UNIVERSITY OF JYVÄSKYLÄ

Final Report

Optimisation and Visualisation of Loading Strategies in 3-Dimesional Truck Containers

Author:
Quentin Hû

Supervised by:
Michael Emmerich & Timo Tiihonen

September 23, 2024

Contents

1 Introduction

This document presents the work conducted as part of a five-month (from January to May), half-time research project with the Multi-Objective Optimization Laboratory at the University of Jyväskylä, supervised by Mr. Michael Emmerich. The project aims to develop and visualize optimal strategies using metaheuristics and reinforcement learning to solve a truck loading problem.

In the field of transportation logistics, the problem of optimally loading the container is often solved by practitioners with the assistance of algorithms. This field play a critical role in our increasingly connected economy where speed and efficiency in delivering goods are central themes. Besides optimal space utilization, the effort to load and unload a container needs to be considered. This project focuses on minimizing the effort of loading and unloading objects into/from a truck container using a fork-lift. It adds complexity by dealing with objects from a storage room grouped by clients.

1.1 Original project plan

A plan has been created at the beginning of the project with the objective of scheduling the various tasks to be completed, estimating their duration and difficulties (see Fig 10 in the appendix).

1.2 Deliverables of the project

Additionally, it includes several deliverables:

- A functional program implementing the developed solution.
- A demonstration tool for this problem that allows the visualization of the container filling process box by box.
- A final report (this document) that provides an account of the project, the work performed, and the results obtained.

2 Container Loading Problem

The container loading problem has been addressed in the literature for many years. A brief overview of the literature will be provided in this section, followed by a more in-depth presentation of the details of the problem that will be tackled in this project.

2.1 Litterature overview

According to the typology introduced by Wäscher, Haußner & Schumann (2007) [WHS07], this problem is in the category of the Single Large Object Placement Problem (SLOPP) in three dimensions and is a NP-Hard problem. This same problem in one dimension (the knapsack problem) is already an NP-Hard problem, so adding dimensions and additional constraints only increases the difficulty of solving it. Consequently, only very few exact algorithms exist and the research on heuristic algorithms is more advanced. In 2011, the issue of simultaneously satisfying several constraints had not yet been satisfactorily addressed. Soft constraints, even though of considerable relevance in practice, have also not been dealt with significantly [BW12]. More than ten years later, much progress has been made like Gajda, Trivella, Mansini and Pisinger in 2021 , they are the first to tackle all of the practical constraints jointly on real instances [GTMP22] .

In general, the number of publications in the area of Container Loading is growing and a lot of different problem and constraints configurations have been studied. [BW12]

2.2 Problem definition

Container loading problems can be interpreted as geometric assignment problems, in which three-dimensional small rectangular items (called *Box*) have to be assigned to three-dimensional, rectangular larger object (called *Container*) to obtain a good solution following different optimization criteria (see Section 2.3) and constrained by the following constrained and specificity:

- The items do not overlap
- The items can be rotated horizontally
- All items lie entirely within the container
- The configuration must be stable, and the objects should not be, even partially, overhanging the void.

2.3 Optimization Criteria

Here is an overview of the different optimization criteria for our problem. It's important to keep in mind that many others are possible depending on the type of loading, the type of transport, or how items are loaded and unloaded.

Weight balance:

As a reminder, we are discussing filling a container, more precisely the trailer of a truck. For safety reasons, the loading inside the container must respect a balance of weight both in width and length. One possible evaluation of this balance proposed here is to determine the center of gravity of the solution and assess the distance between this point and the center of the container. This is a highly simplified view; in reality, numerous other parameters need to be considered, such as the position of the wheel axles, the structure of the trailer, the type of truck, etc.[BG01]

Number of boxes:

Naturally, the number of loaded boxes is an important criterion. A good solution will enable the loading of as many boxes as possible while minimizing empty spaces.

The proximity of boxes from the same client:

Boxes with the same ID come from the same client and thus should be loaded and unloaded together. For efficiency purposes, it is important to prioritize a solution where boxes from the same client are grouped together or, at least, as accessible as possible.

2.4 Goals

In this project in particular we look at these goals

- A heuristic approach to obtain solutions for the truck loading problem.
- An optimization task to achieve the best possible results.
- The solution must be a sequence of boxes in the manner of a loading plan.

3 Presentation of the work

An instance is presented as follows. There is an array for each piece of information concerning the boxes:

n the number of box.

$\{w_1, w_2, w_3, \ldots, w_n\}$ the width w_i for each box i.

$\{h_1, h_2, h_3, \ldots, h_n\}$ the height h_i for each box i.

$\{d_1, d_2, d_3, \ldots, d_n\}$ the depth d_i for each box i.

$\{wgt_1, wgt_2, wgt_3, \ldots, wgt_n\}$ the weight wgt_i for each box i.

$\{id_1, id_2, id_3, \ldots, id_n\}$ the identifier id_i for each box i.

W, D, H the dimensions of the container.

Wgt the maximum possible weight in the container.

3.1 Data structures used

Afterwards, we will manipulate several objects and concepts to ensure a thorough understanding. This section provides an overview of these various objects and their properties.

Container

The truck container is a simple rectangle in three dimensions defined by a width W, a deep D, a height H and a maximum weight Wgt.

Box

A box i is basically an item to put in the truck container. It is defined by a width w_i, a deep d_i a height h_i, an id id_i, a weight wgt_i and a position in the three-dimensional space (x_i, y_i, z_i). The position of the box is its first left-bottom corner. We assume that all boxes are rectangular and have evenly distributed weight. The box can be rotated on the horizontal axe, it results in a switch between the deep and the width. The box's ID links it to the client it is associated with. Indeed, all boxes are divided into subgroups that belong to different clients.

Corner

To obtain the free locations to place a box in the container, we calculate these locations at each corner of the already placed boxes. To do this, we start at a corner, calculate the free location to the right and below of the corner in the case of a $right$ corner type, or to the left and below in the case of a $left$ corner type (see Fig. 4). We suppose that it is not necessary to calculate free locations upwards because we start filling the container from the back gradually towards its front. Finally a corner is defined by its type $left$ or $right$, its dimensions (w, d, h) and its position (x, y, z), the position of a corner is at its top-left for the $right$ corners, top-right for the $left$ ones.

3.2 A greedy algorithm

To obtain an initial acceptable solution for our problem, we develop a greedy algorithm. The development of a greedy algorithm will provide an easily modifiable foundation for generating solutions quickly. It can serve as a basis for future optimization through reinforcement learning. This iterative algorithm follows the simple behavior as follows: choose a box, calculate its future position in the container, if possible, place it, and repeat. Thus, we obtain as desired an iterative sequence of "movements," that is, the selection of a box and its position one by one. The resulting algorithm is as follows:

Algorithm 1 Greedy Algorithm

```
1: function GREEDY(boxList)
2:     solution ← Solution()                    ▷ Start with an empty solution
3:     while boxList do                         ▷ Check if the list is empty
4:         box ← best_new_box(boxList)
5:         del_box(boxList, box)
6:         isPossible ← compute_position(box, solution)
7:         if isPossible then
8:             add_box(solution, box)
9:         end if
10:    end while
11:    return solution
12: end function
```

To achieve this, we need to define several aspects and address several issues. Firstly, the selection of the box, then how to calculate the various available positions (the corners), and finally determine which one is the optimal.

Choice of the box

The choice of the box can follow several strategies. We always take the first box in the list, and the chosen strategy dictates how the list of boxes is sorted. Here is an example of results with different strategies (see Fig. 1 & 2). Subsequently, we will choose to sort the boxes by ID.

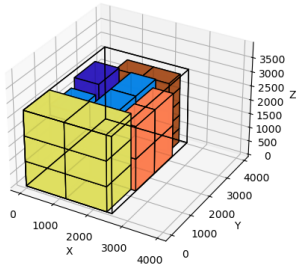


Figure 1: A solution with the strategy of the largest box first.

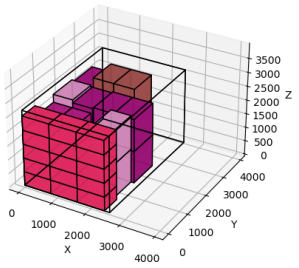


Figure 2: A solution with the strategy of the smallest box first.

Compute the corners

Calculating the available positions is the most challenging aspect of this program. It involves traversing our solution and conducting numerous checks. To do this, we compute a matrix representing the solution from a top-down view, which we'll refer to as the "height-matrix" (see Fig. 9). We then simply traverse this matrix point by point, halting at each encountered corner, and calculate the available space to the right and left of the corner to create the corresponding *Corner* objects (see Fig. 4). We quickly realize that this technique is far from efficient and that much faster solutions are possible. We will discuss this later in the reinforcement learning section (see section 3.3).

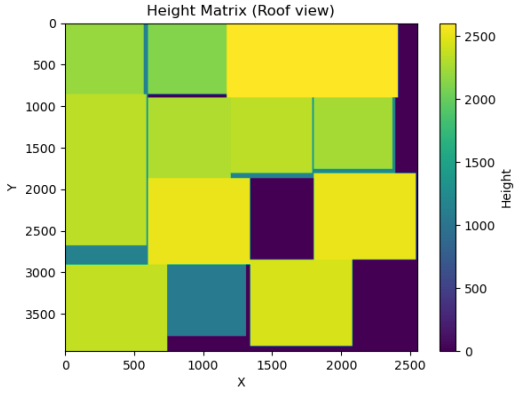


Figure 3: A representation of a solution viewed from the roof of the container. This visualization allows for a better appreciation of the heights and facilitate the calculation of *Corners*.

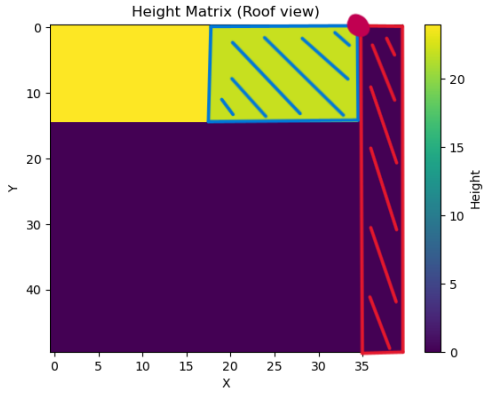


Figure 4: In this figure shows the computation of free *Corners*. We detect a corner on the purple point then we make a double-loop at right and left to compute the corresponding left-type (in blue) and right-type *Corners*

Compute the position

The choice of position can again be subject to different strategies. The most natural one is to start by placing the boxes as far back as possible (i.e., choosing the Corner with the lowest y position). Then, it is necessary to determine if it is advantageous to rotate the box 90° horizontally. The choice made here is to perform this rotation only if it reduces the total volume of the solution, aiming to achieve the most compact arrangement of boxes possible.

Algorithm 2 Compute the position

```
1: function COMPUTE_POSITION(box, solution)
2:     cornersList ← solution.get_cornersList()
3:     cornerList ←sort(cornerList, y)
4:     isPossible ← False
5:     for corner ∈ cornerList do
6:         if fit_in(box, corner) then
7:             if is_better_with_rotation(corner, box, solution) then
8:                 temp ← box.w
9:                 box.w ← box.d
10:                box.d ← temp
11:            end if
12:            box.x ← corner.x
13:            box.y ← corner.y
14:            box.z ← corner.z
15:            return True
16:        end if
17:    end for
18:    return False
19: end function
```

3.3 The reinforcement learning

Ant Colony algorithm

Now that we have an algorithm capable of generating solutions, we need to define two parameters before using machine learning. The first is the evaluation function, which takes a solution as input and returns a quantifiable value to optimize (either maximizing or minimizing). For this, several choices are possible, as discussed in the Optimization Criteria section (see section 2.3). Here, we will simply use the number of boxes in the container to maximize.

Secondly, we need to choose the decision variables, i.e., the parameters where the algorithm will have to make a choice and learn from its choices. In the algorithm presented here, it only involves the choice of the order of the boxes to be placed. Thus, the algorithm will attempt to approach an optimal sequence of boxes that maximizes the total number of boxes in the container.

For this, we will use an algorithm based on the Ant Colony algorithm (ACO). ACO is not the only possibility for solving this type of problem; genetic evolutionary algorithms are also viable solutions [vRERB15]. However, ACO is widely used in the context of sequence optimization, such as finding the shortest path, which is why this algorithm will be chosen here.

This metaheuristic is a probabilistic technique for solving computational problems inspired by the behavior of real ants. [DG97]. The principle is as follows: we assign a probability ϕ to all the boxes $i \in \{1, \ldots, n\}$ for each step $s \in \{1, \ldots, n\}$, at the start : $\phi_0(i, s) = \frac{1}{n}$, this is the pheromones. Then we create a lot of solutions by randomly choosing the boxes at each step with probabilities dictated by ϕ, these solutions are the ants. Now we have a set of solutions S which are a sequence of boxes : $\{b_0, b_1, \ldots, b_n\}$, if a solution is good we multiply the probability to choose the corresponding box at the corresponding step by a positive number ρ_D such that $\phi(b_i, i) \leftarrow \phi(b_i, i) \times \rho_D$, $i \in \{1, \ldots, n\}$. It is the deposition of pheromones. But for the other boxes probabilities we have the evaporation of the pheromones, all the probabilities are multiplied by a value ρ_E less than 1. Then we can iterate the process by regenerating a set of solutions. But we want a solution that shows an iterative way of placing the boxes, a solution will therefore be a sequence of steps $s_1, s_2, s_3, \ldots$, with each step indicating the box, whether it fits in the container, and for example, if it is placed on the right or not. We can have this kind of results :

$$(1, True, False), (7, True, False), (21, True, True), (14, False, True) \ldots$$

To resume we have the following algorithm:

Algorithm 3 Ant Colony Algorithm

```
1: function ANT_COLONY(instance, n, maxIter, maxAnt, ρ_D, ρ_E)
2:     z_best ← +∞
3:     bestSolution ←empty_solution()
4:     φ ←initiate_phi(n)
5:     for i in {1, ..., maxIter} do
6:         for ant in {1, ..., maxAnt} do
7:             newSolution ←generate_solution(φ, instance, i, maxIter)
8:             z ←evaluate_solution(newSolution)
9:             if z ≥ z_best then
10:                z_best ← z
11:                bestSolution ← newSolution
12:            end if
13:        end for
14:        φ ←manage_phi(φ, bestSolution, ρ_D, ρ_E)
15:    end for
16:    return bestSolution
17: end function
```

Efficiency

Naturally, we see that it will be necessary to perform a large number of iterations and generate a large number of solutions to converge towards a good solution. With the greedy algorithm proposed at the beginning of this document, generating a solution can take up to ten seconds. It's clear that this is far too much, and it's not reasonable to iterate hundreds of times. Thus, it is necessary to drastically improve the efficiency of the algorithm to save execution time. To do this, we focus on the calculation of available positions, which is the part of our algorithm that takes the most time. Here is the presentation of several successive ideas that have helped overcome the problem.

Idea 1 - Store the corners and compute only one side: As a reminder, to determine the available positions, we traverse the height-matrix point by point, checking if we are at a corner, and if so, we calculate the available spaces to the left and right. The first step is therefore to avoid a complete traversal of the matrix by storing the locations of the corners beforehand. When adding a box, we record the coordinates x and y, which are x, $x+w$, y and $y+d$, where w is the width of the box and d is the depth of the box. Moreover, it is actually useful to calculate only one of the two sides, as the other side is already calculated by a preceding or succeeding pass. With these improvements, represented in Figure 5, we achieve a significant improvement in time.

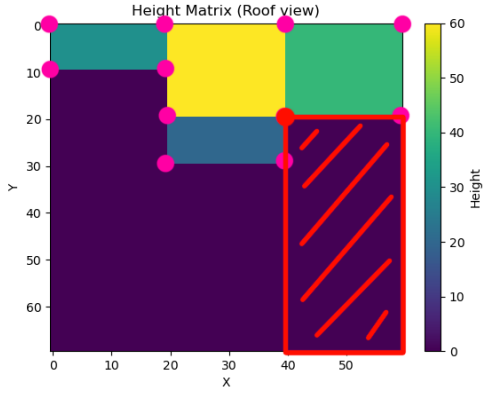
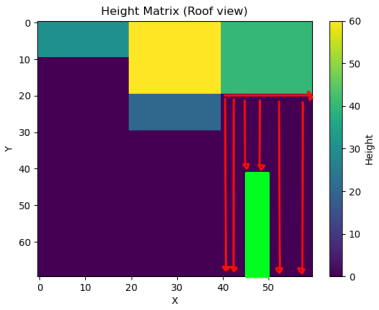
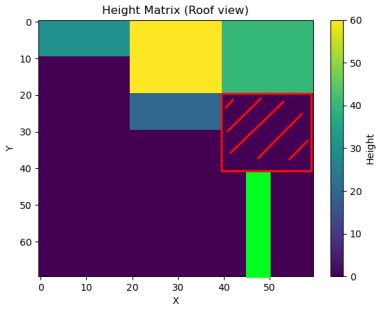


Figure 5: The purple points represent the stored coordinates and thus the possible positions of *Corners*. In red an example of a calculated *Corner*.

Idea 2 - Better calculation of the *Corner* dimensions: The second major improvement lies in how to determine the dimensions of a Corner. Up until now, it has been necessary to test the maximum width, and then for each unit of width, determine the maximum depth. Then, it's sufficient to take the minimum of these depths to obtain the largest possible dimensions of our Corner before encountering an obstacle (see Fig 6).



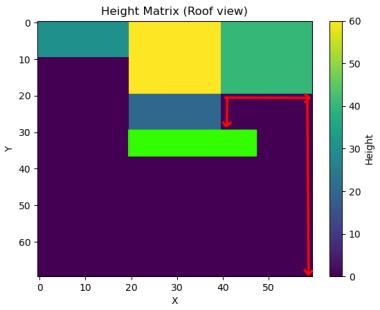
(a) Test of the depth at each unit of width



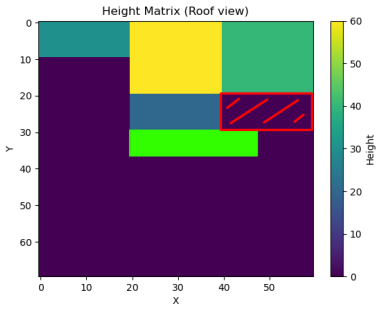
(b) Final dimensions of the calculated *Corner*

Figure 6

But in practice, it is not necessary to test the depth everywhere. With our construction system, it is not possible to have obstacles "in the middle"; they are either to the left or to the right. Therefore, it is sufficient to test only these two locations (see Fig 7).



(a) Test of the depth only at the beginning and at the end



(b) Final dimensions of the calculated *Corner*

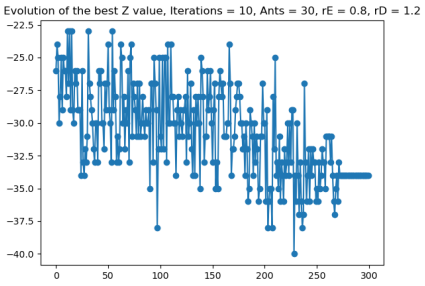
Figure 7

Idea 3 - Cython: The Cython language is a superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes. This allows the compiler to generate very efficient C code from Cython code [Cytay]. We will use Cython only on the parts of the code that are costly in terms of computation time. That requires a significant effort to type all variables and functions comprehensively, as well as the necessity of having clear and well-organized code.

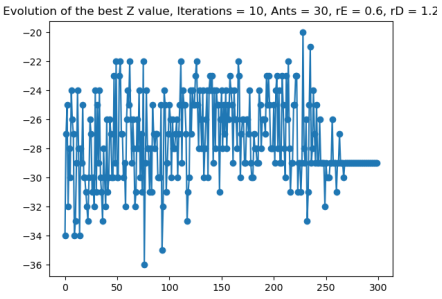
Ultimately, with all these different improvements on the chosen data structures, developed algorithms, libraries, and languages, we achieve a significant improvement that allows us to generate a solution in between 0.3 to 1 seconds compared to almost 10 seconds previously. Of course it's depend of the CPU used, this has been computed with a AMD Athlon Silver 3050U with Radeon Graphics.

3.4 Results

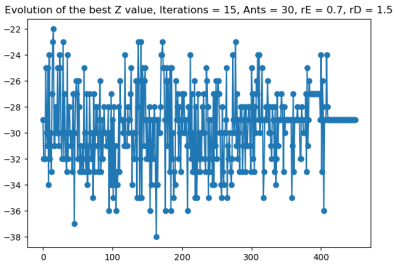
In figure 8 there is an overview of the different solutions found by the ants over the iterations with different values for the input parameters of the algorithm (number of iterations, number of ants per iteration, value of ρ_E and ρ_D). The objective here is to minimize the value on the y-axis representing the number of boxes passed in negative.



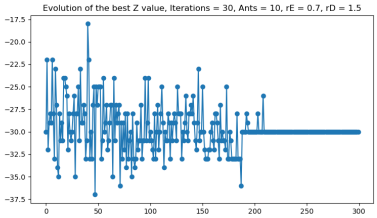
(a) $maxIter = 10$, $maxAnt = 30$, $\rho_E = 0.8$, $\rho_D = 1.2$



(b) $maxIter = 10$, $maxAnt = 30$, $\rho_E = 0.6$, $\rho_D = 1.2$



(c) $maxIter = 35$, $maxAnt = 30$, $\rho_E = 0.7$, $\rho_D = 1.5$



(d) $maxIter = 10$, $maxAnt = 30$, $\rho_E = 0.7$, $\rho_D = 1.5$

Figure 8

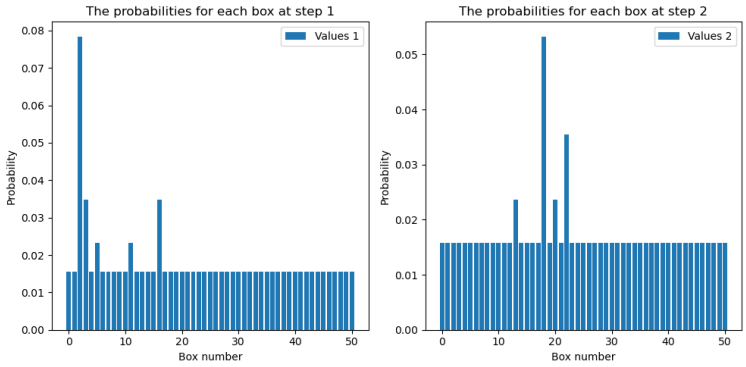


Figure 9: These graphs depict the probability for each box to be chosen at step 1 and at step 2, respectively, on the left and on the right. For an iteration with the following parameters: $maxIter = 10$, $maxAnt = 30$, $\rho_E = 0.8$, $\rho_D = 1.2$

3.5 A visualisation tool

Throughout the project, the visualization of results and solutions was a significant focus. The final objective was to observe the filling proposed by the program. Therefore, a method was developed to manually visualize, box by box, the best solution found by the learning algorithm.

4 Conclusion

4.1 Remaining challenges and solution ideas

Here are some points for improvement and discussion regarding the project and the proposed solution:

- Regarding the calculation of free spaces, it should be noted that often a free space is calculated that is actually already included in a previously calculated space. To avoid this, we would need to re-examine our list of *Corners* to detect "parent" *Corners* that already encompass the one currently being calculated. Storing the *Corners* as a simple list does not allow for efficient traversal of these *Corners*, and this would take a lot of time relative to the gain. A possible solution would be to store them in the form of a tree (binary or quadtree), where traversal would be much faster.

- If we look at the results proposed by ACO, it is clear that the convergence is not occurring as expected. The algorithm does not behave as desired every time. This algorithm has numerous small parameters that need to be meticulously tuned according to our problem and instance to obtain the expected behaviour it requires further work and careful refinement and parameterization. The ant colony algorithm assumes that in the vicinity of an interesting solution, other interesting solutions also appear. This point might be debatable regarding this problem, where even the slightest variation can completely prevent the addition of boxes and result in a radically different solution.

- The choice of the evaluation function is important to consider. Here we have considered the number of boxes in the final solution; this is a simple and easily understandable criterion that allows testing the functionality of the algorithms, which is why this criterion was chosen. However, it seems important to discuss the relevance of such a criterion in the context of our problem. Given that we have many boxes of very different sizes, the algorithm will favor the smaller boxes to fit in as many as possible. Nevertheless, it is common and natural to say that heavy and large items should be loaded first, which calls into question the relevance of such a criterion.

- Incorporating additional real-world constraints, such as handling fragile items or specific loading sequences required for practical operations, could make the solution more applicable in real logistics scenarios.

- In the interim report written in April, it was noted, "The goal of obtaining a prototype by mid-March has been achieved. However, a lot of time has been spent on certain aspects, and their difficulties were underestimated during the writing of the project plan." This underestimation disrupted the initially planned schedule. Additionally, reinforcement learning and the pursuit of efficiency, particularly the rewriting of certain parts of the code in Cython, took a considerable amount of time. This highlights the importance of establishing a plausible plan at the beginning of the project, clearly defining the needs and available resources. It has also been necessary to keep the final objective in sight throughout the project's execution and not get sidetracked by trying to accomplish too many different things.

4.2 Objectives and deliverable discussion

Finally, we have the definition of various clear and easily usable data structures, an effective and functional greedy algorithm, and a prototype of a machine learning algorithm based on an Ant Colony Algorithm that already offers a broad sweep of solutions, retaining the best one.

As planned, visualization tools were developed to observe the creation of a solution step by step. Unfortunately, some aspects, such as the ability to have a truly interactive and visual tool to observe, propose, and compare solutions, could not

be developed in time. However, a more comprehensive and interactive tool could be developed by September and several improvements could be made by then for a demonstration at the Univeristy of Jyväskylä.

Acknowledgments

I would like to extend my special thanks to my supervisor Michael Emmerich for his assistance, guidance, and engaging discussions. I also wish to express my gratitude particularly to Kaisa Miettinen and the Multiobjective Optimization Group at the University of Jyväskylä for their hospitality. Finally, my thanks also go to Timo Tiihonen for providing me with the opportunity to undertake such a project and for supervising it.

References

[BG01] Andreas Bortfeldt and Hermann Gehring. Hybrid genetic algorithm for the container loading problem. *European Journal of Operational Research*, 131:143–161, 02 2001.

[BW12] Andreas Bortfeldt and Gerhard Wäscher. Container loading problems - a state-of-the-art review. 01 2012.

[Cytay] Cython Developers. Cython Documentation. https://cython.readthedocs.io/en/stable/index.html, Reviewed the 17 of May.

[DG97] M. Dorigo and L.M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[GTMP22] Mikele Gajda, Alessio Trivella, Renata Mansini, and David Pisinger. An optimization approach for a complex real-life container loading problem. *Omega*, 107:102559, 2022.

[vRERB15] Sander van Rijn, Michael Emmerich, Edgar Reehuis, and Thomas Bäck. Optimizing highly constrained truck loadings using a self-adaptive genetic algorithm. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 227–234, 2015.

[WHS07] Gerhard Wäscher, Heike Haußner, and Holger Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.

A Appendix

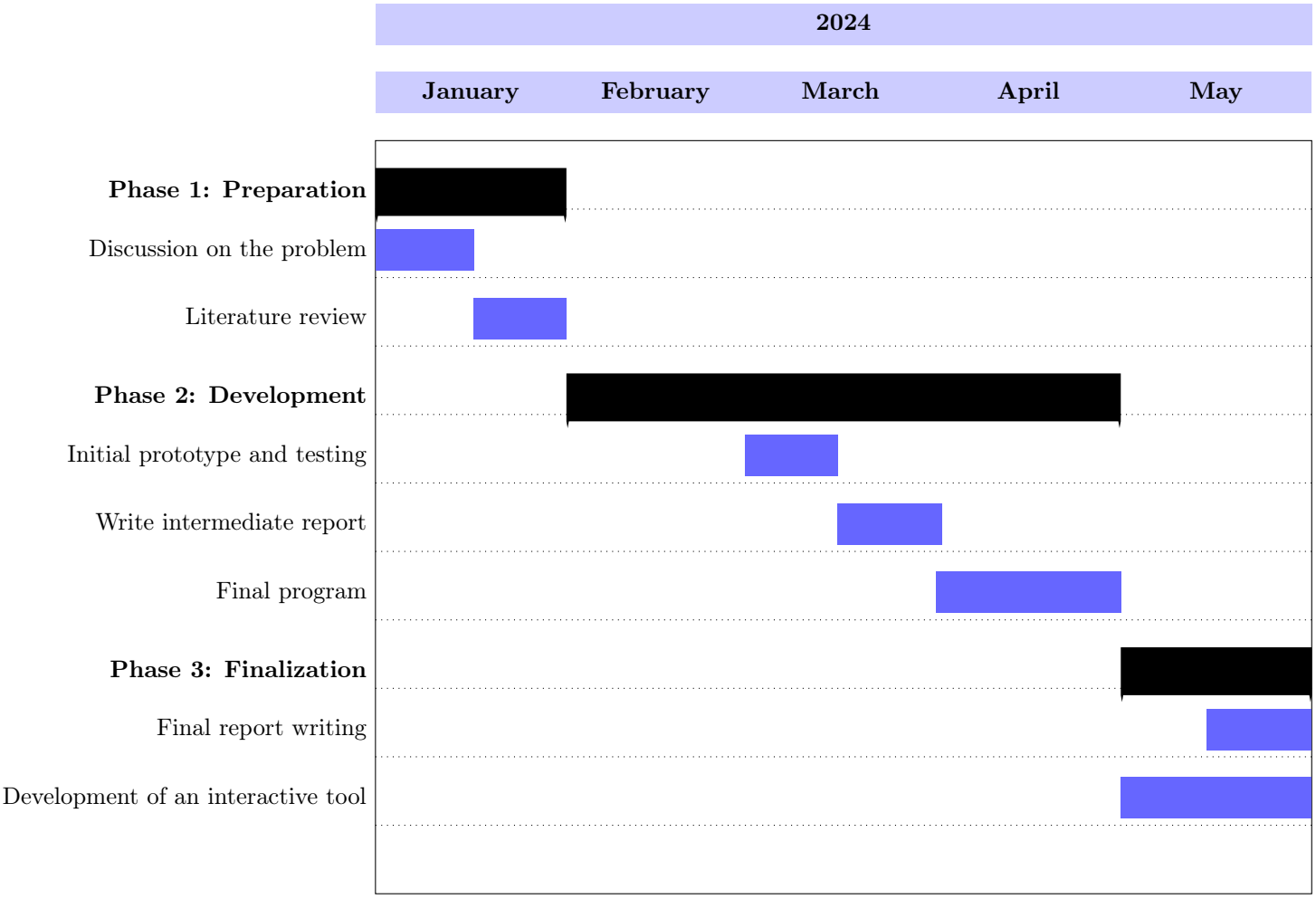


Figure 10: Original project plan