Title

Pedro Henrique Centenaro[*]
Supervisor: Luiz-Rafael Santos

Abstract. TBD

[*]Undergraduate Control, Automation and Computation Engineering student at UFSC/Campus Blumenau.

1 Introduction

TBD

2 Literature review

A vast literature exists on optimization problems where the goal is to fit a set of objects inside of one or more larger objects. Perhaps the simplest example is the knapsack problem (KP), where a set of items is defined, each with a corresponding volume and profit. A subset of items must be selected such that the sum of the profits is maximized, with the constraint that the sum of the volumes does not exceed the volume of a certain knapsack [18]. Mathematically, let $V \in \mathbb{R}$ be the volume of the knapsack, and $p, v \in \mathbb{R}^n$ be the vectors containing, respectively, the profit and volume associated with each item $i \in \{1, \dots, n\}$. It is possible to define a binary programming (BP) model to find out which items must be selected to solve the KP to optimality. Equations (1) to (3) are an example of such a problem. Equation (1) specifies the profit maximization problem, with $x \in \mathbb{R}^n$ being the vector of decision variables that describe whether an item is selected ($x_i = 1$) or not ($x_i = 0$). Equation (2) enforces that the selected items do not exceed the volume of the knapsack. Equation (3) specifies that vector x is binary.

$$\min\ p'x \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} v_i x_i \leq V, \tag{2}$$

$$x_i \in \{0, 1\}, \qquad \forall i \in \{1, \dots, n\}. \tag{3}$$

Many cargo loading problems cannot be solved using a KP model due to its simplicity. For example, many 2D palletization problems require that rectangular shapes be placed on a pallet, maximizing profits while avoiding overlaps between items and respecting the rectangular boundaries of the pallet. These constraints cannot be translated into a KP, and so the model has to be extended. See, for instance, two mixed-integer optimization models proposed by Kalvelagen [14] to solve this type of problem. Similarly, Chen, Lee, and Shen [5] proposed a model to load items into containers, which involves three-dimensional coordinates and even more complex enforcement of overlaps and relative positions. This is done by introducing alternative constraints to the models, which, combined with the hundreds of binary variables that are often necessary to solve the problems, makes such methods computationally unfeasible for many real applications.

The field of container loading problems (CLPs), like [5], is vast and full of research gaps [4]. According to a literature review by Bortfeldt and Wäscher [2], who studied 163 papers on CLPs from 1980 to 2011, a total of ten types of constraints can be identified in the literature:

1. **Weight limits:** Refer to how much weight the container can carry.
2. **Weight distribution:** Impose weight balances, to avoid applying too much force on a specific area of the container.
3. **Loading priorities:** Give more urgency to the placement of certain types of items, such as products that may expire soon.
4. **Orientation:** Constrain the possible rotations of certain items. For example, fragile objects may break if rotated sideways.
5. **Stacking:** Specify what items, and how many, may be placed on top of certain types of items.
6. **Complete shipment:** Force subsets of items to be placed in the same container.
7. **Allocation:** Similar to complete shipment, but also including the fact that certain groups of items may not be placed into the same container.
8. **Positioning:** Attempt, for example, to group similar items together in a container, for easier deployment.
9. **Stability:** Guarantee that items will not fall down (vertical stability) or violently collide with the container walls (horizontal stability).
10. **Pattern complexity:** Attempts to make item placement simple to understand, so that human operators can more efficiently load them.

The model presented by [5] considers only constraints 2 and 4. Yet, due to its exact nature, it is not applicable to problems with hundreds or thousands of items. For this reason, most research on CLPs has been done using inexact methods, which, according to Fanslau and Bortfeldt [9], can be divided into three categories:

(a) Conventional heuristics: Methods that were conceived specifically for the purpose of solving CLPs. Examples include the wall-building [10], layer-building [1] and block-building [8] heuristics. Typically, wall-building methods divide the container into cuboid layers, resembling walls, which are successivelly filled with items by following a set of rules. Layer-building methods are similar, except for the fact that layers are built on top of one another. Block-building methods do not necessarily fill sequential spaces; rather, they search for the best current cuboid spaces to place items in, and attempt to put items of the same type adjacent to each other (thus generating blocks of items).

(b) Metaheuristics: Methods that can be adapted to solve a wide variety of problems. Generally, metaheuristics owe their flexibility to the natural processes they are based on, which can be easily abstracted. Two common examples for solving CLPs are simulated annealing [7] and genetic algorithms [11]. In simulated annealing, a random solution is generated and evaluated. At each iteration of the method, a slight alteration is introduced to the solution (e.g. one item takes the spot of another during the placement process), and the result of this change is evaluated. If the new solution is better, it replaces the old solution. Otherwise, there is still a chance that the new solution replaces the older one, but it decreases over time. This helps the method avoid local optima. Genetic algorithms are a more robust alternative to simulated annealing: Multiple solutions are considered at the same time, and they may be combined and modified through each iteration. Genetic algorithms are discussed in more detail in Section 2.1

(c) Tree search: Methods that create trees of possible loadings in an attempt to filter promising solutions. These include the method by Fanslau and Bortfeldt [9], the improvement step of the block-building heuristic [8], and the tree search method by Liu et al. [17].

2.1 Genetic algorithms

Genetic algorithms (GAs) are brute force methods that utilize principle of natural selection to find good solutions to a problem. Initially proposed by Holland [13] in 1975, GAs have become one of the most widely used heuristics to solve a variety of optimization problems, thanks to its robustness when it comes to representing and solving problems.

The principles of natural selection were initially proposed by Darwin [6], who believed that the different characteristics of individuals in a population were the key to their adaptation in a given environment. The more varied a population is, the greater the chances that at least one type of individual will prove fit to survive in its environment. Hence, the fittest individuals are more likely to live to produce offspring, and so their characteristics carry on to the next generation.

With the subsequent discoveries of chromosomes and DNA, the biological mechanisms by which individuals' traits are perpetuated in a population became clearer. It also clarified why certain individuals can possess characteristics that are drastically different to those of their progenitors – a process known as mutation.

A chromosome is a DNA molecule that carries genetic information pertaining to an individual. It is divided into smaller sections called genes. The contents of each gene – called alleles – specify the different traits of a specimen. For instance, one gene may inform the eye color of an animal, while another may inform the blood type of the animal. Similarly, in a GA, chromosomes are vectorial representations of solutions to a given problem. Hence, they can be schematized like in Figure 1. In this case, the alleles are the different possible values stored in each gene. Often, alleles are binary (0 or 1) [15]; however, this is not always the case – for certain problems, it is better to let alleles assume any natural value [12] or any value in a real interval [11].

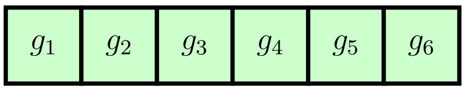


Figure 1: Illustration of a chromosome with 6 genes

It is important to define chromosomes in a way that they can be converted back into the solution that generated them. Once this is done, the next step in the GA is to define a fitness function. This function converts the chromosome back into a solution and evaluates its results. Often, the best the solution, the higher its fitness value. With the fitness function, solutions can be ranked, and thus it is possible to favor the selection of fitter chromosomes for reproduction [15, 20].

With chromosomial structure and fitness function defined, the GA follows a series of steps to generate, evaluate and evolve populations with each generation [20]:

Initialization

A population of n solutions (chromosomes) is generated, either randomly or through a specialized method. Though n is arbitrary, research indicates that if it is too small for a given problem, the GA may get stuck in a local optimum; if it is too large, it may consume more computational resources without improving the population. Different types of problems may require different population sizes, and human expertise is helpful to figure out the size to use [19].

Evaluation

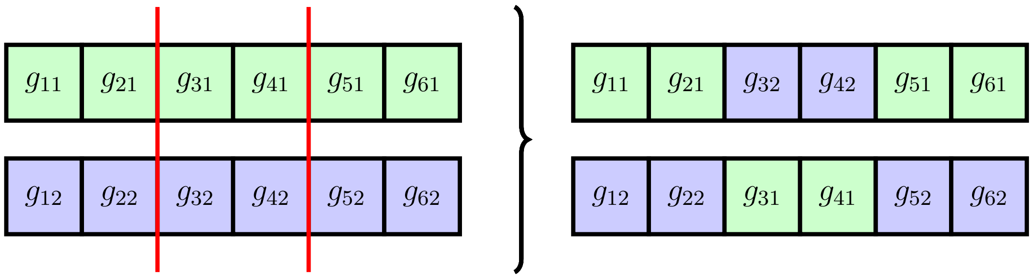
All solutions in the population are evaluated and ranked using the fitness function.

Selection

Chromosomes are selected in pairs to generate offspring in the recombination stage. One common example of selection procedure is *roulette selection*, which works as follows: Suppose a population with $n \geq 2$ individuals. To each solution i we associate a probability $p_i = q_i / \sum_{j=1}^{n} q_j$ of i being selected. We attribute interval $I_1 = [0, p_1]$ to solution 1, and each subsequent solutions k is associated with interval $I_k = \left(\sum_{j=1}^{k-1} p_j, \sum_{j=1}^{k} p_j\right]$. It results that $\cup_{j=1}^{n} I_j = [0, 1]$, so we generate a random value $r \in [0, 1]$ and pick chromosome k such that $r \in I_k$ [3, 20]. If a few solutions are much fitter than the rest, roulette selection may favor them too much and limit genetic variety early on. *Ranking selection* gives value 1 to the worst solution and n to the best solution, and then uses these values to generate the roulette. Thus, it can be used to more uniformly distribute the probabilities of selecting each solution. Another method that increases worse solutions' chances of selection is *tournament selection*, where a subset of the population is picked, and the best chromosome from the subset is selected.

Recombination

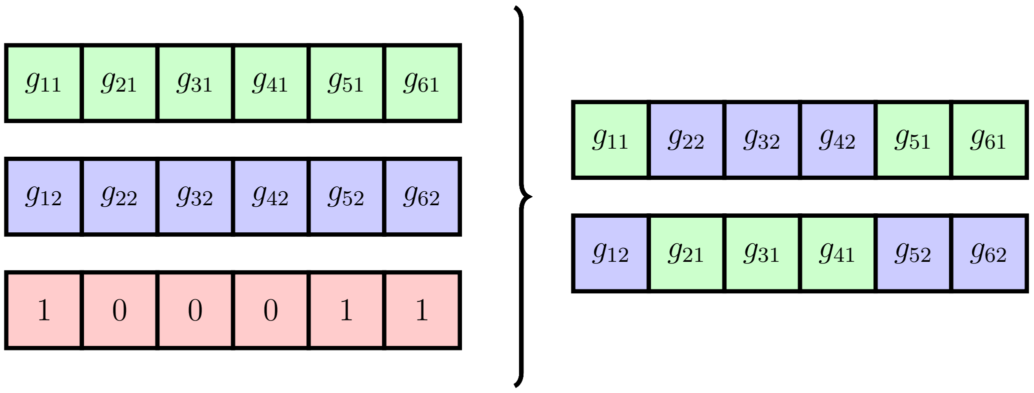
Two selected parents are combined to create new chromosomes. The simplest method is *one-point recombination*, and it works as follows: Suppose two parents with m genes each. A point k such that $1 < k < m$ is defined. The first offspring chromosome receives the first k genes from the first parent and the last $m - k$ genes from the second. The same happens to the second offspring chromosome, but with the order of parents switched. This procedure can be generalized into a *k-point recombination*, as illustrated in Figure 2, which shows a 2-point recombination producing two offspring.



g_{11} g_{21} g_{31} g_{41} g_{51} g_{61}
g_{12} g_{22} g_{32} g_{42} g_{52} g_{62}
g_{11} g_{21} g_{32} g_{42} g_{51} g_{61}
g_{12} g_{22} g_{31} g_{41} g_{52} g_{62}

Figure 2: 2-point recombination example.

Another common recombination method is *uniform crossover*. In this method, a binary vector (mask) is generated, with the same length as the parent chromosomes. One of the offspring takes genes from the first parent where the mask is equal to 0, or the second parent where the mask is equal to 1. The other offspring does the same, but it alternates the parent chromosomes. Figure 3 illustrates this procedure.



g_{11} g_{21} g_{31} g_{41} g_{51} g_{61}
g_{12} g_{22} g_{32} g_{42} g_{52} g_{62}
1 0 0 0 1 1
g_{11} g_{22} g_{32} g_{42} g_{51} g_{61}
g_{12} g_{21} g_{31} g_{41} g_{52} g_{62}

Figure 3: Uniform crossover using a mask.

k-point recombination consumes less resources than uniform crossover; however, the use of masks makes uniform crossover more oriented towards diversity in the next population [15]. Furthermore, the mask can be generated in a way that gives both parents equal odds of contributing genes, or it can favor the chromosome with the better fitness value [20].

Mutation

The solutions resulting from the recombination step may randomly have part of their genes changed. This is important to explore neighboring solutions and avoid convergence to local optima, if the parents are too similar. A common way of mutating chromosomes is to go through each gene and generate a number in $[0, 1]$. If this number is below a probability p, the gene's value is replaced with a random new value [21].

Replacement

The current population is either partially or completely replaced by the offspring of the recombination and mutation processes. The GA returns to the Evaluation step, unless one or more stop criteria are reached – for example, a certain fitness value being reached, or the maximum number of iterations.

3 Implementations

This section details the method used to determine container sequences, as well as two heuristics that were implemented to fill the containers. By container sequence, we mean a list of the containers that will be used to pack items. For example, given container types 1, 2 and 3, a valid container sequence is $(1, 2, 2, 3, 3)$. Heuristics should try to put each item within the first container in the sequence. If that is not possible, they should try to place the item in the second container, and so on.

Determining a container sequence is a complex task: In the best scenario, the sequence is such that no items are left outside of a container, and the cost associated with the set of selected containers is minimal. Since it is computationally expensive to test every possible container sequence for a problem, we propose an integer programming (IP) model to find sequences that should at least come close to meeting these criteria. Afterwards, we adapt two heuristics from the literature to this method, which allows them to solve MCLPs.

3.1 Container sequence determination via integer programming

Suppose a set of n cuboid container types of different widths (W_k), heights (H_k) and depths (D_k), with $k = 1, \ldots, n$. We want to find a sequence of containers that all items can be packed into, with the possibility of repeating containers of each type, while also minimizing the costs associated with each container in the sequence. To do so, we propose the use of the IP presented in Equations (4) to (7), where $c \in \mathbb{R}^n$ represents the cost of each container type, $s \in \mathbb{R}^n$ is the vector of decision variables that indicate how many units of each container to use, and $v \in \mathbb{R}^n$ is the vector containing the volume of each container $(v_i = W_i H_i D_i)$.

$$
\begin{aligned}
&\min\ c's && && (4)\\
&\text{s.t.}\ \sum_{i=1}^{n} v_i s_i \geq V_s + V_c, && && (5)\\
&\qquad s_i \geq 0, && \forall i \in \{1, \ldots, n\}, && (6)\\
&\qquad s_i \in \mathbb{Z}, && \forall i \in \{1, \ldots, n\}. && (7)
\end{aligned}
$$

Equation (4) minimizes the objective function, which is defined as the total cost of all selected containers. Equation (5) imposes that the total volume of the selected containers must be greater than the total volume of the items in stock (V_s) and the containers selected in a previous iteration of the IP model (V_c). At first, $V_c = 0$ and V_s is the sum of all item volumes. Now suppose that the IP model is used to determine a sequence of containers, and the heuristic used is not able to load all the items into the selected containers. In this case, V_c becomes the volume sum of all containers in the previous sequence, and V_s becomes the volume sum of all items that the heuristic was unable to load. This ensures that the IP model searches for a new sequence of containers, since it must now determine a sequence of greater volume than before. Finally, Equations (6) and (7) establish that the containers are unitary.

With the values of s determined, a sequence can be built by repeating container i a total of s_i times, for all $i \in \{1, \ldots, n\}$. There are many ways to build this sequence, but we assume that it is better to start with the largest selected containers and finish with the smallest containers, since that might help to load larger items first, which can facilitate the packing process of smaller items later on.

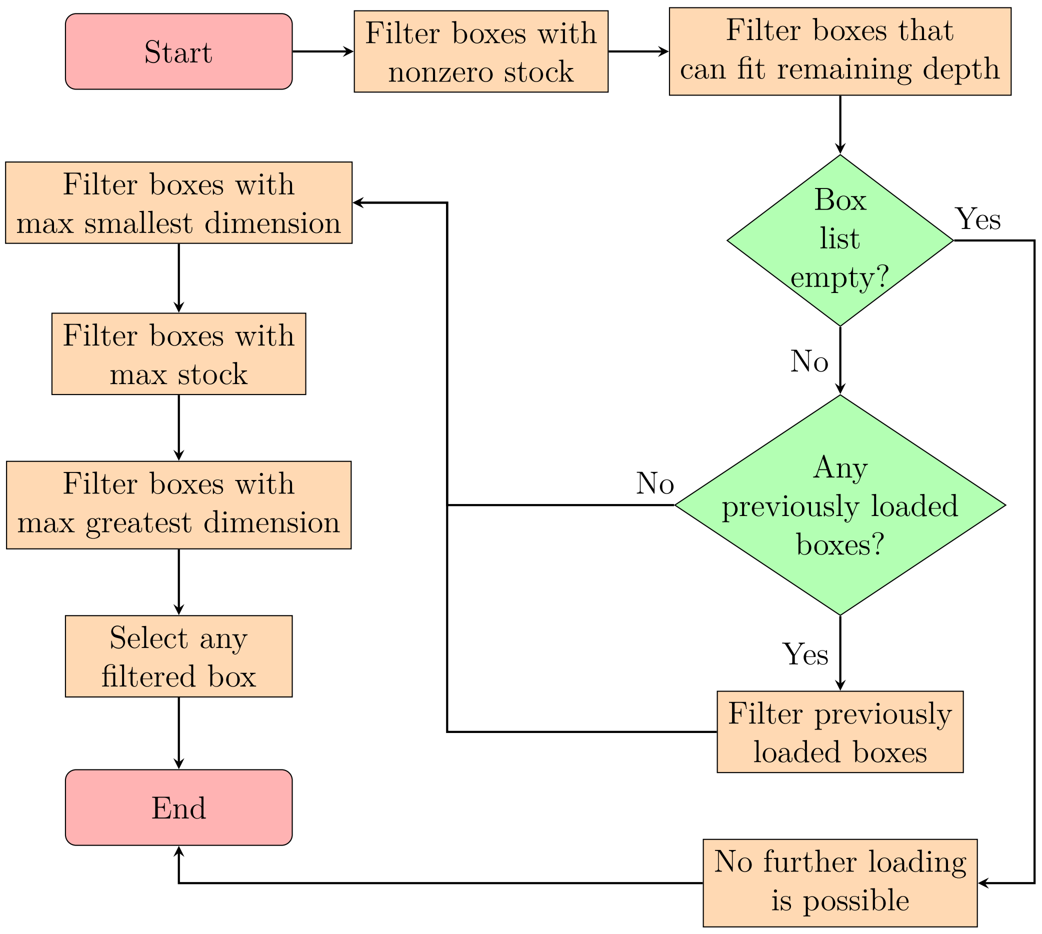
3.2 Wall-building

The wall-building (WB) heuristic attempts to load a single container by stacking cuboids within layers. A layer is a space with the same width and height as the container, but a lesser depth. To determine the depth of a layer, a box type is selected according to Flowchart 4, where, for simplicity, *box* is synonymous with *box type*.

Once a box type is filtered, the feasible rotation with the greatest depth is selected, and the same depth is applied to the layer, as Figure 6 shows. For unconstrained boxes, any of the six cuboid rotations is feasible. However, for boxes with a fixed height, only two rotations are feasible. Figure 5 illustrates this.

From this point on, we refer to the box type that defines a layer's depth as its *primary box type*. In its initial state, a layer contains a single empty cuboid space, which we call *primary space*. Because they have the same depth, primary boxes are always loaded into primary spaces.

After the initial loading process, the primary space is divided into smaller cuboid spaces, which we call *secondary spaces*. Figure 7a illustrates this situation. In this example, two *heightwise* spaces are created above the primary boxes, and one *widthwise* space is created to their right. Figure 7b shows the state of the layer after loading boxes into the widthwise space. In this case, one heightwise space is created on top of the secondary boxes, and one *depthwise* space is created in front of them. The loading process is complete when none of the remaining spaces can be filled with the available box types.


Start
Filter boxes with nonzero stock
Filter boxes that can fit remaining depth
Box list empty?
Yes
No
Any previously loaded boxes?
No
Yes
Filter previously loaded boxes
Filter boxes with max smallest dimension
Filter boxes with max stock
Filter boxes with max greatest dimension
Select any filtered box
End
No further loading is possible

Flowchart 4: Primary box selection procedure

Flowchart 8 is used to determine the box type for filling secondary spaces.

Before we discuss the loading procedure, we must introduce the concept of amalgamation. Suppose that after the layer in Figure 7 is fully loaded, the depthwise space created in Figure 7b remains empty because no box type could fit in it. This depthwise space will be adjacent to the next layer in the container, which means it can be amalgamated with the next layer in an attempt to reduce wasted space. For instance, consider Figure 9, which consists of the top view of two layers. The previous layer contains two empty depthwise spaces, which we assume to be at the same height or below the current layer (otherwise, no amalgamation is possible). Since space Ω has the least depth, any box we place within it is guaranteed not to overlap with other boxes from the previous layer, and thus we choose Ω to amalgamate with. After the amalgamation, the original space is split into two, $S_L = L$ and $S_R = R \cup \Omega$, with widths w_L and w_R, respectively. However, changing these widths during loading might lead to better space utilization, which is why we introduce a *flexible width* parameter, $\hat{w} = \phi w_R$, with $\phi \in [0, 1]$. As we

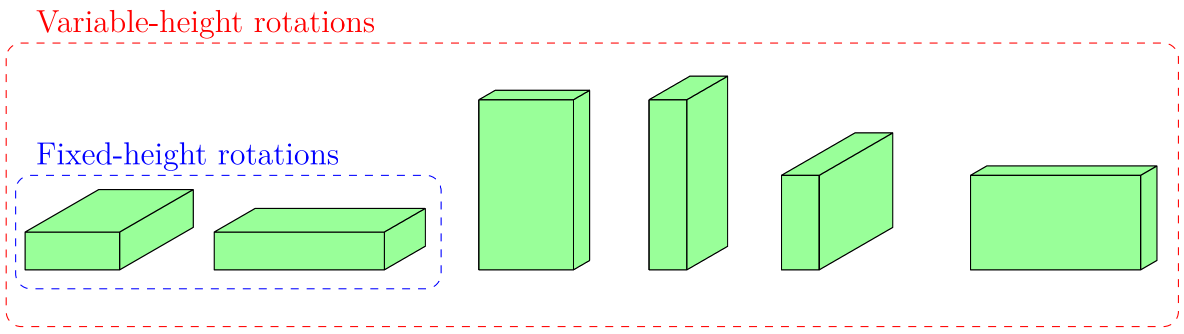


Figure 5: All possible box rotations

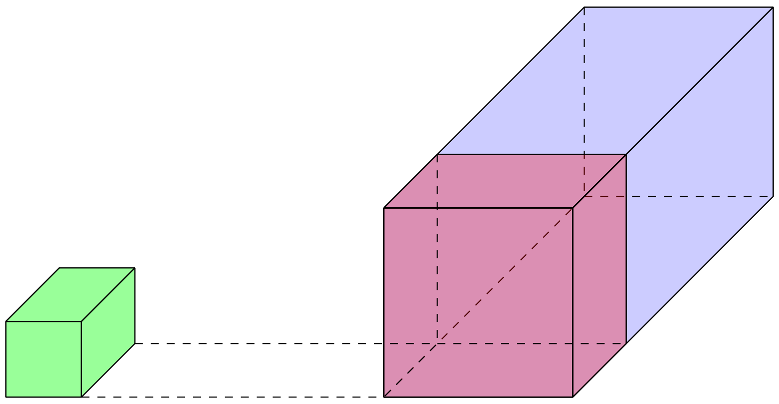


Figure 6: A selected box (left) defining the depth of a container layer (right)

clarify further ahead (Flowchart 10), this parameter allows for the width of S_L to grow to at most $w_L + \hat{w}$, with the width of S_R decreasing accordingly.

We now describe the loading procedure. Let B_{wh} be the box selected through Flowchart 4 or Flowchart 8, and B_{hw} a rotation that swaps its width and height. If B_{wh} has a fixed height, or if B_{hw} does not fit in the space, then we keep B_{wh}. Otherwise, if enough stock exists to complete a column with either rotation, we select the rotation that results in the highest column. If neither rotation completes a column, we choose the rotation with the greatest height. This leads us to Flowchart 10, which describes how spaces are filled using the selected box type. Since spaces are filled by side-by-side columns of a single box rotation, determining the cuboid spaces that remain after loading is trivial.

With this procedure, a single container can be loaded with items. To generalize the loading process to multiple containers, Algorithm 1 is used.

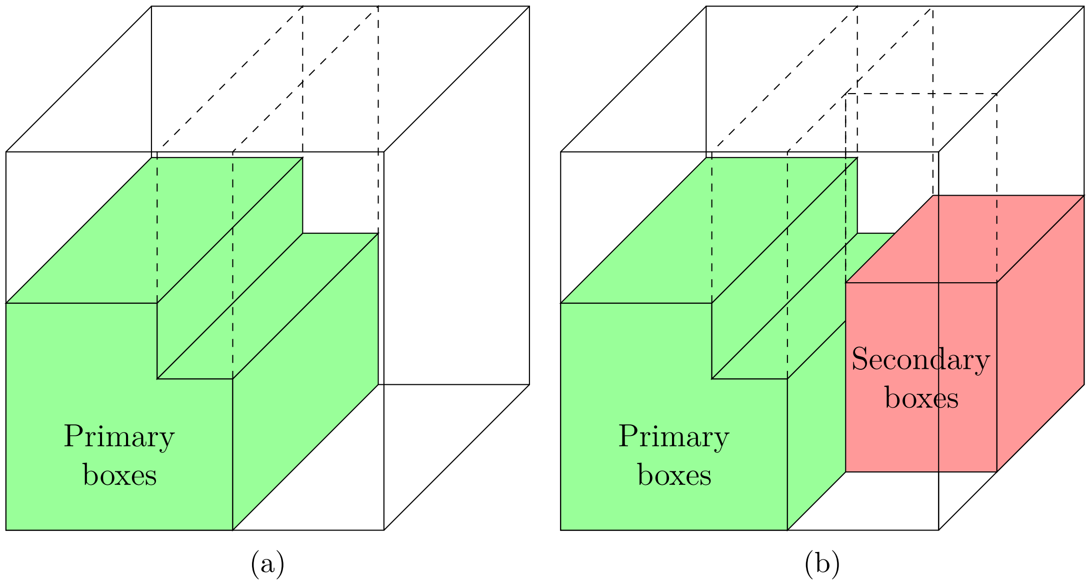


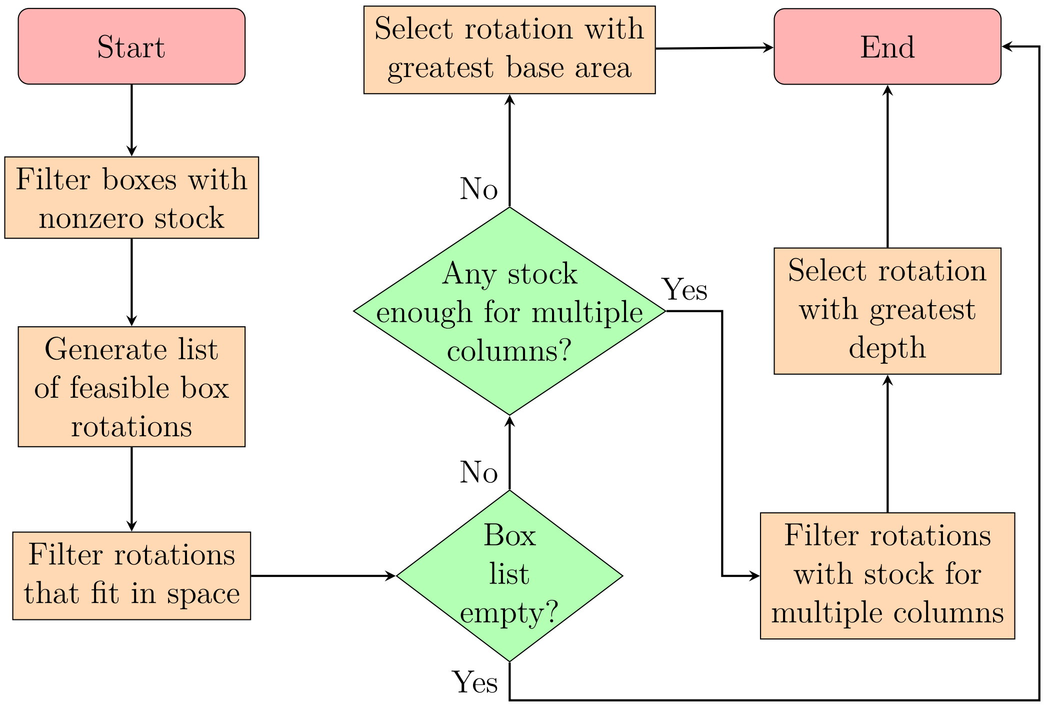
Figure 7: Remaining spaces in the first filling of a layer

3.3 Genetic algorithm

The GA developed is a simplified version of the one presented by Gonçalves and Resende [11]. To understand this method, we first need to enumerate the different item types from 1 to n. Then, let $q_1, \ldots, q_n$ be the quantities of each item type. A non-decreasing sequence S can be defined, containing item k a total of q_k times, for $k = 1, \ldots, n$. For example, given 3 item types with $q_1 = q_2 = 2$, $q_3 = 1$, it follows that $S = (1, 1, 2, 2, 3)$.

For a problem with $Q = \sum_{k=1}^{n} q_k$ items, a population of chromosomes with $2Q$ genes is generated, with values in the $[0, 1]$ interval. The first Q genes specify the order in which the heuristic attempts to place items. This is done by associating each gene to the item type with the same index in S, and then reordering S the same way as needed to put the first Q genes in ascending order. As an example, suppose $S = (1, 1, 2, 2, 3)$, and a chromosome whose first five genes are $(0.93, 0.42, 0.17, 0.48, 0.80)$. If we rearrange these numbers in ascending order, we obtain $(0.17, 0.42, 0.48, 0.80, 0.93)$. By equivalently swapping the items in S, we get its rearrangement, $\bar{S} = (2, 1, 2, 3, 1)$.

The remaining Q genes inform how each item must be placed. Specifically, gene g_{Q+k} is used to determine the rotation and plane of item $\bar{S}_k$, $k = 1, \ldots, Q$. Valid rotations for an item can be either variable- or fixed-height rotations, as shown in Figure 5. The plane can be xy, xz or yz, and it indicates the axes along which the item is placed. Figure 11 illustrates the filling of a cuboid space along the xy plane, supposing that there are four


Start
Filter boxes with nonzero stock
Generate list of feasible box rotations
Filter rotations that fit in space
Box list empty?
Yes
No
Any stock enough for multiple columns?
No
Yes
Select rotation with greatest base area
End
Select rotation with greatest depth
Filter rotations with stock for multiple columns

Flowchart 8: Secondary box selection procedure

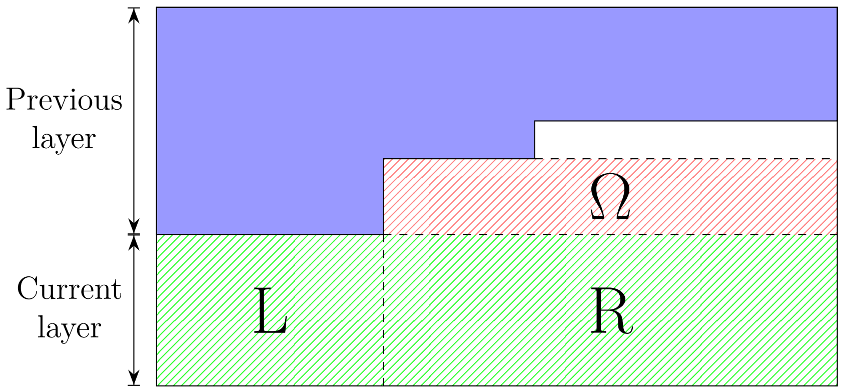
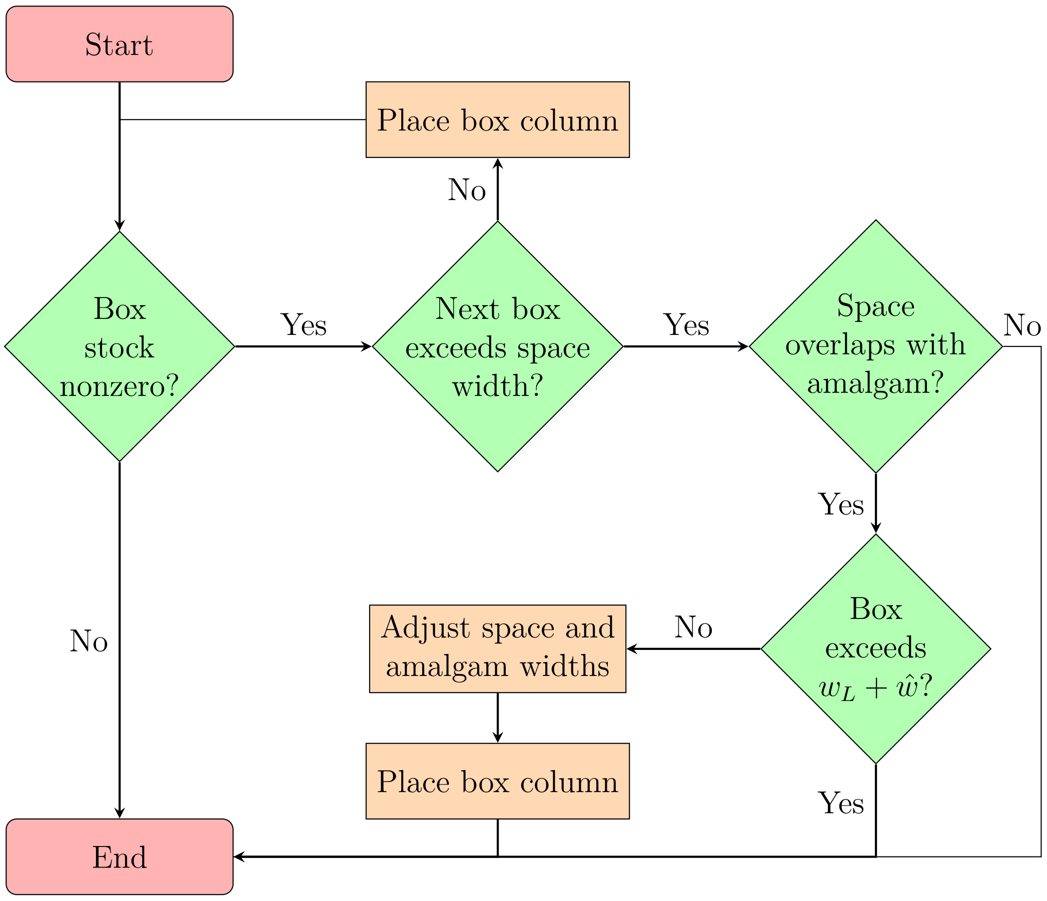


Figure 9: Amalgamation procedure


Start
Place box column
No
Box stock nonzero?
Yes
Next box exceeds space width?
Yes
Space overlaps with amalgam?
No
Yes
No
Adjust space and amalgam widths
No
Box exceeds $w_L + \hat{w}$?
Place box column
Yes
End

Flowchart 10: Box loading procedure

items to place. The process consists of selecting one of the axes, x or y, and attempting to fill it with as many strips of items as possible, such that the resulting packing is a cuboid. In the first packing, the x plane is prioritized. As a result, one horizontal strip with two items is placed. Since there is enough vertical and horizontal space left, another strip of two boxes is placed on top of the first. In the second packing, the y plane is prioritized. Three items can be stacked vertically, which leaves one. This remaining item is not enough to complete another column, therefore it is not placed.

Since there are at most six rotations, and each plane can be filled in two different ways, there is a maximum of 36 possible ways to place an item in a given space. In order to select one of the item configurations, all possibilities are mapped to different subintervals of equal length of $[0, 1]$. Then, whichever interval the respective gene belongs to is used to define the item's placement.

To select the space in which to place an item, the back-bottom-left procedure (BBL) is used. Let E_k be the k-th empty space available for packing. BBL orders spaces such

Algorithm 1 Wall-building heuristic for multiple containers

```
1: $C_{List} \leftarrow$ List of containers obtained through IP model
2: while there are items left to load do
3:     apply primary box selection procedure to remaining item types
4:     $L_{depth} \leftarrow$ depth of the new layer
5:     for $C$ in $C_{List}$ do
6:         $C_{depth} \leftarrow$ depth of remaining unfilled space in $C$
7:         if $C_{depth} \geq L_{depth}$ then
8:             apply placement procedure to $C$
9:             break
10:        end if
11:    end for
12:    if placement procedure not applied then
13:        return cannot place all the items
14:    end if
15: end while
16: return all items placed
```

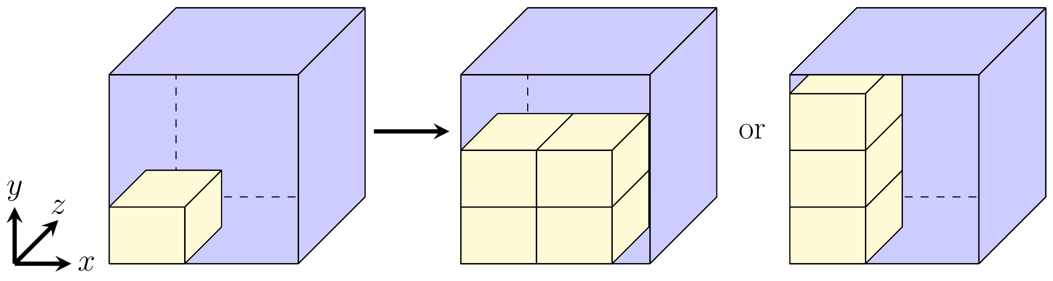


Figure 11: Two possible ways of filling the xy plane of a given space with a cuboid item

that $E_i < E_j$ if $x_i < x_j$, or $x_i = x_j$ and $y_i < y_j$, or $x_i = x_j$ and $y_i = y_j$ and $z_i < z_j$. Then, the first space in the ordering where the item fits is selected. After an item is placed, new empty cuboid spaces are created, adjacent to the item. To calculate these spaces, the method described by Lai and Chan [16] is used, with one difference: The space on top of the item is limited to the area the item occupies. This is to guarantee the vertical stability of items, by ensuring that they are either placed on the ground or that their base area is completely supported by an item directly below.

In order to extend the GA by Gonçalves and Resende [11] to multiple containers, the BBL procedure is applied to the first container in the sequence. If the current item does not fit in any of the remaining spaces, the next container in the sequence is selected, and the BBL procedure is applied again. This process is repeated until a fitting space is found, or there are no containers left to check.

A chromosome's fitness is calculated as follows: Let V_{Ti} and V_{Fi} be, respectively, the total and the filled volume of container i. Let k be the number of containers in the sequence determined by the IP model. We first calculate the fitness of a given chromosome as:

$$F = \frac{100}{k} \sum_{i=1}^{k} \frac{V_{Fi}}{V_{Ti}}. \tag{8}$$

Which gives us the mean percentage of filled volume per container. Now let L be the number of items left out of the containers after the packing process is finished. If $L > 0$, a penalty must be applied to the fitness value. In our case, we chose

$$\bar{F} = \frac{F}{10L} \tag{9}$$

as the new fitness value, because it greatly decreases with the number of items that are left outside of the containers.

Once a population is generated and the fitness value of each solution is determined, the chromosomes are sorted according to their fitness values. A certain subset of the population, containing the best fitness values, is immediately copied to the next population, so as to ensure that the best fitness does not decrease from each iteration to the next. This subset of the population is called its *elite*.

Another part of the next population is made of crossovers between solutions from the elite and solutions from the whole population, both chosen at random. Uniform crossover is used to generate a single offspring, with a higher chance of selecting genes from the elite chromosome. Finally, the last part of the next population is created by randomly generating as many chromosomes as needed to match the number of solutions in the current population. This can be seen as an extreme case of mutation.

References

[1] E.E. Bischoff, F. Janetz, and M.S.W. Ratcliff. “Loading Pallets with Non-Identical Items”. In: *European Journal of Operational Research* 84.3 (1995), pp. 681–692. DOI: 10.1016/0377-2217(95)00031-K. (Visited on 04/23/2024).

[2] Andreas Bortfeldt and Gerhard Wäscher. “Constraints in Container Loading – A State-of-the-Art Review”. In: *European Journal of Operational Research* 229.1 (2013), pp. 1–20. DOI: 10.1016/j.ejor.2012.12.006. (Visited on 04/06/2024).

[3] André Ponce de Leon F. de Carvalho. *Algoritmos Genéticos*. https://sites.icmc.usp.br/andre/research/genetic/. (Visited on 06/05/2024).

[4] Pedro Henrique Centenaro. “An Overview on Container Loading Problems.” In: (2024).

[5] C.S. Chen, S.M. Lee, and Q.S. Shen. “An Analytical Model for the Container Loading Problem”. In: *European Journal of Operational Research* 80.1 (1995), pp. 68–76. DOI: 10.1016/0377-2217(94)00002-T. (Visited on 04/06/2024).

[6] Charles Darwin. “The Origin of Species”. In: (1859).

[7] Jens Egeblad and David Pisinger. “Heuristic Approaches for the Two- and Three-Dimensional Knapsack Packing Problem”. In: *Computers & Operations Research* 36.4 (2009), pp. 1026–1049. DOI: 10.1016/j.cor.2007.12.004. (Visited on 04/06/2024).

[8] Michael Eley. “Solving Container Loading Problems by Block Arrangement”. In: *European Journal of Operational Research* 141.2 (2002), pp. 393–409. DOI: 10.1016/S0377-2217(02)00133-9. (Visited on 04/06/2024).

[9] Tobias Fanslau and Andreas Bortfeldt. “A Tree Search Algorithm for Solving the Container Loading Problem”. In: *INFORMS Journal on Computing* 22.2 (2010), pp. 222–235. DOI: 10.1287/ijoc.1090.0338. (Visited on 04/23/2024).

[10] J.A. George and D.F. Robinson. “A Heuristic for Packing Boxes into a Container”. In: *Computers & Operations Research* 7.3 (1980), pp. 147–156. DOI: 10.1016/0305-0548(80)90001-5. (Visited on 04/06/2024).

[11] José Fernando Gonçalves and Mauricio G.C. Resende. “A Parallel Multi-Population Biased Random-Key Genetic Algorithm for a Container Loading Problem”. In: *Computers & Operations Research* 39.2 (2012), pp. 179–190. DOI: 10.1016/j.cor.2011.03.009. (Visited on 04/06/2024).

[12] Denny Hermawanto. “Genetic Algorithm for Solving Simple Mathematical Equality Problem”. In: (2013).

[13] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[14] Erwin Kalvelagen. *Yet Another Math Programming Consultant: 2d Knapsack Problem*. 2021. (Visited on 08/20/2024).

[15] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. “A Review on Genetic Algorithm: Past, Present, and Future”. In: *Multimedia Tools and Applications* 80.5 (2021), pp. 8091–8126. DOI: 10.1007/s11042-020-10139-6. (Visited on 04/17/2024).

[16] K.K. Lai and Jimmy W.M. Chan. “Developing a Simulated Annealing Algorithm for the Cutting Stock Problem”. In: *Computers & Industrial Engineering* 32.1 (1997), pp. 115–127. DOI: 10.1016/S0360-8352(96)00205-7. (Visited on 07/10/2024).

[17] Sheng Liu, Wei Tan, Zhiyuan Xu, and Xiwei Liu. “A Tree Search Algorithm for the Container Loading Problem”. In: *Computers & Industrial Engineering* 75 (2014), pp. 20–30. DOI: 10.1016/j.cie.2014.05.024. (Visited on 04/23/2024).

[18] Silvano Martello. *Surveys in Combinatorial Optimization*. North-Holland Mathematics Studies 132. Amsterdam New York: North-Holland Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, 1987.

[19] Olympia Roeva, Stefka Fidanova, and Marcin Paprzycki. “Influence of the Population Size on the Genetic Algorithm Performance in Case of Cultivation Process Modelling”. In: (2013).

[20] “Genetic Algorithms”. In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Ed. by Kumara Sastry, David Goldberg, and Graham Kendall. New York: Springer, 2005.

[21] Nitasha Soni and Tapas Kumar. “Study of Various Mutation Operators in Genetic Algorithms”. In: 5 (2014).