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

According to Fanslau and Bortfeldt [6], inexact methods for CLPs 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 [7], layer-building [1] and block-building [5] heuristics.

(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 [4] and genetic algorithms [8].

(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 [6], the improvement step of the block-building heuristic [5], and the tree search method by Liu et al. [10].

Of particular interest to us are the wall-building and genetic algorithm methods, which we describe in greater detail ahead.

2.1 Wall-building heuristic

The literature review on CLPs conducted by Bortfeldt and Wäscher [2] marks the wall-building heuristic (WB) by George and Robinson [7] as the oldest method in the literature. Regardless, WB has proven to be very successful, with a few of the most robust methods in the literature drawing inspiration from it (e.g. [3]). WB builds solutions by sequentially cutting the container into cuboid spaces of same width and height, but lesser depth, called layers, which are then filled with boxes according to a set of rules. Section 3.1 contains a detailed description of our WB implementation.

2.2 Genetic algorithms

Genetic algorithms (GAs) are methods that take inspiration from evolutionary principles to solve problems. To achieve this goal, a GA must first encode solutions in the form of chromosomes. Usually, this means that solutions are converted into strings of numbers, with each position in the string representing a different characteristic of the

solution (a gene). Once the conversion process is properly defined, we must establish a fitness function, which will be responsible for evaluating the quality of a given solution.

According to Sastry, Goldberg, and Kendall [12], GAs take the following steps to evolve the pool of available solutions, known as a population:

1. Initialization: A population of n solutions is generated, either randomly or following a specialized method. Though n is arbitrary, research indicates that if it is too small for a given problem, GA may get stuck in a local optimum; if it is too large, it may consume more computational resources without improving the population [11].

2. Evaluation: All solutions in the population are evaluated using the fitness function.

3. Selection: A subset of the population is chosen to breed solutions for the next generation. This takes the fitness function's evaluations into account.

4. Recombination: Two or more selected parents are combined to create new chromosomes.

5. 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.

6. Replacement: The current population is either partially or completely replaced by the offspring of the recombination and mutation processes.

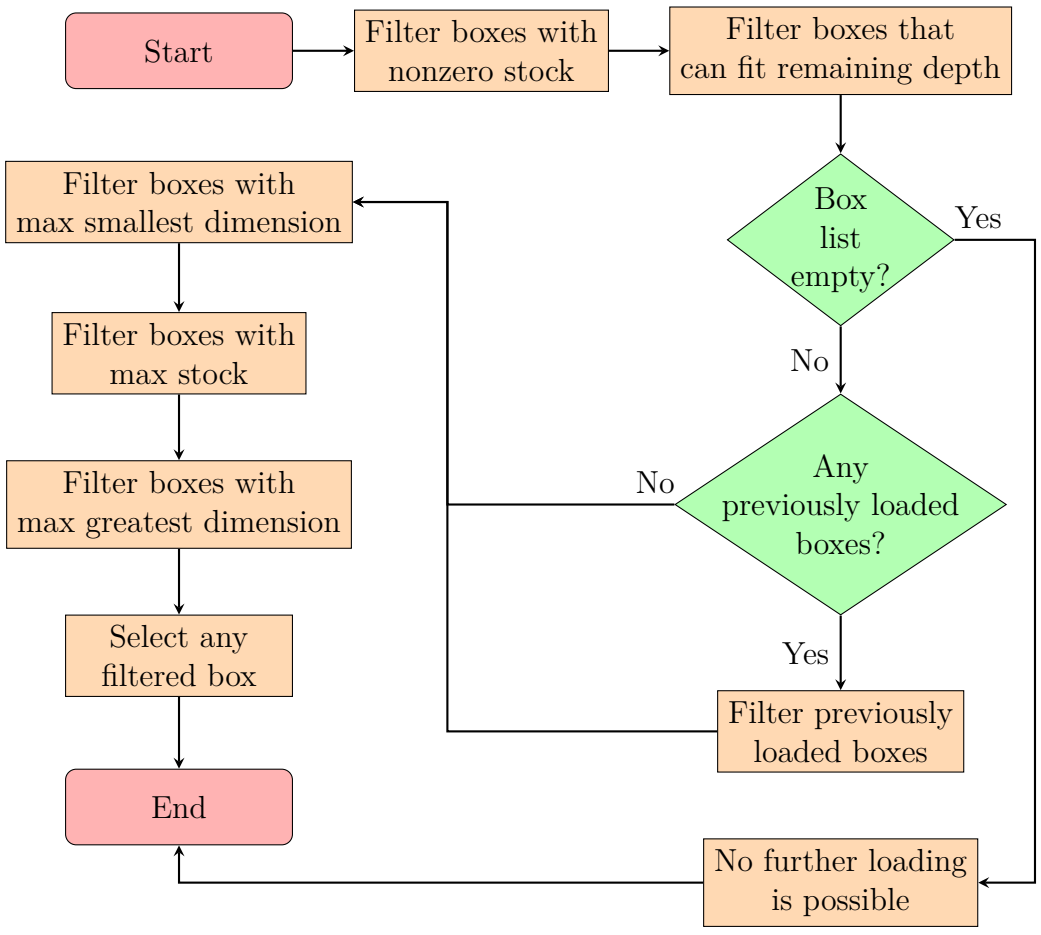
7. Steps 2 through 7 are repeated until contextual criteria are met.

3 Implementations

In this section, we describe our implementations of multiple heuristics, which can be found here.

3.1 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 1, where, for simplicity, *box* is synonymous with *box type*.



Flowchart 1: Primary box selection procedure

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 3 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 2 illustrates this.

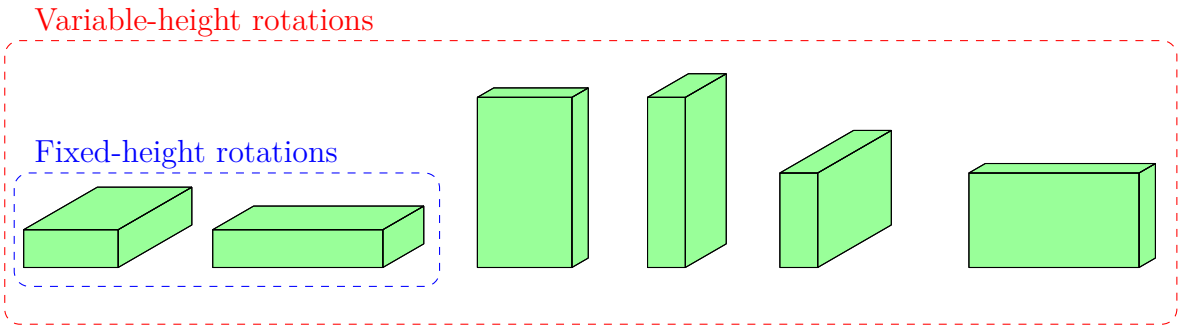


Figure 2: All possible box rotations

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.

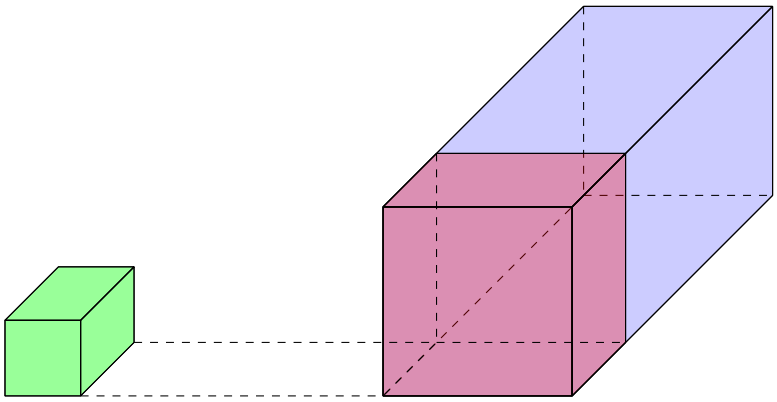


Figure 3: A selected box (left) defining the depth of a container layer (right)

After the initial loading process, the primary space is divided into smaller cuboid spaces, which we call *secondary spaces*. Figure 4a 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 4b 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. Flowchart 5 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 4 is fully loaded, the depthwise space created in Figure 4b remains empty because no box type could fit in it. This depthwise

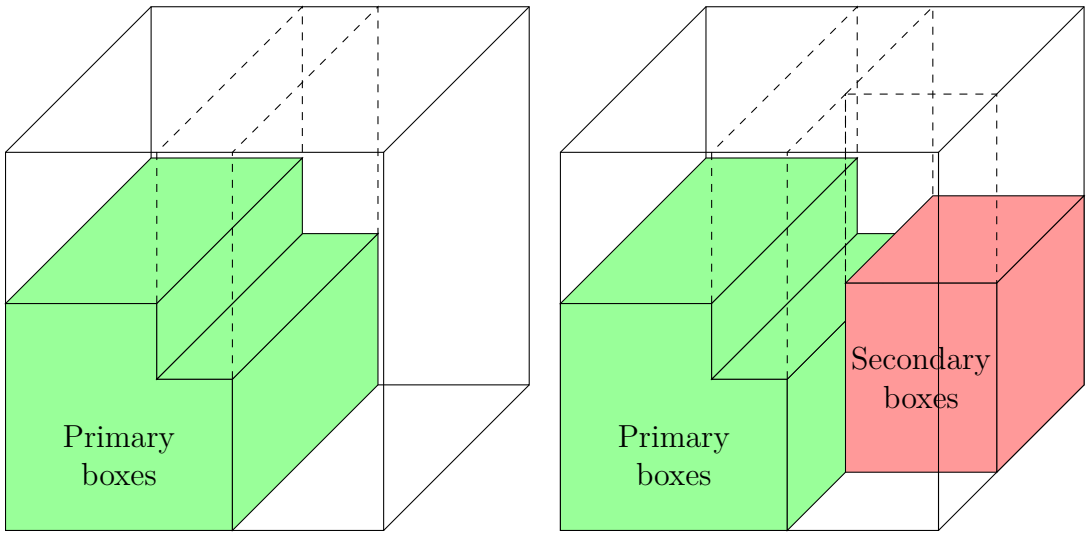
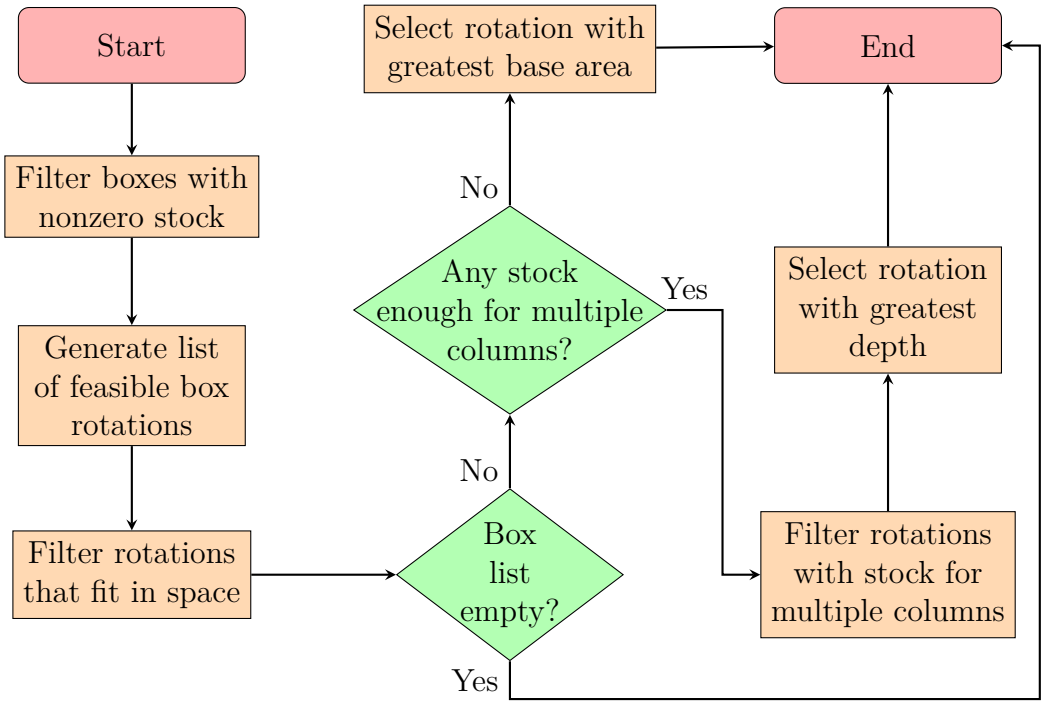


Figure 4: Remaining spaces in the first filling of a layer

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 6, 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 clarify further ahead (Flowchart 7), 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 1 or Flowchart 5, 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 7, 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



Flowchart 5: Secondary box selection procedure

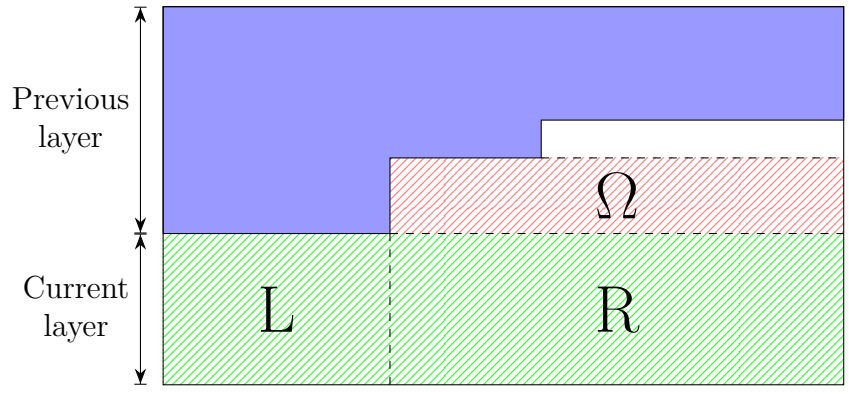
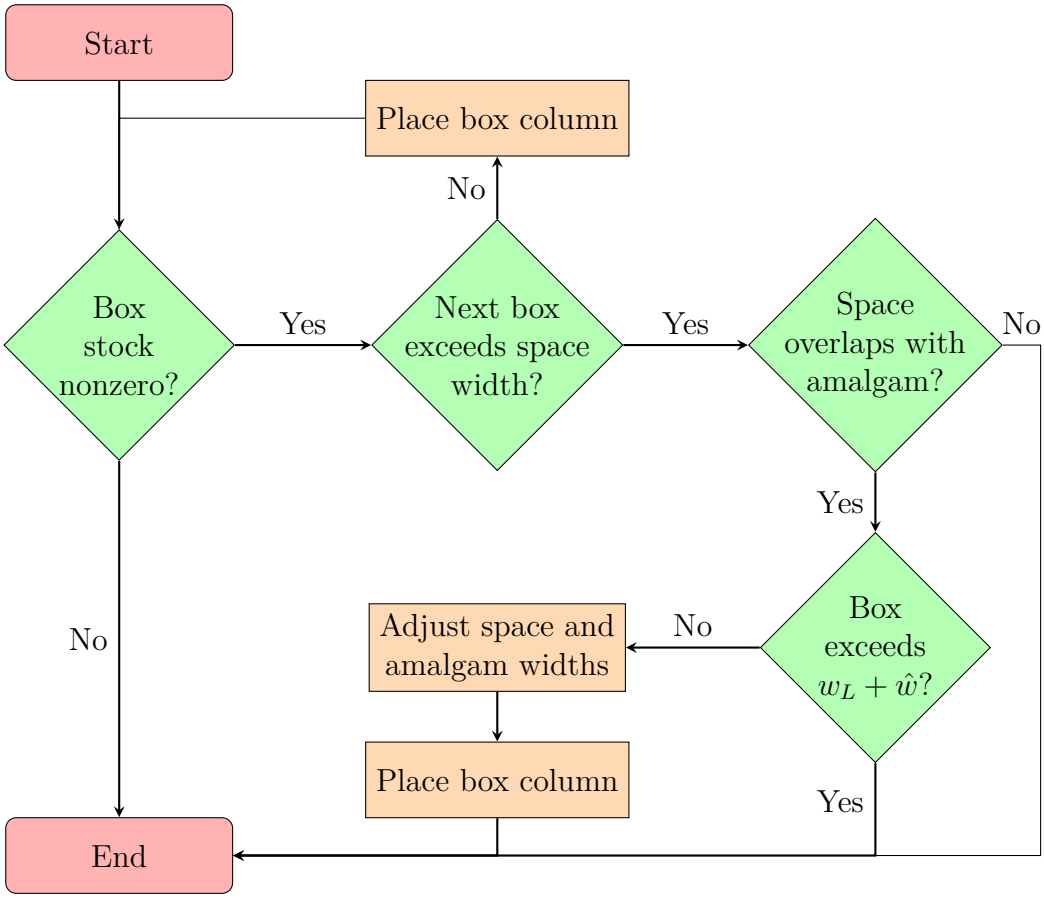


Figure 6: Amalgamation procedure



Flowchart 7: Box loading procedure

loading process to multiple containers, Algorithm 1 is used.

3.2 Genetic algorithm

The GA developed is a simplified version of the one presented by Gonçalves and Resende [8]. 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 $S = (1, 1, 2, 2, 3)$,

Algorithm 1 Wall-building heuristic for multiple containers

```
C_List ← List of containers obtained through IP model
while there are items left to load do
    apply primary box selection procedure to remaining item types
    L_depth ← depth of the new layer
    for C in C_List do
        C_depth ← depth of remaining unfilled space in C
        if C_depth ≥ L_depth then
            apply placement procedure to C
            break
        end if
    end for
    if placement procedure not applied then
        return cannot place all the items
    end if
end while
return all items placed
```

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 2. The plane can be xy, xz or yz, and it indicates the axes along which the item is placed. Figure 8 illustrates the filling of a cuboid space along the xy plane, supposing that there are four 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.

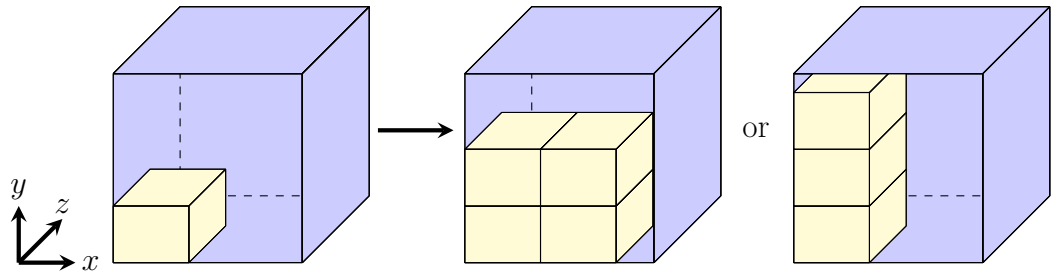


Figure 8: Two possible ways of filling the xy plane of a given space with a cuboid item

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 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 [9] 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 [8] 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:

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

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

$$\bar{F} = \frac{F}{10L} \tag{2}$$

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 number of top solutions, called the population's elite, is immediately copied to the next population. Another set of chromosomes of the next population is randomly generated, which can be interpreted as a form of mutation. Finally, the remaining spots left in the next population are filled by chromosomes resulting from the crossover of randomly selected elite solutions and randomly selected solutions from the entirety of the population.

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. URL: https://linkinghub.elsevier.com/retrieve/pii/037722179500031K (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. URL: https://linkinghub.elsevier.com/retrieve/pii/S037722171200937X (visited on 04/06/2024).

[3] Jens Egeblad, Claudio Garavelli, Stefano Lisi, and David Pisinger. "Heuristics for Container Loading of Furniture". In: *European Journal of Operational Research* 200.3 (2010), pp. 881–892. DOI: 10.1016/j.ejor.2009.01.048. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221709000563 (visited on 04/06/2024).

[4] 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. URL: https://linkinghub.elsevier.com/retrieve/pii/S030505480700264X (visited on 04/06/2024).

[5] 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. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221702001339 (visited on 04/06/2024).

[6] 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. URL: https://pubsonline.informs.org/doi/10.1287/ijoc.1090.0338 (visited on 04/23/2024).

[7] 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. URL: https://linkinghub.elsevier.com/retrieve/pii/0305054880900015 (visited on 04/06/2024).

[8] 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. URL: https://linkinghub.elsevier.com/retrieve/pii/S0305054811000827 (visited on 04/06/2024).

[9] 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. URL: https://linkinghub.elsevier.com/retrieve/pii/S0360835296002057 (visited on 07/10/2024).

[10] 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. URL: https://linkinghub.elsevier.com/retrieve/pii/S0360835214001776 (visited on 04/23/2024).

[11] 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).

[12] "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.