Optimizing Container Placement for a Cargo Ship

Ruben Ahrens, s3677532 Lucas de Wolff, s3672980 Shaoxuan Zhang, s3426505

Abstract

In this paper, we present an optimized container loading plan for a cargo ship with specific dimensions of 8 bays, 4 rows, and 3 tiers. Our solution aims to minimize ship instability and unloading time while carrying up to 96 containers destined for three harbors. To achieve this, we employ NSGA-III, a genetic algorithm designed for multi-objective optimization. By exploring the Pareto front, our approach generates stable and efficient solutions. Our findings offer practical applications and improved container loading strategies for cargo ships.

1. Introduction

1.1. Research Background

Most real-life problems must be solved by optimizing certain objectives, such as maximizing profit or outcome, minimizing cost or time, etc. Meanwhile, there are also different constraints, according to the problems. The optimization process of objectives is required to follow the constraints. For multi-objective problems, finding the optimal value or setting for all objectives is important because only optimizing a single objective may compromise the other ones (Fonseca & Fleming, 1995). Therefore, Multi-objective Optimization (MOO) problem is crucial for real-life decision-making.

Mathematical Programming (MP) is a tool for formalizing the solution of MOO problems. MP defines the variables of the problem and builds objective functions and constraints mathematically which can be programmed easily. The goal of MOO is to find the decision space, which is the space with all possible solutions, and optimize within that space. Nowadays, there exists a plethora of different optimization methods, each valuable in their own way. Such as Pareto optimization, which aims to find the Pareto front in decision space and the corresponding efficient set of the points (Knowles, 2002).

1.2. Problem Overview

In this report, we will be investigating the container loading configuration problem for a cargo ship routed for several harbours, with certain constraints of the containers and ship weighting structure. The goal for the problem is to find the best solution for loading the containers in a way that can not only keep the ship balanced with as many containers as possible but also has the easiest unloading process for each harbour.

1.3. Report Structure

The rest of the report is structured as follows: In Chapter 2, we introduce the real-life problem, container configuration optimization for container ships, and use a mathematical programming scheme to formalize the variables, objectives and constraints. In Chapter 3, with the help of Pymoo (Blank & Deb, 2020), we build the problem optimization structure and give an algorithmic solution to the problem. We discuss important matters and give a summary of the optimization and decision-making for the container configuration problem in Chapter 4.

2. Problem Formalization

The container configuration optimization problem is about a container ship MS Leiden. The ship has a fixed route from Rotterdam, the Netherlands to Copenhagen, Denmark. There are three intermediate stops: Hamburg, Germany, Kiel, Germany and Aarhus, Denmark. As shown in Figure 2. In this problem, the ship departs from Rotterdam, carrying containers destined for Hamburg, Aarhus and Copenhagen. Only in Rotterdam, containers are loaded. only in Hamburg, Aarhus and Copenhagen, containers are unloaded. In Kiel, no containers are loaded or unloaded.

The structure of the container ship is demonstrated in Figure 3. The ship is 8 containers long (bays), 4 containers wide (rows), and 3 containers high (tiers). Figure 1 illustrates the bays, rows, and tiers of the container ship. For simplicity, all the containers loaded on the ship are of the same size. As can be seen in Figure 4, unloading is done with a crane and can only be done by taking the top container off the ship. Unloading times, therefore, rely on the configuration of containers, as a container that is stuck below a container destined for another port will take longer to unload.

The objectives of the problem are:

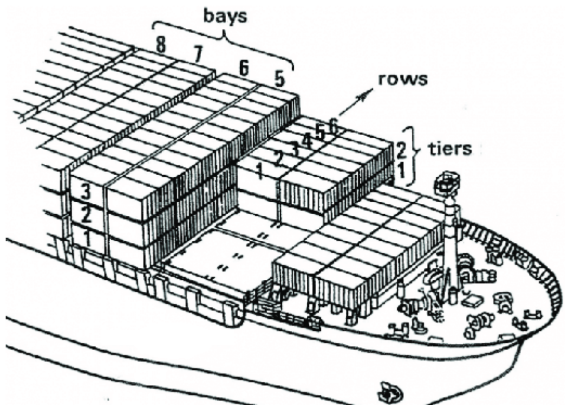

Figure 1. Layout of a container ship. In our problem, we have a ship with 8 bays, 3 tiers, and 4 rows.

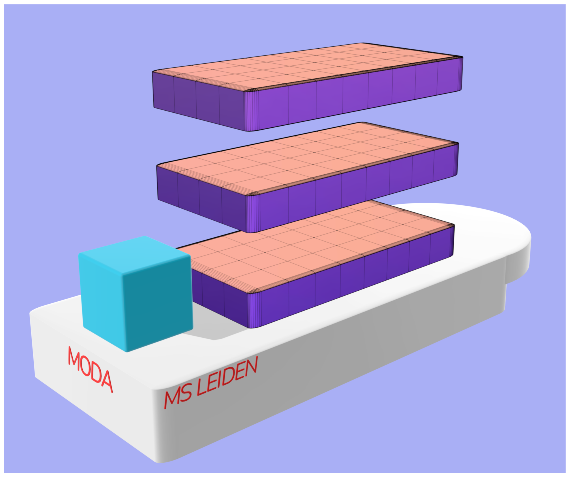

Figure 3. Container ship demonstration

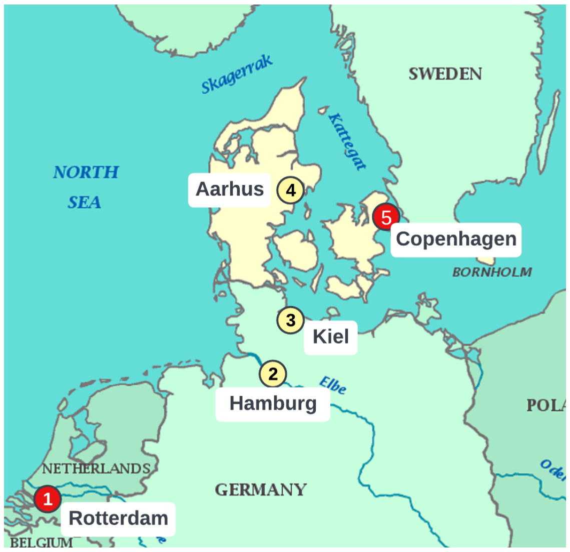

Figure 2. Container Ship Route

Figure 4. Harbour crane unloading containers from a ship (picture from `espo.be`)

- **Balanced ship** The distribution of the containers highly affects the gravitational balance of the ship. The goal is to find an evenly distributed configuration so that the gravitational centre can be as close as possible to the centre of the container section. This will guarantee the ship safe from dangerous rolling and tilting. Rolling is when a ship is not straight with respect to the line along the length of the ship. Tilting is when a ship is not straight along the width of the ship. To prevent rolling and/or tilting, we propose an objective function that minimizes the imbalance by using the weight and position of each container to calculate the centre of gravity of the ship. The centre of gravity is optimal in the middle of the bays and rows as illustrated in figure 5.

- **Easy unloading solution** Since all the containers loaded in Rotterdam are destined for different harbours, based on the unloading method of the container, it is important to place the containers which are destined for earlier harbours above those for later harbours so that we don't have to move a lot of other containers when unloading.

- **Maximised containers** To maximise the profit, the number of containers on the ship should be maximised. However, in our implementation, we did not add this as a third objective. Instead, we made sure our solutions always included all containers.

Additionally, there are some constraints on the problem as well:

- **Non-overlapping containers** Each spot of the cargo ship can only be occupied by one container.

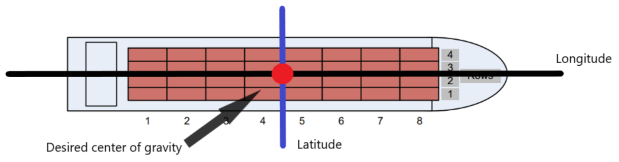

Figure 5. Gravitational centre of the ship

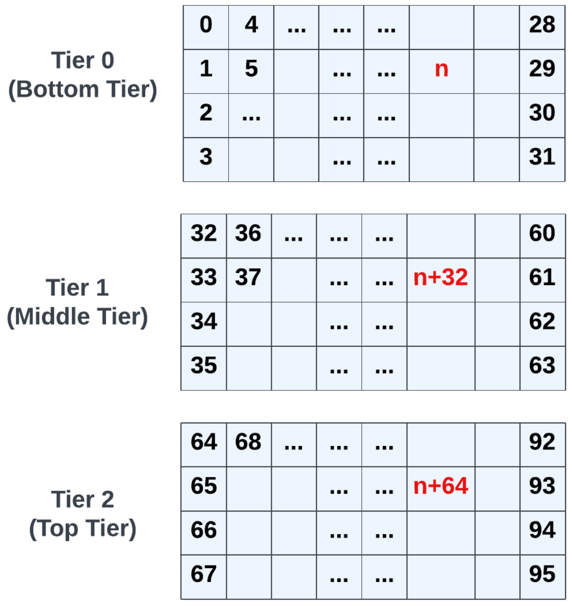

Figure 6. Container position ID demonstration on the ship, from 0 to 95

- **Non-floating containers** Since the ship has 3 tiers, the container that is placed at the middle tier and the upper tier must have another container at tiers below as support.
- **Weight limit of the container** Each container weighs differently, in reality, it is dangerous to put a much heavier container onto another container. In this problem, we specify a weight difference δ (kg) to evaluate container placement. If container A weighs δ (kg) more than container B, A cannot be placed on top of B.

Using the mathematical programming method, we formulate the above objectives and constraints as follows:

- **Variables**:

  1. Number of containers n: $n \in [1, 96]$ and $n \in \mathbb{N}$
  2. Harbour Vector $\mathbf{h}$: $|\mathbf{h}| = n$ and $\mathbf{h}_i \in \{0, 1, 2\}$ indicating Hamburg, Aarhus and Copenhagen, respectively.
  3. Container Weight Vector $\mathbf{w}$: $|\mathbf{w}| = N$ and $\mathbf{w}_i \in \mathbb{R}^+$.
  4. Position Vector $\mathbf{p}$: $|\mathbf{p}| = n$, $\mathbf{p}_i \in [0, 95]$ and $\mathbf{p}_i \in \mathbb{N}$, To each container we assign a position on the ship, as shown in Figure 6

- **Objectives**:

  1. Evenly-distributed:

$$long(p) = \frac{|\sum_{i=0}^{n} w_i \cdot (p_i \text{ mod } 32 \ \% \ 4 - 3.5)|}{\sum_{i=0}^{n} w_i} \tag{1}$$

$$lat(p) = \frac{|\sum_{i=0}^{n} w_i \cdot (p_i \text{ mod } 4 - 1.5)|}{\sum_{i=0}^{n} w_i} \tag{2}$$

  Where w is a vector containing the weight of each stack of containers (summed over the tiers).

$$f_1(p_1, ..., p_n) = \sum^{h} \big(long(p_h) + lat(p_h)\big) \rightarrow min \tag{3}$$

  Where $long$ is the magnitude of the difference in weight of the containers along the length of the ship. Whereas lat is the same for the width of the ship. Since it is possible for the container positions to change in tier every time we reach a new harbour, we repeatedly calculate the instability values after each harbour. For this, we define p_h to be the configuration where the containers of previous harbours have been removed. $\mathbf{h}$ changes accordingly. This final value should be minimized across all rows and bays in f_1.

  2. Unloading time:

$$f_2(p_1, ..., p_n) = \frac{totalUnloadingTime}{n} \rightarrow min \tag{4}$$

  Where *totalUnloadingTime* is the number of containers across all harbour visits that have to be lifted to reach the container underneath it.

- **Constraints**: Subject to:

  1. Non-overlapping containers:

$$p_i \neq p_j \text{ for } i = 1, ..., n, \ j = 1, ..., n, \ i \neq j \tag{5}$$

  2. Non-floating containers:
  Let p be the set of positions defined by $\mathbf{p}$

$$\sum_{i=1}^{n} \begin{cases} 0, & \mathbf{p}_i < 32 \\ -1, & \mathbf{p}_i - 32 \notin p \\ 0, & \text{otherwise} \end{cases} = 0 \tag{6}$$

3. Weight limit: much heavier containers cannot be on top of others, the threshold is δ we define $\delta = 2000$:

$$\sum_{i=1}^{n} \begin{cases} 0, & \mathbf{p}_i < 32 \\ -w_i + w_j + \delta, & \mathbf{p}_i \geq 32 \wedge w_i \leq w_j + \delta \\ & \text{where } j \text{ is the index of the} \\ & \text{container in the layer below} \\ 0, & \text{otherwise} \end{cases} = 0 \tag{7}$$

3. Algorithmic Optimization

3.1. Data generation

For each of the experiments, we generated our own data. The number of containers is defined by the parameter n. Each container is then assigned a weight between 2000 and 40000, and a destination (harbour number). Because the optimization is highly dependent on the generated data, we made sure to use the same data across our experiments for the same n.

3.2. Implementation

To optimize for the objectives and constraints defined in the previous chapter, we chose the Non-dominated Sorting Genetic Algorithm III (NSGA-III), which is an evolutionary algorithm based on reference directions and is the improved version of its predecessor NSGA-II (Deb & Jain, 2013) (Jain & Deb, 2013). The goal of this algorithm is to explore the Pareto front in the objective space by using a non-dominated sorting technique.

During the selection and reproduction process, NSGA-III favors solutions that are closer to the reference points, which guides the exploration of different regions of the Pareto front. The mutation and crossover operators are applied to the selected solutions, introducing genetic variation and recombination to enable the exploration of new solution spaces. The selection operator then determines which solutions are retained for the next generation based on their fitness and diversity, favoring superior and diverse solutions to maintain the population's quality.

By iteratively repeating these steps over multiple generations, NSGA-III progressively improves the population, searching for optimal solutions on the Pareto front. The algorithm continues until a termination condition is met, such as reaching a maximum number of generations or achieving a desired level of convergence.

We initialize the algorithm with a randomly generated population of size 100 and 12 well-spaced reference points, which were generated following the Das and Dennis's structured approach (Das & Dennis, 1998). In our implementation, we used the IntegerSampling method to generate the initial population. While it is likely for this method to generate infeasible solutions, it is great for exploration. We actually found through experiments that a valid starting generation[1], resulted in worse final solutions.

3.3. Experiments

Figure 7 shows the evolution plots of the two main objectives: stability (balance) and unloading time for the experimental run with 50 randomly generated container data with destined harbours and weights. From the figure, it can be observed that after 40000 function evaluations, both objective functions' values of the population converge to a small value. The green vertical line shows the moment where half of the population has reached a valid configuration that doesn't violate any constraints.

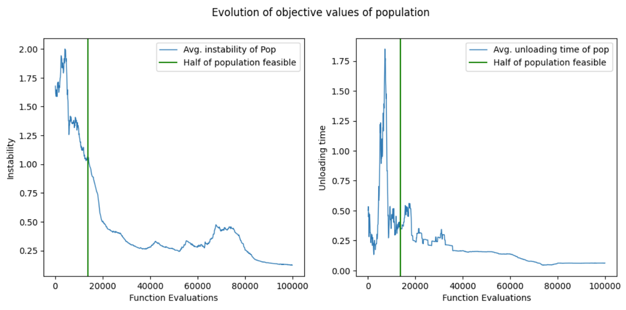

Figure 7. Change in average objective values through generations. The green line indicates the first moment where half of the population is feasible.

Often times, it is difficult to determine the Pareto front mathematically. One way to approximate the Pareto front is by running multiple experiments while keeping all solutions. Then after, one can extract the non-dominated set of points out of all found solutions. In figure 7, we have plotted all solutions found after 100 simulations. For each simulation, we used 50 containers with a fixed weight and harbour configuration (which were randomly generated earlier). This was then run for 1000 generations. As explained in the previous chapter, f_1 represents the balance of the ship and f_2 represents the unloading time of the containers. Both objective functions need to be minimized. Therefore we would want the solution to be as close to the origin as possible in the objective space. An interesting pattern emerges within the objective values of the found solutions, where the values of objective 2 contain gaps. This pattern is likely caused by the discrete nature of objective 2. Essentially objective 2 is an integer function, where we count the number of times

[1]A valid generation was generated by sorting the containers on weight and placing them from heaviest to lightest randomly on the ship.

we need to lift a different container to retrieve the container underneath. We then divide this by the number of containers to achieve the average unloading time per container, however, this will still cause the discrete like behaviour in the objective space. In figure 9, we extracted the non-dominated solution points and plotted them to visualise the Pareto front approximation. Interestingly, the algorithm was able to find a solution that was very close to the ideal point (origin) within 1000 generations. This can be explained due to the relatively small value of n (= 50), compared to the total space on the cargo ship.

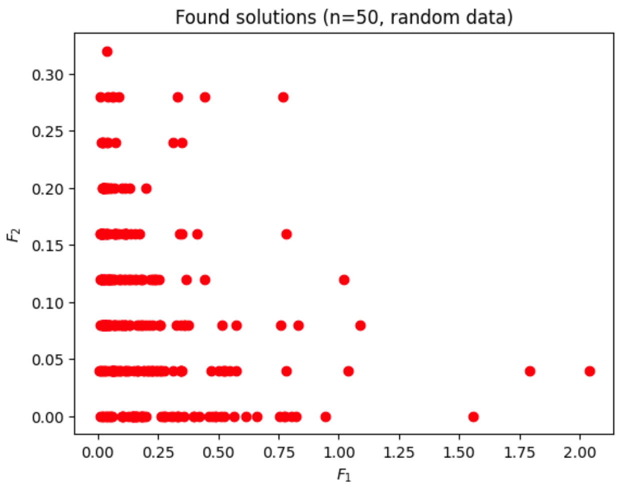

Figure 8. Found solutions after 100 simulations with 50 containers. F_1 represents the stability objective, F_2 represents the unloading time

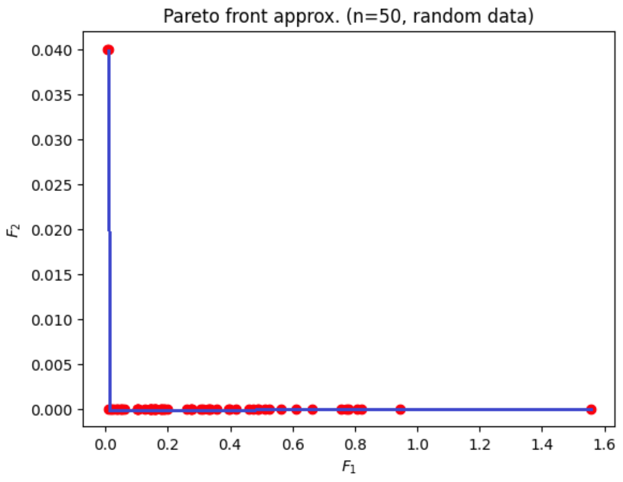

Figure 9. Approximated Pareto front of the problem. F_1 represents the stability objective, F_2 represents the unloading time. The red points are the non-dominated solutions, and the blue line is the approximated Pareto front.

As a final experiment, we used a larger value for n (= 80). The optimization was again simulated for a 100 repetitions. However, because it was considerably more difficult to find valid solutions within a 1000 generations, we increased the number of generations to 2000. The results are shown in figure 10 and 11. Note that this time, no solutions close to the ideal point are found. It might be possible that the approximation of the Pareto front is inaccurate and we need to increase the number of generations even more.

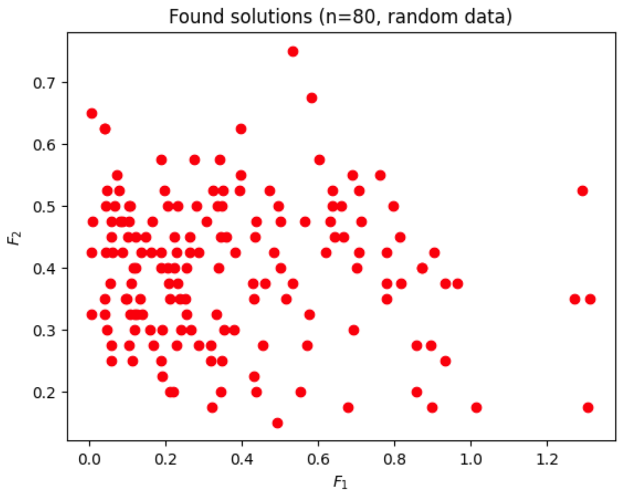

Figure 10. Found solutions after 100 simulations with 80 containers. F_1 represents the stability objective, F_2 represents the unloading time

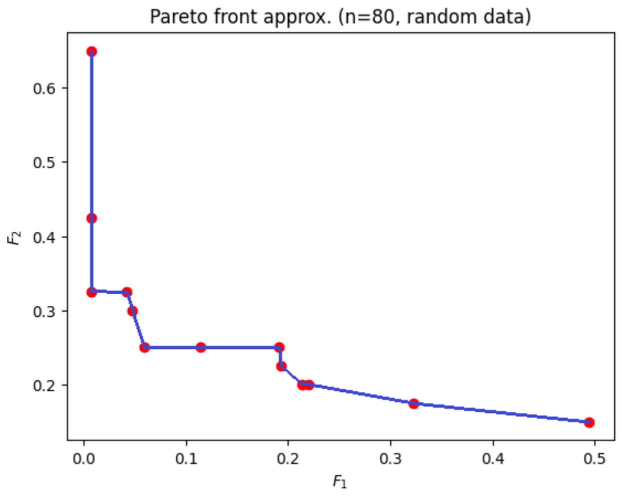

Figure 11. Approximated Pareto front of the problem with 80 containers and random data. F_1 represents the stability objective, F_2 represents the unloading time. The red points are the non-dominated solutions, and the blue line is the approximated Pareto front.

3.4. 3D Rendering of solutions

To demonstrate the optimization process, we used Minecraft to render the found solutions after each generation. We used the *GDMC HTTP interface mod (v1.0.0)* as well as the Python package *GDPC* (Salge et al., 2018) to establish a connection between our Python program and the Minecraft world. We could then programmatically alter worlds and load our found solutions onto the Minecraft cargo ship (Finn-McShipbuilder, 2019).

During the evolution, the configuration is saved every 50 generations. Figure 12 shows the first generation of the evolution process for 50 containers. The yellow containers are destined for Hamburg, the green ones are for Aarhus and the blue ones are for Copenhagen. To demonstrate the weight constraints, We set three weight levels and represent them in different shades/textures of the containers in Minecraft. The darker the colour is, the heavier the container is.

Looking at this figure, some containers appear to be levitating in the upper tiers, with no containers supporting them in lower tiers. However, in generation 1000, which is shown in Figure 13, the algorithm finds a feasible solution that satisfies both the balancing objective and the short unloading time objective without violating the weight constraint.

What we find during the generations is that the algorithm tends to move the containers destined for the first harbour (Hamburg, yellow colour in the figure) to the middle part of the ship. By doing this, when the cargo ship arrives in Hamburg and unloads those containers, the rest containers are still evenly distributed on both sides of the ship. The balance objective can be reached without moving more containers. The strategy is similar to the green ones (Aarhus). Additionally, in most cases, the containers that arrive at the destination earlier are placed on top of those that arrive later.

We also visualized the population every 50 generations of NSGA-III. Using random data with $n = 70$. The evolution of the optimal solution can best be seen in an animated GIF.[2]

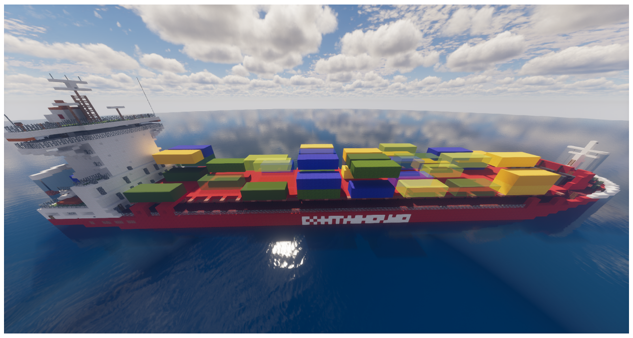
Figure 12. Minecraft evolution visualization for 50 containers with randomly generated destined harbours and weights: Generation round 1, the colours of different containers represents different destination harbours, the shades of the containers depict the weight of the containers, the darker the colour is, the heavier the container weights

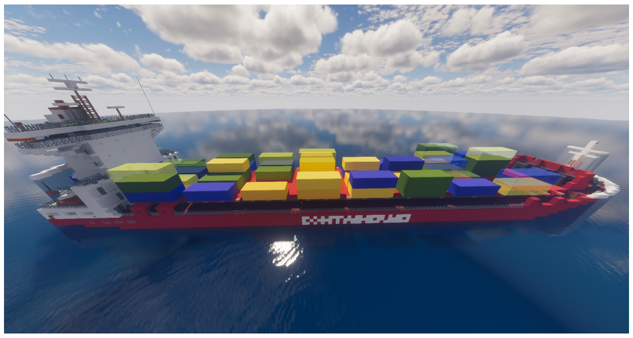
Figure 13. Minecraft evolution visualization for 50 containers with randomly generated destined harbours and weights: Generation round 1000, the colours of different containers represents different destination harbours, the shades of the containers depict the weight of the containers, the darker the colour is, the heavier the container weights

4. Discussion & Conclusion

In this report, we focussed on the container configuration problem for the cargo ship and provided an optimized solution that maximises the stability (balance) of the ship while minimising the total unloading time of the containers. We went from a general problem introduction to a proper mathematical programming formulation and implemented the objective functions and constraints using the Pymoo package. Through the use of the genetic multi-criteria optimization algorithm NSGA-III, we managed to find valid solutions for this problem. Experiments showed that the provided solutions satisfy the stability and unloading time objectives, while also avoiding constraint violations. Minecraft-based 3D rendering of the solutions is used to showcase the process of evolution. The final solutions make sense intuitively and have shown interesting features.

Although we intended to use the DESDEO (Misitano et al., 2021) library, we found that the Pymoo library provided the flexibility needed for this problem. DESDEO could not handle discrete integer values as well as Pymoo could. The constraints were more often violated in DESDEO and it had trouble finding feasible solutions.

To conclude, we want to give our thoughts on some future improvements. First, because we force the loading plan to consider all containers, our solutions may not always be stable and/or have efficient unloading plans. In the future, it might be good to add a third objective that maximises the number of containers taken, i.e. make the number of

[2] GIF of evolution generations

containers variable. Moreover, it would be beneficial to introduce an additional constraint that prevents the instability from dropping below acceptable thresholds. It might be interesting to explore different initial population strategies and to tune algorithm parameters. Finally, the problem could be modified to make the problem more useful in real-world scenarios. For example, when a container is lifted to reach a container underneath, instead of moving the top container back to its original row and bay, we could change its location to minimize instability and future unloading times.

Contributions

Working on this project has been a team effort and all group members contributed equally.

References

Blank, J. and Deb, K. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

Das, I. and Dennis, J. E. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM Journal on Optimization*, 8(3):631–657, 1998. doi: 10.1137/s1052623496307510.

Deb, K. and Jain, H. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE transactions on evolutionary computation*, 18(4):577–601, 2013.

FinnMcShipbuilder. Containerships vii, 2019. URL `https://www.planetminecraft.com/project/containerships-vii/`.

Fonseca, C. M. and Fleming, P. J. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary computation*, 3(1):1–16, 1995.

Jain, H. and Deb, K. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. *IEEE Transactions on evolutionary computation*, 18(4):602–622, 2013.

Knowles, J. D. *Local-search and hybrid evolutionary algorithms for Pareto optimization*. PhD thesis, University of Reading Reading, 2002.

Misitano, G., Saini, B. S., Afsar, B., Shavazipour, B., and Miettinen, K. Desdeo: The modular and open source framework for interactive multiobjective optimization. *IEEE Access*, 9:148277–148295, 2021. doi: 10.1109/ACCESS.2021.3123825.

Salge, C., Green, M. C., Canaan, R., and Togelius, J. Generative design in minecraft (GDMC). In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. ACM, aug 2018. doi: 10.1145/3235765.3235814. URL `https://doi.org/10.1145%2F3235765.3235814`.