Greedy Algorithms for the Freight Consolidation Problem

Zuguang Gao[1]
The University of Chicago Booth School of Business, Chicago, IL, USA

John R. Birge
The University of Chicago Booth School of Business, Chicago, IL, USA

Richard Li-Yang Chen
Flexport, Inc., San Francisco, CA, USA

Maurice Cheung
Flexport, Inc., San Francisco, CA, USA

Abstract

We define and study the (ocean) *freight consolidation problem* (FCP), which plays a crucial role in solving today's supply chain crisis. Roughly speaking, every day and every hour, a freight forwarder sees a set of shipments and a set of containers at the origin port. There is a shipment cost associated with assigning each shipment to each container. If a container is assigned any shipment, there is also a procurement cost for that container. The FCP aims to minimize the total cost of fulfilling all the shipments, subject to capacity constraints of the containers. In this paper, we show that no constant factor approximation exists for FCP, and propose a series of greedy based heuristics for solving the problem. We also test our heuristics with simulated data and show that our heuristics achieve small optimality gaps.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains

Keywords and phrases Freight consolidation, heuristics, greedy algorithm

Digital Object Identifier 10.4230/OASIcs.ATMOS.2022.4

1 Introduction

The spiking high container prices since the COVID-19 pandemic has caused significant issues in global supply chains. In this paper, we consider the (ocean) *freight consolidation problem* (FCP) - a combinatorial optimization problem that is being solved every day and every hour by some of the world leading freight forwarders. In a nutshell, the freight consolidation problem aims to optimize the assignments of shipments to containers at the origin ports, such as Yantian Port (Shenzhen) and Port of Shanghai. In the FCP, there are a set of shipments and a set of candidate containers that can be used. The origin/destinations of each shipment and each container, as well as the estimated departure/arrival dates of each container, are predetermined as the shipment/container becomes available at the port. There are two major costs: cost of assigning a shipment to a container (shipment cost), and cost of procuring a container (container cost). We further explain these costs in slightly more detail:

- The shipment cost takes into account everything related to sending the shipment boxes to their final destinations. Starting from the origin port, the remaining cycle of a shipment includes arriving at a destination port, being sorted and loaded to rail or truck, and delivering to their destinations. If a shipment is assigned to two containers that arrive at different ports, the remaining rail and/or trucking costs will be different. Further

[1] Corresponding author

more, many shipments also have time window requirements, and based on the arrival time of different containers, there may be different lateness costs. Therefore, we have a shipment cost associated with assigning each shipment to each container. If a container is not feasible for a shipment due to time window or destination ports, the corresponding shipment cost (of assigning that shipment to that container) is assigned ∞.

- The container cost is the cost of using a container. There is a set of containers available at the origin port, each with its own destination, departure time, and cost of procurement. If we decide to assign any shipment to a container, then we have to pay the procurement cost for that container.

Moreover, if we find there is no proper container to assign a shipment, there is always an option to "coload" that shipment, i.e., use a third-party shipper, e.g., Shipco, to fulfill that shipment. The cost associated with assigning the shipment to a third-party shipper is called the "coloading" cost. In our formulation, the "coloading" option can be viewed as a container with unlimited capacity, and the coloading costs are equivalently viewed as the shipment cost of assigning a shipment to this "coloading" container.

The freight forwarder aims to fulfill all shipments at hand while minimizing the total cost, which includes both shipment costs and container costs, subject to certain constraints. Specifically, each container has its own size in three-dimensions, as does each shipment. A container also has a maximum weight limit. In reality, we need to ensure that the total weight of all shipments assigned to a container does not exceed the weight limit of that container, and the center of mass (of a loading plan of these shipments) is not too far away from the center of the container. Moreover, these shipments should be able to fit into the container in three dimensions. Assuming a shipment is packed in a three-dimensional box, there are six possible rotations (orientations) of a box when being loaded to the container. Some boxes do not allow all six rotations, and some boxes are not stackable (which means they have to be put on the top). Given all these practical constraints, the problem of loading any given set of shipments to a container is a separate NP-hard problem, which is called the *container loading problem* in literature (see [5] for a comprehensive review). It would be too complicated to consider all container-loading constraints in our freight consolidation model. Therefore, we simplify the constraints by just having a weight capacity constraint and a volume capacity constraint for each container, ignoring the actual three-dimensional packing feasibility constraint. Despite that FCP does not reflect all practical constraints, we believe it is the simplest model to capture the most important features of the problem.

Up till now, a keen reader would recognize that our FCP can be viewed as a combination of the generalized assignment problem (GAP) and the bin packing problem (BPP), in a more complicated version. The shipment costs mimic the costs of assigning jobs in GAP, while in FCP we have two sets of capacity constraints (both weight and volume). The container cost is the cost of using each container (bin), while we have different costs for each container (bin). Therefore, FCP is already complicated in its nature and is expected to be difficult to solve. In this paper, we prove the non-approximability result of FCP, i.e., there is no constant factor approximation to FCP in polynomial time, unless $P = NP$. As a remedy, we propose a series of heuristics. With simulated data that aims to reflect the actual practice, we show that our heuristics return solutions with small optimality gaps.

The remaining of the paper is organized as follows. In Section 2, we provide a comprehensive literature review on the Bin Packing and related problems. In Section 3 we formally introduce the FCP and provide the non-approximability result. In Section 4, we provide main greedy heuristics for solving the FCP. Due to page limits, some of the discussions in Section 2 are delayed to Appendix A.

2 Literature Review of the Bin Packing and Related Problems

2.1 Classical Bin Packing Problem

We first review the classical (one-dimensional) bin packing problem (BPP). In the classical bin packing problem, we are given a set of items, each with a one-dimensional size, and an unlimited number of containers (bins) with the same sizes. The BPP asks to minimize the total number of bins used, subject to the constraints that the total size of items added to each bin does not exceed the size of the bin. BPP is strongly NP-hard [14], meaning that no full polynomial time approximation scheme (FPTAS) exists. Over the years, many heuristics have been developed to provide high-quality solutions for practical purposes. The traditional heuristics are all for the "online" version of BPP, meaning that the list of items are shown one by one, and a decision for each item is made final as soon as the item is shown. Classical heuristics include the following.

- First Fit (FF) [16]: Upon seeing an item, it is inserted to the first bin (according to the indices of the bins) that has room for it. A new bin is opened if the item does not fit into any existing bin.
- Next Fit (NF) [16]: Upon seeing an item, it is inserted to the last existing bin (according to the indices of the bins) that has room for it. A new bin is opened if the item does not fit into any existing bin.
- Best Fit (BF) [20]: Upon seeing an item, it is inserted to the fullest bin that has room for it. A new bin is opened if the item does not fit into any existing bin.
- Worst Fit (WF) [11]: Upon seeing an item, it is inserted to the emptiest bin (among those existing ones) that has room for it. A new bin is opened if the item does not fit into any existing bin.
- Almost Worst Fit (AWF) [11]: Upon seeing an item, it is inserted to the second emptiest bin that has room for it. A new bin is opened if the item does not fit into any existing bin.

For the "offline" problem, on the other hand, we are given access to the full list of items from the beginning (before making any decisions). The above heuristics may also be used, but combined with some sorting of the items. For example, FF-Decreasing uses the First Fit heuristic on the presorted list of items, where the items are listed in decreasing order of their sizes. Other heuristics such as BF-Decreasing, NF-Decreasing, FF-Increasing are defined similarly. We refer to [12] for a survey on the worst-case analysis of the above algorithms.

There are also algorithms that have both online and offline flavor for BPP. One example is the Better-Fit heuristic algorithm (BFH) [10]. In BFH, an existing item from a bin is removed and replaced with the current item if the current item better fills the bin. If the packing of the current item results in a smaller remaining space than the packing of the existing item, then the existing item is removed from the bin it is in. The replaced item is then packed again using BFH. Such procedure continues for all items until better-fit cannot pack a replaced item, in which case it is packed with BF heuristic.

In recent years, there are also developments of more complicated metaheuristic approaches for solving the BPP. Examples include the Whale Optimization Algorithm (WOA) [17] (may be improved with Lévy Flights [1]), (Adaptive) Cuckoo Search (may also incorporate with Lévy Flights) [21], Squirrel Search Algorithm [15], the Fitness-Dependent Optimizer (FDO) [3, 2], and so on. Since BPP is still not so close to our FCP, we do not extend our discussions on these metaheuristics. We refer to [18] for a comprehensive survey of the aforementioned algorithms.

2.2 Variations of BPP

One major restriction of the classical BPP is that the objective is simply minimizing the number of bins used, and these bins are assumed to be identical. In our FCP, however, containers may differ in their size/dimensions, and the costs of containers are different from each other. Luckily, a number of variations of the classical BPP have also been studied in the literature.

2.2.1 Bin Packing Problem with General Cost Structures (GCBP)

In GCBP, the cost of a bin is not one, but depends on the number of items actually inserted into this bin. Specifically, the cost of a bin is given by a function $f : \{0, 1, 2, \ldots, n\} \to \mathbb{R}^+$, where f is a monotonically non-decreasing concave function, and $f(0) = 0$. In words, if the bin has been inserted k items, the cost of that bin would be $f(k)$. GCBP was first proposed in [4], where the worst-case performance of some BPP heuristics was analyzed. Specifically, it was shown that many common heuristics for BPP, such as FF, BF, and NF as described in Section 2.1 do not have a finite asymptotic approximation ratio, while NF-Decreasing was shown to have an asymptotic approximation ratio of exactly 2. Moreover, the BF-Increasing, FF-Increasing and NF-Increasing achieve a better asymptotic approximation ratio of approximately 1.691. It was also shown in [4] that any heuristic that is independent of f has an asymptotic approximation ratio of at least $\frac{4}{3}$. Later, [13] developed an asymptotic fully polynomial time approximation scheme (AFPTAS) and proved the tight bound of 1.5 asymptotic approximation ratio.

2.2.2 Generalized Bin Packing Problem (GBPP)

GBPP was first introduced in [7]. In GBPP, a set of items I with volume and profit has to be loaded into proper bins. Items can be either compulsory or non-compulsory, i.e., the item set is partitioned into two subsets: items in I^C are mandatory to load into any bin, and items in I^{NC} are optional. Bins are also classified in bin types, where bins belonging to the same type have the same capacity and cost. Moreover, for each bin type, there is a maximum number of bins that can be used. The objective is to accommodate all compulsory items and possibly non-compulsory items into appropriate bins in order to minimize the overall cost, which is the total cost of all used bins deducted by the total profit earned from the items.

GBPP differs from FCP in two ways: first, only one set of capacity constraints are considered; second, in GBPP, each item has the same profit (or cost) if inserted into different bins, while in FCP, items would cost differently if inserted into different containers. Even though GBPP is a still a much simplified version of the FCP, it was shown in [8] and [6] that GBPP cannot be approximated by any constant, unless $P = NP$.

2.2.3 Generalized Bin Packing Problem with Bin-Dependent Item Profits (GBPPI)

GBPPI extends GBPP by allowing that when an item is inserted into different bins, the profit earned from that item may be different. In this sense, GBPPI is the closest model to FCP, with the only difference being the absence of an additional set of capacity constraints on containers. GBPPI was introduced in [9], and to the best of our knowledge, there has been no further studies on the same problem since then. Since this is closely relevant to our problem, we provide a more detailed discussion of this problem in Appendix A.

3 Problem Formulation and Non-Approximability Result

In this section, we first define what we call the Freight Consolidation Problem (FCP). Then, we present the non-approximability result of the FCP. An instance of the FCP is given by a set of shipments and a set of containers. Each shipment has a weight and a volume, and each container has its own weight limit (capacity) and volume limit. There is a cost associated with assigning each shipment to each container (shipment cost), and, if any container is used (been assigned any shipment), there will be a procurement cost of that container (container cost). The goal is to assign all shipments to some containers to minimize the overall cost (total of shipment costs and container costs), subject to the volume and weight capacity constraints of these containers. In the following, we formulate the FCP as an integer linear program (ILP).

Sets:

- $\mathcal{S} = \{1, 2, \ldots, |\mathcal{S}|\}$ - set of shipments (indexed by s)
- $\mathcal{C} = \{1, 2, \ldots, |\mathcal{C}|\}$ - set of containers (indexed by c)

Parameters:

- ξ_{sc} - cost of packing shipment piece s into container c, assigned ∞ if cannot ship s with c
- p_c - procurement cost of container c
- ϕ_s - weight of shipment s
- Φ_c - weight limit of container c
- v_s - volume of shipment s
- V_c - volume limit of container c

Binary decision variables:

- $\mu_{sc} = 1$ if s is assigned to container c
- $\mu_c = 1$ if container c is used

The optimization problem (FCP):

$$\min_{\mu_{sc}, \mu_c} \sum_{c \in \mathcal{C}} \sum_{s \in \mathcal{S}} \xi_{sc} \mu_{sc} + \sum_{c \in \mathcal{C}} p_c \mu_c \tag{1a}$$

$$\text{s.t.} \sum_{c \in \mathcal{C}} \mu_{sc} = 1, \quad \forall s \in \mathcal{S}, \tag{1b}$$

$$\sum_{s \in \mathcal{S}} \phi_s \mu_{sc} \leq \Phi_c, \quad \forall c \in \mathcal{C}, \tag{1c}$$

$$\sum_{s \in \mathcal{S}} v_s \mu_{sc} \leq V_c, \quad \forall c \in \mathcal{C}, \tag{1d}$$

$$\mu_c \geq \mu_{sc}, \quad \forall s \in \mathcal{S}, \ \forall c \in \mathcal{C}, \tag{1e}$$

$$\mu_{sc}, \mu_c \in \{0, 1\}, \quad \forall s \in \mathcal{S}, c \in \mathcal{C}.$$

The objective (1a) is to minimize the total cost, which includes both the cost of shipping and the cost of containers. (1b) implies that each shipment must be assigned to one of the containers. (1c) and (1d) ensure that the total weight (resp. volume) of shipments assigned to each container does not exceed the weight (resp. volume) limit of that container. Lastly, (1e) forces us to pay the cost of a container as long as at least one of the shipments is assigned to that container.

The *approximation ratio* of any algorithm that solves FCP is defined as follows.

▶ **Definition 1.** *Given the minimization problem* (1)*, an instance* π *of the problem, an algorithm* ALG*, the optimum* $\text{OPT}(\pi) \geq 0$*, and value* $\text{ALG}(\pi)$ *of the solution computed by the algorithm, the* approximation ratio *of the algorithm* ALG *is the infimum* $\alpha \geq 1$ *such that*

$$\text{ALG}(\pi) \leq \alpha \cdot \text{OPT}(\pi), \quad \forall \pi, \tag{2}$$

i.e., for all instances, the output of the algorithm incurs a total cost that is at most α *times the optimal value.*

We next have the following non-approximability result for FCP.

▶ **Proposition 2.** *For any constant* α*, there is no polynomial-time algorithm for the Freight Consolidation Problem (FCP)* (1) *with approximation ratio* α*, unless* $P = NP$*.*

Proof. We prove by reduction from the decision version of the Bin Packing Problem (BPP). Consider an instance $\hat{\pi}$ of the BPP, which consists of n items, each with a volume v_i for $i = 1, \ldots, n$, and unlimited number of bins, each with a capacity V, where $V \geq v_i$ for all $i = 1, \ldots, n$. The decision version of the BPP asks if it is feasible to assign all items to the bins such that at most k bins are used. This instance $\hat{\pi}$ of BPP can be transformed into an instance π of the FCP as follows. The instance π of the FCP would include n shipments, each with volume v_i for $i = 1, \ldots, n$. The weight of these shipments are all 0. There are also $k + n$ containers with volume capacity V and weight capacity one. The cost of procuring each of the containers $1, \ldots, k$ is one, and the cost of procuring each of the containers $k+1, \ldots, k+n$ are $k\alpha$. All shipment costs ξ are zero. We note that, if $\hat{\pi}$ for BPP has a solution, then the optimal value of the FCP is at most k; otherwise if $\hat{\pi}$ does not have a solution, then the optimal value of the FCP must be greater than $k\alpha$ since at least one container with cost $k\alpha$ must be used.

Suppose that to the contrary a polynomial time algorithm approximating the FCP with a constant $\alpha > 1$ exists, then through such an algorithm we would be able to determine if an instance of the BPP has a solution: the algorithm would return value $\leq k\alpha$ for the instances of the FCP corresponding to the instances of the BPP which have a solution, and the algorithm would return value $> k\alpha$ for those corresponding to the instances of BPP without a solution. Unless $P = NP$, this is impossible since the decision version of the BPP is NP-complete. ◀

Since there is no constant factor approximation for the FCP (assuming $P \neq NP$), we propose in the next section some intuitive greedy heuristics for the problem.

4 Proposed Heuristics

In this section, we propose a series of greedy-type heuristics that find solutions that are (hopefully) close to optimal.

4.1 Greedy Cost-Feasibility Algorithm (GR)

4.1.1 Overview

In this subsection, we propose a greedy heuristic for the FCP, which we call the Greedy Cost-Feasibility algorithm. In this algorithm, we first assign all shipments to the containers such that the shipping cost is the lowest, i.e., for each shipment s, we find one container c' such that $\xi_{sc'} = \min_c \xi_{sc}$, and assign shipment s to container c'. This assignment provides a lower bound on the total shipping costs. The assignment, however, may not be feasible as

some of the capacity constraints of the containers may be violated. In each of the following steps, the algorithm moves one shipment at a time, from one container to another, to make the assignment move towards feasibility, while keeping the increment of the shipping cost at a minimum.

4.1.2 Overflow Score

We define an "overflow score" on each container for any given assignment, and use this overflow score together with the shipping costs to determine which shipment to be moved to which container. For any assignment μ, the overflow score for container c is defined as

$$O_c(\mu) := \beta_1 \cdot \frac{\left[\sum_{\{s|\mu_{sc}=1\}} v_s - V_c\right]^+}{V_c} + \beta_2 \cdot \frac{\left[\sum_{\{s|\mu_{sc}=1\}} \phi_s - \Phi_c\right]^+}{\Phi_c}, \tag{3}$$

where β_1, β_2 are some adjustable parameters that satisfy $\beta_1, \beta_2, \beta_1 + \beta_2 \in [0, 1]$. The first term of the overflow score measures the percentage volume overflow of container c, and the second term measures the percentage weight overflow of container c. These two terms are summed together with weights β_1, β_2 to obtain the overflow score of container c.

The total overflow score of an assignment is then defined as

$$O(\mu) := \sum_c O_c(\mu). \tag{4}$$

4.1.3 Moving Towards Feasibility

After computing the overflow score of each container given the initial assignment, we find those containers with $O_c(\mu) > 0$, i.e., containers that are not feasible. For each shipment in these containers, we try to move the shipment out of its current container to another container, and compute the new overflow score O'. Let μ denote the current assignment, and $\mu^{sc'}$ denote the new assignment that moves shipment s from its current container to container c'. If we move the shipment s from its current container c to container c', we will have the following cost-feasibility ratio:

$$\mathcal{R}(s, c') := \frac{\xi_{sc'} - \xi_{sc}}{O(\mu) - O(\mu^{sc'})}. \tag{5}$$

The algorithm decides to move the shipment s from c to c' that minimizes the above ratio. In other words, in deciding which move to take, we choose the move that incurs least incremental shipping cost per unit reduction of the overflow score.

Since there are always coloading options for those shipments in the overflowed containers, at each round after the move, the overflow score is guaranteed to decrease. We repeat this process until the overflow score decreases to zero, at which time we have a feasible solution.

In the end, we also perform a post-adjustment procedure by looking at each used container (containers with $\mu_c = 1$)[2] and the shipments assigned to it. We will remove that container and coload all shipments assigned to it if it is more profitable to do so.

4.1.4 Algorithm Summary

The complete Greedy Cost-Feasibility (GR) algorithm is given as Algorithm 1.

[2] In the rest of this paper, we also say a container c is "opened" if $\mu_c = 1$, and "closed" if $\mu_c = 0$.

Algorithm 1 Greedy Cost-Feasibility (GR).

Input: shipment info, container info, β_1, β_2 ▷ β_1, β_2 are adjustable parameters
Output: Assignment of each shipment to a container
Greedy Procedure
Assign each shipment to its shipment cost-minimizing container, i.e., assign s to a c' such that $\xi_{sc'} \leq \xi_{sc}, \forall c$. Denote the current assignment by μ.
Compute the overflow score of the current assignment $O(\mu)$.
while $O(\mu) > 0$ **do**
    For each shipment-container pair (s, c), compute the cost-feasibility ratio $\mathcal{R}(s, c)$ if s is reassigned to c.
    Find the pair (s, c) with the minimum $\mathcal{R}(s, c)$. Reassign s to c.
    Compute the new overflow score.
end while
Post-Adjustment Procedure
for each container c with $\mu_c = 1$ **do**
    Find all shipments s that has been assigned to c.
    if $p_c + \sum_{s \text{ assigned to } c} \xi_{sc} > \sum_{s \text{ assigned to } c} \xi_{s1}$ **then**
        $\mu_c = 0$, coload all these shipments. ▷ Coload all shipments in c if more profitable
    end if
end for

4.2 Greedy + Local Search (GRL)

The next heuristic we introduce is Greedy with Local Search (GRL).

4.2.1 Overview

From the solution of GR, we perform local movements of shipments. Specifically, we search in two neighborhoods of a solution: the "shift" neighborhood, which consists of all solutions obtained by reassigning one shipment from the current solution, and the "swap" neighborhood, which consists of all solutions obtained by swapping the assignment of two shipments from the current solution. In searching each neighborhood, there are two standard ways of performing movements: first-admissible (FA) and best-admissible (BA).

- In the first-admissible scheme, we randomly search the neighborhood and take the move as soon as we find a better solution.
- In the best-admissible scheme, we search all possible moves and thus all solutions in the neighborhood, and choose to take the move that leads to the most reduction in the shipment cost.

It has been shown in [19] that for the generalized assignment problem (GAP), BA returns a slightly better solution, but takes much longer time to generate the solution. We therefore choose FA in our implementations for two reasons: first, the (potentially) slightly better solution from BA may not be worth the extra time; second, our problem size is much larger than those that have been experimented upon in the GAP literature.

4.2.2 Searching the "Shift" Neighborhood

The search of the "shift" neighborhood is performed in cycles. In each cycle, we first randomly sort the list of all shipments. Then, starting from the first shipment s in the list, we sort the set of opened containers (those with $\mu_c = 1$ in the GR solution) in increasing order

of μ_{sc}, and try to reassign this shipment to each container in the container list. If the reassignment is feasible, the shipment is reassigned permanently, and a new cycle is started. Otherwise, we move to the next container in the sorted container list. If no container before the current assigned container is feasible, i.e., no reassignment of the current shipment can lead to reduction in cost while keeping feasibility, we skip this shipment and move to the next shipment. This process is repeated until we reach a cycle where no feasible improvement relocation can be made, at which time the solution is locally optimal in its "shift" neighborhood.

4.2.3 Searching the "Swap" Neighborhood

The search of the "swap" neighborhood is also performed in cycles. We first generate a list of all pairs of shipments. In each cycle, we sort this list randomly. Then, starting from the first shipment pair in the list, we try to swap the assignment of the two shipments. If the assignment after the swap is feasible for both containers, and the swap leads to a reduction in the total shipment cost, the swap is made permanent and a new cycle will start. Otherwise, we move to the next pair of shipments. This process is repeated until we reach a cycle where no swaps are made after visiting all shipment pairs, at which time the solution is locally optimal in its "swap" neighborhood.

4.2.4 Local Optimal Solution in Both Neighborhoods

Given any input solution, we first repeatedly search the "shift" neighborhood. We always keep the best solution found so far, and the search is repeated until no better solution is found after $Max\_Nonimprove\_S$ consecutive number of searches. Next, we search the "swap" neighborhood of the best solution found so far (locally optimal within the "shift" neighborhood), after which we reach a locally optimal solution within the "swap" neighborhood. If the new solution is better than the solution before searching the "swap" neighborhood, we will again repeatedly search the "shift" neighborhood and then the "swap" neighborhood. The whole process is repeated until no better solution is found after $Max\_Nonimprove$ consecutive number repetitions, at which point the solution is locally optimal within both neighborhoods.

4.2.5 Algorithm Summary

The complete Greedy + Local Search (GRL) algorithm is given as Algorithm 2.

4.3 Greedy + Local Search + Varying Containers (GRLV)

We now introduce the heuristic that is based on GRL, but tries to vary the set of used (opened) containers.

4.3.1 Overview

This heuristic consists of two layers. In the first layer, we generate a set of "seed" solutions. In the second layer, we try to vary the set of used containers on each "seed" solution, and finally return the best solution found throughout the process.

There are several intuitions behind this heuristic. First, the local search can be combined with the post-adjustment: Every time after running local search and finding a locally optimal solution, we can check again if deleting some containers and coloading all shipments in

Algorithm 2 Greedy + Local Search (GRL).

Input: shipment info, container info, $\beta_1, \beta_2, Max\_Nonimprove\_S, Max\_Nonimprove$
Output: Assignment of each shipment to a container

Run Greedy Procedure (as in Algorithm 1).
Run Post-Adjustment Procedure (as in Algorithm 1), save as "initial solution".
Local-Search Procedure
"best solution" = "initial solution"
$Outer\_counter = 0$
while $Outer\_counter < Max\_Nonimprove$ **do**
    $Inner\_counter = 0$
    "best shift solution" = "initial solution"
    while $Inner\_counter < Max\_Nonimprove\_S$ **do**
        Search the "shift" neighborhood of the "initial solution", save as "shift solution"
        if "shift solution" has lower total cost than "best shift solution" **then**
            "best shift solution" = "shift solution"
            $Inner\_counter = 0$
        else
            $Inner\_counter = Inner\_counter + 1$
        end if
    end while
    Search the "swap" neighborhood of the "best shift solution", save as "swap solution"
    while "swap solution" has lower cost than "best shift solution" **do**
        $Inner\_counter = 0$
        "best shift solution" = "swap solution"
        while $Inner\_counter < Max\_Nonimprove\_S$ **do**
            Search the "shift" neighborhood of the "swap solution", save as "shift solution"
            if "shift solution" has lower total cost than "best shift solution" **then**
                "best shift solution" = "shift solution"
                $Inner\_counter = 0$
            else
                $Inner\_counter = Inner\_counter + 1$
            end if
        end while
        Search the "swap" neighborhood of the "best shift solution", save as "swap solution"
    end while
    if "swap solution" has lower cost than the "best solution" **then**
        "best solution" = "swap solution"
        $Outer\_counter = 0$
    else
        $Outer\_counter = Outer\_counter + 1$
    end if
end while
Return "best solution"

those containers can be more profitable. If such containers exist, we proceed to delete these containers. Then we can redo the local search and the post-adjustment, and repeat this process till the post-adjustment does not delete any more containers. Second, every time we

perform some procedure that might change the set of used (opened) containers, we might do further local search based on the current solution, or we can also build a new solution from scratch, again using the Greedy Procedure, but this time fixing the set of unopened containers , i.e., set $\xi_{sc} = \infty$ for all containers that are not open before applying the Greedy Procedure. Third, every time we try to vary the set of containers, we can either add/delete one container at a time, or we can add/delete a number of containers altogether. In the following, we describe the procedures/subroutines that are used in this heuristic.

4.3.2 Adjusted Local Search

We may combine the Post-Adjustment Procedure with the Local-Search Procedure, then iterate both procedures repeatedly until the set of opened containers no longer changes so that we obtain a local optimum within both neighborhoods. We define the Adjust-Local Procedure as Algorithm 3.

Algorithm 3 Adjust-Local Procedure.

Input: initial solution
Output: updated solution
1: "updated solution" = "initial solution"
2: $Num\_del\_master = 1$
3: **while** $Num\_del\_master > 0$ **do**
4: Run Local-Search Procedure on "updated solution", save as "updated solution"
5: Run Post-Adjustment Procedure on "updated solution", save as "upsated solution"
6: Save the number of deleted containers in the Post-Adjustment Procedure as Num_del_master
7: **end while**
8: Return "updated solution"

4.3.3 Adding One of the Deleted Containers Back

Since the Post-Adjustment Procedure deletes some containers, we try to add one of those deleted containers back to the solution and then perform Adjust-Local Procedure. In the end, we save the best solution found during this process. The Add-One Procedure is defined as Algorithm 4.

4.3.4 Deleting a Chain of Containers

We observe that the GR solution, even after the Post-Adjustment Procedure, uses more containers than the optimal solution returned by the solver. Based on an initial solution, we try to delete a chain of containers. Specifically, we sort the containers in increasing order of their profit, i.e., for each container c, we compute:

$$\text{Profit of using container } c := \sum_{s:\mu_{sc}=1} \xi_{s1} - \left(p_c + \sum_{s:\mu_{sc}=1} \xi_{sc} \right), \tag{6}$$

which is the total coloading cost of the shipments assigned to container c deducted by the total shipping cost of those shipments and the procurement cost of the container. This is the actual "saving" from using container c for these shipments, compared with the cost of coloading all these shipments.

Algorithm 4 ADD-ONE PROCEDURE.

Input: initial solution
Output: updated solution
"updated solution" = "initial solution"
Run ADJUST-LOCAL PROCEDURE on "initial solution", save as "cand solution", save the set of deleted containers as S
Run ADJUST-LOCAL PROCEDURE on "initial solution", save as "updated solution"
for each container in set S **do**
    Reopen the container in the "cand solution", and add the shipments what were assigned to this container in the "initial solution" to this container, save as "current solution"
    if "current solution" has lower total cost than "updated solution" **then**
        "updated solution" = "current solution"
    end if
    Close this container in the "cand solution"
end for
Return "updated solution"

We delete the top k containers in the list from the initial solution and perform the ADJUST-LOCAL PROCEDURE, where k ranges from 0 to *num_cont_del* (a preset parameter). In the end, we output the best solution among these $(k+1)$ solutions. The DEL-CHAIN PROCEDURE is defined as Algorithm 5.

Algorithm 5 DEL-CHAIN PROCEDURE.

Input: initial solution, *num_cont_del*
Output: updated solution
"updated solution" = "initial solution"
"current solution" = "initial solution"
Sort the containers used in the "initial solution" in increasing order of their total profit (6). Save as "sorted list"
for $j \in \{0, 1, 2, \ldots, num\_cont\_del\}$ **do**
    Delete the jth container from the "current solution", coload all shipments previously assigned to that container, save as "current solution"
    Run ADJUST-LOCAL PROCEDURE on "current solution", save as "new solution"
    if "new solution" has lower total cost than "updated solution" **then**
        "updated solution" = "new solution"
    end if
end for
Return "updated solution"

4.3.5 Deleting One More Container

Given an initial solution, we may again sort the containers in increasing order of their profits (6), and try to delete one container from the top *num_cont_del* containers in the sorted list. The best solution is saved in the end. We define the DEL-ONE PROCEDURE as Algorihtm 6.

Algorithm 6 Del-One Procedure.

Input: initial solution, *num_cont_del*
Output: updated solution
"updated solution" = "initial solution"
Sort the containers used in the "initial solution" in increasing order of their total profit (6). Save as "sorted list"
for $j \in \{0, 1, 2, \ldots, num\_cont\_del\}$ **do**
    Delete the jth container from the "initial solution", coload all shipments previously assigned to that container, save as "current solution"
    Run Adjust-Local Procedure on "current solution", save as "current solution"
    if "current solution" has lower total cost than "updated solution" **then**
        "updated solution" = "current solution"
    end if
end for
Return "updated solution"

4.3.6 Deleting Containers One by One

Starting from some initial solution, we can repeatedly perform Del-One Procedure, until further deleting any containers leads to no improvement in the solution. The Del-ObO Procedure is defined as Algorithm 7.

Algorithm 7 Del-ObO Procedure.

Input: initial solution, *num_cont_del*
Output: updated solution
"updated solution" = "initial solution"
Run Del-One Procedure on "initial solution", save as "current solution"
if "current solution" has lower total cost than the "updated solution" **then**
    "updated solution" = "current solution"
    while "current solution" has lower total cost than the "updated solution" **do**
        "updated solution" = "current solution"
        Run Del-One Procedure on "current solution", save as "current solution"
    end while
end if
Return "updated solution"

4.3.7 Algorithm Summary

The complete Greedy + Local Search + Varying Containers (GRLV) algorithm is given as Algorithm 8.

Algorithm 8 Greedy + Local Search + Varying Containers (GRLV).

Input: shipment info, container info, β_1, β_2, $Max\_Nonimprove\_S$, $Max\_Nonimprove$, num_cont_del

Output: Assignment of each shipment to a container

1: Run Greedy Procedure (as in Algorithm 1), save as "GR solution"
2: Run Post-Adjustment Procedure (as in Algorithm 1) on "GR solution", save as "PA solution"
3: Run Adjust-Local Procedure on "GR solution", save as "LC solution"
4: Run Greedy Procedure on "PA" solution, i.e., first set $\xi_{sc} = \infty$ for all containers that are not open (used) in the "PA solution", then run Greedy Procedure. Save the solution as "PA_GR solution"
5: Run Greedy Procedure on "LC" solution, i.e., first set $\xi_{sc} = \infty$ for all containers that are not open (used) in the "LC solution", then run Greedy Procedure. Save the solution as "LC_GR solution"
6: Run Add-One Procedure on "PA solution", save as "PA_one solution"
7: Run Add-One Procedure on "LC solution", save as "LC_one solution"
8: Run Add-One Procedure on "PA_GR solution", save as "PA_GR_one solution"
9: Run Add-One Procedure on "LC_GR solution", save as "LC_GR_one solution"
10: Run Del-Chain Procedure on "PA solution", save as "CHAIN_PA solution"
11: Run Del-Chain Procedure on "LC solution", save as "CHAIN_LC solution"
12: Run Del-Chain Procedure on "PA_GR solution", save as "CHAIN_PA_GR solution"
13: Run Del-Chain Procedure on "LC_GR solution", save as "CHAIN_LC_GR solution"
14: Run Del-Chain Procedure on "PA_one solution", save as "CHAIN_PA_one solution"
15: Run Del-Chain Procedure on "LC_one solution", save as "CHAIN_LC_one solution"
16: Run Del-Chain Procedure on "PA_GR_one solution", save as "CHAIN_PA_GR_one solution"
17: Run Del-Chain Procedure on "LC_GR_one solution", save as "CHAIN_LC_GR_one solution"
18: Run Del-ObO Procedure on "PA solution", save as "OBO_PA solution"
19: Run Del-ObO Procedure on "LC solution", save as "OBO_LC solution"
20: Run Del-ObO Procedure on "PA_GR solution", save as "OBO_PA_GR solution"
21: Run Del-ObO Procedure on "LC_GR solution", save as "OBO_LC_GR solution"
22: Run Del-ObO Procedure on "PA_one solution", save as "OBO_PA_one solution"
23: Run Del-ObO Procedure on "LC_one solution", save as "OBO_LC_one solution"
24: Run Del-ObO Procedure on "PA_GR_one solution", save as "OBO_PA_GR_one solution"
25: Run Del-ObO Procedure on "LC_GR_one solution", save as "OBO_LC_GR_one solution"
26: Return the best solution among {"CHAIN_PA solution", "CHAIN_LC solution", "CHAIN_PA_GR solution", "CHAIN_LC_GR solution", "CHAIN_PA_one solution", "CHAIN_LC_one solution", "CHAIN_PA_GR_one solution", "CHAIN_LC_GR_one solution", "OBO_PA solution", "OBO_LC solution", "OBO_PA_GR solution", "OBO_LC_GR solution", "OBO_PA_one solution", "OBO_LC_one solution", "OBO_PA_GR_one solution", "OBO_LC_GR_one solution"}.

5 Experiments

In this section, we provide experimental results on our proposed heuristics, including GR, GRL, and GRLV. We first generate a set of instances that hopefully reflects part of the reality. Each of these instances are generated as the following:

- **Containers:** We have 150 containers in an instance (not including the "coloading" container), each with a weight capacity $\Phi_c = 28,000$ (kg) and a volume capacity $V_c = 76$ (m^3), which reflects the capacities of the most used containers (40' high-cube container). The container cost p_c is sampled from a truncated Normal distribution (lower bounded at 0) with the mean 9000 and the standard deviation 4000. The "coloading" container, however, has a cost 0, and infinite weight and volume capacities.
- **Shipments:** We have 1000 shipments in an instance, each with its weight and volume sample from the truncated bivariate Normal distribution (lower bounded at 0) with the means $(2000, 10)$ and the covariance matrix $\begin{bmatrix} 250,000,000 & 1,000,000 \\ 1,000,000 & 4,500 \end{bmatrix}$.
- **Shipment costs:** Each shipment has a limited number of feasible non-coloading containers. For each shipment, the number of feasible containers is sampled from the truncated Normal distribution (lower bounded at 0) with the mean 10 and the standard deviation 10. Then, if shipment s has k number of feasible containers, we randomly select k containers from the container set, plus the "coloading" container. The shipment costs ξ_{sc} are sampled from a truncated Normal distribution (lower bounded at 0) with the mean 3500 and the standard deviation $10,000$.

The experiments were run on 20 simulated instances generated as above. These instances have much larger sizes than any of those tested in the Bin Packing or Generalized Assignment Problem literature. In GR, we set the parameters $\beta_1 = \beta_2 = 0.5$. In GRL, we further set the parameters $Max\_Nonimprove\_S = 1$ and $Max\_Nonimprove = 10$. In GRLV, we start with generating different GR solutions by setting different parameters of β_1, β_2 (β_1 ranging from 1 to 5 and β_2 ranging from 1 to 5). We then fix the set of β_1, β_2 that gives the best GR solution, and the parameter *num_cont_del* is set to 5. The benchmark is the solution of the integer linear program (1) returned by the Gurobi solver whose default optimality gap is 0.01%, and the solving time limit is set to 60 seconds. The setups of the experiments are described as follows.

- Program used for implementation: Julia Version 1.7.2.
- Solver used for solving the ILP: Gurobi Version 9.5.1 (academic license).
- Machine used for running: Surface Book 2 with Intel Core i7-8650 CPU @ (1.90 GHz 2.11 GHz) and 16 GB RAM.

The results of the experiments, including the optimality gaps (compared with the optimal solutions returned by the solver) and the runtimes (in seconds) of all heuristics, averaged over the 20 instances, are summarized as Table 1.

Table 1 Summary of experimental results.

Metric	Solver	GR	GRL	GRLV
Average Optimality Gap	0.01%	8.36%	4.56%	3.73%
Average Runtime (s)	26.18	7.99	72.43	3056.92

Finally, we remark that while the solver is able to solve these instances to a smaller optimality gap with shorter runtime, the problem size is expected to grow significantly in the near future. It is likely that the solver will not be able to solve the problem when its

size grows larger in the next few years. Given this expectation, a freight forwarder should be prepared to not rely on the integer linear program solver for the FCP. Therefore, our proposed heuristics will still be practically relevant.

6 Conclusion and Future Direction

In this paper, we have properly defined the freight consolidation problem (FCP) - a proven important and practically relevant problem faced by freight forwarders every day and every hour at the origin ports. We proved the non-approximability result of the FCP, and proposed a series of greedy based heuristics to solve the problem. Our solutions are shown to perform well in the numerical experiments with simulated data. For future improvement of this work, we may consider more generalized definitions of the neighborhood in the local search. We may also generate the set of used (opened) containers by some types of genetic algorithms. Furthermore, it might be helpful to use Tabu list and Tabu search to avoid repeated search of candidate solutions.

References

1 Mohamed Abdel-Basset, Gunasekaran Manogaran, Laila Abdel-Fatah, and Seyedali Mirjalili. An improved nature inspired meta-heuristic algorithm for 1-d bin packing problems. *Personal and Ubiquitous Computing*, 22(5):1117–1132, 2018.

2 Diaa Salama Abdul-Minaam, Wadha Mohammed Edkheel Saqar Al-Mutairi, Mohamed A Awad, and Walaa H El-Ashmawi. An adaptive fitness-dependent optimizer for the one-dimensional bin packing problem. *IEEE Access*, 8:97959–97974, 2020.

3 Jaza Mahmood Abdullah and Tarik Ahmed. Fitness dependent optimizer: inspired by the bee swarming reproductive process. *IEEE Access*, 7:43473–43486, 2019.

4 Shoshana Anily, Julien Bramel, and David Simchi-Levi. Worst-case analysis of heuristics for the bin packing problem with general cost structures. *Operations Research*, 42(2):287–298, 1994.

5 Merve Aydemir and Tuncay Yigit. A review of the solutions for the container loading problem, and the use of heuristics. In *The International Conference on Artificial Intelligence and Applied Mathematics in Engineering*, pages 690–700. Springer, 2019.

6 Mauro Maria Baldi and Maurizio Bruglieri. On the generalized bin packing problem. *International Transactions in Operational Research*, 24(3):425–438, 2017.

7 Mauro Maria Baldi, Teodor Gabriel Crainic, Guido Perboli, and Roberto Tadei. The generalized bin packing problem. *Transportation Research Part E: Logistics and Transportation Review*, 48(6):1205–1220, 2012.

8 Mauro Maria Baldi, Teodor Gabriel Crainic, Guido Perboli, and Roberto Tadei. Asymptotic results for the generalized bin packing problem. *Procedia-Social and Behavioral Sciences*, 111:663–671, 2014.

9 Mauro Maria Baldi, Daniele Manerba, Guido Perboli, and Roberto Tadei. A generalized bin packing problem for parcel delivery in last-mile logistics. *European Journal of Operational Research*, 274(3):990–999, 2019.

10 Avnish K. Bhatia, M Hazra, and SK Basu. Better-fit heuristic for one-dimensional bin-packing problem. In *2009 IEEE International Advance Computing Conference*, pages 193–196. IEEE, 2009.

11 Edward G Coffman, Gabor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: Combinatorial analysis. In *Handbook of Combinatorial Optimization*, pages 151–207. Springer, 1999.

12 Edward G. Coffman Jr, Michael R. Garey, and David S. Johnson. Approximation algorithms for bin packing: A survey. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1996.

13 Leah Epstein and Asaf Levin. Bin packing with general cost structures. *Mathematical Programming*, 132(1):355–391, 2012.

14 Juris Hartmanis. Computers and intractability: A guide to the theory of np-completeness (michael r. garey and david s. johnson). *SIAM Review*, 24(1):90–91, 1982.

15 Mohit Jain, Vijander Singh, and Asha Rani. A novel nature-inspired algorithm for optimization: Squirrel search algorithm. *Swarm and Evolutionary Computation*, 44:148–175, 2019.

16 David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

17 Seyedali Mirjalili and Andrew Lewis. The whale optimization algorithm. *Advances in Engineering Software*, 95:51–67, 2016.

18 Chanaleä Munien and Absalom E Ezugwu. Metaheuristic algorithms for one-dimensional bin-packing problems: A survey of recent advances and applications. *Journal of Intelligent Systems*, 30(1):636–663, 2021.

19 Ibrahim H Osman. Heuristics for the generalised assignment problem: simulated annealing and tabu search approaches. *Operations-Research-Spektrum*, 17(4):211–225, 1995.

20 Wansoo T Rhee and Michel Talagrand. Online bin packing with items of random size. *Mathematics of Operations Research*, 18(2):438–445, 1993.

21 Xin-She Yang and Suash Deb. Cuckoo search via Lévy flights. In *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*, pages 210–214. IEEE, 2009.

A Review of GBPPI

In this section, we discuss the algorithms in [9] in more detail. The overall approach can be described in three steps.

1. Constructive Heuristics. Items are given in a presorted list, and are visited one by one. All containers are closed initially. Let p_{ij} be the profit of inserting i to bin j, and let $\Phi_{res}(j)$ be the remaining space of bin j after inserting i. Upon seeing an item i, compute a weighted profit of inserting item i to bin j for all bins that are opened and has enough capacity for item i. The weighted profit is calculated as

$$\alpha \cdot p_{ij} + (1 - \alpha) \cdot \Phi_{res}(j), \tag{7}$$

where α is some parameter that can be configured. We then insert i to the bin j that results in a maximum weighted profit.

This insertion process may be generalized by looking at N items each time, where N is another parameter to be configured, rather than just one item. Specifically, we look at item i and the succeeding $N - 1$ items in the list. For each item, we find the best bin according to (7), and then select the best item-bin pair that maximizes the weighted profit.

If no bin is feasible, there are two different heuristics to choose a new bin to open:

- Best Profitable (BP). BP heuristics considers item i and the remaining succeeding items in the item list, and selects the bin that maximizes the overall profit, which is the sum of profits of the items that can be inserted into the bin deducted by the cost of that bin. If the overall profit is negative and item i is non-compulsory, then item i is discarded.
- Best Assignment (BA). BA heuristics selects the bin that maximizes the profit for item i.

At the end when all items are inserted to some bins, a post-optimization procedure is performed, which consists of two parts. First, for each bin used in the solution, we try to perform (if possible) the best swap with a bin that has not been used. Second, we remove bins from the solution that are not profitable and do not contain compulsory items.

2. Greedy Adaptive Search Procedure (GASP). GASP, shown as Algorithm 9, is a metaheuristic that uses BA or BP as a subroutine. The MULTI-START INITIALIZATION generates some initial solution and sets the initial parameters of α, N that will be used in the BP or BA constructive heuristics. Before reaching some preset time limit, the algorithm at each round first sort the items uniformly randomly. The BP or BA heuristic is then performed, and if the resulting solution is better than the best one found so far, we replace the best solution as the current one, and perform "1 to 1" swaps to search the neighborhood of the current solution. A swap consists on unloading one item to create sufficient room to insert another item that was not part of the solution. If the heuristic solution is not better than the best one, the counter *numConsecutive* is incremented. If no better solution is found after performing $MAXCONSECUTIVE$ number of constructive heuristics, we jump to the LONG-TERM INITIALIZATION PROCEDURE which will reset different parameters for α, N.

Algorithm 9 The GASP [9].

```
IS : Initial solution provided by the MULTI-START INITIALIZATION procedure
BS : best solution
BS := IS
numConsecutive : number of consecutive non-improving solutions
numConsecutive := 0
while time limit has not been reached do
    sort the items
    perform either the BP or the BA constructive heuristic
    store the resulting solution as CS
    if CS < BS then
        BS := CS
        perform "1 to 1" swaps
        numConsecutive := 0
    else
        numConsecutive := numConsecutive + 1
    end if
    SCORE UPDATE procedure
    if numConsecutive = MAXCONSECUTIVE then
        LONG-TERM REINITIALIZATION procedure
        numConsecutive := 0
    end if
end while
```

3. Model-Based Matheuristic (MBM). MBM is a parallel matheuristic for the GBPPI. During each iteration we feed the MBM a solution from GASP. Then, the set of bins used in the solution is randomly partitioned into P subsets, where P is the total number of threads available for the parallel computing. Each thread then solves the GBPPI problem using a solver with some time limit, e.g. 1 second, where the problem instance only uses a subset of bins, the items loaded to those bins, and the items not loaded in the solution.

The partial solutions returned by the solver are then merged to create a new current solution, and if the current solution is better, we save it as the best solution. This process is repeated until some time limit is reached.

In [9], the above algorithms were also tested using both artificial instances and some instances from the parcel delivery in last-mile logistics.