)$	6	1	21	55	30	1	1	1	
32	$(158, 188)$	$(566, 587)$	$(0, 165)$	6	3	21	30	55	1	1	3	
TPB	TUPB	UPBT					Instance ID			SUR		
226	7	(2,1),(3,1),(9,5)					55			0.975254		

TABLE A.7: The Best Result in Benchmark Four

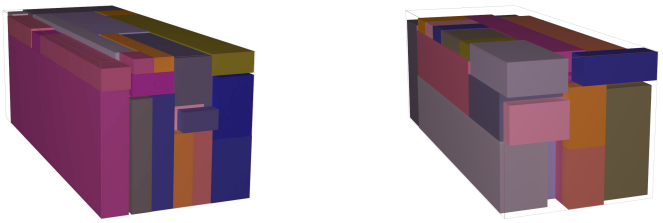


FIGURE A.4: The Benchmark Four

Best Packing Pattern for Benchmark 5 in Overall Experiment												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 43)$	$(0, 570)$	$(0, 123)$	12	15	114	43	41	1	5	3	
2	$(43, 123)$	$(0, 434)$	$(0, 218)$	10	14	62	80	109	1	7	2	
3	$(123, 232)$	$(0, 195)$	$(0, 219)$	8	9	65	109	73	1	3	3	
4	$(123, 231)$	$(195, 492)$	$(0, 98)$	7	11	27	108	98	1	11	1	
5	$(0, 40)$	$(0, 582)$	$(123, 215)$	9	6	97	40	92	1	6	1	
6	$(123, 197)$	$(195, 525)$	$(98, 220)$	5	10	66	74	61	1	5	2	
7	$(197, 233)$	$(195, 500)$	$(98, 210)$	3	10	61	36	56	1	5	2	
8	$(43, 119)$	$(434, 500)$	$(0, 216)$	4	9	22	76	72	1	3	3	
9	$(119, 228)$	$(492, 565)$	$(0, 65)$	8	1	73	109	65	1	1	1	
10	$(43, 116)$	$(500, 565)$	$(0, 109)$	8	1	65	73	109	1	1	1	
11	$(43, 117)$	$(500, 561)$	$(109, 175)$	5	1	61	74	66	1	1	1	
12	$(119, 229)$	$(492, 564)$	$(65, 96)$	2	2	72	55	31	2	1	1	
13	$(197, 229)$	$(500, 543)$	$(96, 213)$	6	3	43	32	39	1	1	3	
14	$(40, 118)$	$(500, 564)$	$(175, 218)$	6	4	32	39	43	2	2	1	
15	$(117, 174)$	$(525, 582)$	$(96, 141)$	11	1	57	57	45	1	1	1	
16	$(174, 196)$	$(525, 580)$	$(96, 207)$	1	3	55	22	37	1	1	3	
17	$(118, 173)$	$(525, 587)$	$(141, 213)$	2	2	31	55	72	1	2	1	
18	$(196, 233)$	$(543, 587)$	$(96, 206)$	1	4	22	37	55	1	2	2	
19	$(43, 117)$	$(561, 583)$	$(109, 164)$	1	2	22	37	55	2	1	1	
20	$(40, 77)$	$(564, 586)$	$(164, 219)$	1	1	22	37	55	1	1	1	
21	$(43, 119)$	$(565, 587)$	$(0, 72)$	4	1	22	76	72	1	1	1	
22	$(77, 114)$	$(564, 586)$	$(164, 219)$	1	1	22	37	55	1	1	1	
23	$(119, 230)$	$(565, 587)$	$(0, 55)$	1	3	22	37	55	3	1	1	
24	$(119, 174)$	$(565, 587)$	$(55, 92)$	1	1	22	55	37	1	1	1	
25	$(174, 229)$	$(565, 587)$	$(55, 92)$	1	1	22	55	37	1	1	1	
26	$(43, 98)$	$(565, 587)$	$(72, 109)$	1	1	22	55	37	1	1	1	
TPB	TUPB	UPBT					Instance ID			SUR		
117	9	(2,3),(6,3),(11,3)					84			0.965213		

TABLE A.8: The Best Result in Benchmark Five

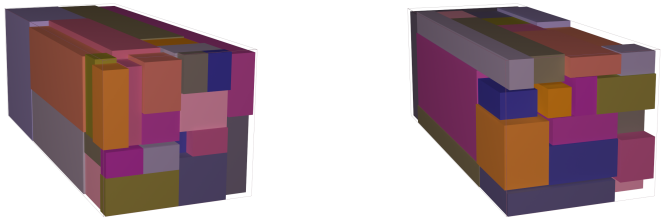
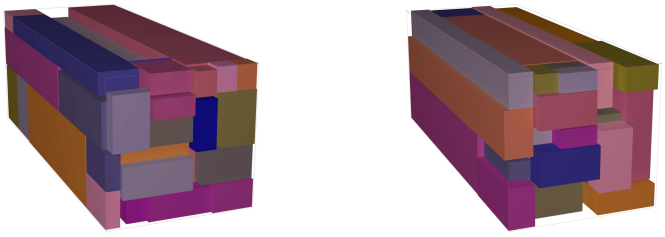


FIGURE A.5: The Benchmark Five

Best Packing Pattern for Benchmark 6 in Overall Experiment												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 101)$	$(0, 350)$	$(0, 180)$	15	14	50	101	90	1	7	2	
2	$(101, 209)$	$(0, 444)$	$(0, 47)$	7	12	37	108	47	1	12	1	
3	$(101, 209)$	$(0, 486)$	$(47, 113)$	2	9	54	108	66	1	9	1	
4	$(101, 170)$	$(0, 480)$	$(113, 200)$	14	12	120	69	29	1	4	3	
5	$(170, 232)$	$(0, 363)$	$(113, 187)$	3	11	33	62	74	1	11	1	
6	$(0, 99)$	$(0, 553)$	$(180, 219)$	9	7	79	99	39	1	7	1	
7	$(0, 99)$	$(350, 429)$	$(0, 117)$	9	3	79	99	39	1	1	3	
8	$(0, 64)$	$(350, 563)$	$(117, 179)$	5	6	71	64	31	1	3	2	
9	$(170, 231)$	$(0, 69)$	$(187, 220)$	4	1	69	61	33	1	1	1	
10	$(170, 231)$	$(69, 552)$	$(187, 220)$	4	7	69	61	33	1	7	1	
11	$(64, 97)$	$(350, 557)$	$(117, 178)$	4	3	69	33	61	1	3	1	
12	$(170, 211)$	$(363, 563)$	$(113, 185)$	8	4	100	41	36	1	2	2	
13	$(209, 231)$	$(0, 83)$	$(0, 110)$	10	2	83	22	55	1	1	2	
14	$(209, 231)$	$(83, 166)$	$(0, 112)$	12	2	83	22	56	1	1	2	
15	$(0, 66)$	$(429, 585)$	$(0, 44)$	13	3	52	66	44	1	3	1	
16	$(0, 99)$	$(429, 478)$	$(44, 116)$	11	2	49	99	36	1	1	2	
17	$(66, 210)$	$(444, 544)$	$(0, 41)$	8	4	100	36	41	4	1	1	
18	$(210, 232)$	$(166, 498)$	$(0, 112)$	12	8	83	22	56	1	4	2	
19	$(0, 99)$	$(478, 527)$	$(44, 116)$	11	2	49	99	36	1	1	2	
20	$(99, 168)$	$(0, 505)$	$(200, 220)$	1	5	101	69	20	1	5	1	
21	$(211, 231)$	$(363, 565)$	$(112, 181)$	1	2	101	20	69	1	2	1	
22	$(99, 168)$	$(480, 581)$	$(113, 153)$	1	2	101	69	20	1	1	2	
23	$(99, 160)$	$(480, 549)$	$(153, 186)$	4	1	69	61	33	1	1	1	
24	$(99, 207)$	$(486, 585)$	$(41, 90)$	11	3	99	36	49	3	1	1	
25	$(66, 166)$	$(544, 580)$	$(0, 41)$	8	1	36	100	41	1	1	1	
26	$(99, 163)$	$(505, 576)$	$(186, 217)$	5	1	71	64	31	1	1	1	
27	$(66, 98)$	$(527, 557)$	$(41, 117)$	6	2	30	32	38	1	1	2	
28	$(99, 209)$	$(486, 569)$	$(90, 112)$	10	2	83	55	22	2	1	1	
29	$(210, 232)$	$(498, 581)$	$(0, 56)$	12	1	83	22	56	1	1	1	
30	$(0, 64)$	$(527, 558)$	$(44, 115)$	5	1	31	64	71	1	1	1	
31	$(209, 231)$	$(498, 581)$	$(56, 111)$	10	1	83	22	55	1	1	1	
32	$(0, 83)$	$(558, 580)$	$(44, 99)$	10	1	22	83	55	1	1	1	
33	$(0, 55)$	$(563, 585)$	$(99, 182)$	10	1	22	55	83	1	1	1	
34	$(168, 223)$	$(565, 587)$	$(112, 195)$	10	1	22	55	83	1	1	1	
35	$(166, 198)$	$(544, 574)$	$(0, 38)$	6	1	30	32	38	1	1	1	
36	$(99, 163)$	$(549, 587)$	$(153, 183)$	6	2	38	32	30	2	1	1	
37	$(64, 96)$	$(553, 583)$	$(178, 216)$	6	1	30	32	38	1	1	1	
38	$(0, 32)$	$(553, 583)$	$(182, 220)$	6	1	30	32	38	1	1	1	
39	$(32, 64)$	$(553, 583)$	$(182, 220)$	6	1	30	32	38	1	1	1	
40	$(64, 96)$	$(557, 587)$	$(99, 175)$	6	2	30	32	38	1	1	2	
TPB	TUPB	UPBT					Instance ID			SUR		
145	10	(6,4),(11,1),(11,2),(13,3)					94			0.964554		

TABLE A.9: The Best Result in Benchmark Six



Best Packing Pattern for Benchmark 7 in Overall Experiment												
Layer ID	Position			BT	BN	Box Size			Layout			
	(x, x')	(y, y')	(z, z')			L_b	W_b	H_b	X	Y	Z	
1	$(0, 200)$	$(0, 116)$	$(0, 98)$	8	5	116	40	98	5	1	1	
2	$(0, 103)$	$(0, 511)$	$(98, 187)$	6	7	73	103	89	1	7	1	
3	$(103, 198)$	$(0, 564)$	$(98, 190)$	2	12	47	95	92	1	12	1	
4	$(0, 80)$	$(116, 491)$	$(0, 75)$	14	5	75	80	75	1	5	1	
5	$(200, 232)$	$(0, 500)$	$(0, 40)$	11	5	100	32	40	1	5	1	
6	$(200, 232)$	$(0, 555)$	$(40, 128)$	15	5	111	32	88	1	5	1	
7	$(198, 230)$	$(0, 111)$	$(128, 216)$	15	1	111	32	88	1	1	1	
8	$(0, 38)$	$(0, 530)$	$(187, 217)$	10	5	106	38	30	1	5	1	
9	$(80, 197)$	$(116, 420)$	$(0, 44)$	5	8	38	117	44	1	8	1	
10	$(80, 153)$	$(116, 556)$	$(44, 98)$	7	8	55	73	54	1	8	1	
11	$(153, 199)$	$(116, 512)$	$(44, 95)$	1	9	44	46	51	1	9	1	
12	$(38, 76)$	$(0, 424)$	$(187, 217)$	10	4	106	38	30	1	4	1	
13	$(76, 166)$	$(0, 564)$	$(190, 220)$	20	6	94	90	30	1	6	1	
14	$(198, 228)$	$(111, 205)$	$(128, 218)$	20	1	94	30	90	1	1	1	
15	$(166, 198)$	$(0, 560)$	$(190, 220)$	19	5	112	32	30	1	5	1	
16	$(198, 230)$	$(205, 317)$	$(128, 218)$	19	3	112	32	30	1	1	3	
17	$(198, 233)$	$(317, 562)$	$(128, 185)$	13	7	35	35	57	1	7	1	
18	$(80, 196)$	$(420, 529)$	$(0, 42)$	16	4	109	29	42	4	1	1	
19	$(0, 78)$	$(116, 580)$	$(75, 98)$	3	8	116	39	23	2	4	1	
20	$(0, 78)$	$(491, 578)$	$(0, 53)$	12	3	87	26	53	3	1	1	
21	$(0, 100)$	$(511, 551)$	$(98, 130)$	11	1	40	100	32	1	1	1	
22	$(0, 100)$	$(511, 551)$	$(130, 162)$	11	1	40	100	32	1	1	1	
23	$(38, 75)$	$(424, 577)$	$(187, 219)$	4	3	51	37	32	1	3	1	
24	$(78, 194)$	$(529, 552)$	$(0, 39)$	3	1	23	116	39	1	1	1	
25	$(0, 87)$	$(551, 577)$	$(98, 151)$	12	1	26	87	53	1	1	1	
26	$(78, 178)$	$(552, 584)$	$(0, 40)$	11	1	32	100	40	1	1	1	
27	$(196, 233)$	$(500, 551)$	$(0, 32)$	4	1	51	37	32	1	1	1	
28	$(153, 190)$	$(512, 544)$	$(42, 93)$	4	1	32	37	51	1	1	1	
29	$(178, 231)$	$(555, 581)$	$(0, 87)$	12	1	26	53	87	1	1	1	
30	$(0, 92)$	$(511, 576)$	$(162, 185)$	18	1	65	92	23	1	1	1	
31	$(0, 62)$	$(491, 587)$	$(53, 75)$	9	2	48	62	22	1	2	1	
32	$(198, 233)$	$(317, 562)$	$(185, 214)$	17	5	49	35	29	1	5	1	
33	$(0, 35)$	$(530, 587)$	$(185, 220)$	13	1	57	35	35	1	1	1	
34	$(78, 140)$	$(556, 578)$	$(40, 88)$	9	1	22	62	48	1	1	1	
35	$(140, 175)$	$(556, 585)$	$(40, 89)$	17	1	29	35	49	1	1	1	
36	$(87, 179)$	$(564, 587)$	$(89, 154)$	18	1	23	92	65	1	1	1	
37	$(92, 184)$	$(564, 587)$	$(154, 219)$	18	1	23	92	65	1	1	1	
38	$(179, 227)$	$(564, 586)$	$(87, 149)$	9	1	22	48	62	1	1	1	
39	$(184, 232)$	$(564, 586)$	$(149, 211)$	9	1	22	48	62	1	1	1	
TPB	TUPB	UPBT				Instance ID			SUR			
137	14	(12,3),(13,1),(16,2),(17,1),(18,1)				98			0.962306			

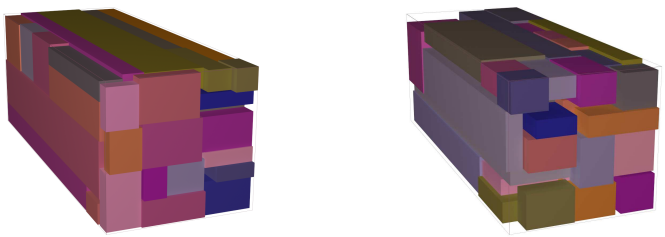


FIGURE A.6: The Benchmark Seven

Bibliography

[1] Jose Fernando Goncalves and Mauricio G.C. Resende. A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers & Operations Research*, 39(2):179 – 190, 2012. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/j.cor.2011.03.009. URL http://www.sciencedirect.com/science/article/pii/S0305054811000827.

[2] JC Bean. Genetics and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6:154–60, 1994.

[3] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2003. ISBN 9783540401841. URL http://books.google.nl/books?id=RRKo9xVFW_QC.

[4] Ratcliff MSW. Bischoff EE. Issues in the development of approaches to container loading. *Omega, International Journal of Management Science*, 23(3):377–90, 1995.

[5] Scheithauer G. Algorithm for the container loading problem. *In:Operational Research Proceedings*, pages 445–52, 1992.

[6] Schepers J Fekete SP. A new exact algorithm for general orthogonal d-dimensional knapsack problems. *Springer*, pages 144–56, 1997.

[7] Vigo D Martello S, Pisinger D. Three-dimensional bin packing problem. *Operations Research*, pages 256–67, 2000.

[8] Bortfeldt A Fanslau T. A tree search algorithm for solving the container loading problem. *INFORMS Journal on computing*, 22(2):222–35, 2010.

[9] Nee AYC Loh TH. A packing algorithm for hexahedral boxes. In *:Proceedings of the conference of industrial automation*, pages 115–26, 1992.

[10] Gehring H Bortfeldt A. A hybrid genetic algorithm for the container loading problem. *European Journal of Operational Research*, 131(1):143–61, 2001.

[11] Robinson DF. George AJ. A heuristic for packing boxes into a container. *computer and Operations Research*, 7(3):147–56, 1980.

[12] Pisinger D. Heuristics for the container loading problem. *European journal of operational Research*, 141:143–53, 2002.

[13] Bortfeldt A Gehring H. A genetic algorithm for solving the container loading problem. *International transactions in Operational Research*, pages 401–18, 1997.

[14] Ratcliff MSW Bischoff EE, Janetz F. Loading pallets with nonidentical items. *European Journal of Operational Research*, 84:681–92, 1995.

[15] Sommerweiss U Riehme J Terno J, Scheithauer G. An efficient approach for the multi-pallet loading problem. *European Journal of Operational Research*, 123(2): 372–81, 2000.

[16] Eley M. Solving container loading problems by block arrangement. *European Journal of operational Research*, 141(2):392–409, 2002.

[17] Mack D Bortfeldt A, Gehring H. A parallel tabu search algorithm for solving the container loading problem. *Parallel Computing*, 29(5):641–62, 2003.

[18] Gehring H Mack D Bortfeldt A. A parallel hybrid local search algorithm for the container loading problem. *international transactions in operational research*, 11: 511–33, 2004.

[19] Arenales M. Morabito R. An and/or-graph approach to the container loading problem. *international transactions in operational research*, 1(1):59–73, 1994.

[20] Bortfeldt A Gehring H. A parallel genetic algorithm for solving the container loading problem. *International transactions in Operational Research*, 9(4):497–511, 2002.

[21] Bischoff E. Three-dimensional packing of items with limited load bearing strength. *European Journal of operational Research*, 168:952–66, 2004.

[22] Oliverira JF Moura A. A grasp approach to the container -loading problem. *IEEE Intelligent Systems*, 20(4):50–7, 2005.

[23] Tamarit JM Oliveira JF Parreno F, Alvarez-Valdes R. A maximal-space algorithm for the container loading problem. *INFORMS Journal on computing*, 20(3):412–22, 2008.

[24] Hifi M. Approximate algorithm for the container loading problem. *International Transcations in operations Research*, 9:747–74, 2002.

[25] Wang Y Lim A, Rodrigues B. A multi-faced buildup algorithm for three-dimensional packing problems. *Omega*, 31(6):471–81, 2003.

[26] Bischoff EE Davies AP. Weight distribution considerations in container loading. *European Journal of operational research*, 114(3):509–27, 1999.

[27] Takahara S. Loading problem in multiple containers and pallets using strategic search method. In *Modeling Decision for artificial intelligence*, pages 448–56, Berlin, Heidelberg, 2005. Springer-verlag.

[28] Wakabayashi Y Miyazawa FK. Approximation aalgorithm for the orthogonal z-oriented three dimensional packing problem. *SIAM Journal on Computing*, 29(3): 1008–29, 1999.

[29] Xu B. Lai KK, Xue J. Container packing in a multi-customer delivering operation. *Computers and Industrial Engineering*, 35(1-2):323–6, 1998.

[30] Mack D Bortfeldt A. A heuristic for the three-dimensional strip packing problem. *European journal of operational Research*, 183(3):1267–79, 2007.

[31] He K. Huang W. A caving degree approach for the single container loading problem. *European Journal of Operational Research*, 196(1):93–101, 2009.

[32] Maculan N Silva JLC, Soma NY. A greedy search for the three-dimensional bin packing problem: the packing static stability case. *International Transcations in operations Research*, 123(2):141–53, 2003.

[33] Laporte G Martello S Gendreau M, Lori M. A tabu search algorithm for a routing and container loading problem. *Transportation Science*, 40(3):342–50, 2006.

[34] Shen QS. Chen CS, Lee SM. An analytical model for the container loading problem. *European Journal of Operational Rsearch*, 80(1):68–76, 1995.

[35] Eley M. A bottlenneck assignment approach to the multiple container loading problem. *OR Spectrum*, 25(1):45–60, 2003.

[36] Bischoff EE. Ratcliff MSW. Allowing for weight considerations in container loading. *OR Spectrum*, 20(1):65–71, 1998.

[37] Thom J. Hodgson. A combined approach to the pallet loading problem. *A I I E Transactions*, 14(3):175–182, 1982. doi: 10.1080/05695558208975057. URL http://www.tandfonline.com/doi/abs/10.1080/05695558208975057.

[38] Beasley JE. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33(1):49–64, 1985.

[39] Chistofides N Hadjiconstantinou E. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83(1):39–56, 1995.

[40] Martins GHA. *Packing in two and three dimensions*. PhD thesis, Naval Postgraduate school, Monterey, California, 2003.

[41] Beasley JE. A population heuristic for constrained two-dimensional non-guillotine cutting. *European Journal of Operational Research*, 156(3):601–27, 2004.

[42] Leonardo Junqueira, Reinaldo Morabito, and Denise Sato Yamashita. Three-dimensional container loading models with cargo stability and load bearing constraints. *Computers & Operations Research*, 39(1):74 – 85, 2012. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/j.cor.2010.07.017. URL http://www.sciencedirect.com/science/article/pii/S0305054810001486. Special Issue on Knapsack Problems and Applications.

[43] Whitlock C. Christofides N. An algorithm for two-dimensional cutting problems. *Operations Research*, 44(2):145–59, 1977.

[44] Chan JWM. Lai KK. Developing a simulated annealing algorithm for the cutting stock problem. *Computers and Industrial Engineering*, 32:115–27, 1997.

[45] Frank Hoffmeister and Thomas Bäck. Genetic algorithms and evolution strategies: Similarities and differences. In Hans-Paul Schwefel and Reinhard Männer, editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer*

Science, pages 455–469. Springer Berlin Heidelberg, 1991. ISBN 978-3-540-54148-6. doi: 10.1007/BFb0029787. URL http://dx.doi.org/10.1007/BFb0029787.

[46] T. Bäck. Evolutionary computation: A guided tour. In G. Paun et al., editor, *Current Trends in Theoretical Computer Science*, volume 1, pages 569–612. World Scientific, 2004.

[47] D. David K. De Jong and D. B. Fogel H.-P. Schwefel. A history of evolutionary computation. *Handbook of Evolutionary Computation*, pages 1–12, 1997.

[48] A. S. Fraser. Simulation of genetic systems by automatic digital computers. II. Effects of linkage on rates of advance under selection. *Australian Journal of Biological Science*, 10:492–499, 1957.

[49] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

[50] Aparna Vishwanath, Ramesh Vulavala, and Sapna U Prabhu. Task scheduling in homogeneous multiprocessor systems using evolutionary techniques.

[51] Otman Abdoun and Jaafar Abouchabaka. A comparative study of adaptive crossover operators for genetic algorithms to resolve the traveling salesman problem. *arXiv preprint arXiv:1203.3097*, 2012.

[52] David E. Goldberg and Kumara Sastry. A practical schema theorem for genetic algorithm design and tuning. In *Proceedings of the Genetic and Evolutionary Computation Conference, 328–335. (Also IlliGAL*, pages 328–335, 2001.

[53] Kumara Sastry, David Goldberg, and Graham Kendall. Genetic algorithms. In EdmundK. Burke and Graham Kendall, editors, *Search Methodologies*, pages 97–125. Springer US, 2005. ISBN 978-0-387-23460-1. doi: 10.1007/0-387-28356-0_4. URL http://dx.doi.org/10.1007/0-387-28356-0_4.

[54] Lawrence Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'85, pages 162–164, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc. ISBN 0-934613-02-8, 978-0-934-61302-6. URL http://dl.acm.org/citation.cfm?id=1625135.1625164.

[55] David E. Goldberg and Robert Lingle, Jr. Alleleslociand the traveling salesman problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 154–159, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc. ISBN 0-8058-0426-9. URL http://dl.acm.org/citation.cfm?id=645511.657095.

[56] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proc. of the International Conference on Genetic Algorithms and Their Applications*, pages 160–168, Pittsburgh, PA, 1985.

[57] J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In *Genetic Algorithms and Simulated Annealing*, London: Pitman, 1987.

[58] L. Darrell Whitley, Timothy Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 133–140, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. ISBN 1-55860-066-3. URL http://dl.acm.org/citation.cfm?id=645512.657238.

[59] C.Shaefer and S. Smith. The argot strategy ii: Combinatorial optimization. In *Thinking Machinesnical Report*, pages RL90–1, 1990.

[60] Jose Fernando Goncalves and MauricioG.C. Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525, 2011. ISSN 1381-1231. doi: 10.1007/s10732-010-9143-1. URL http://dx.doi.org/10.1007/s10732-010-9143-1.

[61] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, February 2000. ISSN 0040-1706. doi: 10.2307/1271432. URL http://dx.doi.org/10.2307/1271432.

[62] Felipe AC Viana. Things you wanted to know about the latin hypercube design and were afraid to ask.

[63] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Artificial Intelligence. Addison-Wesley, 1989. ISBN 9780201157673. URL http://books.google.nl/books?id=3_RQAAAAMAAJ.

[64] David Beasley, RR Martin, and DR Bull. An overview of genetic algorithms: Part 1. fundamentals. *University computing*, 15:58–58, 1993.

[65] Charles Darwin. On the origins of species by means of natural selection. *London: Murray*, 1859.

[66] William Spears. On the virtues of parameterized uniform crossover, 1991.

[67] Hans-Paul Paul Schwefel. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., New York, NY, USA, 1993. ISBN 0471571482.

[68] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, October 1988. ISSN 0001-0782. doi: 10.1145/63039.63042. URL http://doi.acm.org/10.1145/63039.63042.

[69] T. Gau and G. Wäscher. Cutgen1: A problem generator for the standard one-dimensional cutting stock problem. *European Journal of Operational Research*, 84(3):572 – 579, 1995. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/0377-2217(95)00023-J. URL http://www.sciencedirect.com/science/article/pii/037722179500023J. Cutting and Packing.

[70] F. Parreno, R. Alvarez-Valdes, J.F. Oliveira, and J.M. Tamarit. Neighborhood structures for the container loading problem: a vns implementation. *Journal of Heuristics*, 16(1):1–22, 2010. ISSN 1381-1231. doi: 10.1007/s10732-008-9081-3. URL http://dx.doi.org/10.1007/s10732-008-9081-3.

[71] Kun He and Wenqi Huang. Solving the single-container loading problem by a fast heuristic method. *Optimization Methods Software*, 25(2):263–277, April 2010. ISSN 1055-6788. doi: 10.1080/10556780902992761. URL http://dx.doi.org/10.1080/10556780902992761.