F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

ROADEF Challenge 2022: Optimization of truck fleet loading

Bc. Tomáš Hromada

May 2023

Supervisor: Ing. David Woller

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hromada Tomáš** Personal ID number: **483629**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master’s thesis details

Master’s thesis title in English:

ROADEF Challenge 2022: Optimization of truck fleet loading

Master’s thesis title in Czech:

ROADEF Challenge 2022: Optimalizace nakládání flotily kamionů

Guidelines:

This thesis is motivated by ROADEF Challenge 2022, a prestigious international competition focused on solving newly formulated challenging combinatorial optimization problems from industrial applications.
Instructions:
1) Get familiar with standard techniques for solving large-scale problems of combinatorial optimization, especially metaheuristics. Research successful solution approaches to related problems (3D truck loading, delivery fleet management, task assignment) in the existing literature. Identify a suitable approach for the newly formulated competition problem.
2) Design and implement a custom optimization algorithm for the competition problem and participate in the competition.
3) Evaluate your method's performance on the available datasets and compare it with the results of other competitors.

Bibliography / sources:

[1] Potvin, J.Y. and Gendreau, M. eds., 2018. Handbook of Metaheuristics. Berlin/Heidelberg, Germany: Springer.
[2] Berbeglia, G., Cordeau, J.F., Gribkovskaia, I. and Laporte, G., 2007. Static pickup and delivery problems: a classification scheme and survey. Top, 15(1), pp.1-31.
[3] Yüceer, Ü. and Özakça, A., 2010. A truck loading problem. Computers & Industrial Engineering, 58(4), pp.766-773.

Name and workplace of master’s thesis supervisor:

Ing. David Woller Intelligent and Mobile Robotics CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **31.01.2023** Deadline for master's thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. David Woller Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgement / Declaration

I would like to express my sincere thanks to my supervisor, Ing. David Woller, for his exceptional guidance, insightful feedback, and continuous support, which significantly contributed to the accomplishment of this work.

Computational resources were provided by the e-INFRA CZ project (ID:90140), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of the university thesis.

Prague, 26 May 2023

.....

Abstrakt / Abstract

Tato diplomová práce se zabývá problémem optimalizace 3D nakládání kamiónů představeným v rámci týmové soutěže ROADEF/EURO Challenge 2022. Abychom uspokojivě řešili daný problém sestávající z velmi velkých instancí, vyvinuli jsme heuristickou metodu založenou na Iterative Local Search metodě, pro kterou jsme vyvinuli specifické operátory. Dále jsme vytvořili vlastní algoritmus pro vyhledávání validních konfigurací balíků v rámci nákadového prostoru kamiónu, metody vyvažování nákladu a perturbační proceduru. Pomocí dřívější verze navrhované metody jsme se kvalifikovali do finálového kola soutěže, které končí 15. června 2023.

Provedli jsme srovnání mezi řešeními generovanými naší metodou a nejlepšími známými hodnotami řešení zveřejněnými organizátory. Dosáhli jsme vynikajících výsledků na datové sadě A ze sprint kola soutěže a srovnatelných výsledků na datové sadě B z kvalifikačního kola.

Klíčová slova: Optimalizace, Nakládání kamionů ve 3D, Flotila kamionů, Local Search

This master thesis addresses the 3D truck loading optimization problem introduced in ROADEF/EURO Challenge 2022, which is a team competition. To tackle the given problem consisting of very large instances, we developed a heuristic method based on Iterative Local Search with problem-specific local search operators, a custom Depth-First Tree Search algorithm for finding feasible stack arrangements in trucks, Truckload balancing methods, and a perturbation procedure. Using an earlier version of this method we qualified for the competition's final round, which ends on June 15th, 2023.

We have conducted a comparison between the solutions generated by our method and the best-known objectives published by organizers. Our method demonstrates satisfactory performance, achieving strong results in Dataset A from Sprint round, and competitive outcomes in Dataset B from Qualification round.

Keywords: Optimization, 3D Truck Loading, Truck Fleet, Local Search

Contents /

Tables / Figures

Chapter 1
Introduction

The ROADEF/EURO Challenge [1] is a team competition that is held every two years and is organized by the French Society of Operations Research and Decision Support (ROADEF [2]) and the Association of European Operational Research Societies (EURO [3]). This event aims to provide an opportunity for industrial partners to stay up-to-date with the latest advancements in Operations Research and Decision Analysis while giving young researchers a chance to tackle a complex industrial optimization problem in the junior category. Additionally, the challenge serves as a platform for qualified researchers to connect with industrial partners through the senior category. Participating teams have the opportunity to compete for various financial prizes, including prizes for the top three teams and a prize for the best junior team.

This year, ROADEF/EURO Challenge 2022 is organized in cooperation with the Renault Group [4] and it is dedicated to a 3D truck loading optimization problem with numerous real-world constraints. The challenge lasts throughout the whole year [5] - it started in July 2022 and ends in June 2023. During this period, two rounds, the Sprint round (with a deadline on October 30th, 2022) and Qualification round (January 31st, 2023) already took place. Our team succeeded in the qualification round placing 3rd in the junior category and 9th among all 51 teams [6]. Since the qualification round our method was improved significantly and currently we are preparing for the final round which will take place on June 15th, 2023.

The proposed optimization problem addresses the transportation of a large number of items from suppliers to plants by utilizing a fleet of trucks while minimizing the costs. This complex problem combines several subproblems such as dividing items into stacks, assigning stacks to trucks, loading stacks into truck loading space while considering their physical dimensions and weights, and combining the cost of used trucks with items inventory cost. The problem includes many additional constraints on stack creation and stack loading into trucks. The large sizes of competition instances presented in each round in combination with a tight computation budget make this problem even more challenging and beyond the capabilities of generic commercial solvers such as Gurobi or CPLEX.

In this thesis, our heuristic approach to solving this problem is presented. We developed and implemented a method based on Iterated Local Search with a large number of problem-specific components such as Local Search operators, perturbation operators, and a custom depth-first search method for finding geometry-feasible stack arrangement in a truck.

The thesis is organized as follows.

Chapter 2 provides an overview of solutions for similar problems.

Chapter 3 gives a detailed description of the problem, notation, and all constraints.

Chapter 4 is dedicated to our proposed method with a full description of all details.

Chapter 5 provides a proposed method evaluation and a comparison of generated results and best-known results to given datasets.

Finally, **Chapter 6** presents a conclusion and an overview of the work.

Chapter 2
Related Works

The combinatorial problem introduced in this challenge combines several subproblems. One of them is the Container Loading Problem (CLP) - loading items into a truck and meeting all stack, placement, and weight constraints. Another subproblem is a variant of the Generalized Assignment Problem (GAP) [7] as we need to assign items to trucks and decide which trucks will be used. Another related problem can be considered the Capacitated Vehicle Routing Problem (CVRP). In this chapter, we review the existing literature on these problems.

Bortfeldt and Wäscher's work [8] provides a comprehensive overview of CLP, its variants, and the classes of constraints that can be applied. According to this work, CLP can be regarded as a geometric assignment problem, in which three-dimension items must be assigned to three-dimensional, rectangular large objects (containers) such that a given objective function is optimized and two basic geometric feasibility conditions hold, all items must be within the container and items can not overlap. Additionally, CLP can be subjected to other constraints, such as weight limits, weight distribution, forced items' orientations, and many others.

In GAP, the goal is to find an assignment of n tasks to m agents, where each task must be assigned to exactly one agent, while minimalizing cost subject to capacity restrictions on the agents.

In CVRP, we need to deliver demanded items to customers using one or more delivery vehicles with a defined capacity at minimum transit cost. This problem combines both the Bin Packing Problem [9] and the Traveling Salesman Problem [10].

In the following sections, we discuss the existing approaches (both exact and heuristic) to these problems and try to identify a suitable approach for our problem.

2.1 Exact Algorithms

CLP, GAP, and CVRP are known to be NP-hard problems, which makes the use of exact methods impractical for large-scale problems. Nonetheless, exact methods can be useful for small-sized problems. For instance, in [11] authors present a Mixed Integer Linear Program model for solving N-dimensional allocation problems (a variant of CLP). The study demonstrated that for small-sized problems, such as packing 13 items into 4 containers, the optimal solution can be obtained in a few seconds. A Mixed Integer Linear Programming model was also proposed in [12]. This work deals with the planning of truck scheduling for cross-dock centers and presents a case study at Renault company. The problem entities are very similar to entities from the problem introduced in the challenge. However, the work does not contain the Truck loading problem, only the item-truck assignment and truck scheduling. The proposed method was tested on instances with up to 3606 packages and 22 trucks.

Another exact approach is presented in [13], where the authors developed a branch-and-bound algorithm for CLP that can solve instances of up to 90 items. The problem on which the method was tested does not include any other constraints than two basic

geometric constraints on fitting items in the container without overlapping. In [14], a method based on depth-first-search and dynamic programming is proposed. The author studied the unconstrained variant of the CLP. The author's experiments showed the method is able to solve instances of up to 50 items. For the vast majority of 64 instances, an optimal solution was found. The last example of an exact method is presented in [15]. The authors developed a two-level tree search algorithm that can optimally the most of their 150 instances with up to 80 items, but the optimal solution was always found only for instances with up to 20 items. In [16] a branch-and-bound algorithm is presented for solving GAP. The computational results were obtained from instances with up to 4,000 0-1 variables. In [17], authors present an exact algorithm for solving CVRP based on the set partitioning formulation with additional cuts. The algorithm was able to compute CVRP with 76 customers in under one hour.

2.2 Heuristic Algorithms

While there exist some exact methods for solving the CLP their practical utility for large-scale problems is limited. Therefore, most research studies have focused on the development of heuristic and meta-heuristic algorithms. According to the state-of-the-art review [8], heuristic algorithms, particularly metaheuristics, are considered the most important class of algorithms for solving CLP in practice, even in the foreseeable future. This is because only these algorithms are capable of generating solutions of reasonable quality within reasonable computing time for realistic size problems (hundreds/thousands of items).

In [18], the authors present the approach for solving a pallet loading problem (a variant of CLP, where rectangular boxes need to be loaded on a 2D pallet). The first feasible solution is found quickly using a greedy method, then a complex branch-and-bound algorithm is applied repeatedly until the solution is proven optimal or the time limit is reached. A similar heuristic algorithm derived from a branch-and-bound algorithm for loading an open container is presented in [19]. The authors state the method was tested on 800 instances with up to 1000 items.

Several works studied using nature-inspired metaheuristics for these relevant combinatorial problems. For example, a method for solving CLP based on genetic algorithms is presented in [20]. Another example can be considered the algorithm for solving CLP (with a single container) introduced in [21]. The authors propose a method that hybridizes a novel placement procedure with a multi-population genetic algorithm based on random keys. The instance classes were based on additional constraints enforcement and the level of item heterogeneity. The results of experiments conducted by the authors showed that the approach performs well across all instance classes. In [22], a simulation approach is proposed to deal with the designing loading and transportation processes utilizing vehicle fleets in open-pit mines. The proposed approach is based on simulation combined with the firefly metaheuristic algorithm. Authors of other work [23] addressed the two-dimensional loading vehicle routing problem and proposed a hybrid approach to solve it. Specifically, a heuristic algorithm is used to solve the loading part, while the overall optimization is handled by an ant colony optimization (ACO) algorithm.

Many papers explore the usage of various of Local Search metaheuristics. Local Search [24] is a widely used approach to solve combinatorial optimization problems and is particularly suitable for NP-hard problems, for which it is not feasible to find an optimal solution in a reasonable amount of time. Instead, the goal is to find a good

suboptimal solution within a reasonable runtime. Well-known examples of local search metaheuristics are Iterative Improvement, Simulated Annealing (SA), and Tabu Search (TS), Iterative Local Search (ILS), Variable Neighborhood Search (VNS), Guided Local Search (GLS), and Greedy Randomized Adaptive Search Procedure (GRASP). In the following paragraph, several works using these metaheuristics are mentioned and briefly described.

Authors of [25] developed an algorithm for the CLP based on the GRASP paradigm. The method consists of two steps. The algorithm greedily builds a solution, and then it improves the solution with a local search algorithm to the first local optimum. A similar approach is also used in the work [26] in which authors present a heuristic algorithm based on GLS for solving a three-dimensional bin-packing problem. Firstly, a solution with an upper bound number of bins is found using the greedy method. Then it iteratively decreases the number of bins, each time searching for a feasible packing of the boxes. Experiments on instances with up to 200 boxes show a solid performance. Authors of [27] addressed the practical variant of the CVRP. They proposed a hybrid metaheuristic methodology that combines the strategies of TS and GLS. The conducted TS is periodically controlled by a guiding mechanism, which locates and penalizes low-quality features present in the candidate solution. TS heuristic is also proposed in [28] for solving GAP. In [29], the author introduced a solution method for the CVRP with Time Windows. The method uses four improvement operators in a steepest descent search strategy. The resulting algorithm was tested on Solomon's capacitated vehicle routing problems with time windows. The algorithm produces a better solution for longer routes than the other compared methods, for shorter routes it performed slightly worse. However, they have been able to generate 12 new best solutions for Solomon's problems.

After reviewing the literature on the existing solutions for CLP, GAP, CVRP, and other similar problems, we came to the conclusion that developing an efficient exact algorithm for the problem introduced in the challenge is not realistic. Based on the heuristic and metaheuristic approaches discussed in this chapter, we conclude that the Local Search approach is the most suitable solution method for this problem due to its ability to produce good solutions and its good scalability. Scalability is crucial for this problem, as we need to handle large instances with up to 200,000 items and several thousands of trucks while operating within a relatively small computing budget of 30-60 minutes.

Chapter 3
Problem Statement

The problem definition is fully described in official challenge documentation [30–31]. We provide a condensed problem description needed to understand our proposed method. In the rest of this chapter, we introduce the notation, the objective, and the constraints with all the necessary details, based on the challenge documentation. Additionally, we introduce our chosen solution representation.

3.1 Problem Description

The problem involves the transportation of a large number of items from suppliers to plants using trucks. The suppliers produce items, which can be loaded into trucks in a dock. A supplier can have multiple docks and produce multiple types of items. Similarly, a plant can have multiple docks, in which items can be unloaded from a truck. Figure 3.1 shows an example of truck routes.

There is a given set of items that need to be transported from a specific supplier dock to a specific plant dock. Each item must be delivered within a time window that consists of the earliest and the latest arrival time. If an item is delivered before the latest arrival time an inventory cost must be paid for each day before the latest date.

All items loaded into a truck must be packed into a stack. A stack can contain one or more items. There are multiple constraints (Section 3.4.2) on how items can be packed into stacks. For example, all items must share the same width and length, the maximal number of items stacked on each other is limited, the maximal density of a stack is defined, etc.

A set of planned trucks is given. Each truck can visit multiple predefined suppliers. At each supplier, the truck can stop at multiple docks and load stacks from them. Afterwards, the truck goes to a single plant, where it can stop at multiple docks and unload stacks. The order of visited suppliers and docks is given, and the stacks must be loaded onto a truck in the same order. Each truck has a set of items it can pick up. There are several stack placement constraints (Section 3.4.3) and also weight constraints (Section 3.4.4), such as a maximum weight limit for the total load and maximum weight limits for each truck axle that must not be exceeded.

Every planned truck has its cost that is paid only if the truck is used. If necessary, it is possible to duplicate any planned truck and create one or more extra trucks. However, the cost of an extra truck is higher than the cost of a planned truck.

The goal is to deliver all items and minimize the total cost. A solution consists of the definition of all stacks and their assigned trucks, 3D in-truck-coordinates of every item and stack, and a set of used trucks and their loading characteristics.

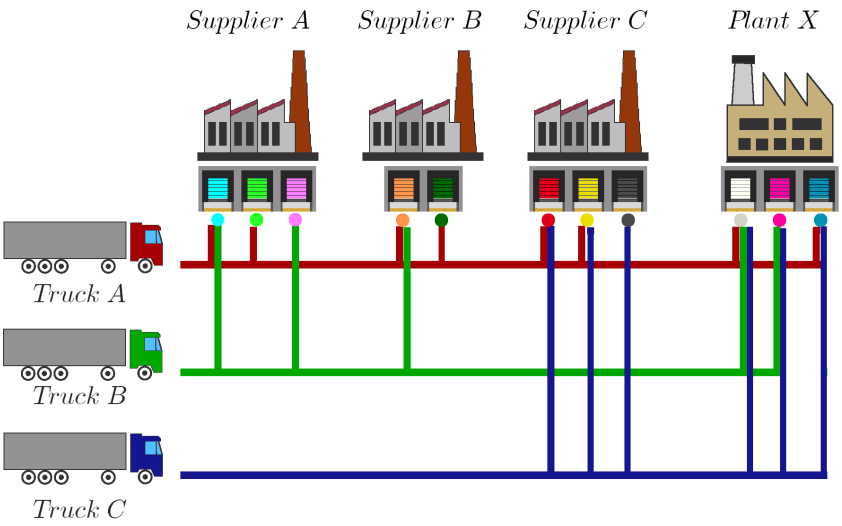

Figure 3.1. Example of routes taken by three different trucks labeled A, B, and C. Truck A stops at Supplier A, where it visits both the cyan and green docks, then proceeds to Supplier B, stopping at both the orange and dark green docks, before moving on to Supplier C, where it stops at the red and yellow docks. Finally, it arrives at Plant X and stops at the white and teal docks. Truck B travels from Supplier A, stopping at the cyan and pink docks, before moving on to Supplier B, where it stops only at the orange dock. Then it arrives at Plant X and stops at the white and pink docks. Truck C travels to Supplier C and stops at all three docks before proceeding to Plant X, where it also visits all the docks available.

3.2 Notations

The following notation is adapted from the challenge documentation. We define input data and solution variables. All constants and variables representing distance are in millimeters (mm), all weights are in kilograms (kg), cost in euros (EUR), and stack density is in kg/m^2. To better understand constraints and variables related to trucks and the computation of truck weight on axles see Figure 3.3.

3.2.1 Input Constants

Input data contains the following constants.

Entity sets

- $\tilde{I}$: set of items i
- $\tilde{P}$: set of plants p
- $\tilde{U}$: set of suppliers u
- $\tilde{G}$: set of plant docks g
- $\tilde{K}$: set of supplier docks k
- $\tilde{\gamma}$: set of products γ - each item has its product code

Items

- IL_i, IW_i, IH_i, $I\hat{H}_i$, IM_i : length, width, height, nesting height (see Figure 3.2), weight of item i
- IS_i : stackability code of item i,
- ISM_i : maximal stackability of item i

- IR_i : product of item i, $IR_i \in \tilde{\gamma}$
- IO_i : forced orientation of item i
- IDE_i, IDL_i : earliest and latest arrival time of item i
- IP_i : destination plant of item i
- IG_i : plant dock of item i
- IU_i : supplier of item i
- IK_i : supplier dock of item i
- IC_i : inventory cost of item i

Trucks

- $T\tilde{R}_t$: set of candidate products picked-up by truck t
- $T\tilde{U}_t$: set of candidate suppliers visited by truck t
- $T\tilde{K}_{ut}$: set of candidate supplier docks k of supplier u loaded into truck t
- $T\tilde{G}_{pt}$: set of candidate plant docks g of plant p delivered by truck t
- TL_t, TW_t, TH_t, TM_t^m : length, width, height, max authorized loading weight of truck t
- TP_t : destination plant of truck t
- TDA_t : arrival time of truck t at TP_t
- $TMM_{t\gamma}$: maximal total weight of all the items packed above the bottom item associated with product γ in any stack of truck t
- TF_t : flag 'stack with multiple docks' for truck [t, $true$ / $false$] - this flag allows loading stacks with multiple plant docks into truck t
- TEM_t : maximal density of stacks in truck t
- TC_t : cost of truck t
- TE_t : supplier loading order for truck t : it is a list indexed by elements of $T\tilde{U}_t$ containing for each supplier its loading order
- TKE_{ut} : dock loading order of supplier u for truck t : it is a list indexed by the elements of $T\tilde{K}_{ut}$ containing for each supplier dock its loading order
- TGE_{pt} : dock loading order of plant p for truck t : it is a list indexed by the elements of $T\tilde{G}_{pt}$ containing for each plant dock its loading order
- CM : weight of the tractor
- CJ^{fm} : distance between the front and middle axles of the tractor
- CJ^{fc} : distance between the front axle and the center of gravity of the tractor
- CJ^{fh} : distance between the front axle and the harness of the tractor
- EM : weight of an empty trailer
- EJ^{hr} : distance between the harness and the rear axle of the trailer
- EJ^{cr} : distance between the center of gravity of the trailer and the rear axle
- EJ^{eh} : distance between the start of the trailer and the harness
- EM^{mr} : max weight on the rear axle of the trailer
- EM^{mm} : max weight on the middle axle of the trailer

Parameters

- α^T : coefficient of transportation cost in the objective function
- α^I : coefficient of inventory cost in the objective function
- α^E : coefficient of cost for extra trucks

3.2.2 Solution Variables

The following variables define a solution and are used in objective computing or constraint checking.

Variables

- $\hat{T}$: set of used trucks
- $\hat{TS}_t$: set of the stacks packed into truck t
- $\hat{SI}_s$: set of the items of stack s
- $\hat{SG}_s$: set of the plant docks of stack s
- sl_s, sw_s, sh_s, sm_s : length, width, height, weight of stack s
- sx^o_s, sy^o_s, sz^o_s : coordinates of the origin point of stack s
- sx^e_s, sy^e_s, sz^e_s : coordinates of the extremity point of stack s. ($sx^e_s = sx^o_s + sl_s$, $sy^e_s = sy^o_s + sw_s$, $sz^e_s = sz^o_s + sh_s$)
- so_s : orientation of stack s
- su_s : supplier of stack s
- sk_s : supplier dock of stack s
- st_s : truck of stack s
- ida_i : arrival time of item i
- tm_t : weight of the stacks loaded into the truck t
- ej^e : distance between center of gravity of stacks and the start of the trailer
- ej^r : distance between center of gravity of stacks and the rear axle of the trailer
- em^h : weight on the harness of the trailer
- em^r : weight on the rear axle of the trailer
- em^m : weight on the middle axle of the trailer

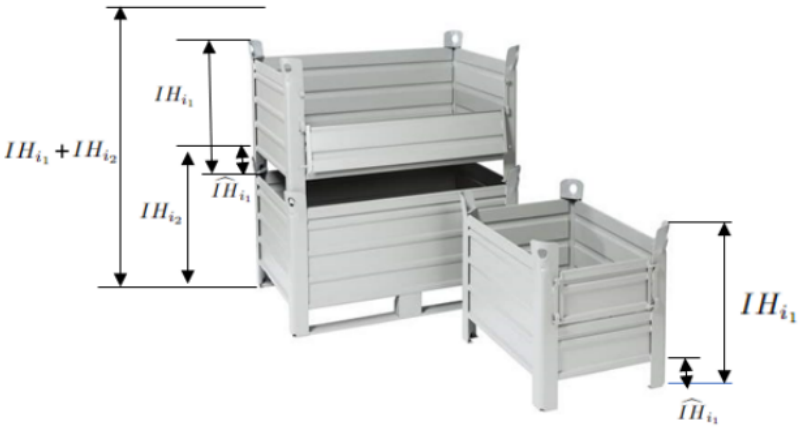

Figure 3.2. Example of a stack with items whose nesting height is greater than 0. Figure source: [30].

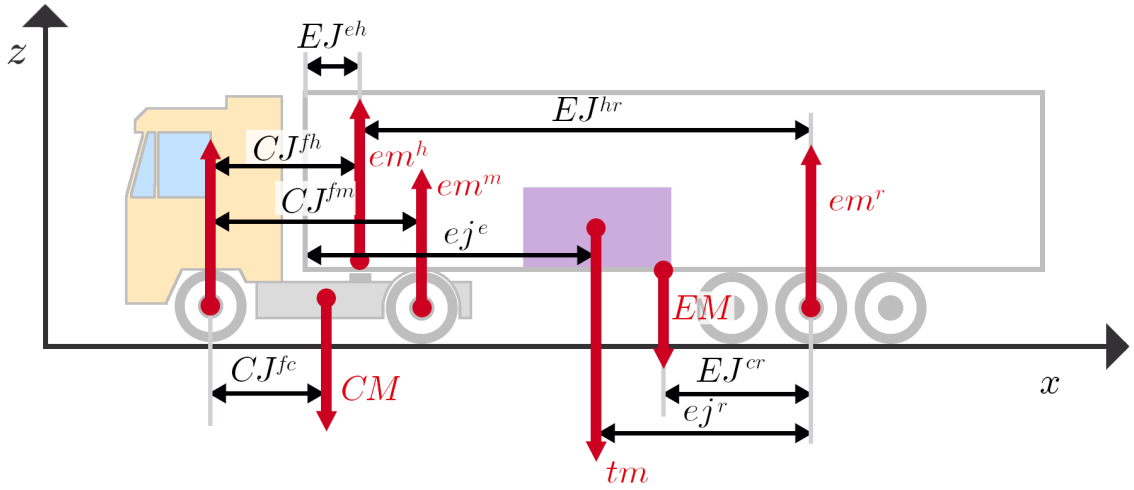

Figure 3.3. Figure showing constants and variables related to axles load computation

3.3 Objective

The goal is to transport all items from their suppliers to their destination plants in the defined time window while minimizing the combined cost of all trucks and the inventory cost of items. See the objective function (1) below.

$$\min \quad \alpha^T \times \sum_{t \in \tilde{T}} TC_t + \alpha^I \times \sum_{i \in \tilde{I}} IC_i \times (IDL_i - ida_i) \tag{1}$$

3.4 Constraints

A valid solution must meet multiple hard constraints. Constraints are divided into four groups: Items Constraints, Stack Constrains, Placement Constraints, and Weight Constraints.

3.4.1 Items Constraints

- **(I1)**: All items must be packed into stacks, which must be loaded into trucks.
- **(I2)**: An item i can be loaded into a truck t only if t arrives at the item's plant IP_i.
- **(I3)**: An item i can be loaded into a truck t only if t can pickup the item's product IR_i.
- **(I4)**: An item i can be loaded into a truck t only if t stops by the item's supplier IU_i
- **(I5)**: An item i can be loaded into a truck t only if t arrives at the plant in the item's time window:

$$IDE_i \leq TDA_t \leq IDL_i \ . \tag{2}$$

3.4.2 Stacks Constraints

- **(S1)**: All the items i packed in a stack s must share the same supplier IU_i, plant IP_i, stackability code IS_i and supplier dock IK_i.
- **(S2)**: For any stack s, packed into truck t and if $(TF_t = no)$, then all the items i of stack s must share the same plant dock IG_i.
- **(S3)**: For any stack s, packed into truck t, if $(TF_t = yes)$, stack s may contain items i_1, i_2 with 2 plant docks IG_{i_1}, IG_{i_2} with consecutive loading orders.
- **(S4)**: If one item i of a stack s has a forced orientation IO_i, then all the items of stack s must share the same orientation $(so_s = IO_i)$.
- **(S5)**: For any stack s, packed into truck t, the total weight of the items packed above the bottom item associated with product γ must not exceed the maximal weight $TMM_{t\gamma}$:

$$\sum_{i \in \hat{SI}_s \,\wedge\, i \neq bottom} IM_i \leq TMM_{t\gamma} \; . \tag{3}$$

- **(S6)**: The number of items packed into a stack s must not exceed the smallest 'max stackability' ISM_i of the items i present in stack s.
- **(S7)**: The density of a stack s must not exceed the maximal stack density defined for the truck t into which the stack s is loaded:

$$\frac{sm_s}{sl_s \times sw_s} \leq TEM_t \; . \tag{4}$$

3.4.3 Placements Constraints

- **(P1)**: The placement of a stack s into a truck t must not exceed the truck's dimensions:

$$\forall t \in \hat{T}, \forall s \in \hat{TS}_t, sx^e_s \leq TL_t \wedge sy^e_s \leq TW_t \wedge sz^e_s \leq TH_t \; . \tag{5}$$

- **(P2)**: The stacks packed into a truck t cannot overlap:

$$\forall t \in \hat{T}, \forall s1, s2 \in \hat{TS}_t \; (sx^o_{s1} \leq sx^o_{s2} \;\wedge\; sx^o_{s2} < sx^e_{s1}) \;\Rightarrow\; sy^o_{s2} \geq sy^e_{s1} \vee sy^e_{s2} \leq sy^o_{s1} \; . \tag{6}$$

- **(P3)**: Any stack must be adjacent to another stack on its left on the X axis, or if there is a single stack in the truck, the unique stack must be placed at the front of the truck (adjacent to the truck driver). Figure 3.4 provides an example of incorrect and correct stack configurations in a truck.

$$\forall t \in \hat{T}, \forall s1 \in \hat{TS}_t \; (sx^o_{s_1} > 0) \Rightarrow$$
$$\Rightarrow \exists s_2 \in \hat{TS}_t, \;\; sx^e_{s_2} = sx^o_{s_1} \wedge (sy^o_{s_2} \in [sy^o_{s_1}, sy^e_{s_1}] \;\vee\; sy^e_{s_2} \in [sy^o_{s_1}, sy^e_{s_1}]) \tag{7}$$

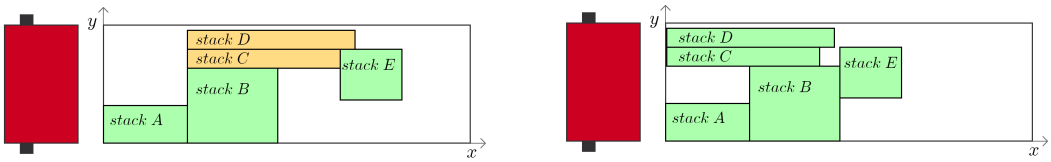

Figure 3.4. Figure demonstrates P3 constraints. Stacks C and D in the left truck violate the P3 constraint because they are not adjacent to any stack or front of the truck. While all stacks in the right truck are placed correctly with respect to the P3 constraint.

- **(P4)**: The stacks loaded onto the truck must follow a specific order. (1) They must be placed in ascending order from the front to the rear of the truck in accordance with the supplier's pickup order. See Figure 3.5. (2) Among the stacks from the same supplier, they must also be arranged in ascending order based on the supplier dock loading order. (3) If there are multiple stacks from the same supplier and supplier dock, they must be organized in increasing order according to the plant dock loading order. See Figure 3.6. The goal is to fulfill these three requirements. It is not required to optimize the unloading process.

$$\forall t \in \hat{T}, \forall s_1 \in \hat{TS}_t, \forall s_2 \in \hat{TS}_t,$$

$$(1)\ \ (TE_{u_{s_1}} < TE_{u_{s_2}}) \Rightarrow sx^o_{s_1} \leq sx^o_{s_2} \tag{8}$$

$$(2)\ \ (u_{s_1} = u_{s_2} \wedge KE_{k_{s_1}} < KE_{k_{s_2}}) \Rightarrow sx^o_{s_1} \leq sx^o_{s_2} \tag{9}$$

$$(3)\ \ u_{s_1} = u_{s_2} \wedge KE_{k_{s_1}} = KE_{k_{s_2}} \wedge GE_{g_{s_1}} < GE_{g_{s_2}}, \forall g_{s_1} \in \hat{SG}_{s_1}, \forall g_{s_2} \in \hat{SG}_{s_2}) \Rightarrow$$
$$\Rightarrow sx^o_{s_1} \leq sx^o_{s_2} \tag{10}$$

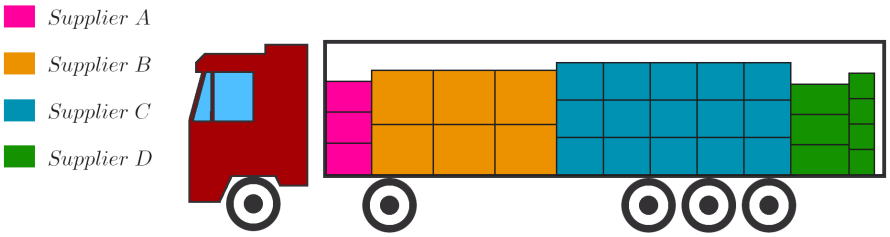

Figure 3.5. Truck loaded with stacks from four different suppliers meeting the P4 constraint, where $TE_{SupplierA} < TE_{SupplierB} < TE_{SupplierC} < TE_{SupplierD}$.

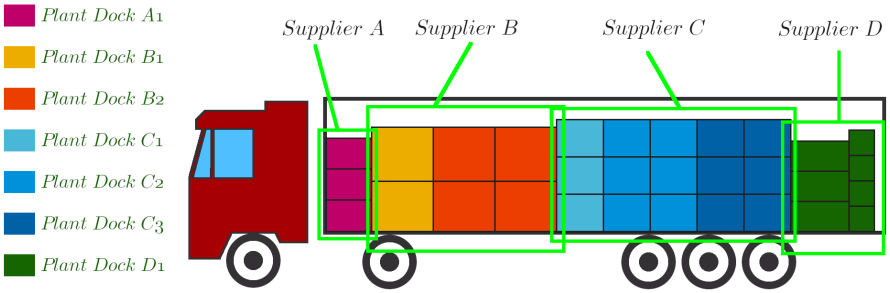

Figure 3.6. Same truck as presented in Figure 3.5, Plant docks of stacks are shown, where $GE_{PlantDockB1} < GE_{PlantDockB2}$ and $GE_{PlantDockC1} < GE_{PlantDockC2} < GE_{PlantDockC3}$.

3.4.4 Weight Constraints

- **(W1)**: The weights of the stacks packed in a truck t should not exceed the truck's maximal loading weight:

$$\forall t \in \hat{T}\ \ tm_t \leq TM^m_t,\ \ tm_t = \sum_{s \in S_t} sm_s \tag{11}$$

- **(W2)**: The weights on the middle axle and on the rear axle of a truck must not exceed the max weights authorized for these 2 axles:

$$em^m \leq EM^{mm}\ \wedge\ em^r \leq EM^{mr} \tag{12}$$

Figure 3.3 provides a clearer understanding of the weights of truck axles. The constraint (W2) must be satisfied every time the truck is on the road. For example, if the truck stops at supplier A, and at supplier B, then goes to plant X, the constraint must be satisfied with only items from supplier A in the truck while driving from supplier A to B, and also with all the items from suppliers A and B, when the truck goes from supplier B to plant X.

3.5 Solution Representation

We chose the following representation of a solution for the presented problem. A solution is encoded as a vector of nodes with variable length. There are four types of nodes: truck start node, item node, stack orientation node, and stack position node.

A solution vector is divided by the truck start node into sub-vectors representing each truck's load. Additionally, the truck start node encodes the identifier of the used truck. After the truck start node, the rest of the sub-vector defines stacks and items loaded into this truck. Before each sector representing a stack, k, $k \geq 0$ stack position nodes can be inserted. Number k encodes the chosen position of the stack (see Section 4.1 for more details). After these nodes, item nodes follow. Each item node defines a single item and its type packed into the current stack. The order of item nodes matches the order of items packed into the current stack bottom-up. A stack is terminated with a stack orientation node, which encodes the stack orientation. An example of a simple solution vector is shown in Figure 3.7.

This representation has been chosen, because it is suitable for performing Local Search operations, such as swapping stacks in trucks. Also, it can be efficiently implemented using a C++ vector of integers.

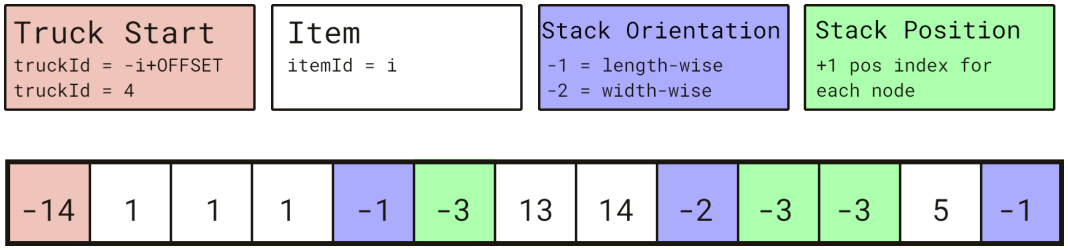

Figure 3.7. Example of solution encoding: the vector encodes solution with one truck of type id=4, 3 stacks, where the first stack is located at a position with index 0, contains 3 items of type 1, and is oriented length-wise. The second stack contains items 13 and 14, it is located at a position with index 1 and is oriented width-wise. The last stack contains only one item with id = 5, is located at a position with index 2, and is oriented length-wise.

Chapter 4
Proposed Method

In this chapter, we present our proposed method for solving the problem introduced in the previous chapters. Our method is based on the Iterated Local Search (ILS) metaheuristic, augmented with a custom Depth-First Tree Search (DFTS) algorithm that explores feasible stack arrangements in trucks. To address the specific requirements of this problem, we have designed six Local Search operators tailored to optimize the solution. Additionally, our method incorporates an algorithm for truckload balancing, which ensures that the weight is distributed evenly allowing adding new stacks to a truck.

The solution generation consists of three main steps. First, an initial solution is obtained using a greedy method (line 3 Alg. 4.0). This solution serves as a starting point for subsequent improvement.

Next, the Local Search operators (line 4 Alg 4.0) are applied to iteratively refine the solution by exploring the local neighborhoods. The operators make incremental changes to the solution, aiming to improve the objective function. This iterative process continues until a local minimum is reached, where no further improvement can be made.

Finally, to avoid local optima, the method employs a perturbation strategy (line 7 Alg. 4.0). The best current solution is repeatedly perturbed to escape from the local optimum, which increases the chances of finding a better global solution. If a better solution is found the solution is stored as the current best solution (lines 10-13 Alg. 4.0). This process continues until a time limit is reached (line 6 Alg. 4.0).

In the following sections, we will provide a detailed description of each component of our proposed method and implementation details.

```
1  # Main method
2  time_start ← getTime()
3  sol_init ← findSolutionGreedy()
4  sol_best ← reachLocalMinimum(sol_init)
5  obj_best ← calculateObj(sol_best)
6  while getTime() - time_start < timeLimit do
7      sol_new ← perturbate(sol_best)
8      sol_new ← reachLocalMinimum(sol_new)
9      obj ← calculateObj(newSol)
10     if obj < bestObj then
11         obj_best ← obj
12         sol_best ← sol_new
13     end if
14 end while
15 return sol_best
```

Alg. 4.0. Pseudo-code of the proposed method described in high-level.

4.1 Stack Placement

All items must be packed in a stack. These stacks are placed into a truck, there are several constraints (see Section 3.4.3) that a stack placement has to satisfy. Stacks are loaded into a truck in the same order as the order in the solution vector. Considering the vast number of possibilities for arranging stacks in a truck, we propose the following structure of Loading space, which thanks to its properties assures meeting constraints P1, P2, and P3 without any additional checks.

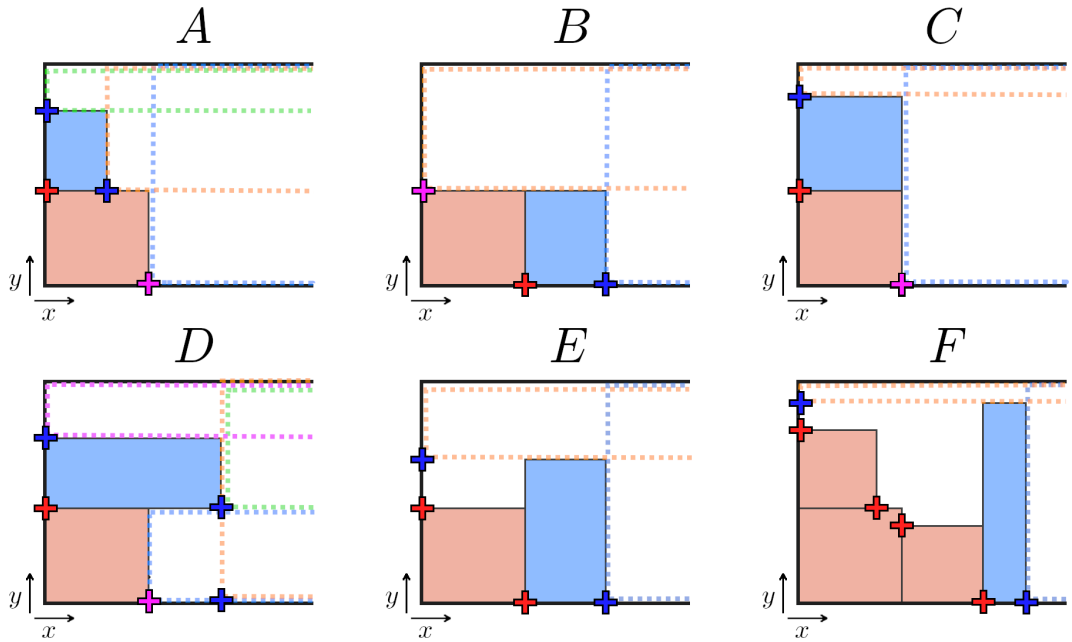

Figure 4.1. The figure illustrates updating of the candidate positions list on six examples. The red boxes represent stacks already placed in the truck, the blue boxes are newly added stacks, red and purple crosses represent candidate positions before insertion while blue and purple crosses denote candidate positions after the insertion. Dashed lines demarcate available areas for each candidate position.

The Loading space structure comprises a list of candidate positions where a stack can be placed. Each candidate position is represented as a tuple $(x, y, length, width, mW)$, which defines an available area. Here, x and y denote the coordinates of the area's origin, $length$ and $width$ represent the dimensions of the area, and mW indicates the minimum required width for a stack to fit within this area. The minimum width is specifically determined for the case illustrated in the example D in Figure 4.1, for the candidate position represented by the blue cross with $y = 0$, it must be defined mW greater than zero. Otherwise, a new stack could not be adjacent to the blue stack.

The candidate list is ordered primarily in ascending order of x and secondarily in descending order of y.

The candidate positions list is defined for a given stack arrangement. If a stack is added or removed the list must be updated accordingly. When a truck t is empty, the initial candidate position $(0, 0, TL_t, TH_t, 0)$ is added to the candidate list. Subsequently, as stacks are placed in the truck, the candidate list is updated. The new position candidates are derived from the corners of the added stack. The algorithm for updating candidate positions is illustrated in examples in Figure 4.1.

The algorithm used for generating the candidate list is designed to be simplified, which means it may not discover all possible stack arrangements. Also, this approach

has a tendency to create gaps in the loading space and narrow unoccupied spaces. In examples D, E, and F shown in Figure 4.1, it can be observed that placing a stack to some of the generated candidate positions results in the creation of small unutilized areas between stacks. Despite these limitations, the algorithm is capable of finding the vast majority of arrangements. Based on the empirical evaluation, we have determined that this approach is satisfactory for our purposes. An example of stack arrangements with data from an introduced instance is shown in Figure 4.2.

As mentioned in the previous chapter in Section 3.5 presenting solution representation, stack position in a truck is defined using position nodes. The number of position nodes inserted before a stack defines the index of the selected candidate position from the candidate position list. For example, if a new stack s was added to the truck shown in Figure 4.2 and 3 position nodes are preceding, the stack s is placed at the position marked 4 with coordinates $(sx_s^o, sy_s^o) = (130000, 0)$ in the figure. It is important to note that the position indexes depend on the stacks loaded before the current stack s. If a stack preceding stack s is added or removed, the candidate positions list must be updated, along with the number of position nodes for stack s.

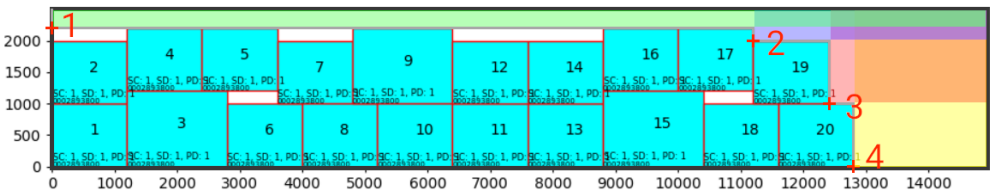

Figure 4.2. The figure illustrates the algorithm for finding free space for a stack placement. The corners of the position candidate areas are marked as red crosses with numbers representing the order in the list, the corresponding area is shaded. Note the white gaps between stacks.

4.2 Initial Solution

The initial solution is obtained using a greedy algorithm. The algorithm begins with an empty set of used trucks $\hat{T}$. The items $\tilde{I}$ are pre-sorted by supplier dock, plant dock, and time window. Then, each item $i \in \tilde{I}$ is added to a truck one by one. The item i can be added in one of three ways: 1) to an already existing stack in a truck $t \in \hat{T}$, 2) to a newly created stack placed in a truck $t \in \hat{T}$, or 3) to a newly created stack in a truck t, which will be added to $\hat{T}$. The option that is both feasible and minimizes weighted increment of objective value is selected. The weighted increment of objective value for ways 1) and 2) is computed as $\alpha^I \times IC_i \times (IDL_i - TDA_t)$, while for option 3), it is computed as $\gamma \times \alpha^T \times TC_t + \alpha^I \times IC_i \times (IDL_i - TDA_t)$. Here, γ is a parameter to weigh adding new trucks. If two options have the same value, the option with a lower index is chosen. The position for a newly created stack in a truck is determined as the first feasible position from the candidate positions list. This process continues until all items are loaded into trucks $\hat{T}$.

This algorithm quickly generates a feasible solution. Best results for instances from Dataset A and B were achieved when the parameter γ was set to a small positive number (e.g., 0.0005). The algorithm, with this setup, generates solutions that minimize the inventory part of the objective. This approach is based on the observation that the inventory cost of items is significant and can be much higher than the cost of trucks. Also, if the initial solution contains more trucks than needed, Local Search

operators perform better due to the larger search neighborhood available. Furthermore, we leverage the fact that an initial solution contains more trucks, and we specifically design our Local Search operators to focus on deleting trucks. However, for some instances from Dataset C, which have a high value of α^T, it is possible that a different value of the parameter γ may be more suitable. Note, that at the time of writing this work, the method has not been fully tuned for instances from Dataset C yet.

4.3 Truck Load Balancing

One of the constraints introduced in the challenge is that maximal limits on the middle and rear truck axle can not be exceeded (W2). The following formulas (1), (2), (3), (4), (5) stated in [30] can be used for the computation of weights on both axles.

$$ej^e = \frac{\sum_{s \in T\hat{S}_t} (sx_s^o + \frac{(sx_s^e - sx_s^o)}{2}) \times sm_s}{tm_t} \tag{1}$$

$$ej^r = EJ^{eh} + EJ^{hr} - ej^e \tag{2}$$

$$em^h = \frac{tm_t \times ej^r + EM \times EJ^{cr}}{EJ^{hr}} \tag{3}$$

$$em^r = tm_t + EM - em^h \tag{4}$$

$$em^m = \frac{CM \times CJ^{fc} + em^h \times CJ^{fh}}{CJ^{fm}} \tag{5}$$

Note that stacks are loaded onto a truck from the beginning of the loading space toward its end and the middle axle has a significantly smaller weight limit than the rear axle (e.g., $EM^{mm} = 12000$, $EM^{mr} = 31500$). We observed that sometimes stacks are loaded only in the first half of the truck and no more stacks can be added because it would overload the middle axle. However, if some stacks were placed further than the rear axle or even if ej^e was greater, it would lower the weight on the middle axle and allow adding new stacks into the truck.

Based on this motivation we introduce a method that iterates over trucks and searches for trucks t that have em^m near their limit while loading length is less than 80% of TL_t. Stacks in these trucks are split into two in order to move the load center of mass ej^e toward the end of the truck.

The method is implemented in the following way. The first stack s with the lowest sx_s^o (closest to the beginning of truck loading space), and containing more than one item, is selected for splitting. The top item of this stack is moved to a newly created stack s_{new}, now with one item. If possible, stack s_{new} is placed before the original stack s, with the two stacks being adjacent to each other, such that $sx_s^o = sx_{s_{new}}^e$. Otherwise, the search for other splittable stacks continues. If the stack was successfully split and placed, the truckload center of mass is shifted toward the rear axle, which is shown in the upper half of Figure 4.3. This process is repeated until either the middle axle is no longer nearly overloaded or no more feasibly splittable stacks are found.

This approach, however, introduces a new problem. Let's consider the following example: a truck's middle axle becomes nearly overloaded, the method splits stacks and increases ej^e, effectively lowering the weight on the middle axle. This adjustment

enables the addition of new heavy stacks, which are placed beyond the rear axle, further lowering the weight on the middle axle. However, the truck loading space becomes fully occupied and no more stacks can be added. In such cases, where split stacks are no longer needed, we created an inverse method that allows merging stacks and lowering the ej^e.

The implementation of this method is the following: method searches for trucks with the length of load approaching its maxima and $\frac{em^m}{EM^{mm}} < 85\%$. Within each truck meeting these criteria, the method attempts to merge stacks if possible. The stacks are checked in ascending order along the x-axis, with the rearmost stacks being merged first. This ordering ensures that the middle axle does not start overloading again. The merging process is repeated until either the condition is no longer met or no feasible stack merges are found. An example of merging stacks is demonstrated in the bottom-half of Figure 4.3.

These truckload balancing methods are run during the local search between selected local search operators.

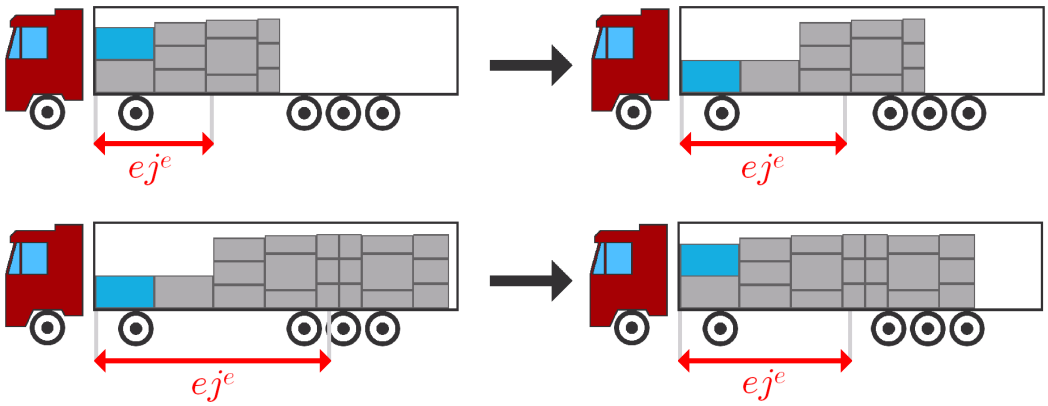

Figure 4.3. Example illustrating the truck balancing method in action. The upper half of the figure demonstrates the splitting of stacks to move the center of gravity (represented as the distance ej^e) towards the rear axle, effectively balancing the weight distribution. Conversely, the second half showcases the inverse method, where stacks are merged to create additional space for new potential stacks.

4.4 Local Search

Local search [32] is an approach used for addressing hard combinatorial optimization problems. Many optimization problems do not have an algorithm that guarantees finding an optimal solution in a reasonable time. That is why heuristic methods returning good suboptimal solutions are often used. One of them is the local search approach, which can flexibly trade solution quality against computation time.

The local search starts with an initial solution $s_{init} \in S$, S is a set of solutions (line 2 Alg. 4.4). A neighborhood function $N: S \rightarrow P(S)$, where $P(S)$ is the set of all subsets of solutions in S, is defined. Then, using calling all neighborhood functions N sequentially, we obtain a set S' of reachable solutions (line 6 Alg. 4.4). We choose somehow a solution $s \in S'$ (line 11 Alg. 4.4), for example, we can choose a solution with minimal objective value, and we call the neighborhood functions on this solution $S'' = N(s)$ again. We iteratively repeat this process until we arrive at a local minimum.

In our implementation, the neighborhood function N is defined by a set of Local Search operators. Specifically, we have designed six operators tailored to the problem

```
1  # Local Search
2  s ← s_init
3  while true do
4      S' ← ∅
5      for N ∈ N' do
6          S' ← S' ∪ N(s)
7      end for
8      if S' = ∅ then
9          break
10     end if
11     s ← argmin(obj(s')), s' ∈ S'
12 end while
```

Alg. 4.4. Local Search written in pseudo-code, N' set of operators

at hand, taking advantage of the both problem's and the initial solution's inherent properties. We propose 4 operators working on the stack level: Stack Swap, Stack Move, Truck Delete, and Truck Replace, these operators consider a stack as static, meaning that it can not be changed. Additionally, we describe two operators working on the item level: Item Swap and Item Move, which are able to modify stacks. The changes they make are relatively minor in comparison with operators on the stack level.

The operators are executed in a predefined order, with each operator exhaustively generating all possible improvements it can find. Once an operator has completed its turn, the next operator in the sequence proceeds. After the last operator in the sequence finishes, the first operator is proceeded again, creating a cycle. This cycle continues until no further improvements are discovered. If an operator fails to find any improvements during its turn, it is deactivated and skipped in the subsequent cycles. The deactivation of operators is primarily implemented to enhance performance speed, as it is possible that an operator may find an improvement in future cycles when the current solution has changed. However, based on our observations, the negative impact of deactivating operators is negligible or even non-existent in some instances.

4.4.1 Parameters Controlling Local Search Cycle

The Local Search cycle can be customized using various parameters. Each operator can be activated or deactivated for the entire Local Search run. The maximum number of cycles can be controlled by the parameter $maxCyclesLS$. By setting the parameter $twoCycles$ to $true$, two cycles are created. In the first cycle, operators working with stacks are executed, while in the second cycle, only item operators are executed. The parameter $truckLoadBalance$ allows the utilization of truckload balancing methods described in Section 4.3.

The default order of operators and auxiliary methods is as follows: Truck Load balancing, Stack Move, Truck Delete, Truck Replace, Stack Move, Stack Swap, Item Move, Item Swap. However, by setting the parameter $firstStackMove$ to $false$, the first Stack Move Operator can be skipped. We determined the order of operators empirically.

4.5 Stack Swap operator

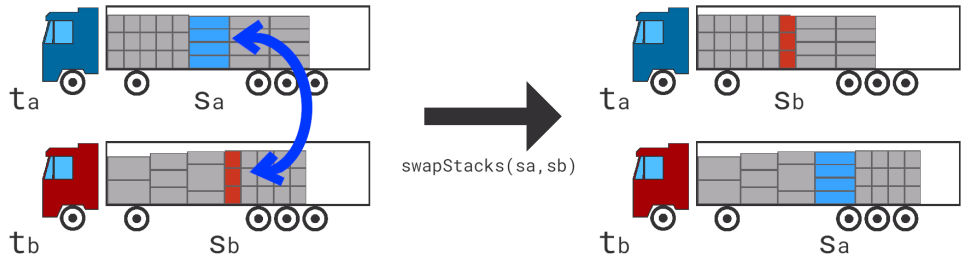

Figure 4.4. Stack Swap Operator

The Stack Swap operator illustrated in Figure 4.4 is designed to reduce the sum of inventory cost of two stacks using swapping their places. Stacks, denoted as s_a and s_b, are located in two trucks, in truck t_a and truck t_b, respectively, where $t_a \neq t_b$.

Application of this operator will result in a state where stack s_a is loaded into truck t_b in the same location and orientation as stack s_b originally was. Similarly, stack s_b will be moved to the original location and orientation of stack s_a in truck t_a. No other stacks in these two trucks are manipulated during the process.

4.5.1 Constraints to check

Before applying the Stack Swap operator to stacks s_a and s_b, the following constraints must be checked for both stacks in their respective new trucks, stack s_a in truck t_b and stack s_b in truck t_a.

- **I2** - New truck must arrive at the plant, where the new stack must be delivered.
- **I3** - New truck might not be able to pick up all items from the new stack.
- **I4** - New truck might not stop at the supplier where the new stack is distributed from.
- **I5** - Not all items in the stack might be delivered in the time window.
- **S4** - Stack orientation may be forced and new orientation may be in conflict with it.
- **S5** - Maximal weight above the bottom item is related to a truck. The new truck's limit for the bottom item might be lower than the original.
- **S7** - Maximal density of a stack is related to a truck. The maximal density might be lower in the new truck.
- **P1** - New stack might be bigger which may result in some stacks will not fit into the truck.
- **P4** - New stack might have a different supplier, supplier dock, or plant dock which may result in violating this constraint.
- **W1** - New stack can weigh more than the old one which may result in overloading the truck.
- **W2** - A different weight of the new stack may overload one of the axles.

4.5.2 Operator Application

Ordered stacks are iterated one by one (see line 2 Alg. 4.5). The currently iterated stack is denoted as s_a and is considered fixed. Then, it is searched for the best candidate stack s_b to swap with s_a. To do so, the operator checks all other stacks that come after s_a in the order (line 5 Alg. 4.5). The swap that results in the greatest improvement in the objective value and satisfies all constraints is then applied (line 16 Alg. 4.5). If no

```
# Stack Swap operator
  for s_a in S do
    bestImprove ← (−inf)
    bestStack ← (−1)
    for s_b in S \ {s, index(s) <= index(s_a)} do
      obj_oriq ← IC(s_a, truck(s_a)) + IC(s_b, truck(s_b))
      obj_new ← IC(s_a, truck(s_b)) + IC(s_b, truck(s_a))
      improve ← obj_oriq − obj_new
      if improve ≤ bestImprove ∨ ¬isSwapValid(s_a, s_b) then
        continue
      end if
      bestStack ← s_b
      bestImprove ← improve
    end for
    if bestStack ≠ −1 then
      swapStacks(s_a, bestStack)
    end if
  end for
```

Alg. 4.5. Pseudo-code of Stack Swap Operator Application. S = set of all stacks, $IC(s,t)$ = inventory cost of stack s in truck t, $truck(s)$ = truck where stack s is placed

feasible swap is found, the stack s_a is skipped. In both cases, the iteration continues to the next stack.

4.6 Stack Move operator

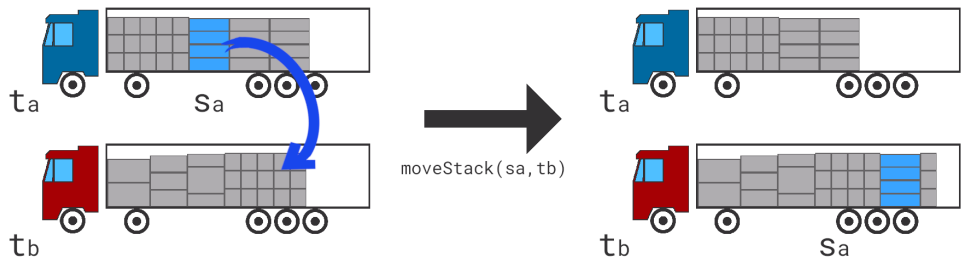

Figure 4.5. Stack Move Operator

The Stack Move operator illustrated in Figure 4.5 is designed to reduce the inventory cost of a stack. A stack, denoted as s_a, currently located in truck t_a (donor) is moved into truck t_b (receiver), where $t_a \neq t_b$. Moving a stack out of or into a truck may violate a constraint, therefore stacks in both trucks t_a and t_b need to be rearranged. The new positions and orientations of stacks that come after stack s_a in a solution vector, in either the original or new location, are recomputed.

If the stack s_a is the only stack in the truck t_a and stack s_a is moved into a different truck, the whole truck t_a is deleted.

4.6.1 Constraints to check (donor truck)

The following constraints must be checked in a truck t_a (donor) before applying the Stack Move operator.

- **P4** - Moving out a stack might result in violating the constraint for stacks' positions and loading orders.
- **W2** - Moving out a stack might result in overloading one of the axles.

4.6.2 Constraints to check (receiver truck)

The following constraints must be checked in a truck t_b (receiver) before applying the Stack Move operator.

- **I2** - New truck must arrive at the plant, where the new stack must be delivered.
- **I3** - New truck might not be able to pick up all items from the new stack.
- **I4** - New truck might not stop at the supplier where the new stack is distributed from.
- **I5** - Not all items in the stack might be delivered in the time window.
- **S4** - Stack orientation may be forced and new orientation may be in conflict with it.
- **S5** - Maximal weight above the bottom item is related to a truck. The new truck's limit for the bottom item might be lower than the original.
- **S7** - Maximal density of a stack is related to a truck. The maximal density might be lower in the new truck.
- **P1** - New stack might not fit into the truck.
- **P4** - New stack's supplier dock or plant dock might result in violating this constraint.
- **W1** - New stack can exceed the truck weight limit.
- **W2** - A weight of the new stack may overload one of the axles.

4.6.3 Operator Application

The Stack Move operator is designed to apply only the best-known improvements. It iterates through all possible stacks (line 6 Alg. 4.6) to find valid moves that result in improvement. The currently iterated stack is denoted as s_a and is considered fixed. The algorithm then searches for the best candidate truck t_b (line 11 Alg. 4.6) where stack s_a can be moved. In the implementation, all trucks are sorted for each stack based on the inventory cost of the stack in ascending order. This ensures that the first truck into which the stack can be loaded is the best candidate that can be used. If the inventory cost of the currently tested truck is greater than the current stack's inventory cost, it means no improving candidate can be found for this stack, and no other trucks are checked before proceeding to the next stack (line 14 Alg. 4.6). If a moving stack s_a to t_b is not valid, the algorithm proceeds to the next truck (line 17 Alg. 4.6).

After cycling through all the stacks and trucks, if a valid improving move is found (line 26 Alg. 4.6), it is applied to create a new solution. This process is repeated until no further valid improvement moves are found (line 28 Alg. 4.6).

```
1  # Stack Move operator
2  do
3      best_improve ← 0
4      best_stack ← Null
5      best_truck ← Null
6      for s_a in S do
7        obj_oriq ← IC(s_a, truck(s_a))
8        if obj_oriq == 0 then
9          continue
10       end if
11       for t_b in T \ {t_b = truck(s_a)} do
12         obj_new ← IC(s_a, truck(s_b))
13         improve ← obj_oriq − obj_new
14         if improve ≤ best_improve then
15           break
16         end if
17         if ¬isMoveValid(s_a, t_b) then
18           continue
19         end if
20         best_improve ← improve
21         best_truck ← t_b
22         best_stack ← s_s
23       end for
24     end for
25     if best_truck ≠ Null then
26       moveStack(best_stack, best_truck)
27     else
28       break
29     end else
30 while true
31
```

Alg. 4.6. Pseudo-code of Stack Move Operator Application. S = set of all stacks, T = set of used trucks, $IC(s,t)$ = inventory cost of stack s in truck t, $truck(s)$ = truck where stack s is placed

4.7 Truck Delete operator

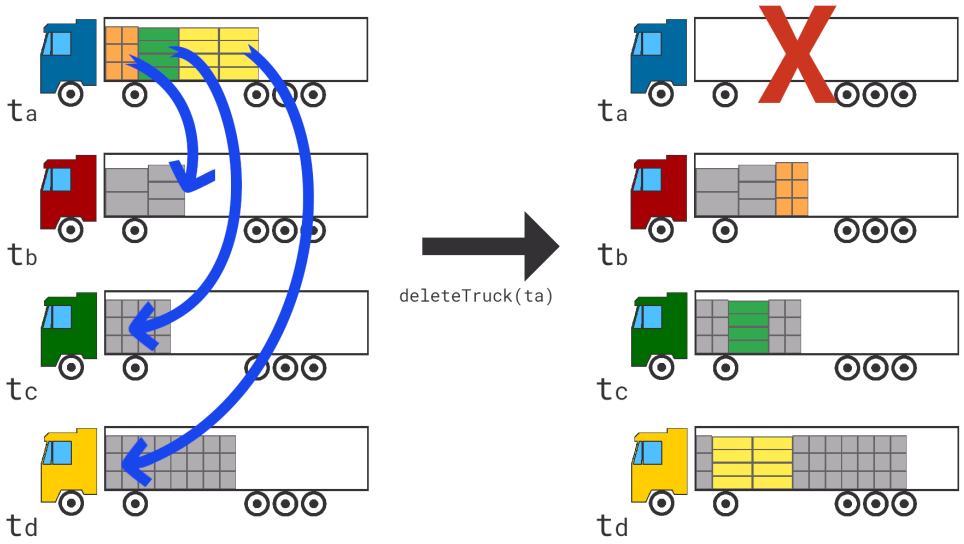

Figure 4.6. Truck Delete Operator

Truck Delete operator illustrated in Figure 4.6 is introduced to reduce the transport part of the objective value. The basic idea is to empty a given truck, denoted by t_a, by moving all of its stacks to other trucks. If all stacks are moved to other trucks, truck t_a can be deleted. It is important to note that this operator may result in an increase in the inventory costs of the moved stacks. For this reason, the sum of the increased inventory costs must still be lower than the cost saved by deleting truck t_a.

This operator plays a major role in the solution improvement, because the way how initial solution is found is very permissive in adding new trucks into the solution.

4.7.1 Constraints to check

If all stacks are moved from the donor truck t_a successfully, the truck is deleted and no constraints are needed to check.

The constraints needed to check whether a stack can be loaded into a receiving truck are the same as constraints for the Move Stack operator 4.6.2.

4.7.2 Operator Application

The Truck Delete operator is implemented similarly to the Stack Move operator, also only best-known improvements are applied.

It is iterated over all trucks (line 6 Alg. 4.7). For each truck t_a, the current solution is stored (line 7 Alg. 4.7). Then, it is iterated over all stacks s loaded in truck t_a (line 11 Alg. 4.7). It searches for a valid stack s moved to another truck, following a similar approach as the Stack Move operator. If a valid move is found, stack s is moved t_b, $invIncrease$ is updated, and the move is stored (lines 16-20 Alg. 4.7). The process then proceeds to the next stack. If no feasible move for the stack is found, the algorithm continues to the next truck (lines 24-26 Alg. 4.7). If it is possible to move all stacks to other trucks, resulting in an improvement $improve > best_{improve}$, the truck delete action is stored (lines 30-34 Alg. 4.7). After processing the truck, the original solution is restored (line 35 Alg. 4.7).

After iterating all trucks, it checks if $best_{truck}$ is not null, and if so, the truck deletion is applied (lines 37-39 Alg. 4.7). The process continues until no more truck deletion is found.

4.8 Truck Replace operator

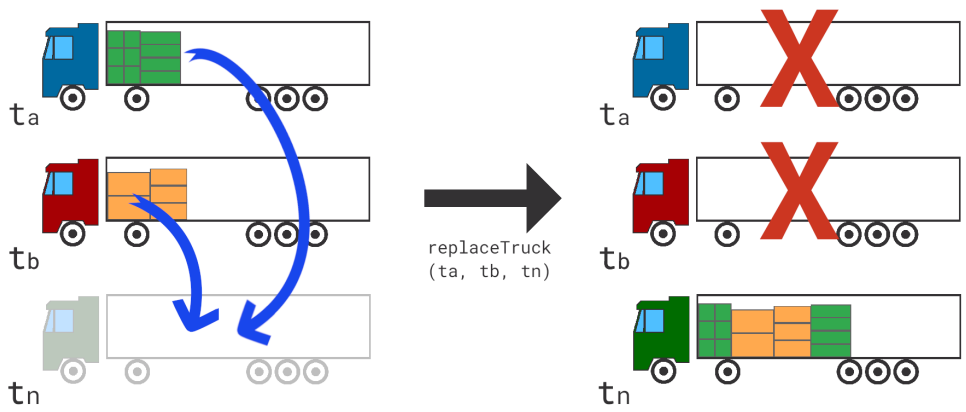

Figure 4.7. Truck Replace Operator

The Truck Replace operator illustrated in Figure 4.7 is designed to reduce the number of used trucks, which leads to lower transportation costs. Unlike the Truck Delete operator, which deletes only one truck, the Truck Replace operator deletes two trucks and moves all stacks from these trucks to a newly added truck. A newly added truck is usually a truck type that is not in the solution yet. Similarly to the Truck Delete operator, moving stacks into other trucks may increase their inventory cost. The operator is applied only if the sum of saved truck costs is greater than the increase in stack inventory costs.

This operator was introduced because of the way how the initial solution is found. The solution contains many unique trucks with few stacks, that can not be moved to any other trucks, because there are no other compatible trucks with these stacks in the current solution. Neither the Stack Move nor Truck Delete operators can move these stacks, making it necessary to introduce a new operator such as this, which is able to add new truck types.

This approach can be generalized to delete not only two trucks but to delete k trucks and add one new truck containing all stacks from deleted trucks. However, in small-scale experiments, the operator version for $k \leq 3$ showed only a negligible improvement compared to $k = 2$, while the $k \leq 5$ version produced the same results as the $k \leq 3$ version. Both of these versions consumed significantly more processor time than the $k = 2$ version. Therefore, for the sake of time efficiency, the operator is kept to delete only two trucks.

4.8.1 Constraints to check

If all stacks are moved from the donor trucks successfully, the trucks are deleted and no constraints are needed to check.

The constraints needed to check whether a stack can be loaded into a receiving truck are the same as constraints for the Move Stack operator 4.6.2.

```
1  # Truck Delete operator
2  do
3      best_improve ← 0
4      best_stackMoves ← Null
5      best_truck ← Null
6      for t_a in T do
7        sol_origin ← sol_cur
8        allStacksMoved ← true
9        stackMoves ← [ ]
10       invIncrease ← 0
11       for s in stacks(t_a) do
12         obj_orig ← IC(s, t_a)
13         stackMoved ← false
14         # T is ordered by inventory cost of s in t_b
15         for t_b in T \ {t_a} do
16           if isMoveValid(s_a, t_b) then
17             moveStack(s_a, t_b)
18             invIncrease ← invIncrease + (IC(s_a, t_b) − obj_orig)
19             stackMoves ← stackMoves ∪ (s_a, t_b)
20             stackMoved ← true
21             break
22           end if
23         end for
24         if ¬stackMoved ∨ invIncrease > TC(t_a) then
25           allStacksMoved ← false
26           break
27         end if
28       end for
29       improve ← TC(t_a) − invIncrease
30       if allStacksMoved ∧ improve > best_improve then
31         best_improve ← improve
32         best_truck ← t_a
33         best_stackMoves ← stackMoves
34       end else
35       sol_cur ← sol_origin
36     end for
37     if best_truck ≠ Null then
38       moveStacksToOtherTrucks(best_stackMoves)
39       truckDelete(best_truck)
40     else
41       break
42     end else
43 while true
44
```

Alg. 4.7. Pseudo-code of Truck Delete Operator Application. T - list of used trucks, $IC(s,t)$ = inventory cost of stack s in truck t, $stacks(t)$ = stacks in truck t

4.8.2 Operator Application

Pseudo-code Alg. 4.8 is designed for $k = 2$. To apply the operator, a list of potential trucks *shortTrucks* to remove based on their loading length is created. The method starts an iteration over the potential trucks (line 2 Alg. 4.8), with the currently iterated truck denoted as t_a. It iterates over all truck types, pT, from the input files. If the stacks from truck t_a can not be added into truck type t_n, it proceeds to the next t_n (lines: 4-6 Alg. 4.8).

For each t_n a candidate list of trucks $trucks_{cand}$, is created to which all stacks from t_a can be loaded (lines 7-13 Alg. 4.8). The candidate list is sorted in ascending order (line 14 Alg. 4.8).

The algorithm then iterates over $trucks_{cand}$ in order to find a suitable t_b that allows improving the objective by deleting trucks t_a and t_b while moving all stacks from these trucks to the newly created truck t_n. If a suitable pair of trucks, t_a and t_b, is found, they are deleted, and truck t_n is added to the solution loaded with all stacks from the deleted trucks (lines 17-21 Alg. 4.8).

The iteration then continues to the next potential truck t_a, regardless of whether the trucks were deleted or not.

```
1  # Truck Replace operator
2  for t_a in shortTrucks do
3    for t_n in pT do
4      if ¬canBeIn(stack(t_a), t_n) then
5        continue
6      end if
7      trucks_cand ← [ ]
8      obj_t_a ← IC(stacks(t_a), t_n)
9      for t_b in T \ {t_a} do
10       if canBeIn(stacks(t_b), t_n) then
11         trucks_cand ← trucks_cand ∪ ((IC(stacks(t_b), t_n)), t_b)
12       end if
13     end for
14     sort(trucks_cand)
15     for t_b, obj_t_b in trucks_cand do
16       if  TC(t_a) + TC(t_b) > TC(t_n) + obj_t_a + obj_t_b then
17         if canBeLoaded(stacks(t_a) ∪ stacks(t_b), t_n)) then
18           moveAllStacksToNewTruck((stacks(t_a) ∪ stacks(t_b), t_n)
19           deleteTruck(t_a)
20           deleteTruck(t_b)
21           shortTrucks ← shortTrucks \ {(obj_t_a, t_a) ∪ (obj_t_b, t_b)}
22           break for loop at line 3
23         end if
24       end if
25     end for
26   end for
27 end for
```

Alg. 4.8. Pseudo-code of Truck Replace Operator Application. pT - list of planned truck types, $IC(S, t)$ = inventory cost of stacks S in truck t, $stacks(t)$ = stacks in truck t

4.9 Item Swap operator

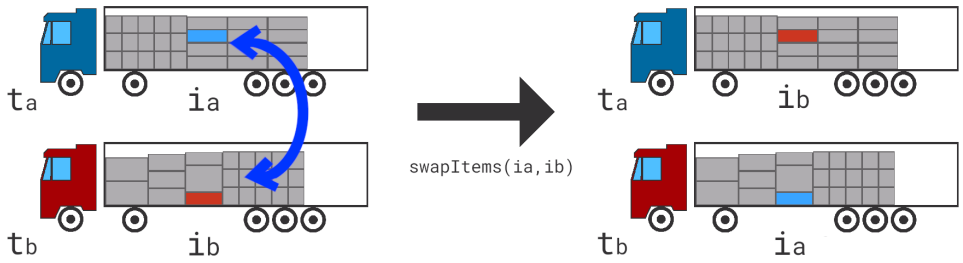

Figure 4.8. Item Swap Operator

The Item Swap operator (see Figure 4.8) works very similarly to Stack Swap operators (described in Section 4.5). It is designed for reducing the inventory part of the objective function. The idea is the same, the only difference is that this operator works on the item level and can modify stacks. An item i_a from stack s_a and truck t_a is swapped for another item i_b from stack s_b and truck t_b, where $t_a \neq t_b$.

4.9.1 Constraints to check

Before applying the Item Swap operator to items i_a and i_b, the following constraints must be checked for both items in their respective new stacks and trucks, item s_a in stack s_b and truck t_b and item i_b in stack s_a and truck t_a.

- **I2** - New truck must arrive at the plant, where the new item must be delivered.
- **I3** - New truck might not be able to pick up a new item.
- **I4** - New truck might not stop at the supplier where the new item is distributed from.
- **I5** - The item might not be delivered in the time window.
- **S1** - New item might not be compatible with a stack
- **S4** - Stack orientation may be forced and new item's orientation may be in conflict with it.
- **S5** - Maximal weight above the bottom item might be exceeded.
- **S7** - Maximal density of a stack item might be exceeded.
- **W1** - New item can weigh more than the old one which may result in overloading the truck.
- **W2** - A different weight of the new item may overload one of the axles.

4.9.2 Operator Application

The operator is applied in a 'first improve' fashion due to a large number of items. It is iterated over all pairs of items i_a and i_b which do not share the same truck (lines 2-6 Alg. 4.9). If the objective of swapped items is lower than the original one, items are swapped and it proceeds to the next item (lines 9-11 Alg. 4.9).

```
# Item Swap
for i_a in Î do
    for i_b in Î do
        if truck(i_a) == truck(i_b) then
            continue
        end if
        obj_orig ← IC(i_a, truck(i_a)) + IC(i_b, truck(i_b))
        obj_new ← IC(i_a, truck(i_b)) + IC(i_a, truck(i_b))
        if obj_new < obj_orig ∧ validSwap(i_a, i_b) then
            swapItems(i_a, i_b)
            break
        end if
    end for
end for
```

Alg. 4.9. Pseudo-code of Item Swap. $IC(i,t)$ = inventory cost of item i in truck t, $truck(i)$ = truck where item i is placed.

4.10 Item Move operator

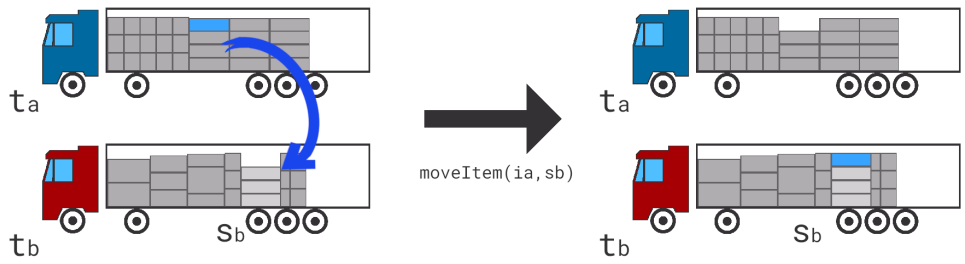

Figure 4.9. Item Move Operator

The Item Move operator (see Figure 4.9) is an item equivalent to Stack Move Operator (described in Section 4.6). It is designed to reduce the inventory part of the objective function. This operator is also able to modify stacks by adding and removing items. An item i_a from stack s_a and truck t_a is moved to stack s_b in truck t_b, where $t_a \neq t_b$.

4.10.1 Constraints to check (donor stack)

Both following constraints must be checked if an item is removed from a stack.

- **S5** - If an item is removed from the bottom of the stack, a new bottom item can have a lower maximal weight above the bottom item.
- **W2** - Removing an item from a stack can overload an axle.

4.10.2 Constraints to check (receiver stack)

The constraints needed to check whether an item can be added to a stack in a different truck are the same as constraints for the Item Swap operator 4.9.1.

4.10.3 Operator Application

The operator is just like the Item Swap operator applied in a 'first improve' fashion due to a large number of items. Pseudo-code Alg. 4.10 describes the iteration over all

```
1  # Item Move
2  for i_a in Î do
3      for t_b in T \ {truck(i_a)} do
4          for s_b in stacks(t_b) do
5              if IC(i_a, t_b) < IC(i_a, truck(i_a)) ∧ validMove(i_a, s_b) then
6                  moveItem(i_a, i_b)
7                  break for loop at line 3
8              end if
9          end for
10     end for
11 end for
```

Alg. 4.10. Pseudo-code of Item Move. $IC(i,t)$ = inventory cost of item i in truck t, $truck(i)$ = truck where item i is placed.

items i_a (line 2 Alg. 4.10), where it tests whether loading i_a into truck t_b results in a better solution. If a better solution is found and the move is valid, the item is moved from its original stack to a new stack s_b (line 6 Alg. 4.10).

4.11 Depth-First Tree Search Algorithm

By default, if we want to insert a stack into a truck, we do not change the order of stacks already loaded in a truck. The stack is inserted into the solution vector and the positions of the following stacks are recalculated. This approach is fast and works well if a truck is relatively empty.

However, in cases where a truck is nearly fully loaded, the previous approach may struggle to find a feasible position for the new stack. In such situations, rearranging the existing stacks by changing their order, orientations or positions may result in finding a feasible position for a new stack as it is shown in an example in Figure 4.10.

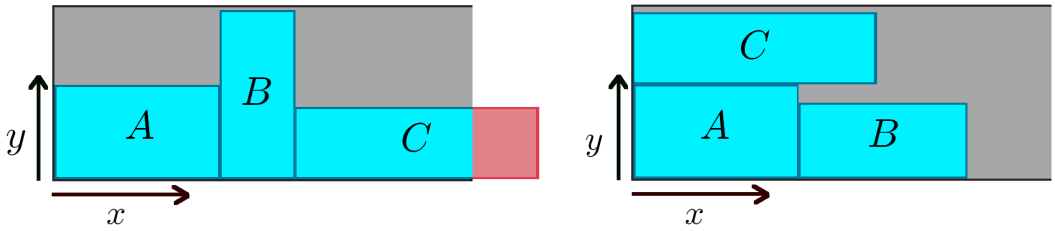

Figure 4.10. The figure illustrates two loading spaces containing the same three stacks but arranged differently. On the left loading space, the arrangement is unfeasible due to stack C exceeding the dimensional limits of the truck. Conversely, the right loading space displays an alternative arrangement which is feasible.

To address the problem of adding a new stack to an already nearly fully loaded truck, we introduced a custom Depth-First Tree Search (DFTS) algorithm. The algorithm explores different stack permutations, stack orientations, and stack positions in order to determine feasible arrangements for all original stacks as well as the newly added stack. The generated tree is illustrated in Figure 4.11. The depth-first search begins at a node representing an empty truck and expands the node to identify all possible stacks and their configurations that can be added. It then traverses to one of the child nodes representing a feasible stack arrangement and inserts the corresponding stack.

This process continues until a feasible arrangement of all stacks is found or until the limit of visited nodes is reached.

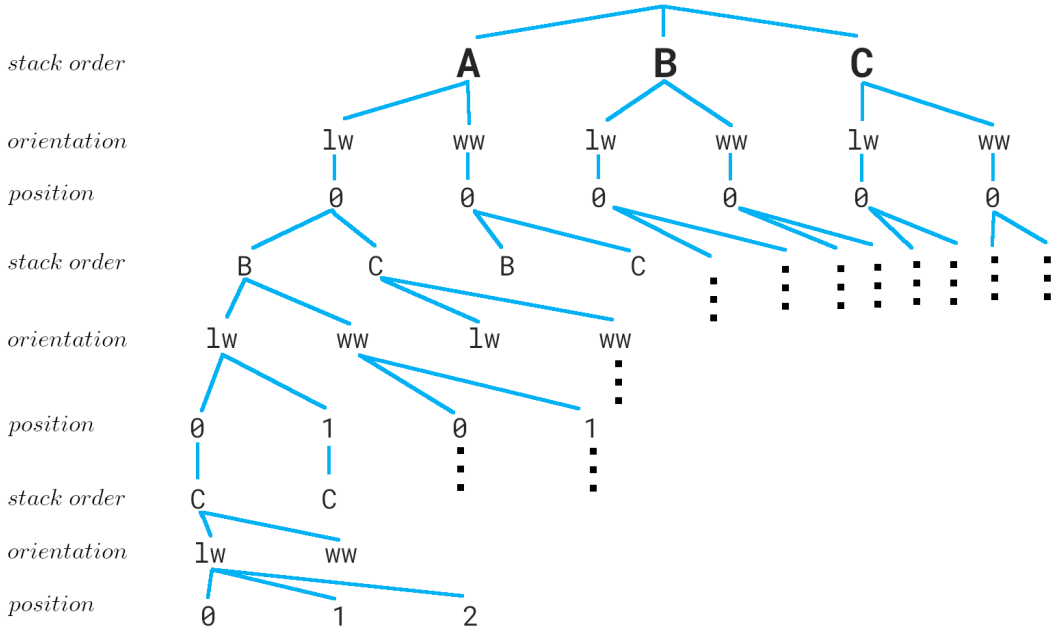

Figure 4.11. An expanded tree representing stack arrangements for 3 stacks A, B, C.

The number of arrangements is enormous. For instance, considering n stacks, the number of arrangements can be approximately up to $n! \times 2 \times nP$, where n is on average ≈ 14 and can be up to 24 (values are based on solutions for Dataset B instances) and nP represents the number of positions. To manage this vast search space, our method incorporates several pruning techniques.

Firstly, stacks are sorted into sectors based on supplier loading orders, supplier dock loading orders, and plant dock loading orders. These sectors are then loaded one by one in only one feasible order, significantly reducing the number of permutations to $\sum_{sec \in Sec} |sec|!)$, Sec denotes the list of sectors. Additionally, we maintain the available area within the truck during the search, terminating the process if the area required by the unloaded stacks exceeds the available space.

Furthermore, certain stacks have predefined orientations, further reducing the number of possible arrangements. Additionally, some positions may not be feasible for certain stacks, thereby narrowing down the potential search space.

4.11.1 Tree Search Parameters

The behavior of the DFTS algorithm is governed by three parameters: $maxVisNodes$, $searchPolicy$, and $repetitions$. The parameter $maxVisNodes$ sets the maximum number of visited nodes during the search. Once this limit is reached, the search process is terminated.

In our observations and experiment described in Section 5.4, we have noticed a pattern that if a feasible solution exists, it is typically found relatively quickly. Conversely, if the search runs for an extended duration without finding a solution, it is likely that no feasible arrangement exists. Therefore, it is often effective to set the $maxVisNodes$ parameter to a relatively low value, such as 800, to avoid prolonged search attempts for not-likely feasible arrangements. This approach is especially beneficial considering the requirement for the algorithm to execute swiftly, as it is run multiple times.

The *searchPolicy* parameter defines the order in which the algorithm explores the children nodes. It can take one of two values: *random* or *static*. When *searchPolicy* is set to *random*, the children nodes are shuffled to ensure a randomized exploration. In contrast, *static* follows the original stack order in the truck.

Additionally, the parameter *repetitions* specifies the number of independent runs for the search. By performing multiple runs, the algorithm can explore different regions of the search space increasing the chance in finding a feasible stack arrangement. It is important to note that this approach is effective only when the *searchPolicy* is set to *random*, as repeated searches with a *static* policy would yield identical results, covering the same search space repeatedly.

4.12 Perturbation

When we reach a local minimum using the Local Search, we would like to escape this minimum and try to explore a different part of the solution space and potentially find a better local minimum or even a global minimum. For small instances, local search converges relatively quickly and the majority of the given time budget for runtime would be unused. For these reasons, a simple perturbation [33] procedure was implemented.

The current best solution is perturbated in the following way. The solution is composed of trucks filled with stacks made of items. The p of randomly selected trucks (line 4 Alg. 4.12), where p is a parameter representing the percentage of trucks that will be removed, is deleted (line 6 Alg. 4.12), and all stacks loaded into deleted trucks are separated into single items which are stored in a list of items (line 5 Alg. 4.12).

This solution is not valid, because it does not contain all items. Therefore, all items from deleted trucks stored in $items_{delted}$ are loaded onto the trucks in the same way the initial solution is created. (lines 9-11 Alg. 4.12). That means that the inventory cost of these items will be minimal. During this process, it is possible to plan new trucks and new extra trucks.

After this procedure solution is valid and hopefully after using local search it converges to a different local minimum. If this solution is better than the original solution from which this procedure started, it becomes the new best solution and other perturbation procedure runs will start from this solution.

The process of how perturbation improves solution quality is visually represented in Figure 4.12. In the plot, each blue point corresponds to the objective value of a solution after it has undergone the perturbation procedure and reached a local minimum.

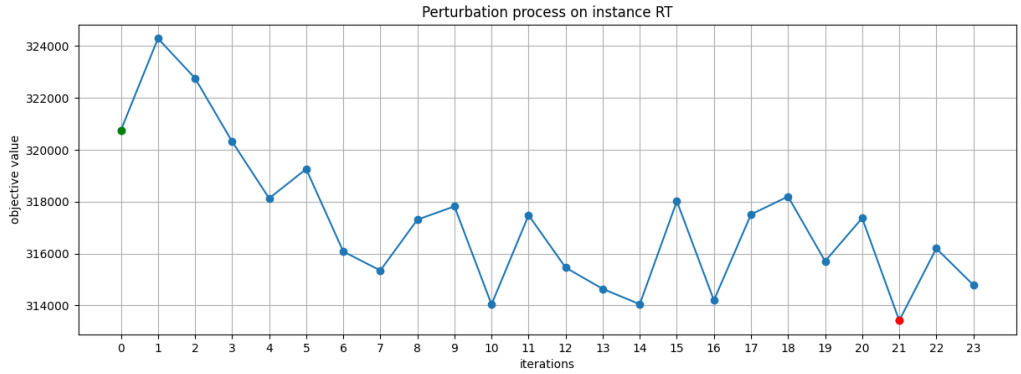

Figure 4.12. The perturbation progress on instance RT (from Dataset A), the green marker shows the solution after the first local search and the red marker shows the best solution found.

```
1  # Pertubation
2  items_deleted ← [ ]
3  for t in trucks(sol) then
4      if random(0, 100) < p then
5          items_deleted ← items_deleted ∪ items(t)
6          deleteTruckFromSolution(t, sol)
7      end
8  end for
9  for i in items_deleted do
10     loadItemToATruckGreedy(t, sol)
11 end for
```

Alg. 4.12. Pseudo-code of Perturbation procedure. sol = current solution, $trucks(s)$ = used trucks in solution s, $items(t)$ = set of all items placed in truck t.

4.13 Parameters

Several parameters are used in the implementation of the proposed method, particularly for controlling the DFTS and Local Search components. A comprehensive list of all the parameters, including their types, default values, and brief descriptions, can be found in Table 4.1.

Parameters are incorporated to provide flexibility in tuning our method, specifically for the available problem instances, aiming to achieve optimal results in the competition. By leveraging parameters, it becomes possible to disable specific operators and adjust the behavior of the DFTS to enhance performance and meet the time constraints, particularly for larger instances. This adaptability allows us to optimize the method's effectiveness based on given instances.

We successfully employed the irace tool [34] for parameter tuning on the Dataset B instances during the Qualification round. For the upcoming Final round, we intend to follow a similar approach, leveraging the publicly available Dataset C. However, it should be noted that Dataset X, which will also be used for evaluation in the Final round, is not public. As a result, our plan is to generate multiple parameter sets based on the characteristics of the instances, in particular instance size.

4.14 Implementation notes

The program implementing the designed method and solving the given problem was written in C++, because of the high-performance demands created by relatively small time limits and large data instances. OpenMP [35] was used for parallelization to increase computing power utilization.

Input and output files are in CSV format and the structures of these files are defined by organizers. The program reads from its arguments paths for input files and paths to output files.

The challenge is still in progress. In this thesis, we present the properties and results of the implementation available in Appendix A.

Parameter	Type	Default	Description
seed	integer	477 167	Random generator seed
γ	float	0.0005	Weight used in initial solution
$maxVisNodes$	integer	800	DFTS: limit for max visited nodes
$searchPolicy$	enum	random	DFTS: order of nodes static/random
$repetitions$	integer	2	DFTS: number of clear starts
$maxCyclesLS$	integer	-1	LS: max number of LS cycles, $-1 \rightarrow \infty$
$firstStackMove$	bool	true	LS: start LS with Stack Move operator
$twoCycles$	bool	false	LS: separate operators to two cycles
$truckLoadBalance$	bool	true	LS: allow truckload balancing
$stackMoveUsesTS$	bool	true	LS: allow DFTS in Stack Move
$truckDeleteUsesTS$	bool	true	LS: allow DFTS in Truck Delete
$stackSwap$	bool	true	LS: allow Stack Swap operator
$stackMove$	bool	true	LS: allow Stack Move operator
$truckDelete$	bool	true	LS: allow Truck Delete operator
$truckReplace$	bool	true	LS: allow Truck Replace operator
$itemSwap$	bool	true	LS: allow Item Swap operator
$itemMove$	bool	true	LS: allow Item Move operator
p	integer	15	Perturbation parameter, percentage of deleted trucks

Table 4.1. Table of parameters used in the implementation

Chapter 5
Results

In this chapter, we present a comparison of our method results on the available datasets (Dataset A and B). We also compare the current method with the older version which our team submitted for the qualification round. Then, we present the comparison of the DFTS method for finding feasible stack arrangements in a truck with an Integer Linear Programming model. Finally, we provide an overview of the contribution of each Local Search operator and its time consumption.

Please note that during writing this work Roadef challenge is still ongoing, the results presented are generated approximately 6 weeks before the final round deadline. We are still tuning our method implementation. The method implementation on which the following experiments were conducted can be found in Appendix A.

5.1 Instances

The organizers of the competition introduced 4 datasets with instances, Datasets A, B, C, and X. Datasets A, B, and C are public, they were released during the competition for the evaluation of different rounds. Dataset X is private, it is meant to be used together with Dataset C for the evaluation of the final round and will be released after it [5].

An instance is composed of 3 files: *input_items.csv*, *input_trucks.csv*, and *input_parameters.csv*. The files are in the comma-separated values (CSV) format.

The *input_items.csv* file contains all information about items. The *input_trucks.csv* file contains all information related to planned trucks. The *input_parameters.csv* includes 4 parameters, α^I, α^T, α^E, and the time limit. The used set of parameters among all datasets can be seen in Table 5.1 while Table 5.2 shows the properties of datasets.

Dataset	Set	α^I	α^T	α^E	Time Limit [s]
A	1	10.0	1.0	0.2	-
A	2	30.0	1.0	0.5	-
B	1	1.0	1.0	0.2	1800
B	2	5.0	1.0	0.2	1800
C	1	1.0	1.0	0.2	3600
C	2	5.0	1.0	0.2	3600
C	3	1.0	5.0	0.2	3600

Table 5.1. Parameter sets of available Datasets

Dataset A [36] was published at the beginning of the competition on 5th July 2022 [5]. It was used for the evaluation of the Sprint round and consists of 30 instances. These instances are described by organizers as small/medium size instances [31]. It should be noted that although a time limit was present in input files, it was not used during the Sprint round, and teams submitted only the best solutions they obtained.

Dataset B [37] was published along with the results of the Sprint round on 30th October 2022 [5]. This dataset was used for the evaluation of the Qualification round and consists of 40 instances. These instances are described by organizers also as small/medium size instances [31].

Finally, Dataset C [37] was released after the qualification results [5]. This dataset, together with Dataset X, will be used for the evaluation of the final round. It is composed of 50 instances. These instances are considered regular-size instances according to [31]. The time limit for these instances is one hour.

Dataset	Attribute	Min Value	Max Value
A	Single Item Count	5 017	85 435
	Item Classes	953	19 114
	Truck Count	220	5 821
	IC_i	0	25 630
	TC_t	1 500	1 500
B	Single Item Count	736	84 607
	Item Classes	203	21 063
	Truck Count	30	6 098
	IC_i	0	6 780
	TC_t	1 500	1 500
C	Single Item Count	2 881	208 318
	Item Classes	684	65 274
	Truck Count	66	6 775
	IC_i	0	4 941
	TC_t	1 500	1 500

Table 5.2. Instance properties of available Datasets

5.2 Results for Dataset A

Dataset A was used for the evaluation of the Sprint round. The organizers then presented the best-known objectives [38] for each instance. As the teams only submitted solution files, we do not have information about their hardware specifications or runtimes. Objectives in Table 5.3 obtained using the proposed method were generated with a time limit of 16 hours, which we set. We ran the proposed method on Metacentrum [39] cluster, which consists of an AMD EPYC Processor (with IBPB) 2.400 GHz (8 threads), 8 GB RAM, and Debian BULLSEYE. The method used default parameters described in Section 4.13.

It is worth noting that the Sprint round took place in October 2022, and we are comparing the Sprint results to the solutions generated by the latest version (May 2023) of the proposed method. Since the Sprint submission, we have improved our method significantly, and it is reasonable to assume that other teams have also improved their methods. Considering all the above, the best-known objectives listed in the Sprint results may now be outdated.

However, the proposed method obtained 13 new best objectives out of 30 instances. If these results were included in the scoreboard, our team would have found the most best-known objectives out of all teams, with team S30 finding 12 best-known objectives and team S41 finding 5. The average gap between Sprint's best-known objectives and

obtained objectives is -0.65%. The maximal gap is 2.90% (FL instance), while the minimal gap of -7.89% was achieved for instance CC. Overall, the proposed method performed very well on Dataset A.

Instance	Sprint[1]	Proposed method	Gap[2]
AS	395 040	396 300	0.32%
BU	520 110	**496 850**	**-4.47%**
BY	3 273 230	3 343 050	2.13%
BY2	3 644 520	3 708 900	1.77%
CA	790 800	**790 430**	**-0.05%**
CC	922 930	**850 150**	**-7.89%**
CI	1 677 310	1 706 030	1.71%
CI2	1 844 250	1 883 940	2.15%
CL	1 719 300	1 753 090	1.97%
DO	1 726 530	1 737 930	0.66%
DO2	1 957 890	1 958 040	0.01%
FC	237 720	**223 400**	**-6.02%**
FL	1 786 880	1 838 660	2.90%
FS	490 200	**460 990**	**-5.96%**
MA	2 552 530	2 592 920	1.58%
MA2	2 846 220	2 892 150	1.61%
MT	1 038 000	1 059 720	2.09%
PA	3 159 800	3 169 170	0.30%
PA2	3 516 780	**3 507 840**	**-0.25%**
RM	807 610	**803 270**	**-0.54%**
RM2	877 680	**859 710**	**-2.05%**
RT	320 650	**298 490**	**-6.91%**
SA	3 323 430	3 331 410	0.24%
SA2	3 739 860	3 769 080	0.78%
TA	2 758 140	**2 711 380**	**-1.70%**
TA2	3 025 500	3 060 720	1.16%
TR	1 963 680	**1 922 900**	**-2.08%**
TR2	2 097 750	2 124 330	1.27%
VA	3 037 030	**2 974 880**	**-2.05%**
VA2	3 369 000	**3 299 550**	**-2.06%**
Average			**-0.65%**

Table 5.3. Comparison of objective values of the best solutions from the sprint round and objective values of solutions found using the proposed method
[1]Best-known value from Sprint round [2]Gap computed as $\frac{obj-BestKnown}{BestKnown}$

5.3 Results for Dataset B

In the Qualification round, Dataset B was used to evaluate the submitted methods. We compared the best-known objective values among all competitors with those generated by our proposed method, as well as with results from an older version of our proposed method that we submitted for the Qualification round. Using the qualification scoreboard [6], which has already been published, we are able to compare our results with other teams' programs.

It should be noted that the competition organizers based their scoreboard and best-known objectives on results obtained by submitted programs running on their hardware, which consisted of a Google Cloud Platform virtual machine with 8 CPUs, 32GB of RAM, and CentOS 7 [31]. In contrast, the proposed method was tested on a laptop with an Intel(R) Core(TM) i7-1065G7 CPU - 1.30 GHz (8 threads), 16 GB RAM, and Windows 10. Both setups were run for 30 minutes for each instance. We performed parameter tuning specific to each instance, and these tuned parameters were used also in both setups. The organizers revealed only the best-known objective for each instance and the score of each qualified team. Therefore, these hardware specifications are important to consider when interpreting the results.

Table 5.4 presents the best-known objectives, as well as the objectives obtained using the proposed method in both a version that was used in the Qualification round and the current version, along with their percentage gaps. The average gap between the best-known objectives and the old version is 20.49%, and the biggest gap is 30.61% for instance CA. However, the method was able to find a solution with the same objective as the best-known objective for two instances (RT, RT2). Despite the average gap being relatively high, we were able to place 9th among all teams and 3rd among junior teams, which is significant considering 51 teams worldwide registered, 20 of which were junior teams.

Since the qualification round, the proposed method has been significantly improved, with the average gap between the obtained objectives and best-known objectives dropping to 8.70%. The maximum gap dropped to 15.59% (instance CI). The current proposed method still managed to obtain the best-known objectives for RT and RT2 instances. Furthermore, for instances AS and CC2, the proposed method generated solutions that were very close to the best-known values, with gaps less than or equal to 2.0%.

5.4 Comparison of Depth-First Tree Search and Integer Linear Program Model

In this section, we compare our custom algorithm, based on Depth-First Tree Search (DFTS) described in Section 4.11, to the Integer Linear Programming (ILP) model solved by Gurobi [40] for finding a feasible stack arrangement in trucks. The task is to load a set of stacks $S = S' \cup s$ into the truck t, satisfying all constraints. It is known that a set of stacks S' is feasible to load into a truck t, and s is a new stack to add.

One alternative way to solve this problem is to create an ILP model and solve it using a solver. The Gurobi Solver, which is listed in the rules of the competition [31] as an available program, can be used for this purpose. However, this approach may be too time-consuming for our purposes compared to our custom tree search. Therefore, we conducted an experiment to compare the performance of these two approaches.

5.4.1 Experiments

We created an ILP model without an objective function, incorporating constraints derived from the problem, such as placement constraints P1, P2, P3, P4 (Section 3.4.3), stack constraint S4 (Section 3.4.2), and weights constrain W2 (Section 3.4.4). The model was then implemented into the program, and the success rate and time needed to run these methods were measured for both methods (ILP model and DFTS) on several instances from Dataset B. All data were collected from one run of the first delete operator. For the DFTS method, we used the following parameters: $maxVisNodes =$

Instance	Qualification[1]	Old[2]	Gap[3]	Current[4]	Gap[3]
AS	912 146	1 044 568	14.52%	962 164	5.48%
AS2	988 795	1 056 880	6.89%	1 008 505	1.99%
BU	449 984	558 164	24.04%	491 235	9.17%
BU2	495 355	588 740	18.85%	522 725	5.53%
BY	2 779 694	3 284 573	18.16%	3 073 669	10.58%
BY2	3 037 000	3 544 190	16.70%	3 332 165	9.72%
CA	689 674	900 805	30.61%	758 408	9.97%
CA2	747 790	948 110	26.79%	780 270	4.34%
CC	636 928	747 003	17.28%	662 718	4.05%
CC2	698 115	798 125	14.33%	712 110	2.00%
CI	2 225 621	2 828 960	27.11%	2 572 675	15.59%
CI2	2 445 335	3 024 375	23.68%	2 791 760	14.17%
CL	1 192 162	1 457 361	22.25%	1 297 437	8.83%
CL2	1 276 130	1 521 145	19.20%	1 339 860	4.99%
DO	1 523 537	1 834 915	20.44%	1 712 133	12.38%
DO2	1 722 765	1 913 710	11.08%	1 842 225	6.93%
FC	281 660	356 525	26.58%	311 220	10.49%
FC2	324 140	386 670	19.29%	339 590	4.77%
FL	1 504 229	1 847 710	22.83%	1 685 566	12.06%
FL2	1 681 390	2 019 725	20.12%	1 789 095	6.41%
FS	505 146	623 284	23.39%	536 899	6.29%
FS2	562 170	662 980	17.93%	593 765	5.62%
MA	2 415 304	3 108 594	28.70%	2 760 723	14.30%
MA2	2 809 290	3 258 175	15.98%	3 008 840	7.10%
MT	1 176 550	1 519 494	29.15%	1 296 990	10.24%
MT2	1 303 555	1 626 250	24.75%	1 387 605	6.45%
PA	2 282 468	2 962 467	29.79%	2 501 929	9.62%
PA2	2 618 020	3 153 550	20.46%	2 820 260	7.72%
RM	842 282	1 077 148	27.88%	952 458	13.08%
RM2	949 855	1 127 895	18.74%	1 020 960	7.49%
RT	43 500	**43 500**	**0.00%**	**43 500**	**0.00%**
RT2	43 500	**43 500**	**0.00%**	**43 500**	**0.00%**
SA	2 536 199	3 269 450	28.91%	2 914 087	14.90%
SA2	2 774 020	3 552 155	28.05%	3 137 090	13.09%
TA	2 135 459	2 677 396	25.38%	2 411 171	12.91%
TA2	2 328 060	2 862 065	22.94%	2 575 195	10.62%
TR	1 429 757	1 727 387	20.82%	1 649 503	15.37%
TR2	1 551 015	1 850 675	19.32%	1 754 415	13.11%
VA	2 160 686	2 623 032	21.40%	2 408 352	11.46%
VA2	2 394 030	2 756 455	15.14%	2 618 365	9.37%
Average			**20.49%**		**8.70%**

Table 5.4. Comparison of best-known objectives, an older version of the proposed method used in the Qualification round, and current proposed method on Dataset B.
[1]Best-known objectives from Qualification round [2]Objectives obtained using proposed method submitted for the Qualification round [3]Gap computed as $\frac{obj-BestKnown}{BestKnown}$ [4]Objectives obtained using current proposed method

500, *searchPolicy* = *random*, and *repetitions* = 1. The ILP model was tested with time limits ranging from 0.1 s up to 10 s. The program for both methods was run on a computer equipped with Intel(R) Core(TM) i7-1065G7 CPU - 1.30 GHz, 16 GB RAM, and Windows 10.

5.4.2 Results

The results, shown in Table 5.5, indicate that DFTS found significantly more feasible solutions than ILP in every test. For instance, in the case of instance MA2, the ILP model with a time limit of 0.5 seconds found 444 feasible solutions out of 3 739 cases, with a total CPU time of 1 482 seconds, which is almost the whole of 1 800 seconds time budget for an instance. In contrast, DFTS found 895 solutions in only 145 seconds. For the RT instance, DFTS found a feasible solution for every one of the 98 runs in an average time of 1.1 milliseconds per solution, while ILP found only 61 even with a time limit of 10 seconds. These results clearly demonstrate that DFTS outperformed ILP significantly both in terms of speed and ability to find feasible solutions.

Moreover, considering that a time limit of 0.1 seconds is hardly feasible for the stated problem and the number of found feasible solutions by ILP model is very low, we have concluded that our custom DFTS algorithm is more appropriate for this problem, and we will not further investigate the Gurobi ILP model approach for this problem.

		ILP					DFTS			
Instance	Runs	TL[1]	F[2]	Ft[3]	NFt[4]	Tt[5]	F[2]	Ft[3]	NFt[4]	Tt[5]
AS	4 155	0.1	4	51.9	117.3	487.1	119	4.3	21.8	88.5
FC	1 671	0.1	4	60.3	64.4	107.7	140	11.1	33.4	46.0
FC	1 671	0.2	9	134.5	114.3	191.7	140	11.1	33.4	46.0
FC	1 671	0.5	10	190.6	252.7	421.8	140	11.1	33.4	46.0
MA2	3 739	0.1	322	45.1	102.0	390.3	895	9.7	48.0	145.4
MA2	3 739	0.5	444	113.1	434.8	1,482	895	9.7	48.0	145.4
RT	98	0.5	29	132.9	543.8	41.4	98	1.1	-	0.1
RT	98	1.0	36	234.0	1037.5	72.8	98	1.1	-	0.1
RT	98	5.0	52	1,089	5,051	289.0	98	1.1	-	0.1
RT	98	10.0	61	1,920	10,069	489.7	98	1.1	-	0.1

Table 5.5. Comparison of ILP solved by Gurobi and DFTS for finding stack feasible arrangements.
[1]Time limit of Gurobi run [s] [2]Number of runs ended with a feasible solution found [3]Average time of run with a feasible solution found [ms] [4]Average time of run with no solution found [ms] [5]Total Time [s] spent on all runs

5.5 Contributions of operators

In this section, we present an analysis of the contribution of each Local Search operator to a solution improvement. Our program was run for 1 hour for each instance from dataset B. We measure solution improvement and runtime per operator for all instances. We combine all improvements of each operator and analyze contribution, time consumption, and contribution per time ratio. The contribution is determined as a percentage of the total improvement in solution objectives across all instances and local search runs.

The results in Table 5.6 show the measured values. The Truck Delete operator was found to have the highest contribution to total solutions improvement at 64.73%, while consuming 25.37% of the total time. The Stack Move operator was the second contributing 28.34% to total improvements. It should be noted that the Stack Move operator behaved similarly to the Truck Delete operator in some cases when a truck contains only one stack, which explains its significant contribution. The Stack Move operator also consumed a significant amount of time at 49.09%, as it was run more times than Truck Delete or Truck Replace. The third most effective operator was Truck Replace, which contributed 4.968% and consumed 22.38% of the total time spent on solution improvements. The Item Move Operator had a contribution of 1.12%, followed by Stack Swap with 0.74%, and Item Swap with the least contribution of 0.1%.

The high contributions of the Truck Delete, Stack Move, and Truck Replace operators can be attributed to their ability to lower the transportation cost component of the objective function. This cost is initially high due to the way the initial solution is found. In contrast, the Item Move, Stack Swap, and Item Swap operators provide only minor improvements to a solution as they only focus on the inventory component of the objective.

The Truck Delete Operator is the most important operator with a significant contribution to the total improvement achieved. It is also the most time-effective operator, with an average improvement of 2.84 EUR per millisecond. Another highly effective operator is the Stack Swap operator, which has an average improvement of 2.29 EUR per millisecond. On the other hand, the least helpful operator is the Item Swap, which has a very low contribution. During the tuning phase of Dataset C, the Item Swap operator will likely be turned off for many instances.

Operator	Contribution [%]	Time[1] [%]	Obj[2] [EUR/ms]
Truck Delete	64.734	25.370	2.844
Stack Move	28.336	49.093	0.643
Truck Replace	4.968	22.372	0.247
Item Move	1.120	1.424	0.877
Stack Swap	0.742	0.361	2.291
Item Swap	0.100	1.381	0.080

Table 5.6. Contribution of each Local Search Operator, percentage of time consumed, and the ratio of solution improvement per millisecond over all instances in Dataset B.
[1]Time consumption [2]EUR per millisecond

Chapter 6
Conclusion

In this thesis, we have studied the introduced Truck loading problem and developed a method based on Iterative Local Search to address this problem. We proposed six Local Search operators specifically designed for this problem, as well as truck load balancing methods, the DFTS algorithm, and a perturbation procedure.

I, the author of this thesis, developed and implemented the entire codebase for incorporating the proposed methods, and I also conducted all the experiments. The team registered for the competition consists of myself and my supervisor, Ing. David Woller.

With an earlier version of our proposed method, our team successfully qualified for the Final round of the Roadef Challenge 2022, achieving 9th place out of 51 teams overall and 3rd place in the junior category. Since then, we have made significant improvements to our method, as evidenced by the reduction in the average gap on Dataset B instances from 20.49% to 8.70%. For two instances (RT, RT2) from Dataset B, we found 2 solutions with the best-known objective. Additionally, on Dataset A, we discovered 13 new best solutions out of 30 instances, with an average gap of -0.65%.

Despite the promising results achieved by our proposed method, we acknowledge certain limitations. Notably, there are instances with relatively high gaps between published best-known results and ours, such as the 15.37% gap for the TR instance and the 15.59% gap for the CI instance, both from Dataset B. Additionally, the loading space of trucks could be utilized more effectively as we currently employ a simplified approach for loading stacks. The upcoming Final round will serve as a true test of the quality of the solutions generated by our method.

In the experiments, I conducted a comparison between DFTS and the Gurobi ILP model. The results demonstrated that the proposed DFTS outperformed the ILP model in terms of both speed and the number of feasible solutions found for the given problem. Additionally, I provided an overview of the contribution of each Local Search operator and its time consumption.

Our future steps involve fine-tuning our method specifically for Dataset C and exploring the optimal setup of Local Search operators to maximize the trade-off between solution quality and computation time. Additionally, we aim to estimate parameter sets suitable for the unknown Dataset X.

Overall, we firmly believe that our team, armed with this method, has a strong potential to achieve interesting results in the competition, particularly within the junior category.

References

[1] Roadef.org. *Challenge ROADEF/euro 2022: Trucks loading problem.* `https://www.roadef.org/challenge/2022/en/`.

[2] *Société française de Recherche Opérationnelle et d'aide à la décision.* `https://www.roadef.org/`.

[3] *EURO - The Association of European Operational Research Societies.* `https://www.euro-online.org/`.

[4] *Renault Group, car manufacturer - Official website.* `https://www.renaultgroup.com/`.

[5] Roadef.org. *SCHEDULE OF THE ROADEF/EURO CHALLENGE 2022.* `https://www.roadef.org/challenge/2022/en/calendrier.php`.

[6] Roadef.org. *CHALLENGE ROADEF/EURO 2022 QUALIFICATION RESULTS.* `https://www.roadef.org/challenge/2022/en/qualifresult.php`.

[7] Dirk G. Cattrysse, and Luk N. Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research.* 1992, 60 (3), 260-272. DOI https://doi.org/10.1016/0377-2217(92)90077-M.

[8] Andreas Bortfeldt, and Gerhard Wäscher. Constraints in container loading – A state-of-the-art review. *European Journal of Operational Research.* 2013, 229 (1), 1-20. DOI https://doi.org/10.1016/j.ejor.2012.12.006.

[9] Silvano Martello, and Paolo Toth. Bin-packing problem. *Knapsack problems: Algorithms and computer implementations.* 1990, 221–245.

[10] Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. *Chapter 4 The traveling salesman problem.* Handbooks in Operations Research and Management Science. 1995. `https://www.sciencedirect.com/science/article/pii/S0927050705801215`.

[11] Joakim Westerlund, Lazaros Papageorgiou, and Tapio Westerlund. A MILP model for N-dimensional allocation. *Computers Chemical Engineering.* 2007, 31 1702-1714. DOI 10.1016/j.compchemeng.2007.02.006.

[12] Christian Serrano, Xavier Delorme, and Alexandre Dolgui. Scheduling of truck arrivals, truck departures and shop-floor operation in a cross-dock platform, based on trucks loading plans. *International Journal of Production Economics.* 2017, 194 102-112. DOI https://doi.org/10.1016/j.ijpe.2017.09.008. Special Issue: Innovations in Production Economics.

[13] Silvano Martello, David Pisinger, and Daniele Vigo. The three-dimensional bin packing problem. *Operations research.* 2000, 48 (2), 256–267.

[14] Mhand Hifi. Exact algorithms for unconstrained three-dimensional cutting problems: a comparative study. *Computers & Operations Research.* 2004, 31 (5), 657–674.

[15] Sándor P. Fekete, Jörg Schepers, and Jan C. Van der Veen. An exact algorithm for higher-dimensional orthogonal packing. *Operations Research*. 2007, 55 (3), 569–587.

[16] G. Terry Ross, and Richard M. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical programming*. 1975, 8 (1), 91–103.

[17] Roberto Baldacci, Nicos Christofides, and Aristide Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*. 2008, 115 351–385.

[18] Johannes Terno, Guntram Scheithauer, Uta Sommerweiß, and Jan Riehme. An efficient approach for the multi-pallet loading problem. *European Journal of Operational Research*. 2000, 123 (2), 372-381. DOI https://doi.org/10.1016/S0377-2217(99)00263-5.

[19] Andreas Bortfeldt, and Daniel Mack. A heuristic for the three-dimensional strip packing problem. *European Journal of Operational Research*. 2007, 183 (3), 1267-1279. DOI https://doi.org/10.1016/j.ejor.2005.07.031.

[20] Andreas Bortfeldt, and Hermann Gehring. A hybrid genetic algorithm for the container loading problem. *European Journal of Operational Research*. 2001, 131 (1), 143-161. DOI https://doi.org/10.1016/S0377-2217(00)00055-2.

[21] José Fernando Gonçalves, and Mauricio G.C. Resende. A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers Operations Research*. 2012, 39 (2), 179-190. DOI https://doi.org/10.1016/j.cor.2011.03.009.

[22] H Hadizadeh Ghaziania, Masoud Monjezi, Amin Mousavi, Hesam Dehghani, and Ezzeddin Bakhtavar. Design of loading and transportation fleet in open-pit mines using simulation approach and metaheuristic algorithms. *Journal of Mining and Environment*. 2021, 12 (4), 1177–1188.

[23] Guenther Fuellerer, Karl F. Doerner, Richard F. Hartl, and Manuel Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem. *Computers Operations Research*. 2009, 36 (3), 655-673. DOI https://doi.org/10.1016/j.cor.2007.10.021.

[24] W. Michiels, E. H. L. Aarts, and J. Korst. *Theory of Local Search*. In: Rafael Marti, Panos M. Pardalos, and Mauricio G. C. Resende, eds. *Handbook of Heuristics*. Cham: Springer International Publishing, 2018. 299–339. ISBN 978-3-319-07124-4.
`https://doi.org/10.1007/978-3-319-07124-4_6`.

[25] A. Moura, and J.F. Oliveira. A GRASP approach to the container-loading problem. *IEEE Intelligent Systems*. 2005, 20 (4), 50-57. DOI 10.1109/MIS.2005.57.

[26] Oluf Faroe, David Pisinger, and Martin Zachariasen. Guided local search for the three-dimensional bin-packing problem. *Informs journal on computing*. 2003, 15 (3), 267–283.

[27] Christos D. Tarantilis, Emmanouil E. Zachariadis, and Chris T. Kiranoudis. A Hybrid Metaheuristic Algorithm for the Integrated Vehicle Routing and Three-Dimensional Container-Loading Problem. *IEEE Transactions on Intelligent Transportation Systems*. 2009, 10 (2), 255-271. DOI 10.1109/TITS.2009.2020187.

[28] Juan A. Diaz, and Elena Fernández. A tabu search heuristic for the generalized assignment problem. *European Journal of Operational Research*. 2001, 132 (1), 22–38.

[29] Philip Kilby, Patrick Prosser, and Paul Shaw. *Guided Local Search for the Vehicle Routing Problem with Time Windows.* In: Stefan Voß, Silvano Martello, Ibrahim H. Osman, and Catherine Roucairol, eds. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization.* Boston, MA: Springer US, 1999. 473–486. ISBN 978-1-4615-5775-3. https://doi.org/10.1007/978-1-4615-5775-3_32.

[30] Christian Serrano Alain Nguyen, Mohamed-Amine Khatouf. *Challenge ROADEF/euro 2022.* https://github.com/renault-iaa/challenge-roadef-2022/blob/1cfa2ed09b45c6e1432b6e1bcc3cd9c252762b3a/challenge_ROADEF_2022.pdf.

[31] Christian Serrano Alain Nguyen, Mohamed-Amine Khatouf. *Challenge ROADEF/euro 2022.* https://github.com/renault-iaa/challenge-roadef-2022/blob/main/Rule_Challenge_2022.pdf.

[32] Abraham Duarte, Jesus Sanchez-Oro, Nenad Mladenovic, and Raca Todosijevic. *Theory of Local Search.* In: Rafael Marti, Panos M. Pardalos, and Mauricio G. C. Resende, eds. *Handbook of Heuristics.* Cham: Springer International Publishing, 2018. 299–338. ISBN 978-3-319-07124-4. https://doi.org/10.1007/978-3-319-07124-4_9.

[33] Madalina M Drugan, and Dirk Thierens. Stochastic Pareto local search: Pareto neighbourhood exploration and perturbation strategies. *Journal of Heuristics.* 2012, 18 (5), 727–766.

[34] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. The irace package: Iterated Racing for Automatic Algorithm Configuration. *Operations Research Perspectives.* 2016, 3 43–58. DOI 10.1016/j.orp.2016.09.002.

[35] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP.* Morgan kaufmann, 2001.

[36] Renault-Iaa. *Challenge-roadef-2022/dataseta.zip at main · Renault-IAA/Challenge -ROADEF-2022.* 2022. https://github.com/renault-iaa/challenge-roadef-2022/blob/main/datasetA.zip.

[37] Renault-Iaa. *Challenge-roadef-2022/dataset_b.zip at main · Renault-IAA/Challenge -ROADEF-2022.* 2023. https://github.com/renault-iaa/challenge-roadef-2022/blob/main/dataset_B.zip.

[38] Roadef.org. *CHALLENGE ROADEF/EURO 2022 SPRINT RESULTS.* https://www.roadef.org/challenge/2022/en/sprintresult.php.

[39] *MetaCentrum (MetaVO) - Virtual Organization.* 2023. https://metavo.metacentrum.cz/.

[40] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual.* 2023. https://www.gurobi.com.

Appendix A
Source code

`src/`	Folder containing source code of the implementation
`CMakeLists.txt`	CMake configuration file
`params.conf`	An example of a parameters file
`README.MD`	Instructions on how to build and run the program