VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

SOFTWARE ENGINEERING STUDY PROGRAM

Optimization of marine container loading

Jūrinių konteinerių krovos optimizavimas

Master's Thesis

Author: Pavlo Ivashchenko ___________

Supervisor: Assoc. Prof. Dr. Algirdas Lančinskas ___________

Reviewer: Assist. Prof. Dr. Linas Litvinas ___________

Vilnius – 2024

Abstract

This work presents a tower-building algorithm for packing boxes of different sizes into one container. The essence of the procedure is to generate a set of towers, which consist of different boxes, using a genetic algorithm, and then place these towers on the floor of a container, also using a genetic algorithm. Several heuristic algorithms have been developed for the genetic algorithm to fill the bottom of a container with towers of boxes. Heuristic mutation and crossover operators have been developed. The crowding technique was applied. The heuristic algorithm was also parallelized for efficient execution on many CPU cores. Also, stability and orientation constraints are included in the work. In addition to this, a script was developed to visualize loading, which can operate in several modes. The heuristic algorithm was tested on 2 datasets and compared with algorithms from other authors. This showed very good results that were above other metaheuristic approaches.

Keywords: Genetic Algorithm, Container loading problem, Tower building.

Santrauka

Šiame darbe pristatomas bokštų statymo algoritmas, skirtas įvairių dydžių dėžių pakavimui į vieną konteinerį. Iš skirtingų dėžių pasitelkus genetinį algoritmą sudaromi bokštai. Tada bokštai yra išdėstomi konteinerio dugne taip pat pasitelkus genetinį algoritmą. Keli heuristiniai algoritmai buvo sukurti genetiniam algoritmui, kuris užpildytų konteinerio dugną bokštų dėžėmis. Darbe apibrėžti heuristiniai mutacijos ir kryžminimo operatoriai. Dar buvo pritaikyta perpildymo technika. Heuristinis algoritmas buvo paralelizuotas efektyviam vykdymui su dideliu skaičiumi procesorių branduolių. Darbe įtraukti stabilumo ir orientacijos apribojimai. Buvo sukurta konteinerių pakrovimo vizualizacijos programa, kuri gali veikti keliuose režimuose. Heuristinis algoritmas buvo išbandytas su 2 duomenų rinkiniais ir palygintas su kitų autorių algoritmais. Palyginimas parodė labai gerus rezultatus, viršijančius kitus metaheuristinius metodus.

Raktiniai žodžiai: Genetinis algoritmas, Konteinerio pakrovimo problema, Bokštų statymas.

TABLE OF CONTENTS

INTRODUCTION ... 4
1.LITERATURE REVIEW ... 7
1.1. Problem description and types ... 7
1.2. Constraints ... 8
1.3. Methods for solving the problem ... 10
1.3.1. Exact and approximation algorithms ... 10
1.3.2. Heuristic algorithms ... 11
1.3.3. Wall building algorithms ... 11
1.3.4. Layer building algorithms ... 14
1.3.5. Cuboid building algorithms ... 16
1.3.6. Metaheuristic algorithms ... 17
1.4. Test sets ... 23
1.5. Summary of literature ... 24
2. GENETIC TOWER-BUILDING ALGORITHM FOR SOLVING CONTAINER LOADING PROBLEM ... 26
2.1. Solution concept ... 26
2.2. Tower heuristic ... 28
2.3. Container floor filling heuristics ... 30
2.3.1. Recursive floor filling heuristic ... 30
2.3.2. Packing floor filling heuristic ... 31
2.4. Genetic algorithm ... 33
2.4.1. Genetic Algorithm main loop ... 34
2.4.2. Parallelization ... 36
2.4.3. Crossover and mutation operators ... 36
2.4.4. Heuristic mutation and crossover ... 38
2.5. Loading visualization ... 38
2.6. Test of Tower-building GA ... 40
RESULTS AND CONCLUSIONS ... 45
REFERENCES ... 46

Introduction

Marine container loading is a special case of the better known and widely studied container loading problem. Most often, the task is to arrange 3-dimensional boxes in containers, which are most often also boxes of a larger size.

An effective solution to this problem is particularly important, as it directly affects the efficiency and speed of delivery of goods by land and water. Every year the volume of goods transported by sea grows by 2 percent; over the past 20 years, volumes have increased by two and a half times.

In addition to the economic importance of the issue, there is also an environmental aspect. Over the past 10 years, the volume of carbon dioxide emissions associated with the transportation of containers by sea has increased by 11 percent. Efficient use of containers during transportation leads to a decrease in the required number of shipments, which leads lower amount of emissions. Also, it is important to mention that during the resolution of the main task, indirect subtasks can also be solved, for example, ensuring the stability of the cargo, or the segregation of certain types of products, which increases the safety of transportation. There are also algorithms that load containers in such a way that it would be as simple as possible for workers to implement this load into reality, so the algorithm indirectly facilitates the work of workers.

Even though the problem is very relevant, it has received much less attention than its 2-dimensional counterpart. Although in recent years, the number of studies has begun to grow, but most of them offer new algorithms and approaches to solving the problem often without considering the limitations of real life, which makes them mostly inapplicable or in need of improvement.

The problem is that specific life circumstances lead to certain restrictions, which cannot be ignored. The most common of these are restrictions on weight, stability, orientation of goods and many others.

In addition to the limitations, there are also different kinds of this problem, with different input data types, and different algorithm goals. Most studies only work on the 3-4 main types of problem, while others are almost completely ignored.

The goal of this work is to develop an algorithm that can compete with other algorithms in terms of recycling space in the container, while it will take into account how feasible the finished result will be in reality. Since containers are usually loaded by forklifts, it would be most logical to arrange towers of boxes inside the container, so loading will require minimal manual effort and

can be automated. In addition, the use of towers simplifies the problem to 2D, where it is need only to place the bases of the towers on the floor, as close to each other as possible.

When working with CLP, two approaches are usually used: exact and approximate. Precise methods guarantee an optimal result, although they can spend a lot of time searching for it, on the other hand, approximate methods can give a very good result, close to the optimal one, in a relatively short period of time. In other works, similar techniques are used, but they create either cubes or layers or walls from boxes, which greatly limits the possible field of solutions, the hypothesis is that when using towers, the field of solutions will be larger, since they can be placed more flexibly, which should lead to better results within the use of space.

For approximate algorithms, there are heuristic and metaheuristic methods. One of the most commonly used metaheuristic algorithms is the genetic algorithm, which is based on the principles of evolution and applies the same concepts and mechanics to gradually improve the result. One of the main advantages of such algorithms is that they can cope well with heterogeneous problems, that is, those in which there are objects with different properties; for CLP, such objects are boxes; the more types of boxes, the more heterogeneous the problem is.

The object of research in this work is a tower-building genetic algorithm to optimize the solving of the marine container loading problem.

Thus, the goal of the work will be to study and create a genetic algorithm with a new approach to building towers from boxes.

In order to achieve the goal, the following tasks have been formulated:

- study the problem in detail, review and compare existing approaches, and refine the concept of the algorithm, taking into account current research.
- Conceptualize the CLP problem in terms of genetic algorithm, develop the structure of chromosomes, crossover and mutation operations, and heuristic algorithms for creating towers, and for placing them on the floor of a container.
- Create a genetic algorithm, that will use created heuristic algorithms and data structures to solve CLP.
- Make sure that the algorithm works properly, by visualizing the results of the algorithm.
- Test the algorithm by comparing it with the algorithms of other authors.

- Analyze the results, and find possible drawbacks and improvements in the algorithm.

The work can be considered successful if the tasks are completed, and the algorithm shows its effectiveness in tests, and the algorithm must produce a result that will be easy to implement in practice, including the constraints that are usually associated with CLP.

1. **Literature Review**

1.1. Problem description and types

The problem is summarized as follows, given 2 sets of objects:

- Large objects(input)
- Small objects (which need to be placed in large ones, output)

The task is to group small objects and assign them to large ones, so that a group of small objects does not go beyond the boundaries of a large one, that is, it fits completely into it, while maximizing the value of small objects (output maximization) or minimizing the number of large ones (input minimization) [WHS07].

Even though the problem sounds quite simple, and humans often solve this problem in everyday life, there are many types of it. In literature, the types of problems are distinguished mainly by the number of different types of small and large objects. Depending on the number of different types, the following categories are distinguished [WHS07]:

- Identical – when there is only one type of object.
- Weakly heterogeneous – small number of different types of objects.
- Strongly heterogeneous - big number of different types of objects.

Based on these classification Bortfeldt A. and Wäscher, G. distinguish the following main types of problems [BW13]:

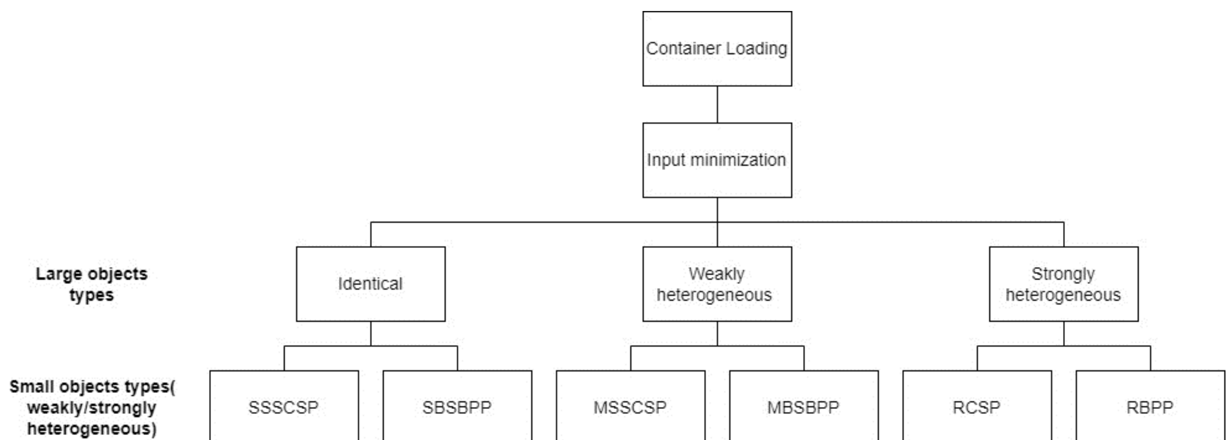

Figure 1. Input minimization CLP types

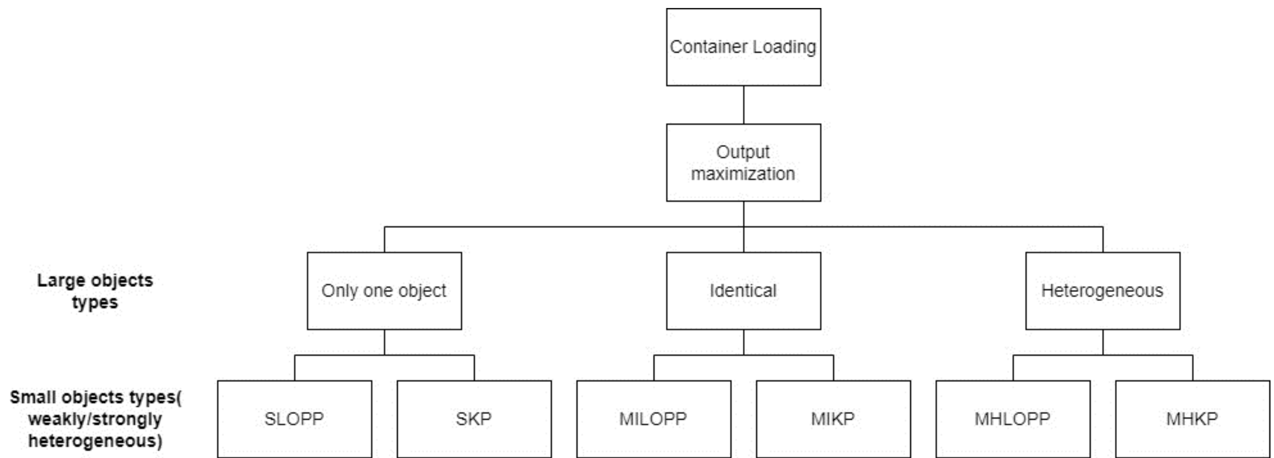

Figure 2. Input minimization CLP types

Also, on Figure 1 and Figure 2, it is not highlighted, but there is 1 more type, which is considered in a relatively small number of works, this is the Identical Item Packing Problem (IIPP) - pack the maximum number of identical small objects into 1 large. Bortfeldt, A., & Wäscher also distinguish separate types depending on the number of dimensions [BW13], but since this work is only focused on 3 dimensions a detailed study of these classifications is not required.

Since containers are usually loaded with goods destined for one destination, it is easier to load one container at a time, otherwise it will be needed to handle different options when goods may go to different places, Therefore, it would be more logical to choose a problem type with one object only. The number of types of goods can be different depending on the specifics of the company's activities, so it is difficult to single out a specific type here. Thus, CLP is best characterized by 2 types of tasks SLOP and SKP, in which different boxes are loaded into one container.

1.2. Constraints

It is also important to analyze the different types of constraints that are encountered in literature in order to determine which constraints should be considered when solving the problem specifically for marine containers.

In addition to the 3 main restrictions that come from the formulation of the problem:

- All small objects must be placed inside a large one.
- Small objects cannot intersect each other.
- Small objects should be placed in such a way that their sides are parallel to the sides of a large object.

Bortfeldt, A. and Wäscher G. distinguish the following main types of constraints [BW13]:

1. Weight limit
2. Weight distribution
3. Loading priorities
4. Orientations
5. Stacking
6. Complete shipping
7. Allocation
8. Positioning
9. Stability
10. Complexity

Let's consider only those that make sense in terms of loading marine containers.

Orientation - is the most encountered constraint in literature. The essence is to limit the rotation of some of the goods [BW13]. If an ordinary product has 6 sides on which it can be placed, then in practice there is often a limitation that some items cannot be placed upside down or on their sides, which limits the number of positions.

The stability constraint is the second most popular constraint in the literature, which is quite logical because in the real world, it is mandatory to consider the force of gravity and calculate everything in such a way that the boxes in the container do not fall but stand stably. Usually, this restriction is simplified to the form when the box must stand either on the container's floor or on the other boxes. Most often, 100 percent support for the box is implemented, which means that it must completely stand on some surface, sometimes incomplete support is also considered, but this option is not always reliable, because it depends on the center of gravity of each of the boxes.

Thus, we have identified 2 restrictions that are the most critical for CLP, some other restrictions would also be nice to implement, but they are not critical, and rather improve the quality of the load, or the ease of its implementation for workers.

The importance of considering these constraints is also in the fact that in most studies the problem of container loading is considered in isolation from the real world, and even if some limitations are considered, it is often not enough for this algorithm to be really used in the industry. This problem is highlighted by many studies [BW13], and it is noted that it would be nice to determine the industrial limitations, as well as to make test sets that are close to the data that is

encountered in reality. That is why it is crucial to consider these limitations because this study is trying to solve a real case, not an abstract problem.

1.3. Methods for solving the problem

In this part, the main methods for solving the container loading problem that have already been proposed by other authors will be considered. It is needed to analyze and compare each of them in order to understand which of the known algorithms is more suitable for the task, what are the trends, and perhaps which areas are less explored or what are the gaps. Thus, in conclusion, we can decide in which direction it is worth conducting further research.

1.3.1. Exact and approximation algorithms

Since the problem of loading containers is NP-hard, there are very few exact algorithms that try to solve it, but in any case, it is worth considering this niche, because the advantage of such algorithms is that they guarantee a certain performance, depending on the quality of the solution. In addition, the literature does not consider options for exact algorithms that also consider limitations, which is a gap in this area of research.

One of the most important works in this area was the publication of Martello S., Pisinger D. and Vigo D. in which they describe the exact algorithm built on the branch-and-bound algorithm [MPV00]. The essence of the algorithm is to build a decision tree of the problem and check each branch according to some criterion, and if this branch does not lead to an improvement in the result - it makes no sense to calculate it, so in the best case, the algorithm can find a solution by calculating only one branch, and discarded all the others, in the worst case, it will have to calculate all the branches completely, and the complexity in this situation is equal to brute force. The authors tested their algorithm in tasks up to 90 objects, but only in tasks up to 20 objects, the algorithm was able to find the optimal solution in a given amount of time. This is the disadvantage of algorithms of this kind, with a large number of elements, the tree can grow very widely, and it will take a very large amount of time to completely calculate it.

It is also worth considering a newer algorithm from O.X. do Nascimento, T.A. de Queiroz and L. Junqueira, this work is distinguished by the fact that it considers as many as 12 types of constraints [NAJ21], most of which have never been considered in the field of exact algorithms.

The essence of the algorithm itself was to divide the execution into 3 parts and weed out most of the solutions in the first two steps, as they are executed quite quickly. The first step of the algorithm is a simple selection from a set of small objects, a subset in which their total volume is

maximized, but does not exceed the volume of a large object. In the second step, the authors created an Integer programming model with simplified restrictions on objects overflow. Thus, it can quickly check if the current solution is impossible and reject it. On the other hand, if the solution passes this test, it does not mean that it can be implemented, for this, step 3 is necessary, which is the loading algorithm itself, it checks if solution is feasible, and if the solution passes the test, then this solution is considered optimal.

The authors also describe how to implement different types of constraints, and for each type of constraint he formulates a mathematical model [NAJ21], accurately describing them with formulas. The author also noted that some types of constraints can be checked at the stage of choosing a candidate for a solution. For example, constraints such as complete shipment or weight limit can be defined at the very beginning of the algorithm, but this only works in cases when the solution is generated immediately, and not object by object. This approach has its advantages, because such constraints are quite easy to check, unlike those that have to be checked at the packaging check stage.

However, the execution speed of the algorithm is much worse than other exact algorithms, since it uses an additional third step to improve the packing quality. At the same time, there are algorithms that solve this problem using only the integer linear programming model, but in this case, it becomes much more difficult to model the constraints, which makes such models not suitable for use in practice.

1.3.2. Heuristic algorithms

The most popular type of algorithms for solving container loading problems are heuristic algorithms. A very large number of them have been invented, but only some types that are closest to solving the problem of this research will be considered here.

The main part of most heuristic methods is the choice of how to stack boxes in a container, most often it is either stacking with walls, when walls are built from identical boxes and then the container is filled with such walls or building layers of boxes. In some situations where the task is strongly heterogeneous, boxes may be packed one at a time.

1.3.3. Wall building algorithms

George J. and Robinson D. were the first to propose a method of building walls, and this work is one of the very first to solve this problem [GR80]. Their algorithm had 2 stages, at the first stage they determined the remaining empty space (at the beginning of execution it is the entire container), after that they chose a box that would become the basis of the wall of boxes. They

chose the first box based on 3 criteria. First, select the box with the largest smallest side, because it will be harder to place them at the end, if there is a tie, then choose the type of box with the biggest quantity, because it will be easier to build a wall with identical boxes, if it is still a tie, then select the box with the longest largest dimension, it helps to avoid boxes with awkward long dimensions, and place them at the start. This also gives priority to "open" boxes. A box is considered open if other boxes of the same type have already been used to build the wall.

When the remaining depth of the container no longer allows to build new walls. comes the 2nd phase of the algorithm, which processes the empty spaces, it merges the empty spaces as shown in the figure below, and fills them with the remaining boxes, if possible.

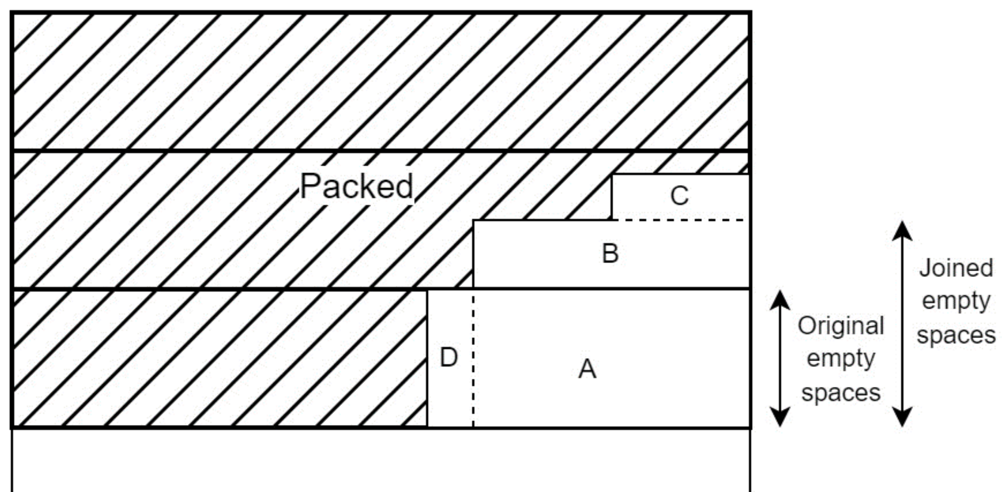

Figure 3. Empty space joining method.

As shown on Figure 3, empty spaces A and B are combined, and one large empty space is created on the basis of them, while places C and D are already considered as separate empty spaces. To join, empty spaces must be at the same height, and the adjacent empty space must have a smaller width, otherwise, the original empty space may be cut off to create a larger block when combined with others.

It is worth reviewing another work in which the algorithm for building walls is slightly modified, instead of building complete walls, it builds towers, and using these towers it builds walls [Pis02], as it is shown on the figure below.

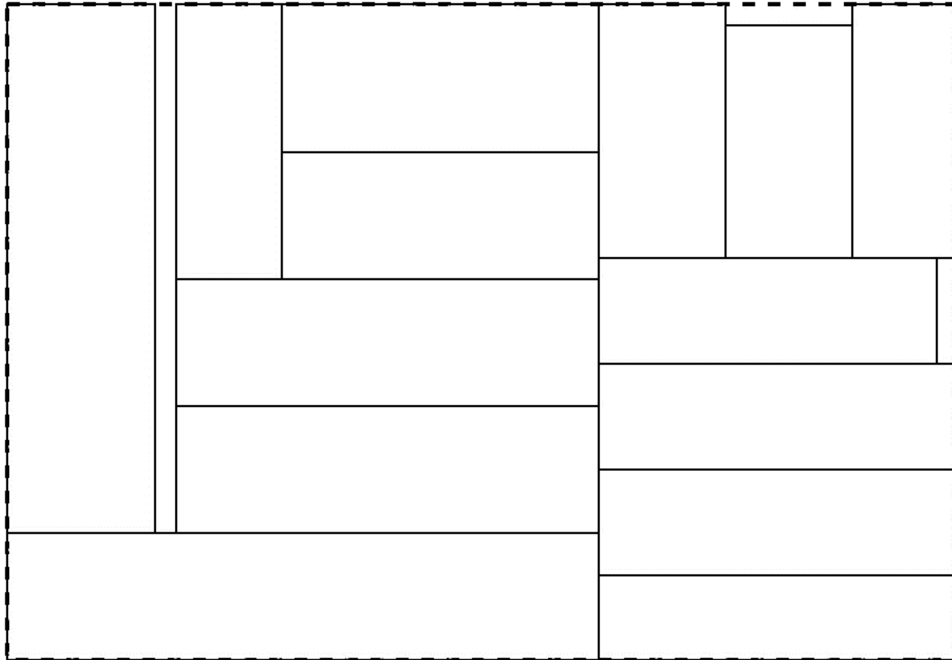

Figure 4. Box wall build with towers

As shown Figure 4, the wall is built from towers, and the towers can be located both vertically and horizontally, this allows to build small towers from the boxes of the same type more easily, unlike building a whole wall, where it is unlikely to be built using one type of box.

Instead of determining the next box based on three criteria (as in previous work), it uses a tree search to help avoid greedy choices that could lead to a non-optimal decision in the future. To limit the size of the tree, only some fixed number of sub-nodes is used, in this case, when we do not consider all available options, we need to determine only the best of them, for this the authors developed several frequency functions, and 9 priority rules, which in sum gave 27 different combinations, by what method to determine the best sub-nodes.

Using this best candidate selection function, the algorithm first chooses the best depth for the new tower, and then the best width. Having length and width, we essentially need to solve the 2-dimensional Knapsack problem, this step is very important since most of the time of the algorithm is spent on solving these small problems, so it would be logical to choose the fastest algorithm here, the authors use their own algorithm called «minknap».

The algorithm showed good efficiency in the situation when the total volume of the boxes is many times greater than the volume of the container, and in this case the algorithm was able to pack the container by 95 percent in a relatively short period of time.

1.3.4. Layer building algorithms

Another type of algorithms based on the principle of placement is the algorithms for building layers, the essence of the algorithm is similar to the previous one, the main difference is that now the layers are filled from the bottom up, that is, the floor of the container is first filled, then new boxes are loaded onto the formed layer, and so on, until the roof of the container will be reached. This method has its pros and cons, on the one hand, some constraints are better suited for this type of algorithm, for example, the stability constraint is often solved by itself, since boxes always have a reliable basis. On the other hand, in practice, loading with layers will be more difficult for workers to implement, because after loading the first layer, it will no longer be possible to enter the container. Despite some advantages, much less works uses this approach, in addition, very often the authors call their algorithm "layer building", meaning the building of walls.

Bischoff E. and Ratcliff M. were the first to propose this approach. In their work, they developed two similar algorithms, one is a classic one that builds layers from boxes, this algorithm solves the stability constraint [BR95], while comparing it with other wall-building algorithms, they showed that building layers gives results no worse and sometimes even better. better than competitors.

They also developed another algorithm in which they solved the multi-drop problem, the essence of which is that one container can contain boxes that need to be unloaded in different places. And if, for example, the box that needs to be unloaded first is placed at the very end of the container, then workers have to unload the entire container to get it. The classical algorithm for building layers also cannot cope with this problem, since this will most likely lead to a loss of stability, and workers will still have to do permutations of the boxes. The main point is to build not just layers from boxes, but to divide the free space and build towers, the principle is like the construction of towers by Pisinger D. [Glo90], but here they can only stand vertically. To choose which box to put, a subset of boxes is considered, and only those that fit in an empty area are selected from it. After that, they are matched by 3 criteria, that is, at each step, the algorithm makes a local optimal choice, which in fact can lead to a non-optimal result, on the other hand, this makes the algorithm faster.

Even though this version of the algorithm solves the multi-drop problem, the stability of such a construction is greatly reduced, moreover, the first version of the algorithm generally generates better results than the second. Therefore, it makes little sense to apply the 2nd algorithm, since wall building methods are better at handling the same tasks.

The layer building algorithm is also used in a task adjacent to the container loading problem - pallet loading. The essence of the task is very similar and consists in loading a pallet in such a way as to minimize the number of required pallets or maximize the value of the load on one pallet. The main difference is that the pallet does not have walls, like a container, and therefore loading stability is even more important here than in CLP, in addition, the restriction on consignments should also be considered, because different boxes most likely need to be unloaded in different places, thus, the boxes that are to be unloaded first should be at the very top of the pallet.

Let's consider another work that solves pallet loading problem [TSS+00]. Proposed algorithm consists of 3 main steps:

1. The first step is to find the lower and upper bounds on the number of containers needed to pack the consignment. For this, a greedy heuristic algorithm is used, and the upper bound is determined from its result. The lower bound can be determined by a simple calculation of how many pallets are needed to load the total volume of all boxes, but such a calculation does not take into account any restrictions and gives a rather weak lower bound. To get a stronger lower bound, the author uses the normalization of raster points.
2. The next step is to divide the consignment by the number of consignments equal to the upper bound minus one. The consignment should be divided in such a way that their weight is similar, and that the number of heavy and light boxes is also similar, so that heavy boxes can be placed below, and light ones can be placed on top of them.
3. After that, from the selected consignments, a full-fledged loading is created using a heuristic algorithm. The heuristic algorithm uses the branch and bound strategy with a limited number of branches, which was discussed earlier. The algorithm tries to group identical boxes together to reduce the number of branches even more. The author uses 2 different algorithms, the first to pack boxes of the same type, or with the same height, and the second to pack boxes of different types. After that, if the packing was successful, then the upper bound is reduced by one and the algorithm returns to the second step. The execution terminates when the lower bound is reached or the consignments were not packed, then the result is the best solution.

The algorithm showed a pretty good result in comparison with the algorithm for solving CLP, although the work was published in 2000, so the comparison may not be very relevant now, as faster algorithms have appeared. What is important here is that this work also considered the

limitation in stability, and the problem with multiple consignments, which is considered quite rarely.

1.3.5. Cuboid building algorithms

Besides the main two types of heuristic algorithms mentioned above there are couple of other types of algorithms. One of them is creating cuboids from boxes of the same or different types, and filling containers with them. This approach has its advantages, since cuboids can be defined at the very beginning of the algorithm, this reduces the execution time, since boxes grouped into a cuboid can be considered as one object. Cuboids can be created with stability in consideration, then container loading stability will be much better, also cuboids can be created by consignments. When unloading the container, it will be easier to reassemble the boxes back.

In their algorithm, Ren, J., Tian, Y. and Sawaragi, T. build cuboids from boxes and fill the empty spaces of the container with them [RTS11]. For the construction of blocks, they defined several restrictions, firstly blocks are built from boxes of the same type, and the number of boxes of a certain type must be sufficient to build a block, as well as block dimensions should not be larger than the size of the empty space where this block will be placed. After that, all possible blocks are created and they are evaluated according to 5 criteria, evaluation of the smallest dimension, evaluation of 2 dimensions together, and evaluation of 3 dimensions. All these checks run simultaneously, and the blocks with the best result of each of the checks are selected.

These selected boxes become branches of the tree and based on these solutions, solutions that can follow them are considered, they are also evaluated by the same 5 checks, and the best one is selected. The algorithm continues until the empty space cannot be filled by any of the cuboids.

The authors also offers an extended version of the algorithm, where the shipping priority constraint is considered. To do this, the authors, in addition to the previous 5 restrictions, also introduces the fact that cuboids with high priority cannot be packed before cuboids with low priority, and such solutions are not considered.

The algorithm showed a pretty good result in terms of the quality of the result, moreover, with good average performance. The algorithm modified to support the shipment priority constraint takes much more time to complete but still generates a very good result. The author also noted that the disadvantage of the algorithm is that it does not support the multi-drop constraint [RTS11], the problem is most likely that the support of this constraint increased the execution time even more and then the algorithm would work too slowly to be used in real life.

1.3.6. Metaheuristic algorithms

In addition to the usual heuristic algorithms that solve CLP, there are also encountered add-ons for them, which, among the many results of the execution of heuristic algorithms, find the best one, while reducing the number of calculations. Thus, metaheuristic algorithms help to find the best result, but at the same time they can worsen the execution time.

1.3.6.1. Tabu search

Tabu search is a metaheuristic algorithm that uses a local search method with additional memory usage to solve an optimization problem, it was introduced by Glover F. [Glo90].

The simplified algorithm has the following steps:

1. Start with some solution.
2. Explore neighborhood solutions.
3. Evaluate neighborhood solutions.
4. Choose the best solution.
5. Update tabu list.
6. Check termination criteria.

At each iteration, the algorithm takes the current state and checks neighboring solutions for the best one, which may lead to worsening of the current result, thus the algorithm can avoid the local optimum trap when the next step does not improve current solution, but the step after it can give significant improvement in results.

Also in the algorithm, it is mandatory to define a list of tabus, so as not to calculate the same solutions repeatedly. Long-term memory is also often used, for example, it can store the best result of all, or the history of decisions at different iterations, this memory should be used to guide the algorithm into unknown areas of solutions that can lead to a better result. Aspiration criteria can also be used to bypass the tabu rule in a certain situation, for example if the new solution is better than the previous best, but it cannot be chosen due to the tabu.

The execution ends when the termination criteria is met, it can be a check for the quality of the result, or the maximum number of iterations. The result is the best solution of all that has been calculated. If the criterion is not met, then steps 2-6 are repeated.

To better understand how the algorithm works, consider an example of solving the Traveling Salesman Problem, the essence of which is to find a route between cities in which each city is visited 1 time and at the end returns to the starting point.

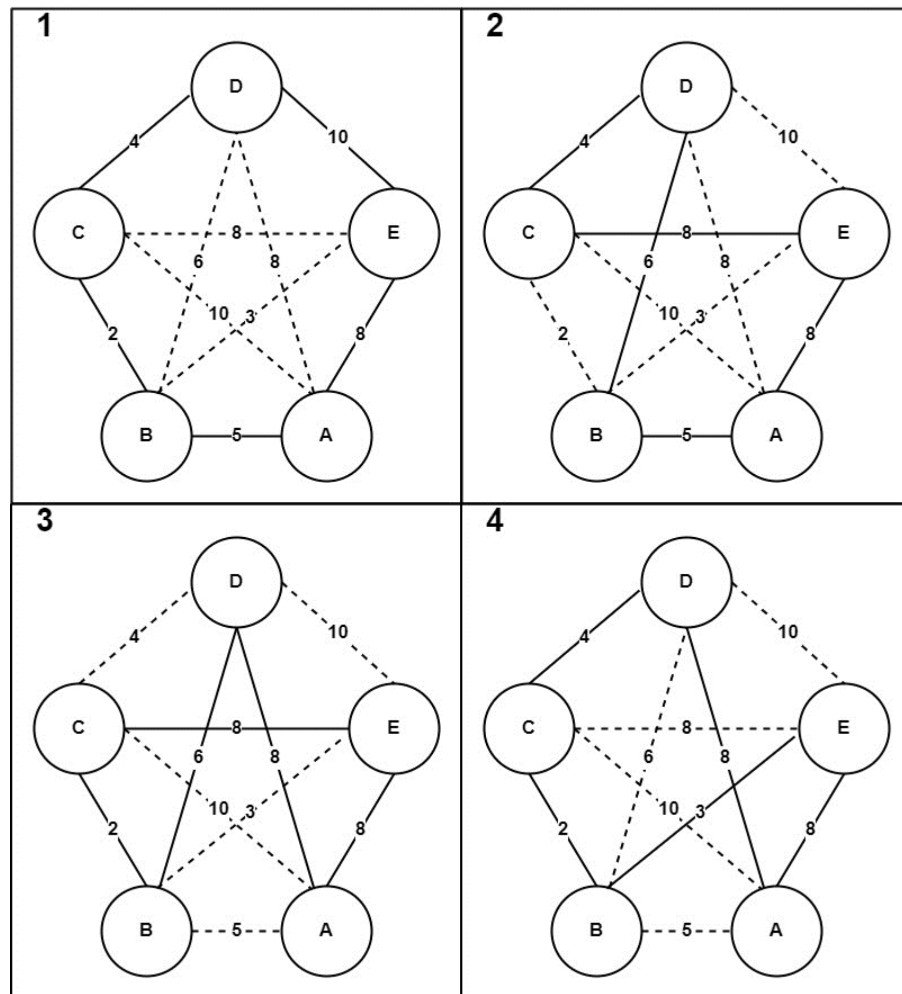

Figure 5. Example of tabu search arlghorithm for Traveling Salesman Problem

As shown on Figure 5, at the first step, the problem starts with the initial solution A->B->C->D->E->A, total cost of such a route is 29. At the second iteration, the algorithm checks all neighboring solutions by swapping neighboring nodes. Having considered all the options, and having calculated their cost, it selects the option with the lowest cost, at the second iteration it is A-> B-> D-> C-> E-> A (changed D and C), the cost of this route is 32. add swap D and C to tabu list to forbit it in future operations. Thus, at 4 iterations, algorithm found the optimal solution- A->D->C->B->E->A with a cost of 25.

Bortfeldt A., Genring, H. and Mack, D suggested how tabu search could be used to solve CLP [BGM03]. At the beginning of the algorithm, an initial solution is generated using wall-building heuristics, after which this solution is encoded to use it in a tabu search, and based on the encoded solution, neighboring solutions are found, the author suggests 2 options for how to search for neighboring solutions:

- large neighborhood - this is a variant of the original solution in which one index of the encoded solution is different.
- small neighborhood - is essentially the same as large, but the index can differ only in some range, which decreases with each iteration. That is, at the first iteration, all options are considered, at the second iteration, all but the first one is considered, and so on.

The beast found solution is added to tabu list on each iteration to avoid cycles and repetitions. Aspiration criteria is not used.

The author also suggested how to parallelize the execution of the algorithm, for this the set of neighboring solutions must be divided into the number of parallel elements, and these subsets should be distributed to these elements. After that, each of them finds the best result and compares it with others. After that, the best result of the algorithm is repeated for a new best result.

The algorithm was compared with other algorithms by the same author, and among all, it showed the best result, which, however, is not surprising because other algorithms did not use any metaheuristics. At the same time, the parallel algorithm almost did not improve the result quality.

Even though the tabu search helps to avoid local optima, as well as cycles and repeated solutions, it also has its drawbacks, the number of iterations of the algorithm can be very large, in addition, there are a large number of parameters and rules on which the efficiency of the algorithm directly depends. The algorithm gives only the concept of execution, and the rules themselves need to be determined independently, often experimenting and finding the best option.

1.3.6.2. Genetic algorithms

The genetic algorithm is a metaheuristic algorithm created on the basis of the process of natural selection in nature. It mimics the selection phases to find the best candidate. The algorithm was proposed by J.H. Holland in the 1970s. The algorithm that he proposed is also called Simple Genetic Algorithm (SGA) [SP94]. The SGA algorithm consisted of the following key steps:

1. At the first step, the algorithm takes some initial population as an input. The genetic algorithm works only with encoded solutions, usually the encoded solution is a string of bits, although this of course also depends on the problem and different problems can be encoded in different ways. For example, in the traveling salesman problem discussed above, the solution can be encoded as a string of bits that will indicate whether a graph edge was included in the solution.

2. Next, the program evaluates the population using a special function called “fitness”. It receives a candidate from the population as input, and as output it must give a number that corresponds to the value of this solution. For the same traveling salesman problem, this function should simply calculate the total length of the candidate's route, and the longer the route - the worse the candidate.
3. Further from the current population, it is mandatory to make a selection to create a new population, there are different ways to do this. The SGA algorithm uses roulette wheel selection [SP94]. The main point is that the better the candidate's fitness value, the more chances it has to get into the next population, however, even the best candidate still has a chance not to pass the selection, since the probability of passing is not equal to 1.
4. After the best candidates for the next lecture have been selected, a crossover must be carried out. This function mixes two candidates by swapping bits. In SGA crossover, the function has a certain chance of triggering, therefore it is applied only to a part of the candidates, the essence of the function is to take two candidates, divide it in half on a random index of their encoded code, and swap these halves between two candidates. Thus, it creates a new candidate, part of which is inherited from the first, and the other part from the second.
5. Then the mutation function is executed. It, like crossover, is needed to create new candidates. In SGA [SP94], this function also has a certain chance of triggering, that is, only some of the candidates are mutated, the point is to take the encoded candidate code and change the bit to the opposite at a random place, thereby obtaining a new candidate. The use of this function is that it helps to avoid the situation of stuck for certain bits, for example, if all the population in the encoded code has 0 bits at some position, and in the optimal solution there should be 1, then crossover will not be able to find such a solution, but during mutations, this bit can be changed and a solution will be found.
6. Steps 2 - 5 are repeated until termination criteria are met. this can be, for example, the maximum number of populations, or the degree of similarity in the population (when most of the bits of different candidates are equal, and further iterations do not change the result much)

Genetic algorithms were applied in much research in CLP domain. It is worth mentioning Bortfeldt’s A.and Gehring H. work on this topic. The algorithm starts with generating towers from the boxes, which could be placed both vertically and horizontally [Geh97]. After that, these towers make up the initial population of the genetic algorithm, they are encoded as an array of data

structures, where each data structure is the index of the tower and its position (horizontal or vertical).

As a fitness function, a heuristic algorithm is used that places these towers, and returns the percentage of space used. As a selection, the value fitness function is used, but it is not just equal to the result of the objective function, that is, the percentage of filling the container, it is only indirectly related to it. The fitness function uses a ranking based system on the objective function, so there will not be a big gap between the first- and second-best results, although the objective function may show that it is large. This helps to avoid guiding the algorithm to a local minimum, since the best candidate will have a strong advantage over the others, and it helps to better explore various solutions without greatly increasing the search area.

Unlike SGA [SP94], either a crossover function or a mutation is used here, with equal probability. Crossover function and mutations also work with binding to CLP, specifically in this algorithm they change the base box of the selected towers, the mutation function also changes the position and orientation of the tower.

The genetic algorithm has been compared with other heuristic algorithms for solving CLP. And the results proved that it can be used to get better result quality, in comparison with other heuristic algorithms, it showed a good increase in container space usage. At the same time, with an increase in the number of box types, the algorithm shows a better result than its competitors that use conventional heuristics. The mean computing time required by this genetic algorithm amounts to less than three minutes for all test cases.

The main strengths of genetic algorithms are that they are very good at exploring the solution space, avoiding local optimum and recalculations, moreover they can be easily parallelized, making execution faster. Also, the genetic algorithm can be quite adaptive to changing conditions, by using techniques such as incremental genetic algorithms or dynamic parameter tuning, GAs can handle dynamic optimization problems. On the other hand, it also has its drawbacks, just like in the tabu search, the use of this algorithm increases the execution time at times, although the quality of the result may not be much higher, this is due to the multiple calls of the fitness function, which is quite heavy. Also, from the results of various studies, it can be noticed that with a small number of boxes in the problem, the genetic algorithm is not as efficient as a simple heuristic one. It is also worth noting that the genetic algorithm needs to be considered for each problem separately, so the functions of encoding and decoding must be developed for a specific heuristic algorithm,

considering its features, which is often quite difficult to do. In addition, the fitness function and mutation should also take into account the specifics of a particular task.

1.3.6.3. Other approaches

In addition to genetic algorithms and tabu search, other metaheuristic algorithms are also sometimes used. Greedy Randomized Adaptive Search Procedure (GRASP) is used in a large number of works, this algorithm, like tabu search, is an add-on to the local search algorithm, the essence of the algorithm is to iteratively create candidates using greedy choices or randomly (randomness helps to avoid local optimum), on the next step the local search algorithm tries to improve the candidate. Execution continues until the program stop criterion is met. By combining greedy selection and randomization, GRASP provides a trade-off between exploitation and exploration, making it a useful approach for solving combinatorial optimization problems.

It is also quite common to use the Simulated annealing algorithm which was inspired by the annealing process in metallurgy. The algorithm is like the genetic algorithm in that it also uses a candidate scoring function and does perturbation in the candidate. In addition, it uses such a concept as temperature, on the basis of which it decides whether to consider a new candidate who has a worse result or not. The higher the temperature, the greater the chance that a poor candidate will be considered, while better candidates are selected automatically without any checks. At the start of the algorithm, the temperature is high, and decreases during execution. Like other metaheuristic algorithms, it has no end, and relies on a certain termination criterion. The acceptance of worse solutions at the beginning of the search allows SA to explore the solution space broadly and escape local optima. As the temperature decreases, the algorithm becomes more selective, favoring solutions with improved objective function values. This cooling process allows SA to converge towards a good-quality solution

A small number of works also use Variable Neighborhood Search (VNS). This is also an algorithm that uses local search. At the beginning, it accepts some initial candidate, after which it conducts a local search to improve it, when the algorithm reaches a local optimum, the algorithm switches to a neighboring solution, and starts exploring it, the algorithm stops when the stopping criteria is met. By iteratively applying local search within different neighborhoods, VNS allows for a more extensive exploration of the search space.

One of the main disadvantages of these local search algorithms is that they are highly dependent on the initial solution, if the initial solution is far from optimal, then such an algorithm most likely will not be able to find it. Therefore, when using these algorithms, it is also necessary

to develop a method for generating good initial solutions. In addition, local search algorithms typically converge slowly, especially when dealing with large and complex search spaces. They often require a large number of iterations to explore space thoroughly and converge to an acceptable solution. This can be computationally expensive and time-consuming.

1.4. Test sets

After creating an algorithm, it is often necessary to check it in order to understand whether it works correctly, and with what performance, of course, it would be possible to randomly generate a certain number of problems, and just see how the algorithm works, but in this case, it would be difficult compare the new algorithm with existing ones, for this purpose there is created various test sets for CLP, with already defined tasks, these test sets are very often used, so it is worth considering the most popular of them.

Table 1. Tests sets for different problem types with constraints considered.

Problem type	Authors	Number of instances	Contrainst considered
SLOPP	Bischoff & Ratclif	700	Orientation, Vertical stability
SKP	Davies & Bischoff	800	Orientation, Vertical stability
SSSCSP	Ivancic, Mathur & Mohanty	47	None
SBSBPP	Martello, Pisinger & Vigo	320	Orientation
ODP/W ODP/S	Bortfeldt & Gehring	100	Orientation
ODP	Bortfeldt & Mack	100	Orientation

Bortfeldt A. and Wäscher G. identified the most popular test sets in their state of art review, these test sets are shown in the table above [BW18]. The most popular of these test sets is the Bischoff & Ratclif test set, most of the researches where there is a comparison of performance or the quality of the result of algorithms use this particular test set, it can be divided into 7 parts, according to the number of different types of boxes - 3, 5, 8, 10, 12, 15, and 20. For each part, 100 test problems were created, that is, 700 problems in total. The dimensions of the container are fixed

and equal to the dimensions of a 20 lb sea container, the maximum size of the boxes is also limited. The test set takes into account only 2 limitations - vertical stability and orientation.

As it can be seen from the table, no test cases have been developed for the type of problems that this research are trying to solve SLOP and SKP, so first two test sets can be used for testing.

As for constraints, it was previously determined that it makes sense to consider the following constraints - orientation, stability. Among all the tests, the first 2 are most suitable, they are most often used in literature, so it will be easy to compare the results.

When testing algorithms, authors often indicate only the test set, and which average percentage of filling was achieved, they also often indicate the standard deviation, while the computer time required to execute the algorithm is rarely indicated. In papers where many algorithms are compared, some exact algorithms and genetic algorithms show good results [ZBD14].

1.5. Summary of literature

In this work, the container loading problem was defined, and its different types were considered, of all these types - SLOP and SCP turned out to be the most suitable for marine container loading, since this is the simplest solution that will not require processing of different consignments, and situations where goods going to different places will be loaded into one container, 2 types are considered because it is difficult to determine how heterogeneous the task will be for different types of companies.

Various constraints for CLP were also considered. Of these, only 2 were suitable for the marine CLP problem - orientation, and stability. In addition to these 2, there are other constraints that could be added, and they would improve the result, but for the problem they are not critical, so they may not be considered.

After we decided on the type of problem and constraints, almost all types of algorithms that are relevant and used by other authors were considered. Thus, we considered exact algorithms, various types of heuristic algorithms, as well as metaheuristic algorithms that work in conjunction with heuristic algorithms. As a heuristic algorithm, the easiest way to solve the problem would be to implement a layer-building algorithm. Since the constraints that are defined above can be easily solved using this algorithm - the stability constraint will be solved almost automatically with such loading, the weight distribution will also be easier to solve, and the weight limit and orientation do not depend much on the algorithm. Also, in combination with this heuristic algorithm, a genetic

algorithm should be applied that would help improve the result. Other metaheuristic algorithms can also be used, but as shown by other studies with comparisons, the genetic algorithm the best the rest in terms of the final utilization of free space. Exact algorithms also show very good results, but it is quite difficult to implement constraints with them since they need to be modeled separately, there are works that consider constraints for exact algorithms, but they do not consider several constraints together.

The first 2 test cases, which were created for the SLOP and SKP problems, which also includes the necessary constraints - orientation and stability. Since these tests are among the most popular, it will be easy to compare your work with other authors to evaluate the effectiveness of the algorithm.

In general, in this literature review, almost all parts of the CLP have been considered. Starting from typology, constraints, and various algorithms, and ending with methods for evaluating algorithms. Gaps in knowledge on this topic were identified and a new algorithm was proposed that would partially close these gaps, offering a new solution for CLP from a practical point of view.

2. Genetic tower-building algorithm for solving container loading problem

2.1. Solution concept

In this work, a new concept was tried, with the construction of towers from boxes instead of walls or layers, because this will give greater flexibility when searching for results since the towers themselves are smaller than walls or layers.

A meta-heuristic approach was also used. A genetic algorithm was adapted to the needs of the task. The general concept of the solution can be formulated as follows:

- Generate towers set using a genetic algorithm, in a way that sum of the free container space waste is minimized. To generate towers a *tower-generating* heuristic is used, which will be described below in more detail.
- Using towers generated in the previous step, initialize the population by randomly shaffling the tower set.
- For each generation apply crossover and mutation functions, to create a new child generation. After this count the distance between chromosomes, match them, and select the fittest (so-called crowding technique). The fitness score is calculated using a *fill* heuristic which places towers on the container flow as tight as possible. For this work, there were two *fill* heuristics designed, namely:
    - Recursive filling heuristic. Which recursively packs space of the container floor, by greedily choosing the best fitting box.
    - Packing filling heuristic. Which fills towers layer by layer, and then moves them back by the x-axis, if there is a space to do so.
- Stop evaluation when the specified number of generations past, or time limit is up.

In this solution following concepts are used:

1. Container, which is a box, defined by Wc, Lc, and *Hc* values, which are the depth, length, and height of the container. Container volume is denoted as *vc.*
2. Boxes are a list of elements defined by their length, depth, height (Li, Wi, Hi) and rotation variants, denoted by *orD, orH,* and *orL* which can have values of either 0 or 1. In this case, box rotation is calculated as follows:
   $bR = 2 \times orD + 2 \times orH + 2 \times orL$

So, in case, when all rotation is allowed, *br* will be equal to 6.

3. Box Group, for simplicity heuristics use box groups instead of separate boxes, box group is defined by depth, length, height, and rotations, which are equal to the box's *bR*, and quantity of the boxes.
4. A tower, is a set of boxes, stacked on each other. A tower is defined by *Wt, Lt, Ht*, which is a tower's depth, length, and height, A tower has a reference to boxes, with which it is built, and the tower could have another tower on its top, or side, or in front.

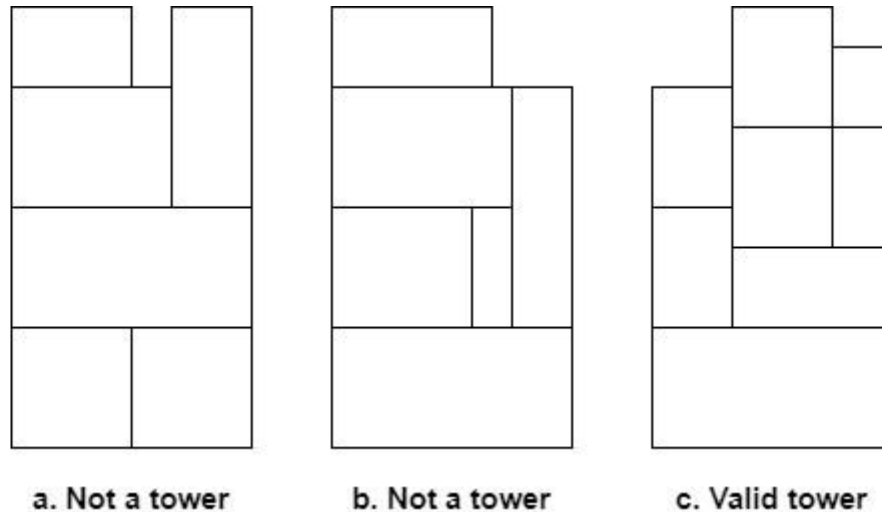

Figure 6. Example of valid and invalid towers.

Figure 6, shows different variations of the towers built from boxes, but in this solution, only towers as the third one are built, so the basic rules for tower building are the following:

- A box can only be supported by one box below.
- A box should only be placed within the sides of another box, in other words only full support is allowed.
- Tower dimensions should not exceed container dimensions.

5. Tower Placement, is the placement of a certain tower on the container floor. It is defined by the tower index which is a reference to the towers list, rotation which can be 0 or 1, and *x* and *y* coordinated on the container floor. The list of tower placements defines the complete loading plan and represents the result of the execution.
6. Empty Space, this concept is used in the recursive filling heuristic, to represent space left after placing the tower on the container floor. It is defined by *x* and *y* coordinates which

are the coordinates of a corner, and depth and length parameters, this combination describes a rectangle in the container floor.

2.2. Tower heuristic

At the start of execution, the algorithm creates towers from all boxes present, using a genetic algorithm and recursive tower-building heuristic called *fillingTower*. The rough algorithm for *fillTower* is presented below:

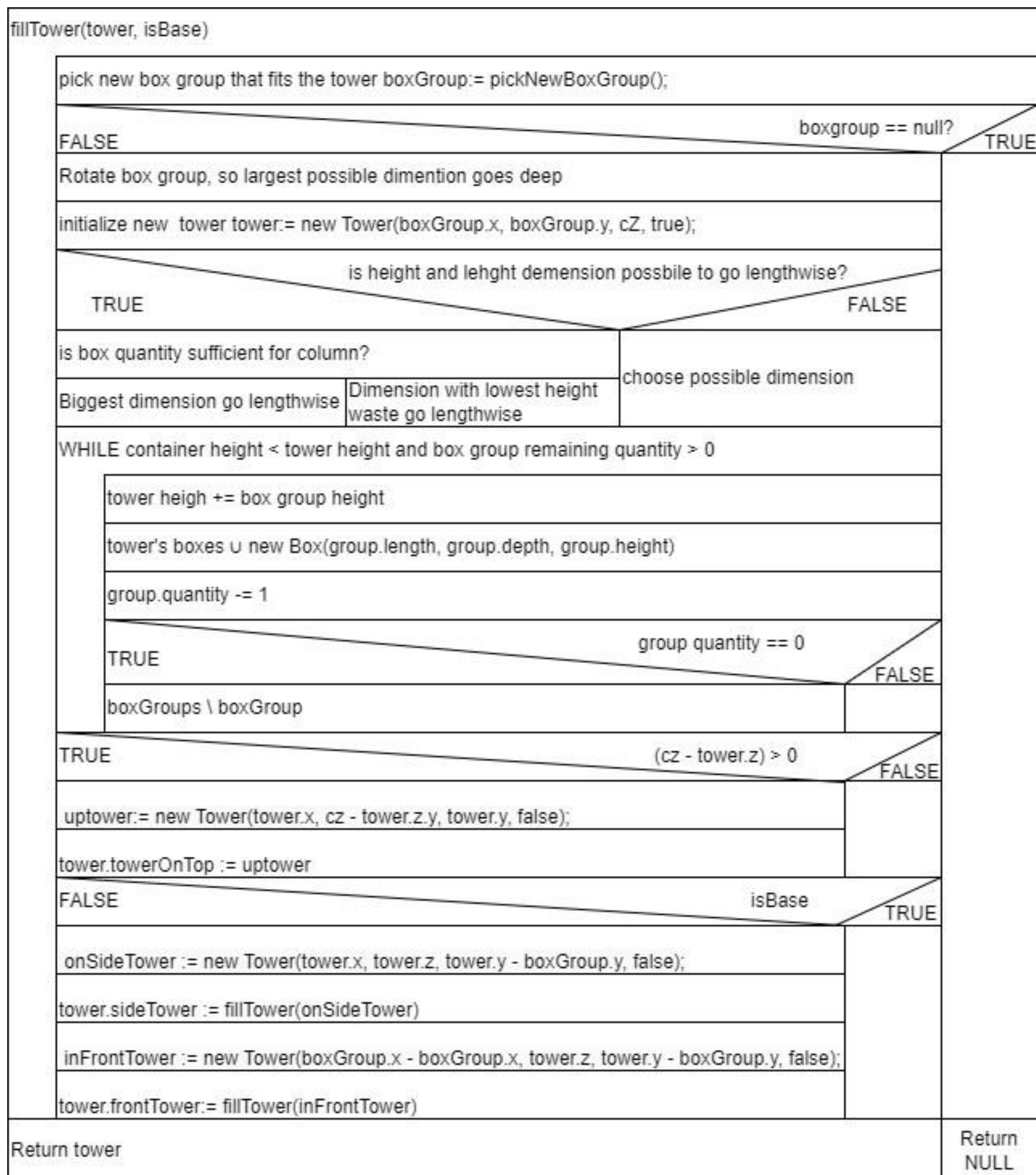

Figure 7. Rough algorithm for fillTower.

- First of all algorithm picks a box group that is feasible to put in the current tower. It rotates the box in all allowed rotations and if there is at least one possible rotation, the box group is chosen.
- Then the box is rotated in a way that the largest dimension goes deep-wise, orientation restrictions are also considered.
- Then choose length dimension according to algorithm rules. And Fill the tower up with the box group.

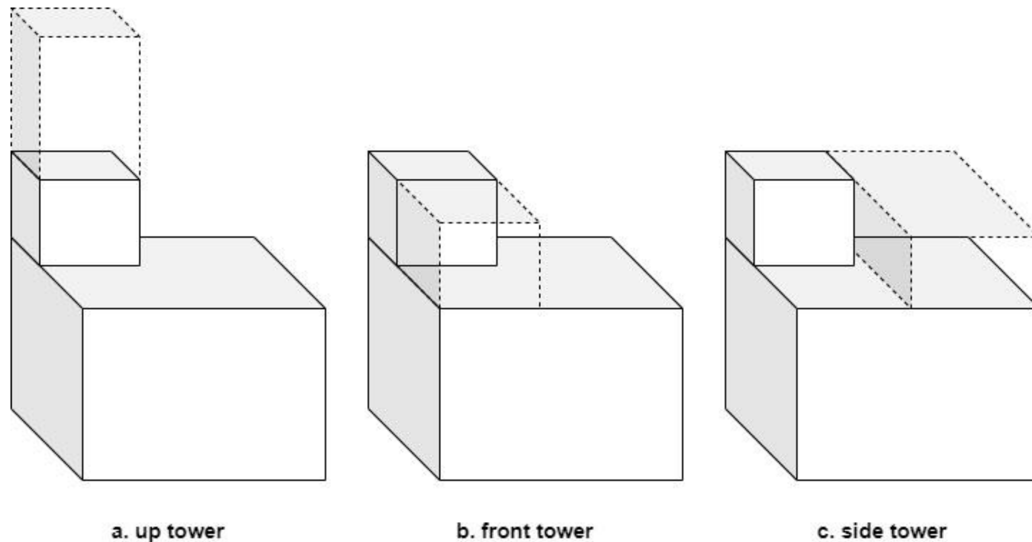

Figure 8. Residual spaces in the tower.

- If the remaining tower height is more than zero. Initialize the new tower with tower depth and length and remaining height and recursively call *fillTower* with new up the tower (Figure 8, a).
- If the tower is not a base tower, then create two new towers in front of the tower and on the side of it, and fill them using the *fillTower* function (Figure 8, b, and c).

fillTower function is called in a while loop until eventually all box groups are fully used. Created set of towers then used later in genetic algorithm to find the best set of towers, which uses the maximum amount of space in the container.

2.3. Container floor filling heuristics

For purposes of this solution, two filling algorithms were developed, each of them has its pros and cons, and they could be used interchangeably for different problem types. Algorithms of these two heuristics are described below.

2.3.1. Recursive floor filling heuristic

This heuristic is simpler than the second one and can perform better when filling containers with towers where base boxes are small, so there will be more small towers on the floor, another advantage is that it fills towers in straight rows, so it is much easier for workers to load such loading plan.

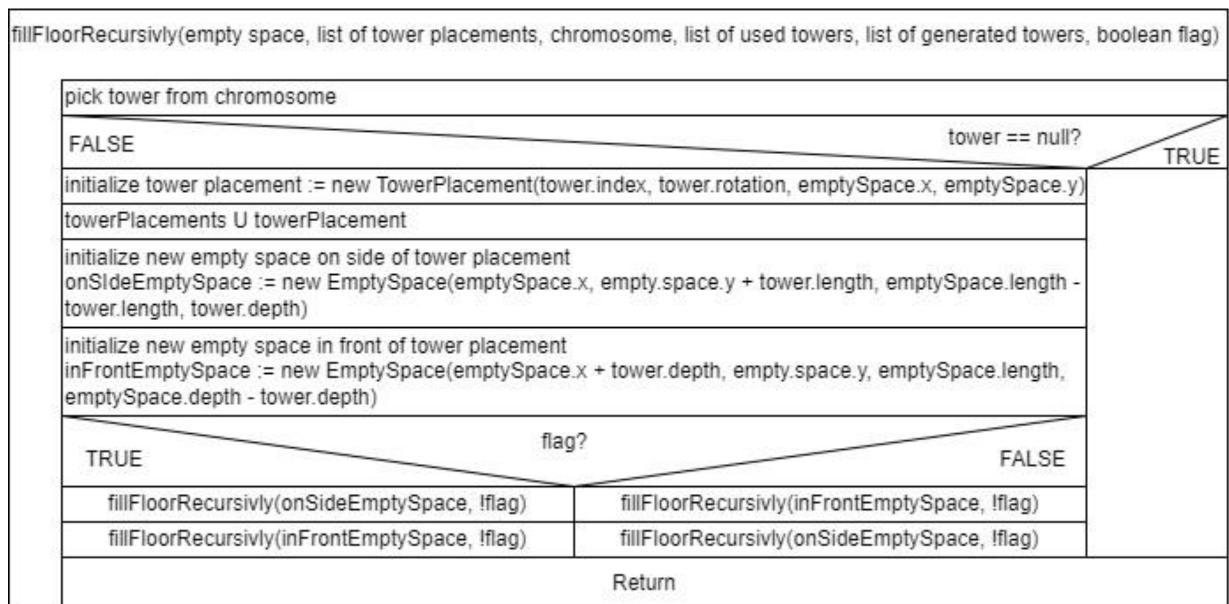

Figure 8. Rough algorithm for recursive floor filling heuristic

- At the start of execution, the program creates a new empty space with 0,0 coordinates, and length and depth equal to the container once, in other words whole container is empty space, it also creates an empty list of used towers, an empty list of tower placements and flag is set to true. Using all these parameters *fillFloorRecursivly* function is called.
- In the first step algorithm picks the tower from the chromosome (the structure of the chromosome will be described later, for now, it is enough to know, that the chromosome contains a list of towers in a certain order and their rotations). To pick a tower, the algorithm goes through chromosome tower sets and picks the first tower, that fits into empty space, and hasn't been used before.

- Then, if a suitable tower is found, new tower placement is created, and residual space is divided into two sections (like the tower building heuristic, except here only spaces on the sides are used).
- Then, there is the flag, that tells which side to fill first, in front or on the side, this flag was introduced to add sort of randomness in the filling because otherwise, it would fill the container row by row, which was proven by experiments to be not efficient.
- After this *fillFloorRecursivly* is called once again leading to recursion. The algorithm fills the container floor until there is no space left where another tower could be put. The result of filling is saved to the tower placements list, which is passed as a function parameter.

This heuristic is simple and efficient, but the main disadvantage of it is that it is time-consuming, and because of that when, using this heuristic it is better to decrease the number of generations in the genetic algorithm, also performance is very related to the number of towers if a number of towers generated will be very big, it could become dramatically slow.

2.3.2. Packing floor filling heuristic

The main idea of this heuristic is to greedily place towers in a row one by one, until there is no space for new towers, and then pack these rows by moving them backwards as tightly as possible. This algorithm is divided into two parts: building rows and packing.

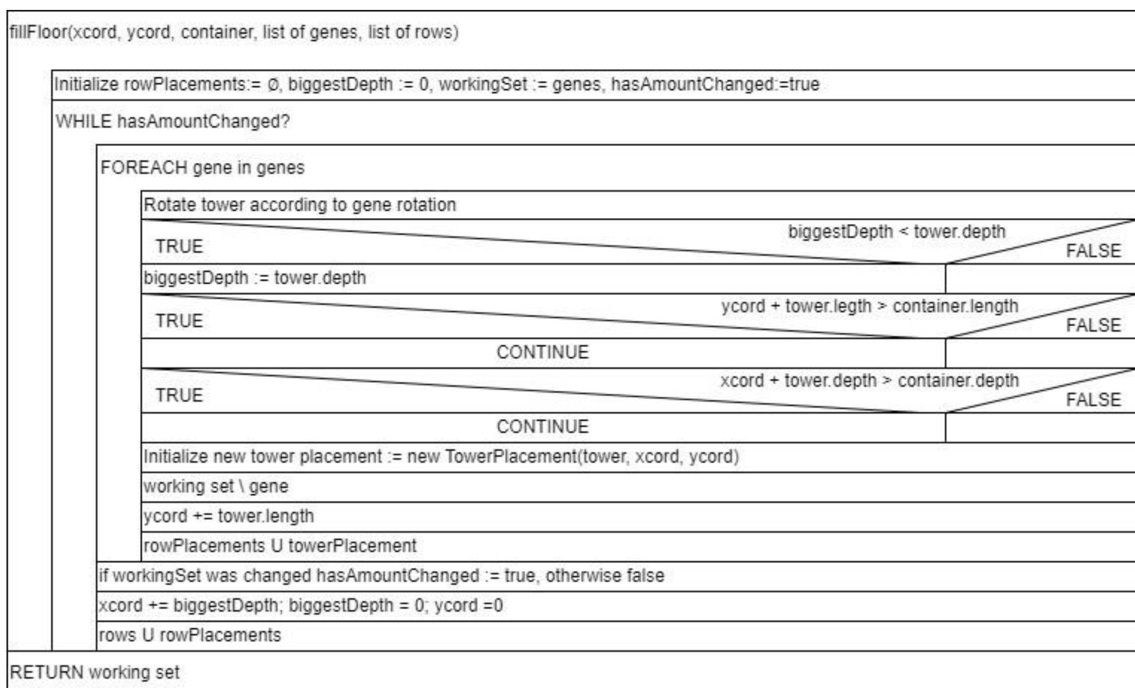

Figure 9. Rough algorithm for filling floor heuristic

- Filling algorithm as the input takes starting x and y coordinates, for the first algorithm execution they are zeros. Also, the algorithm takes a container, a list of genes, and a list of rows, to save the result.
- Then the algorithm goes through all the genes, which represent the tower and its rotation, and if the tower fits a row, it is saved to the row, and removed from the genes list.
- The algorithm continues until there is no tower used from any of the genes left. It returns genes that were not used for rows, and rows are saved to the input parameter.

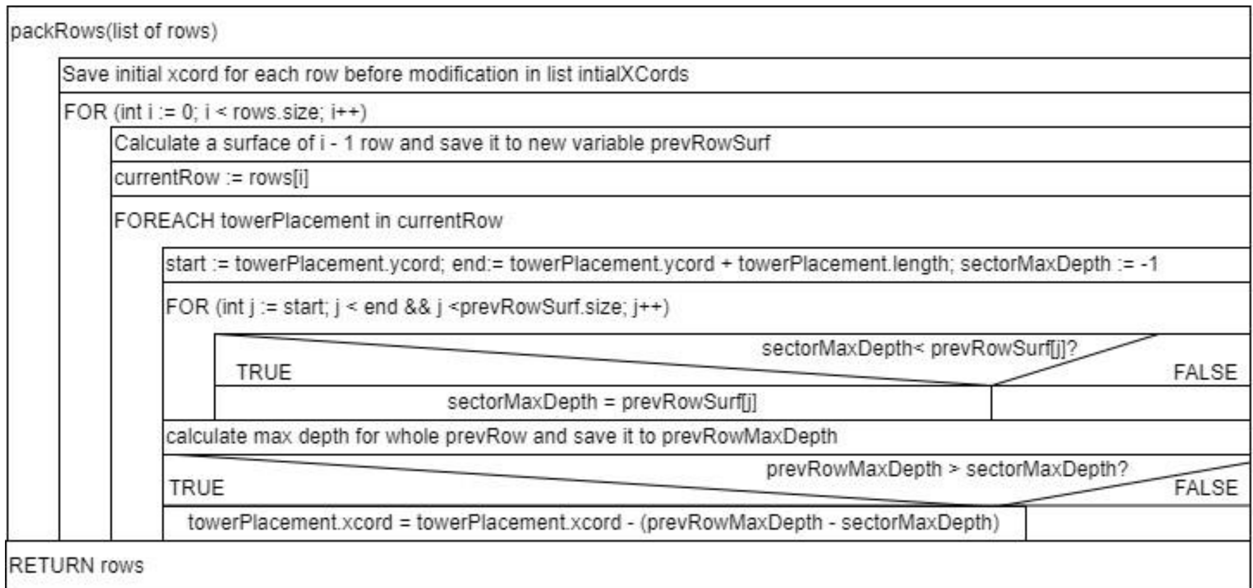

Figure 10. Rough algorithm for packing rows heuristic

- After rows were created, the algorithm packs these rows using the *packRows* function. As input, this function takes a list of rows.
- The first step is to save the initial x coordinates of each row into a separate list because it will be needed further for calculating the row's surface.
- After this algorithm calculates the surface for the previous row, for the first iteration this will be the first row, and the second row will be shifted backward. The surface is an array of integers of the size of a row, where each element equals the depth of a row in this coordinate minus how much it was shifted backward from its initial position.
- Then it calculates the biggest depth in a sector of the surface in which current towers are placed. If the biggest depth in a sector is smaller than the whole row's biggest depth, that means we can shift backward the tower for the difference between these two depths.
- The algorithm goes through all tower placements and tries to pack them, at the end, it returns a modified list of rows.

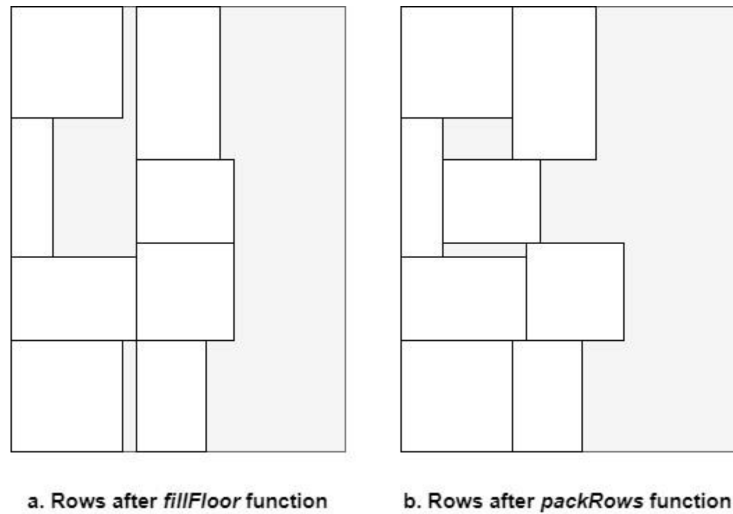

Figure 11. Rough algorithm for packing rows heuristic

Figure 11 shows a basic idea of how this heuristic works. It continues to create rows and pack them, until, there are no towers left to be placed, or there is no space left to incorporate any of the towers.

2.4. Genetic algorithm

The basic concept of the genetic algorithm was presented before in the literature review, however, in this work, it was adapted to the problem and tuned by using some advanced techniques. This solution has two-step execution, each of them uses a genetic algorithm. First, there is a tower-building genetic algorithm, the purpose of which is to create a tower set in a way that will have the least possible container space wastage. The second one is container-filling GA, which places towers on the container floor to load them as tightly as possible. These two GAs use the same techniques, and the only difference between them is gene structure and fitness evaluation.

Tower-building GA uses chromosomes, that are permutations of box groups, so each gene in a chromosome is just an index of box groups in box groups global list, accordingly, there are as many genes in a single chromosome as there are groups of boxes. For fitness evaluation, it uses a tower-building heuristic, which generates a tower set. After, for each tower, it calculates tower space wastage – the difference between the total volume of the tower and maximum volume (cube's volume with base box's length and width and container's height). Then, the sum of the tower's space wastage – is a fitness score of the chromosome, so the fitness function for tower-building GA, can be defined as:

$fintess = \sum_{i=1}^{N}((\sum_{j=1}^{m} L_j \times W_j \times H_j) - Lt_i \times Wt_i \times H_c)$, where i is a number of towers and j is a number of boxes. GA tends to decrease this value.

Container-filling GA is based on the chromosomes, which are permutations of towers. Each gene consists of a tower index and rotation of the tower's base (there are only two rotations). Using these genes, GA fills the container floor with towers, using one of the filling heuristics. The fitness score for these chromosomes is the whole container loading total volume, so the algorithm goes through all the towers that were placed by heuristic and calculates total volumes, and the sum of these volumes is a fitness score, so the fitness function for container-filling GA, can be defined as:

$$fintess = \frac{\sum_{i=1}^{n} L_i \times W_i \times H_i}{L_c \times W_c \times H_c},$$ where i is a number of boxes to be loaded.

GA works towards maximizing the total volume of the towers.

2.4.1. Genetic Algorithm main loop

Typically, genetic algorithms select the best candidates, and thus often quickly lose diversification of the population, and accordingly tend to a local minimum. To prevent this and maintain diversification, a special technique called "crowding" [MG08] was used in this work.

There are different selection mechanisms in genetic algorithms, and crowding is commonly associated with techniques such as crowding tournament selection or niche selection. These methods involve evaluating the fitness of individuals not only based on their absolute performance but also in relation to the performance of their neighboring individuals. By favoring diverse solutions, crowding helps in exploring the solution space more effectively and avoiding the loss of potentially valuable genetic material.

The method used in this work is like tournament selection, with some modifications, it can be described as follows:

- Randomly select S parents from the population, where S is the size of a tournament.
- In the second step, perform crossover on the parents with some defined probability P_c and do a mutation with probability P_m , add these new chromosomes to the child list.
- In the third step, the algorithm calculates distances between each pair of chromosomes, say if S is equal to 2, then we will have p_1 and p_2 as parents and c_1 and c_2 as children, then we will have a 2-dimensional array 2x2 *distance*, where *distance*[0,0] is dist(p_1 , c_1), *distance*[0,1] is dist(p_1, c_2), and so on. dist function, in its order, can be formulated as:

Let function d, be a function of two arguments, x and y, where x and y, are sets of length m, then for $x_i \in x$ and $y_i \in y$, where $1 \le i \le m$, $d(x_i, y_i) = 0$ if $x_i = y_i$ and 0, otherwise. Then function dist is defined:

$$dist(x, y) = \sum_{i=1}^{m} d(x_i, y_i)$$

- In the fourth step, the algorithm matches parents and children in pairs, in a way that distance between them is minimized. For case when S is equal to 2, we will have two possible permutations of parent-child pairs:
  - $perm_1 = \{(p_1, c_1), (p_2, c_2)\}$
  - $perm_2 = \{(p_2, c_1), (p_1, c_2)\}$

  Then, using these permutations we calculate the total distance between each pair:

  - $d_1 = dist(p_1, c_1) + dist(p_2, c_2)$
  - $d_2 = dist(p_2, c_1) + dist(p_1, c_2)$

  Then, algorithm choose d_1 ,if $d_1 < d_2$, and d_2, otherwise.

  Clearly, it is not a very efficient step, because its complexity is O(S!), and with a big tournament size, it will slow down execution dramatically. Because of this, in this work, the maximum tournament size is fixed to 6, because it is not affecting algorithm performance very much.

- In the fifth step, the algorithm performs selection in each pair, the chromosome that has a bigger fitness score is selected to the next generation.
- In the last step, there is little adjustment to the original crowding algorithm, which removes duplicates from the new generation, and replaces them with new, randomly generated chromosomes. This step doesn't allow the population, to converge towards some fittest individual, preserving some level of diversity, and adding randomness to a search, which allows to explore solutions field wider.

Besides that, before execution of the genetic algorithm itself, on the stage of forming the initial population, to have a better population from the start, the algorithm generates 5000 individuals, and sorts them according to their fitness, after this, every 100th of this set goes to initial population, doing this, we have a chance to have strong individuals, and at the same time preserve diversity by selecting also weak ones.

2.4.2. Parallelization

In this work, the "island parallelization" method was used. The essence of which is that many genetic algorithms are executed in parallel and sometimes exchange chromosomes, which allows achieving better diversity in all populations.

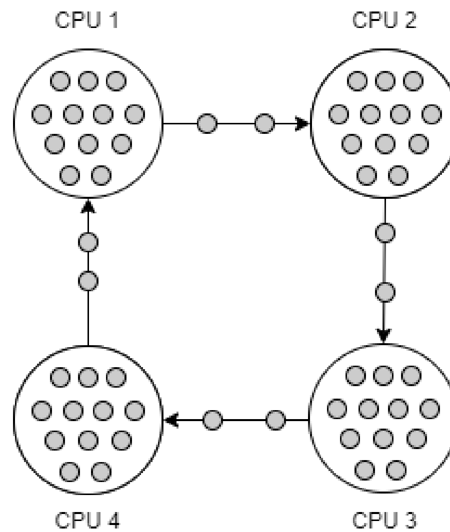

Figure 12. Island parallelization scheme

As shown in Figure 12, each genetic algorithm runs on a separate island, isolated from the others. Every 100 iterations, genetic algorithms synchronize and exchange chromosomes. Every fifth chromosome in the population is subject to exchange. To maintain elitism, the top 10 percent of the population remain untouched and are not exchanged. The islands are connected in a ring, so each one exchanges chromosomes with the island to its right. In addition, in order to add even more diversity to populations, each genetic algorithm has its own parameters for the probability of mutation and crossover, thus, all populations develop differently and take longer to converge to one common solution.

2.4.3. Crossover and mutation operators

Since chromosome genes are generally just a permutation of tower list, it is important to use special crossover and mutation that will avoid the generation of invalid offspring. For this purpose, many kinds of operators were designed, in this work modified version of crossover operator OX1 was used.

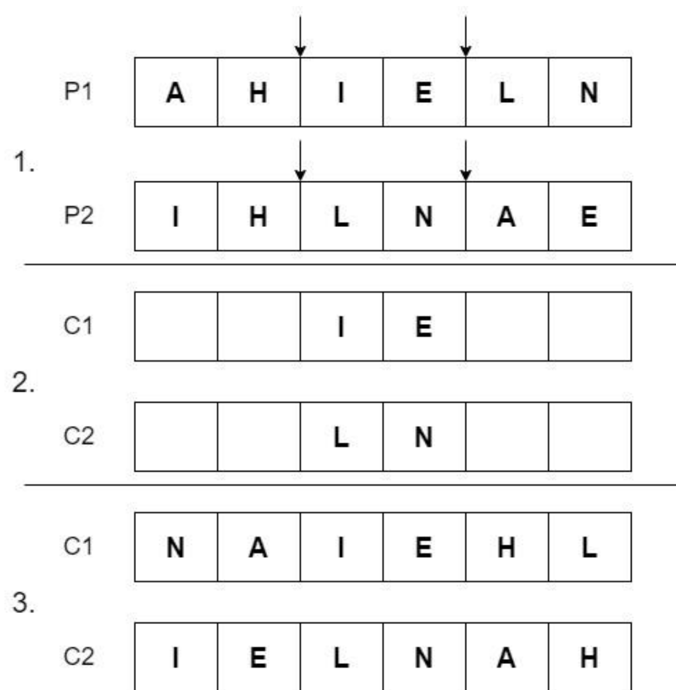

Figure 13. OX1 crossover operator scheme

Figure 13, schematically, shows the principal of OX1 crossover. At first, there are P_1 and P_1 chromosomes, that were selected for reproduction. The algorithm selects a random sector of the chromosome, and it is moved to the child chromosome, genes from P_1 are moved to child C_1 and from P_2 to C_2. Finally, missing genes are inserted in chromosomes, starting from the end of the sector, from another parent, in the order in which they appear in it, duplicates are skipped. So missing genes in C_1 are taken from P_2 and for P_1 for C_2. For this work, version with two sectors is used, however it could be adjusted to use *n* sectors.

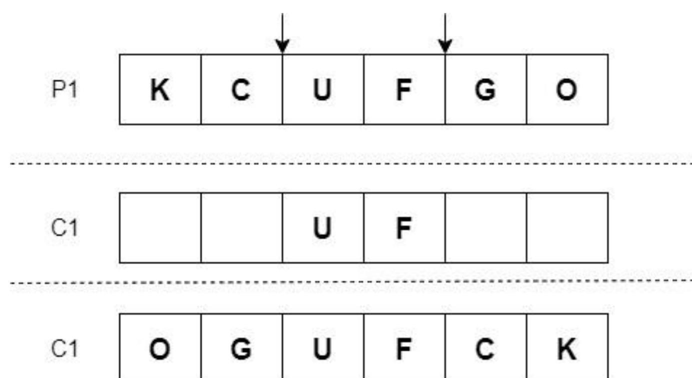

Figure 14. Inversion mutation operator

As a mutation operator, it was chosen to use inversion mutation. This mutation also was designed for permutation-based chromosomes. In the first step, it randomly chooses a sector in the parent chromosome, and this sector is moved to a child without modification, other genes are filled

from the parent in reversed order, also for filling a chromosome it randomly changes the tower's rotation.

2.4.4. Heuristic mutation and crossover

For this work, heuristic crossover and mutation also was designed. The idea was to create these operators specifically for CLP, when inversion mutation and order crossover create new individuals, they do it efficiently, but randomly because they are not aware of problem specifics. The basic idea for heuristic operators – is preserving the best rows of towers and constructing from them new individual. Heuristic mutation, preserves best rows from parent, while other rows are generated randomly. Heuristic crossover constructs a new child from the best rows from each parent, it is also very important, that rows should not overlap, which means that towers, that are used in these rows, should be unique.

2.5. Loading visualization

While developing the algorithm, it was decided to create a visualization engine, that would help to debug the algorithm, since it is a lot easier to check the correctness of loading when you can see it on the screen. For this purpose, the algorithm was extended with one additional step, that writes to the file x, y, and z coordinates of each box in the best loading that was achieved, also it writes the length, width, and height of each box and its group number. Also, at the start of the file, it writes container dimensions.

This file is then used in the engine, which draws the container and the boxes inside of it. Boxes are 50% transparent, so users can see through them, also every box group has its unique color, so it is easier to distinguish between different box types, the maximum number of groups supported is 12, since the more colors there are, the harder it becomes a select set of colors that are not similar. Finally, it is possible to rotate and move the camera in every direction, zoom in and zoom out.

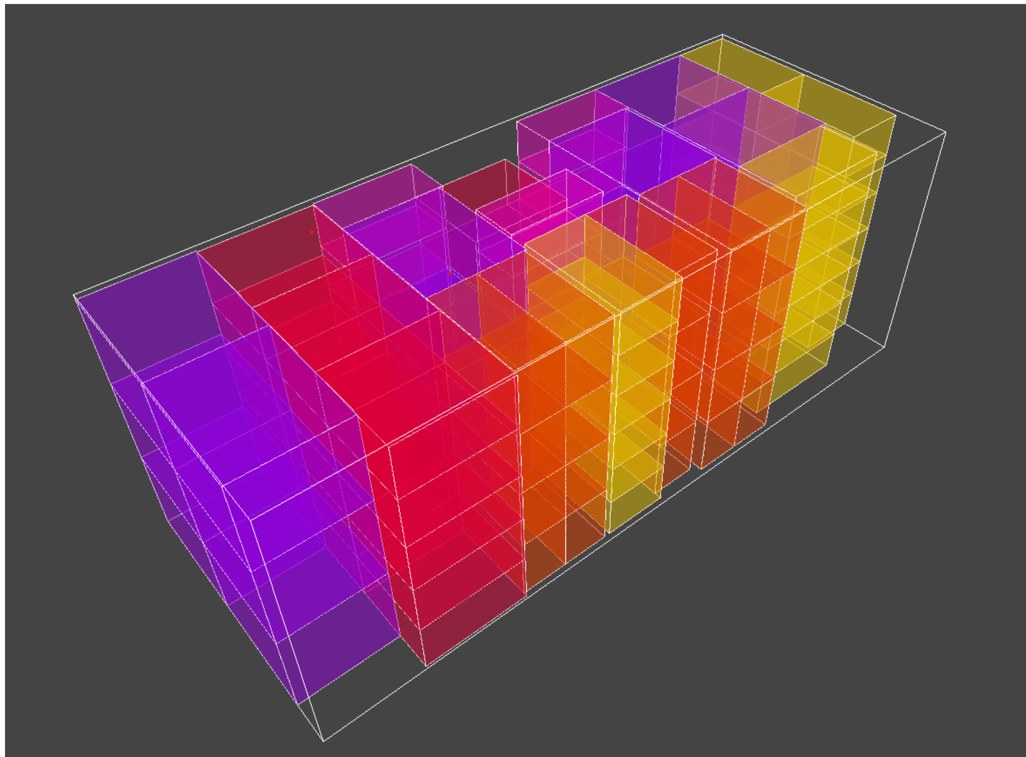

Figure 15. Visualization engine, full loading example

Figure 15, shows an example of visualization of the loading for CLP with 5 group types with %89 of volume utilization. It can be noticed that the algorithm tends to build towers with only one box type, it is relatively rare, that a tower has more than 2 box types in it.

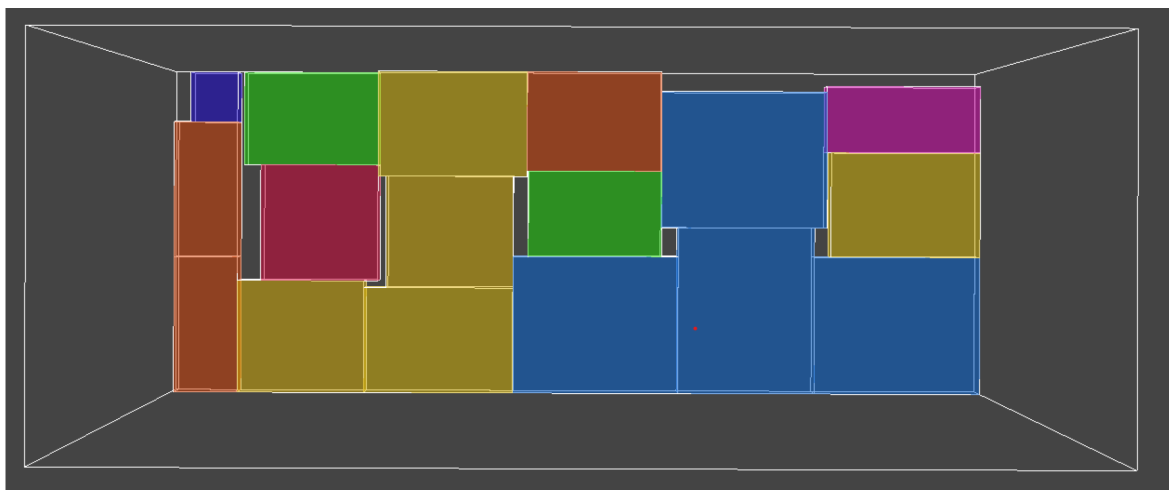

Figure 16. Visualization engine, floor-filling example

The algorithm can also be switched to draw only tower bases, so it is easier to see how the filling algorithm performs. In Figure 16, there is an example of how the packing filling heuristic fills the floor, it can be seen, that boxes are placed not in straight rows, but moved to the left side, where it is possible.

2.6. Test of Tower-building GA

To test the tower genetic algorithm, it was decided to use several test sets, one of them is created using real-live data from a company that ships Procter & Gamble products, this data set was taken from [DD10].

Table 2. Procter & Gamble's products test set.

Product Number	Description	width/ depth/height	Quantity
1	Detergent	40/36/28	325
2	Bleaching liquid	54/28/30	25
3	Personal care	54/28/30	75
4	Detergent	39/29/32	75
5	Shaving product	15/10/20	10
6	Baby care	42/37/25	150
7	Toothpaste	36/18/18	3
8	Shampoo	18/17/22	25
9	Shampoo	22/17/22	50
10	Shampoo	12/11/16	5
11	Bleaching liquid	30/27/40	20
12	Shaving product	19/7/21	3

In total, there are 12 product types and 766 boxes. The container size is 530 × 220 × 210, which is smaller than the standard size of the container. Using this data, the algorithm ran 30 times with different filling heuristics, and different mutation and crossover operators, also it was not mentioned does this data set has some orientation restriction, but to test orientation constraint rotation was fixed to two (just rotation of a box base).

Table 3. Experiments parameters

Algorithm type	Population size	Mutation Probability	Crossover Probability	Number of Iterations	Number of Processes
Basic	100	30%	90%	3000	1
Parallel	60	10%-50%	60%-100%	3000	12

Table 3 shows the parameters that were used for the experiments, all of them were found experimentally. The basic algorithm has fixed parameters, while the parallel one has different probability parameters for each of the processes, the range of these parameters is indicated in the table. Also, the parallel algorithm has a smaller population size, since in this case reducing the size does not affect the result, while increasing the execution speed.

Table 4. Procter & Gamble's product test result.

Algorithm	Container volume utilization/(dispersion)
Recursive heuristic + Inversion mutation + Order crossover	90.98% / (4.6)
Packing heuristic + Inversion mutation + Order crossover	90.46% / (4.8)
Packing heuristic + Heuristic mutation + Heuristic crossover	84.31% / (3.2)
Recursive heuristic + Heuristic mutation + Heuristic crossover	87.83% / (2.3)
Parallel + Packing heuristic + Inversion mutation + Order crossover	**92.17%** / (3.8)
Parallel + Recursive heuristic + Inversion mutation + Order crossover	**91.78%** / (2.9)
Genetic, SLFH [GML14]	91.13%
Genetic, MLFH [GML14]	91.4%
Simulated annealing [DD10]	87.51%

From Table 4, it can be seen that without parallelization the recursive algorithm works better than the packing one, the best result achieved by the recursive algorithm is 92.48%. This is due to the fact that the recursive algorithm itself is simpler, and it is easier for it to find a more optimal result faster, while the packing algorithm lacks the diversity and time to come to a better result. But this can be changed, with the help of parallelization of the algorithm, the parallel packing algorithm was able to bypass all other algorithms that worked with this dataset. The best result achieved is 93.01%.

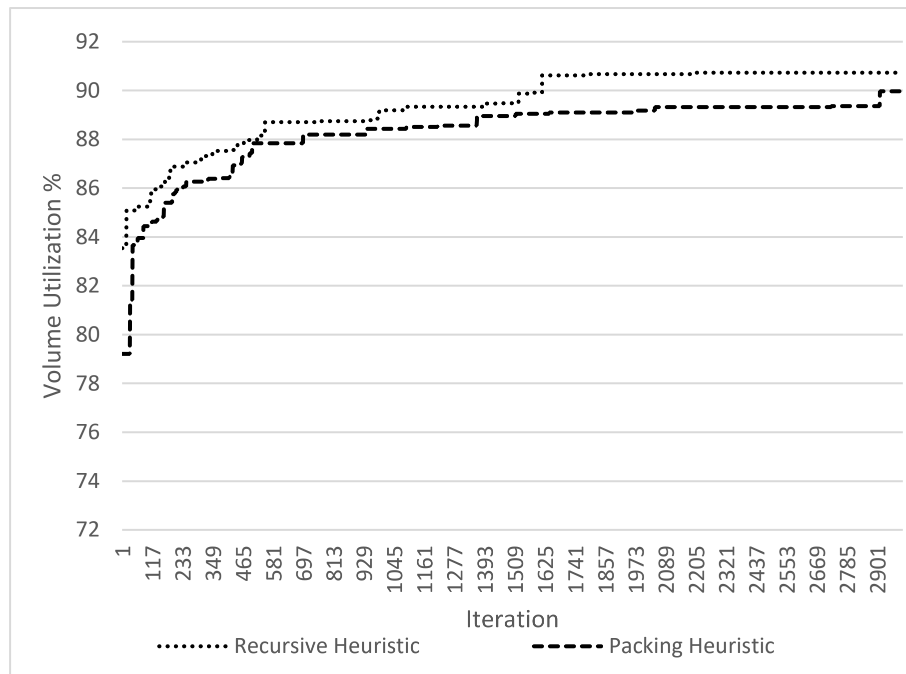

Figure 17. Recursive vs. Packing Heuristic

Figure 17, shows the difference between the execution of recursive and packing heuristics. The packing heuristic starts from a worse point (79.21%) and improves quickly, but slows down closer to the end of the execution, the final result is 89.97%. The recursive heuristic starts from a better point(83.54), and improves quicker during mid of run, but stacks in the third term, final result is 90.73%. From this plot, the suggestion could be made, if adjust genetic algorithm, so the packing heuristic will have a better starting point, it is possible to improve results.

It is also noticeable that heuristic mutation and crossover performed worse than simple ones, the reason for this is that they are very slow in exploration and tend to converge to local optimum. Preserving the best rows from chromosomes helps to build a stronger child, but in the same time, it lacks randomness to expand the search field.

Also, it is worth mentioning, that building tower constraints result field in a way that boxes can be placed only when the box is fully supported by the box underneath. On one side, it is harder to achieve good volume utilization, on the other side, such loading will be more stable during transportation, and it is easier to container with such loading because works usually use forklifts, that are loaded with towers.

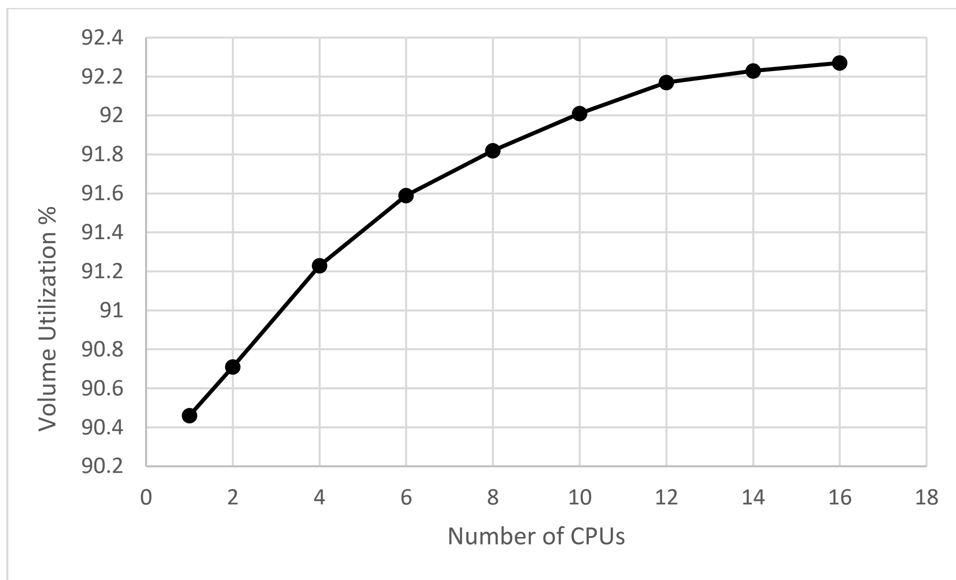

Figure 18. dependence of the final volume utilization on the number of CPUs

Figure 18 shows a chart of the increase in container volume utilization depending on the number of parallel processes. As it can be seen, after 12 processes the increase becomes insignificant, the difference between 14 and 16 processes is only 0.03 percent, which can practically be perceived as an error, because the algorithm is based on randomness. However, the best result that was achieved was 92.27 percent with 16 CPUs, using the packing heuristic with standard operators, population size fixed to 60 individuals.

Besides the test with Procter & Gamble products, it was decided to conduct another test with the generally used test set BR10. This test set consists of 1000 weakly heterogeneous test cases from Bishoff and Ratcliff[BR95] and Davies & Bischoff[DB99], number of box groups starts from 3 for BR1 and ends with 50 for BR10. Since this test is used in many other researches, it will help to compare this solution with others. Algorithm will be tested, with simple inversion and mutation operators, orientation constraints are also included. Parameters used for this experiment , which makes the algorithm work on average 50 seconds per problem for the packing heuristic.

Table 5. Comparison of results with BR7 test set.

Test case \ Algorithm	Parallel + Packing heuristic	Simulated annealing [DD10]	Tabu search [BG98]	Basic heuristic [BR95]	Wall-building Genetic Algorithm [Geh97]
BR1	87.56(3.5)	86.38	92.41	83.79	85.80
BR2	87.82(3.1)	87.70	92.33	84.44	87.26
BR3	89.26(2.5)	87.06	91.57	83.94	88.10
BR4	89.05(1.7)	86.61	91.26	83.71	88.04
BR5	89.25(1.6)	86.10	90.40	83.80	87.86
BR6	89.61(1.0)	85.47	89.57	82.44	87.85
BR7	89.10(1.4)	84.49	88.18	82.01	87.68
BR8	88.28(1.1)	-	86.26	-	87.09
BR9	86.62(1.2)	-	84.85	-	86.12
BR10	85.31(2.9)	-	83.65	-	85.24
Mean	88.73	86.26	89.1	83.45	87.10

As can be seen from Table 5, the new algorithm outperforms most other algorithms. It can be seen that the algorithm copes better with a large number of box options; the best result was achieved for the BR6 test, which has 15 different types of boxes. The Tabu search algorithm showed the best result, but as problem become more heterogeneous, created tower-building genetic algorithm outperforms it. It is also worth noting that the algorithm works much faster, single problem is solved in about 50 seconds, while the tabu search algorithm spends 250 seconds.

Results and Conclusions

During this work, an algorithm was developed for the optimization of the container loading problem. The algorithm is built using a genetic algorithm approach, to the novel idea of building towers from boxes and filling containers using these towers. Also, stability and orientation constraints were implemented in the algorithm.

For the genetic algorithm, heuristic mutation and crossover operators were created. In combination with GA, three heuristic algorithms were developed, two container floor filling heuristics (recursive filling and packing filling) and a tower building heuristic. With the algorithm itself, a visualization application was created, to visualize the achieved result, and debug the algorithm during implementation.

The algorithm was tested on a real-life Procter & Gamble products test set, where a parallel version of the algorithm with packing heuristic overtook every other algorithm, with an average result - 92.17%, which is 0.77% more than the previous best by MLFH genetic algorithm. For more broader comparison, a new approach was also tested with the BR10 test set, which is widely used in literature, it showed comparatively good results, by outperforming some of the most known algorithms.

Some algorithms can give better results in terms of free space utilization, but this particular approach stands out because it works comparatively fast, while for other metaheuristic algorithms, it takes much longer to calculate decent results. Another advantage of the tower building is naturally supported stability constraint, and much simpler loading from a practical point of view, because towers are the easiest form of shape to load, using a forklift.

Considering all this, the goal and tasks of this work were completed, and the test results proved the effectiveness of the algorithm.

References

[BG98] BORTFELDT, Andreas and GEHRING, Hermann. Ein Tabu Search-Verfahren fr Containerbeladeprobleme mit schwach heterogenem Kistenvorrat. *OR Spectrum*. 1998. Vol. 4, no. 20, p. 237–250.

[BGM03] BORTFELDT, A, GEHRING, H and MACK, D. A parallel tabu search algorithm for solving the container loading problem. *Parallel Computing*. May 2003. Vol. 29, no. 5, p. 641–662. DOI https://doi.org/10.1016/s0167-8191(03)00047-4.

[BR95] BISCHOFF, E.E. and RATCLIFF, M.S.W. Issues in the development of approaches to container loading. *Omega*. August 1995. Vol. 23, no. 4, p. 377–390. DOI https://doi.org/10.1016/0305-0483(95)00015-g.

[BW13] BORTFELDT, Andreas and WÄSCHER, Gerhard. Constraints in container loading – A state-of-the-art review. *European Journal of Operational Research*. August 2013. Vol. 229, no. 1, p. 1–20.

[BW18] BORTFELDT, Andreas and WÄSCHER, Gerhard. Container Loading Problems : a State-of-the-Art Review. *Working Paper Series*. 21 September 2018.

[DB99] DAVIES, A.Paul and BISCHOFF, Eberhard E. Weight distribution considerations in container loading. *European Journal of Operational Research*. May 1999. Vol. 114, no. 3, p. 509–527. DOI https://doi.org/10.1016/s0377-2217(98)00139-8.

[DD10] DERELI, Türkay and SENA DAS, Gülesin. A HYBRID SIMULATED ANNEALING ALGORITHM FOR SOLVING MULTI-OBJECTIVE CONTAINER-LOADING PROBLEMS. *Applied Artificial Intelligence*. 28 May 2010. Vol. 24, no. 5, p. 463–486. DOI https://doi.org/10.1080/08839514.2010.481488.

[GML14] GONZÁLEZ, Yanira, MIRANDA, Gara and LEÓN, Coromoto. A Multi-level Filling Heuristic for the Multi-objective Container Loading Problem. *Semantic Scholar*. Online. 2014. [Accessed 20 October 2023].

[GR80] GEORGE, J.A. and ROBINSON, D.F. A heuristic for packing boxes into a container. *Computers & Operations Research*. January 1980. Vol. 7, no. 3, p. 147–156. DOI https://doi.org/10.1016/0305-0548(80)90001-5.

[Geh97] GEHRING, Hartmut. A genetic algorithm for solving the container loading problem. *International Transactions in Operational Research*. 1 November 1997. Vol. 4, no. 5-6, p. 401–418. DOI https://doi.org/10.1016/s0969-6016(97)00033-6.

[Glo90] GLOVER, Fred. Tabu Search: A Tutorial. *Interfaces*. August 1990. Vol. 20, no. 4, p. 74–94. DOI https://doi.org/10.1287/inte.20.4.74.

[MG08] MENGSHOEL, Ole J. and GOLDBERG, David E. The Crowding Approach to Niching in Genetic Algorithms. *Evolutionary Computation*. September 2008. Vol. 16, no. 3, p. 315–354. DOI https://doi.org/10.1162/evco.2008.16.3.315.

[MPV00] MARTELLO, Silvano, PISINGER, David and VIGO, Daniele. The Three-Dimensional Bin Packing Problem. *Operations Research*. April 2000. Vol. 48, no. 2, p. 256–267. DOI https://doi.org/10.1287/opre.48.2.256.12386.

[NAJ21] NASCIMENTO, Oliviana Xavier do, ALVES DE QUEIROZ, Thiago and JUNQUEIRA, Leonardo. Practical constraints in the container loading problem: Comprehensive formulations and exact algorithm. *Computers & Operations Research*. April 2021. Vol. 128, p. 105186. DOI https://doi.org/10.1016/j.cor.2020.105186.

[Pis02] PISINGER, David. Heuristics for the container loading problem. *European Journal of Operational Research*. Online. 1 September 2002. Vol. 141, no. 2, p. 382–392. [Accessed 21 June 2021]. DOI https://doi.org/10.1016/S0377-2217(02)00132-7.

[RTS11] REN, Jidong, TIAN, Yajie and SAWARAGI, Tetsuo. A tree search method for the container loading problem with shipment priority. *European Journal of Operational Research*. November 2011. Vol. 214, no. 3, p. 526–535. DOI https://doi.org/10.1016/j.ejor.2011.04.025.

[SP94] SRINIVAS, M. and PATNAIK, L.M. Genetic algorithms: a survey. *Computer*. June 1994. Vol. 27, no. 6, p. 17–26. DOI https://doi.org/10.1109/2.294849.

[TSS+00] TERNO, Johannes, SCHEITHAUER, Guntram, SOMMERWEISS, Uta and RIEHME, Jan. An efficient approach for the multi-pallet loading problem. *European Journal of Operational Research*. June 2000. Vol. 123, no. 2, p. 372–381. DOI https://doi.org/10.1016/s0377-2217(99)00263-5.

[WHS07] WÄSCHER, Gerhard, HAUSSNER, Heike and SCHUMANN, Holger. An improved typology of cutting and packing problems. *European Journal of Operational Research*. Online. 2007. Vol. 3, no. 183, p. 1109–1130. [Accessed 6 January 2022]. DOI https://doi.org/10.1016/j.ejor.2005.12.047.

[ZBD14] ZHAO, Xiaozhou, BENNELL, Julia A., BEKTAŞ, Tolga and DOWSLAND, Kath. A comparative review of 3D container loading algorithms. *International Transactions in Operational Research*. Online. 7 May 2014. Vol. 23, no. 1-2, p. 287–320. DOI https://doi.org/10.1111/itor.12094.