# 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 [FB10], 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 [GR80], layer-building [BJR95] and block-building [Ele02] 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 [EP09] and genetic algorithms [GR12].

(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 [FB10], the improvement step of the block-building heuristic [Ele02], and the tree search method by Liu *et al.* [LTXL14].

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 [BW13] marks the wall-building heuristic (WB) by George and Robinson [GR80] 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. [EGLP10]). 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 and Goldberg [SG05], 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 [RFP13].

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.

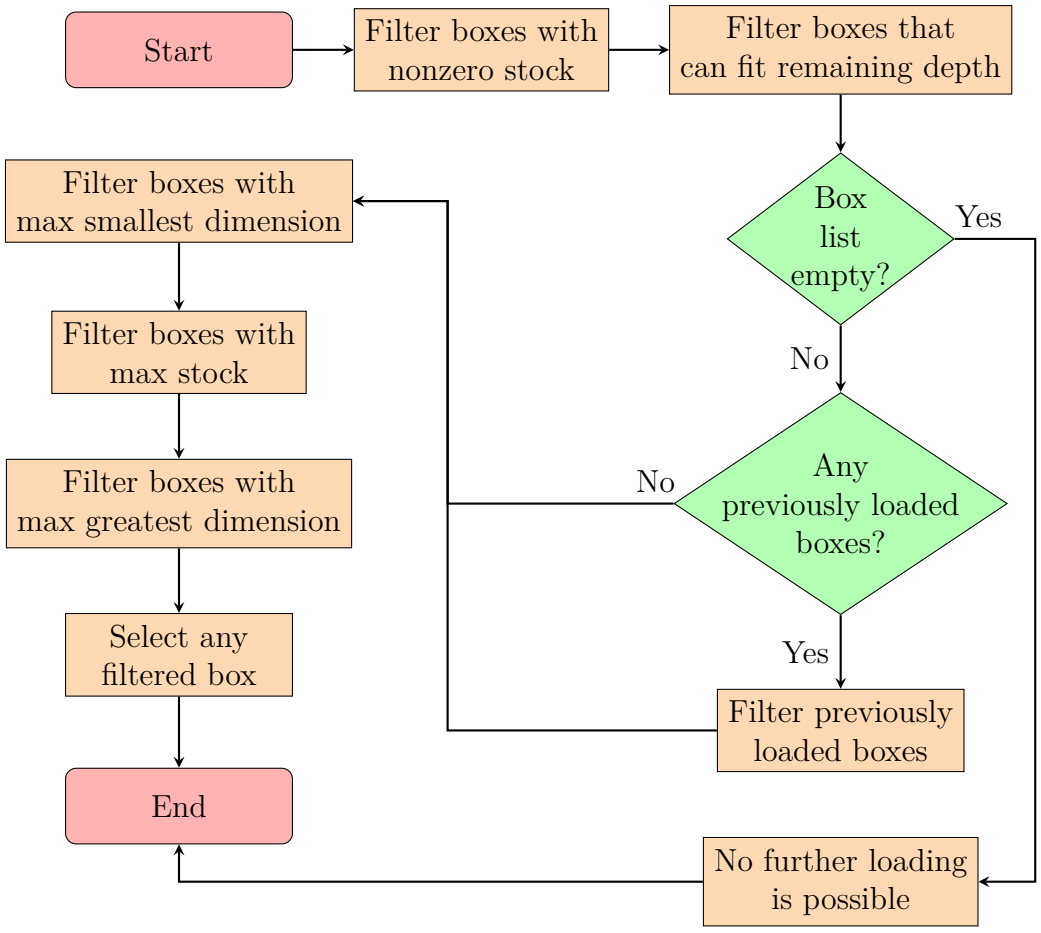
7. Steps 2 through 7 are repeated until contextual criteria are met.

# 3 Heuristics

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.

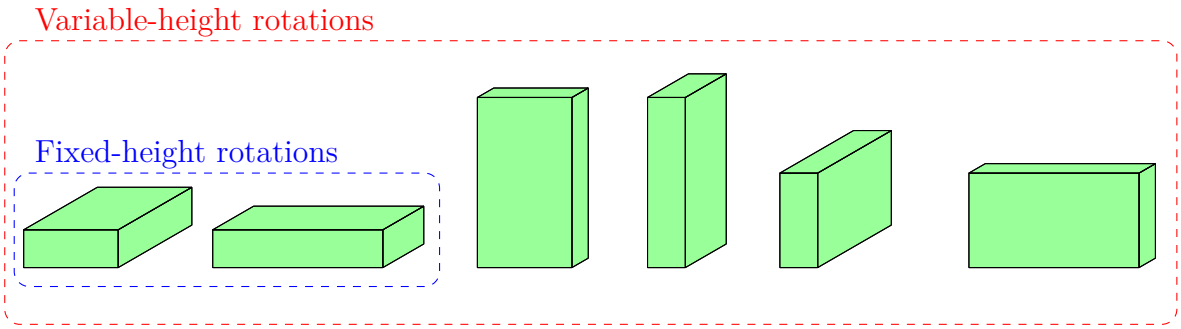



Figure 2: All possible box rotations

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.

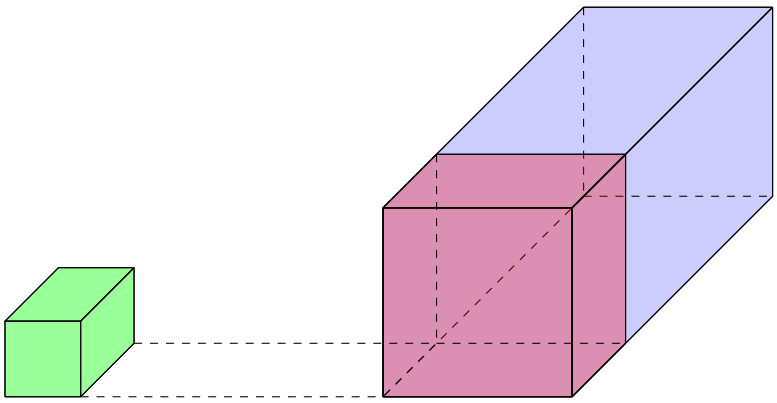


Figure 3: A selected box (left) defining the depth of a container layer (right)

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

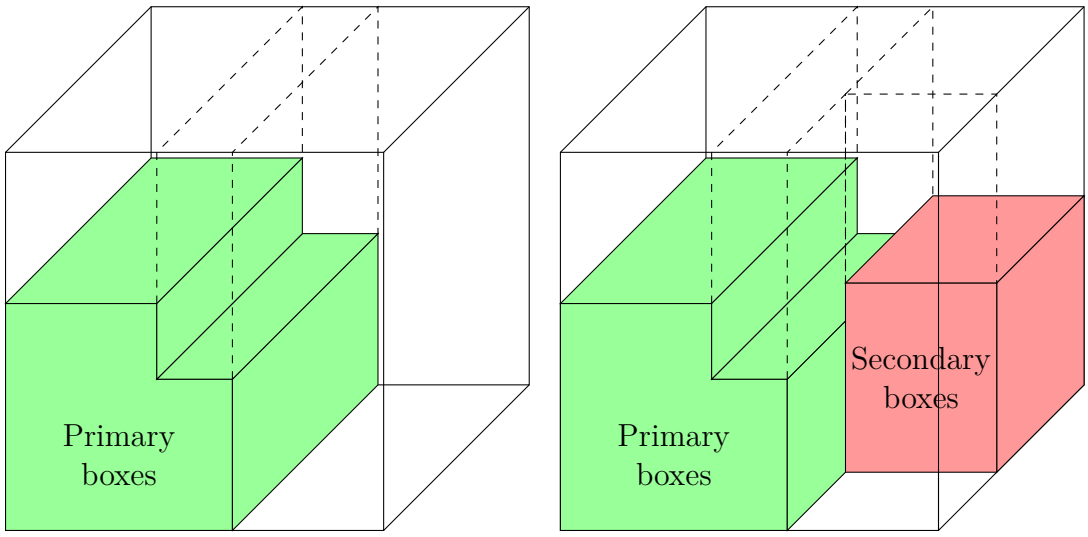
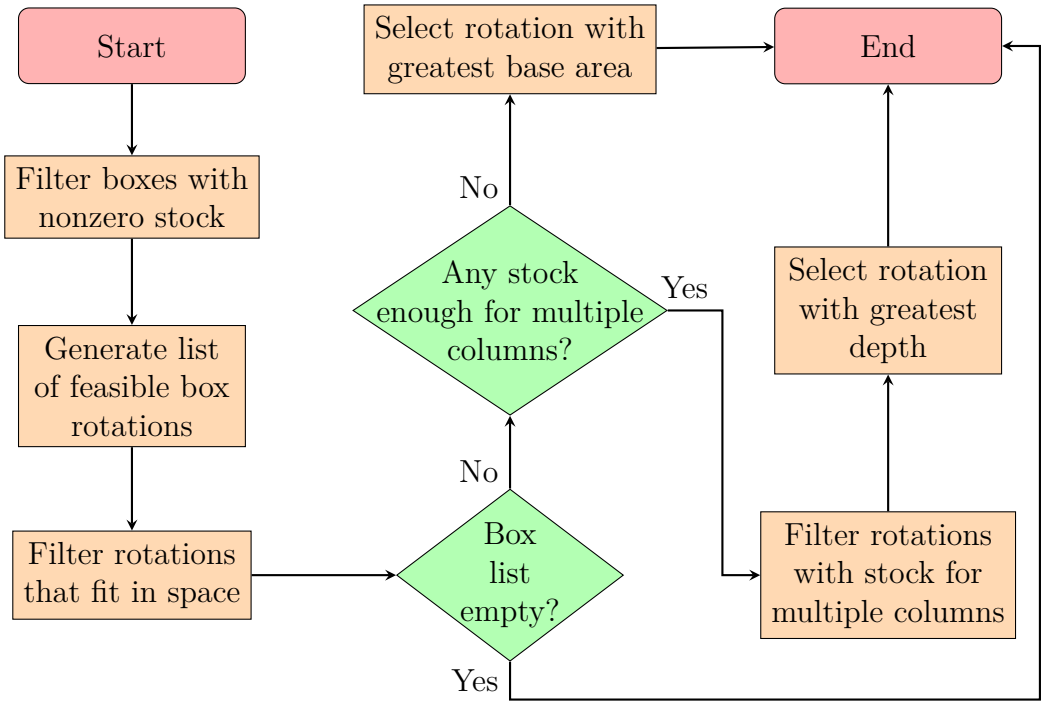



Figure 4: Remaining spaces in the first filling of a layer

created in Figure 4b 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 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 $\Omega$ 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 $\Omega$ 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.




Flowchart 5: Secondary box selection procedure

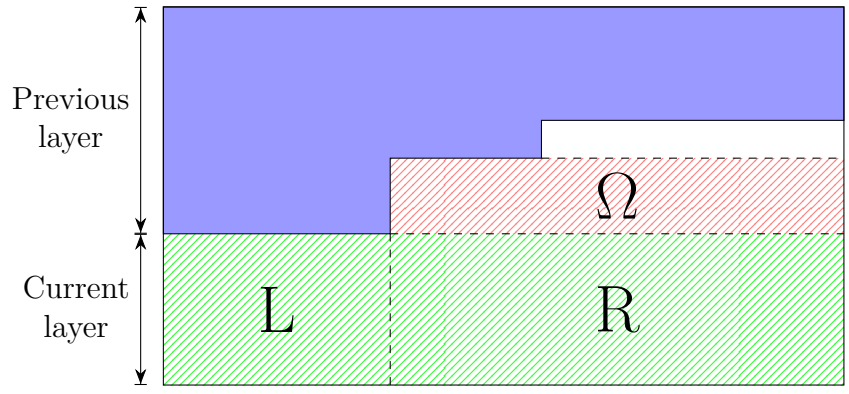
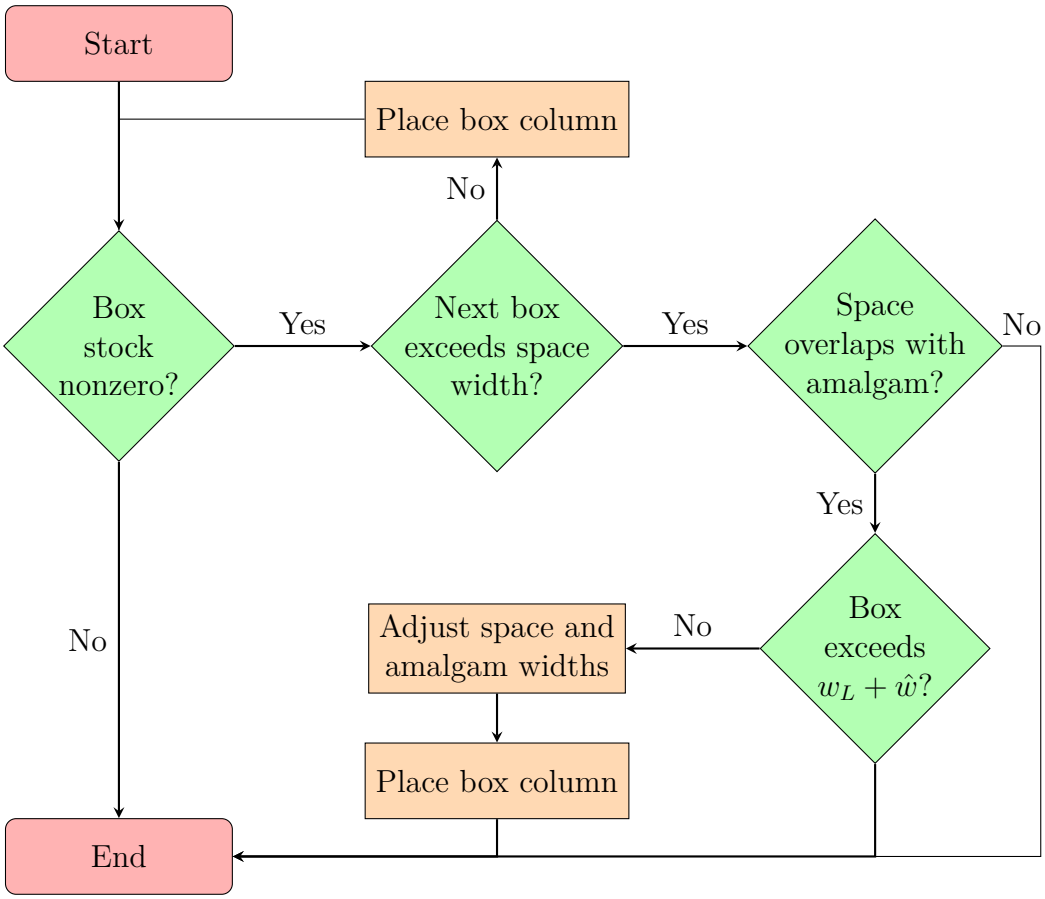



Figure 6: Amalgamation procedure




Flowchart 7: Box loading procedure

# References


[BJR95] E. Bischoff, F. Janetz, and M. Ratcliff, "Loading pallets with non-identical items," en, *European Journal of Operational Research*, vol. 84, no. 3, pp. 681–692, 1995. DOI: 10.1016/0377-2217(95)00031-K. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/037722179500031K (visited on 04/23/2024).

[BW13] A. Bortfeldt and G. Wäscher, "Constraints in container loading – A state-of-the-art review," en, *European Journal of Operational Research*, vol. 229, no. 1, pp. 1–20, 2013. DOI: 10.1016/j.ejor.2012.12.006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S037722171200937X (visited on 04/06/2024).

[EGLP10] J. Egeblad, C. Garavelli, S. Lisi, and D. Pisinger, "Heuristics for container loading of furniture," en, *European Journal of Operational Research*, vol. 200, no. 3, pp. 881–892, 2010. DOI: 10.1016/j.ejor.2009.01.048. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0377221709000563 (visited on 04/06/2024).

[EP09] J. Egeblad and D. Pisinger, "Heuristic approaches for the two- and three-dimensional knapsack packing problem," en, *Computers & Operations Research*, vol. 36, no. 4, pp. 1026–1049, 2009. DOI: 10.1016/j.cor.2007.12.004. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S030505480700264X (visited on 04/06/2024).

[Ele02] M. Eley, "Solving container loading problems by block arrangement," en, *European Journal of Operational Research*, vol. 141, no. 2, pp. 393–409, 2002. DOI: 10.1016/S0377-2217(02)00133-9. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0377221702001339 (visited on 04/06/2024).

[FB10] T. Fanslau and A. Bortfeldt, "A Tree Search Algorithm for Solving the Container Loading Problem," en, *INFORMS Journal on Computing*, vol. 22, no. 2, pp. 222–235, 2010. DOI: 10.1287/ijoc.1090.0338. [Online]. Available: https://pubsonline.informs.org/doi/10.1287/ijoc.1090.0338 (visited on 04/23/2024).

[GR80] J. George and D. Robinson, "A heuristic for packing boxes into a container," en, *Computers & Operations Research*, vol. 7, no. 3, pp. 147–156, 1980. DOI:

10.1016/0305-0548(80)90001-5. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0305054880900015 (visited on 04/06/2024).

[GR12] J. F. Gonçalves and M. G. Resende, "A parallel multi-population biased random-key genetic algorithm for a container loading problem," en, *Computers & Operations Research*, vol. 39, no. 2, pp. 179–190, 2012. DOI: 10.1016/j.cor.2011.03.009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0305054811000827 (visited on 04/06/2024).

[LTXL14] S. Liu, W. Tan, Z. Xu, and X. Liu, "A tree search algorithm for the container loading problem," en, *Computers & Industrial Engineering*, vol. 75, pp. 20–30, 2014. DOI: 10.1016/j.cie.2014.05.024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0360835214001776 (visited on 04/23/2024).

[RFP13] O. Roeva, S. Fidanova, and M. Paprzycki, "Influence of the Population Size on the Genetic Algorithm Performance in Case of Cultivation Process Modelling," en, 2013.

[SG05] "Genetic Algorithms," en, in *Search methodologies: introductory tutorials in optimization and decision support techniques*, K. Sastry and D. Goldberg, Eds., New York: Springer, 2005.