LIVERPOOL JOHN MOORES UNIVERSITY

LJMU Research Online

Lersteau, C, Nguyen, TT, Le, TT, Nguyen, HN and Shen, W

Solving the problem of stacking goods: mathematical model, heuristics and a case study in container stacking in ports

http://researchonline.ljmu.ac.uk/id/eprint/14255/

Article

Citation (please note it is advisable to refer to the publisher's version if you intend to cite from this work)

Lersteau, C, Nguyen, TT, Le, TT, Nguyen, HN and Shen, W Solving the problem of stacking goods: mathematical model, heuristics and a case study in container stacking in ports. IEEE Access. ISSN 2169-3536 (Accepted)

LJMU has developed **LJMU Research Online** for users to access the research output of the University more effectively. Copyright © and Moral Rights for the papers on this site are retained by the individual authors and/or other copyright owners. Users may download and/or print one copy of any article(s) in LJMU Research Online to facilitate their private study or for non-commercial research. You may not engage in further distribution of the material or use it for any profit-making activities or any commercial gain.

The version presented here may differ from the published version or from the version of the record. Please see the repository URL above for details on accessing the published version and note that access may require a subscription.

For more information please contact researchonline@ljmu.ac.uk

http://researchonline.ljmu.ac.uk/

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Solving the problem of stacking goods: mathematical model, heuristics and a case study in container stacking in ports

CHARLY LERSTEAU[1,2], TRUNG THANH NGUYEN[2], TRI THANH LE[3], HA NAM NGUYEN[4], WEIMING SHEN[1].

[1]State Key Laboratory of Digital Manufacturing Equipment & Technology, Huazhong University of Science and Technology, Wuhan, 430074, China (e-mails: charly.lersteau@gmx.fr, wshen@ieee.org)
[2]Faculty of Engineering and Technology, Liverpool John Moores University, Byrom Street, Liverpool L3 3AF, United Kingdom (e-mail: T.T.Nguyen@ljmu.ac.uk)
[3]Faculty of Information and Technology, Vietnam Maritime University (e-mail: thanhlt@vimaru.edu.vn)
[4]Information Technology Institute, Vietnam National University, Hanoi (e-mail: namnh@vnu.edu.vn)

Corresponding author: Trung Thanh Nguyen (e-mail: T.T.Nguyen@ljmu.ac.uk).

This work was supported by Newton Institutional Links grant no. 172734213 by the UK BEIS.

ABSTRACT Stacking goods or items is one of the most common operations in everyday life. It happens abundantly in not only transportation applications such as container ports, container ships, warehouses, factories, sorting centers, freight terminals, etc., but also computing systems, supermarkets, and so on. We investigate the problem of stacking a sequence of items into a set of capacitated stacks, subject to stacking constraints. In every stack, items are accessed in the last-in-first-out order. So at retrieval time, getting any lower item requires reshuffling all upper items that are blocking the way (called blocking items). These reshuffles are redundant and expensive. The challenge is to prevent reshuffles from happening. For this purpose, we aim at assigning items to stacks to minimize the number of blocking items with respect to the retrieval order. We provide some mathematical analyses on the feasibility of this problem and lower bounds. Besides, we provide a mathematical model and a two-step heuristic framework. We illustrate the applications of these models and heuristic framework in the real cargo handling process in an Asian port. Experimental results on real scenarios show that the proposed model can eliminate almost all reshuffles, and thus decrease the number of stacking violations from 62.6 % to 0.9 %. We also provide an empirical analysis of variants of the heuristic framework.

INDEX TERMS Combinatorial optimization, Containers, Heuristic algorithms, Linear Programming, Logistics, Optimization methods, Stacking

I. INTRODUCTION

THE problem of stacking goods/items (we call it the Stack Loading Problem, abbreviated as SLP) arises in many applications such as container terminals, warehouses, factories, supermarkets, computer memory, and so on. In these environments, items (or goods) arrive in a given order and are assumed to be loaded immediately in one or multiple stacks, one item on top of another. The arrangement of items in the stacks is called a configuration. These items can be retrieved later but not necessarily in the same order as they arrive. In many settings, the stacks can only be accessed from the top. It means that if an item has to be retrieved before the items above it, all the upper items, called blocking items, will have to be reshuffled. Similarly, if some of the loaded items in the stacks violate stability, load-bearing, or other stacking requirements (e.g. heavier items are on top of lighter ones), reshuffles will also be needed. Besides, some applications strictly forbid putting some items above some other items. Such restrictions are called hard stacking constraints. For example, they occur when lighter items cannot bear heavy upper items, or when some items contain dangerous goods.

Reshuffles can lead to an excessive number of redundant moves and a significant increase in cost and/or time. Take the case of container terminals as an example. Published tariffs from ports worldwide, e.g. Liverpool (Europe) [24], Portland (America) [23] and Klang (Asia) [20] indicate that the cost for a single reshuffle move can be very expensive, equal to 25-44 % the total cost of handling, storing and transporting a container through all stages of the port. Given that 90 % of the world's dry/non-bulk manufactured goods are shipped in ocean containers [6], container reshuffling in stacks is a significant issue.

This paper attempts to minimize reshuffles in stacks by minimizing the number of blocking items while making sure that no item violates hard stacking constraints. It has the following contributions: (1) Lemmas on the feasibility of the problem and lower bounds, (2) A mathematical model which allows the problem to be solved to optimality, (3) Applications of the proposed model on a real-world problem in an Asian port, showing a significant improvement in stacking efficiency, (4) A two-step heuristic framework with several variants, (5) An empirical analysis of these variants. Please note that the newly proposed model can be seen as an extension of already existing models such as [15].

a: Related work

In a comprehensive survey, Lehnfeld and Knust [19] gave a classification scheme of stacking problems in three categories: loading, pre-marshalling, and unloading problems. The problem investigated in this paper is a loading problem according to the classifications from [19]. Using the three-field notation detailed in [19], our problem can be denoted by L|π^{in},s_{ij}|BI, where BI is an objective function defined in Section II. In this section, we provide a literature review of related works.

Kim et al. [15] proposed IP models and heuristics for relaxed versions of SLP, i.e. without hard stacking constraints and stack height limit. They tackle two cases: when reshuffled items are pushed back to their stack of origin, and when they are not. Boysen and Emde [2] tackled another relaxed SLP, called PSLP. The objective is to minimize the number of blockages, i.e. the number of pairs of adjacent items such that the upper item blocks the lower one. They presented IP models, a dynamic programming procedure, and two heuristics. Boge and Knust [1] further studied several objective functions for the PSLP: the number of blockages, the number of blocking items, and the number of reshuffles. Whereas the arrival order of items is imposed and reshuffles are forbidden, the PSLP does not include hard stacking constraints, i.e. arbitrarily imposing that an item cannot be put above another one. As solution methods, MIP formulations and a simulated annealing algorithm were given. Bruns et al. [4] presented complexity results on several loading problems. One of them consists in minimizing the number of unordered stackings with hard stacking constraints but assuming that each stack cannot store more than two items. They proved that the latter can be solved in polynomial time. Delgado et al. [8] proposed an integer and a constraint programming models to optimize a weighted sum of four objectives, including the number of blocking items. However, they assumed that the arrival order of items is not imposed. Parreño et al. [22] extended the previous problem to handle items transporting dangerous goods and an additional objective. Guerra-Olivares et al. [13] analyzed the sensitivity of three stacking strategies (horizontal, vertical, and diagonal) to minimize the number of reshuffles when items arrive randomly at the storage area. They extended the analysis given in [7] concluding that the diagonal stacking strategy results in fewer reshuffles. In their experiments, horizontal stacking yielded the best performance but was sensitive to every factor studied.

The following related works deal with uncertainty. Kim et al. [17] distinguished three groups of items corresponding to retrieval priorities and assumed that the group of incoming items is not known in advance. They described a dynamic programming model based on the probability of the group of the next arriving item, to minimize the expected number of reshuffles. Zhang et al. [25] showed that the previous model contained an error and gave a correction. Kang et al. [14] solved a similar problem by simulated annealing, where the probability distribution of retrieval of items is available from past statistics. Olsen and Gross [21] gave an online heuristic to use as few stacks as possible with hard stacking constraints, assuming that the stacking restrictions of the next incoming items are unknown. Goerigk et al. [11] tackled a robust loading problem under stacking and payload constraints, where the item weights are subject to uncertainty. Exact and heuristic approaches were developed. Le and Knust [18] aimed at minimizing the number of used stacks under uncertain stacking constraints and proposed several formulations as mixed-integer programs.

Although much research has been made on optimizing stacking problems, loading problems have still attracted little attention in the literature [19]. Hard stacking constraints and stack height limits occur frequently in real-world applications such as container terminals. To the best of our knowledge, the loading problem including the latter constraints has not been extensively studied.

b: Organization

In Section II, we provide our formal description of the problem, and we study the properties. Section III provides a mathematical model. Section IV describes a heuristic framework and its variants for solving the problem. Experimental results are discussed in Section V. Finally, Section VI concludes this paper.

II. PROBLEM DESCRIPTION

The problem investigated in this paper is named as Stack Loading Problem (SLP). A sequence of incoming items has to be put in a given order in the storage area arranged as stacks. The objective is to reduce the unloading effort afterward, by minimizing the number of blocking items with respect to their retrieval order while satisfying the stacking constraints. In this section, we give a formal definition.

a: Definitions

Let $I = \{1, \dots, n\}$ be a set of items, $M = \{1, \dots, m\}$ be a set of stacks defining the storage area. Each stack can store at most b items. The set of items is partitioned into two subsets I^{fix} and I^{in}. I^{fix} is the set of initial items indexed from 1 to $|I^{\text{fix}}|$ and placed beforehand in the storage area. I^{in} is the set of incoming items indexed from $|I^{\text{fix}}|+1$ to n. When unspecified, we consider that $I^{\text{fix}} = \emptyset$ and $I^{\text{in}} = I$ by default. The position of each initial item is represented by a coordinate (k, h), where k is the stack index and h the position of the item in stack k (e.g. $h = 0$ for bottommost items). We index initial items in a stack in increasing order of their position h. We also define an array k^{fix} whose element k_i^{fix} represents the stack of item i. Thus, the order of initial items in a stack is implicitly defined from their item indices. Incoming items arrive at the storage area one after another, in increasing order of their indices. So the ingoing sequence of items is (1,2,...,n). In addition, reshuffles are forbidden. Thus, an item i will never be put above another item j if $i < j$. Besides, we need additional constraints to determine whether item i can be put above item j when $i > j$. We define two $n \times n$ binary matrices (r_{ij}) and (s_{ij}) expressing respectively soft and hard stacking constraints as follows:

$$r_{ij} = \begin{cases} 1 & \text{if item } i \text{ will be retrieved after item } j \\ 0 & \text{otherwise} \end{cases}$$

$$s_{ij} = \begin{cases} 1 & \text{if item } i \text{ can be stacked above item } j \\ 0 & \text{otherwise} \end{cases}$$

Then the binary matrix (r_{ij}) describes also the outgoing order of items. A pair of items may verify $r_{ij} = r_{ji} = 0$, e.g. if they have equal retrieval times. In this case, these items can be retrieved in any order and are not blocking each other. Soft and hard stacking constraints may define a total order when the matrices are built by comparison of times, weights, or sizes. For example, a commonly used hard stacking constraint is that larger and/or heavier items cannot be put above smaller and/or lighter ones. Stacking constraints induced by specific item conflicts may lead to an arbitrary structure. For example, items containing hazardous contents may not be stacked together or may not be stackable with some other items. Note that our constraints apply regardless of whether items are vertically adjacent or not. When $s_{ij} = 0$, item i cannot be put above item j in the same stack even if items i and j are not adjacent. Moreover, since reshuffles are forbidden, if item i arrives after item j, $s_{ij} = 0$ ensures that items i and j are located in different stacks. Table 1

TABLE 1: Problem input

I	Set of items: $\{1, \dots, n\}$
M	Set of stacks: $\{1, \dots, m\}$
b	Maximum stack capacity
k^{fix}	Stack positions of initial items
(r_{ij})	Soft stacking constraints
(s_{ij})	Hard stacking constraints

summarizes the necessary input. Loading an incoming item to a stack is called a placement. Moving an existing item from a stack to another is called a reshuffle and is not allowed at loading time. An item i is said to be blocking if it is stacked above another item j for which $r_{ij} = 1$.

b: Assumptions

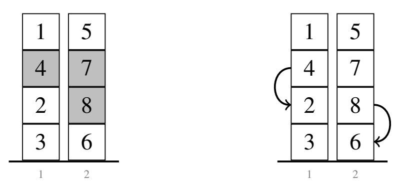
SLP has the following assumptions.

- A1: There are m stacks of capacity b.
- A2: An initial configuration (could be empty) is known in advance.
- A3: Items in a stack are accessed in the last-in-first-out order.
- A4: Items can only be put on top of a stack that can be either already loaded or empty.
- A5: Incoming items have to be put to the stacks in the order of their arrival, which is indicated by their index.
- A6: No item leaves the storage area at loading time.
- A7: Items are subject to hard stacking constraints (s_{ij}).
- A8: Reshuffles are forbidden at loading time.

c: Objective

The motivation of our work is to reduce the number of reshuffles at retrieval (unloading) time. However, we choose a surrogate objective function, minimizing the number of blocking items, for the following reasons. First, evaluating the exact minimum number of reshuffles may be very time-consuming on large instances, since it requires to solve a Blocks Relocation Problem, which is NP-hard [5]. Second, the number of blocking items is a valid lower bound on the number of required reshuffles. Indeed, every blocking item is to be reshuffled at least once. Finally, the expected minimum number of reshuffles converges to the expected number of blocking items, as shown in [10]. Note that given an arbitrary configuration, a methodology was proposed in [16] to estimate the expected number of reshuffles, but we cannot use it since it assumes that the retrieval order of items is unknown. In SLP, the objective is to minimize BI, the number of blocking items in the final configuration. Note that BI was proposed in [15, 8, 22], referred to as the number of overstows or shifts.

Apart from BI, US_{adj} can also be considered as a surrogate objective function for minimizing the number of reshuffles. US_{adj} counts every pair of adjacent items for which the upper item blocks the lower one [2]. Figure 1 shows an arbitrary



(a) BI $= 3$ (b) $\text{US}_{\text{adj}} = 2$

FIGURE 1: A stack configuration where BI and US_{adj} have different values

configuration where items are numbered by their retrieval time and shaded items represent blocking items. Item 4 is blocking both items 2 and 3. Items 7 and 8 are blocking item 6. In this example, the two objective functions have different values. Since item 7 is not adjacent to item 6, this is not counted in US_{adj}, even if item 7 requires a reshuffle. Both BI

and US_{adj} give a lower bound on the number of reshuffles, but the former is stronger than the latter. This example illustrates the relevance of choosing BI as our objective function.

d: Solution representation

A solution of SLP is expressed as a sequence of stacks $(k_1, \ldots, k_q)$ where k_j is the stack in which the j^{th} incoming item is placed. Thus, a feasible solution of SLP consists of any assignment of items to stacks satisfying the maximum stack height (b) and hard stacking constraints (s_{ij}).

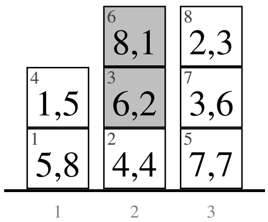
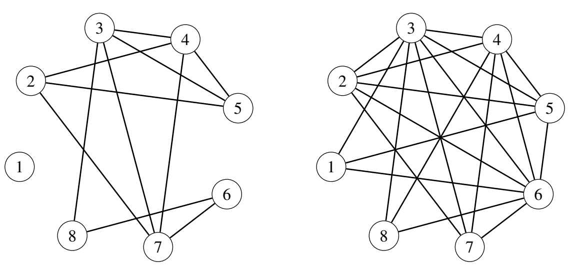


FIGURE 2: An optimal solution of SLP

e: Example

We consider a small instance with $n = 8$ items and $m = 3$ stacks of capacity $b = 3$. Each item i is associated with a retrieval time d_i and a weight w_i. The retrieval times are defined by the vector $d = (5, 4, 6, 1, 7, 8, 3, 2)$ and the weights by $w = (8, 4, 2, 5, 7, 1, 6, 3)$, both ordered with respect to the item indices. We define for every pair $(i, j) \in I^2$, $r_{ij} = 1$ if $d_i > d_j$, 0 otherwise. Moreover, we assume that a heavier item cannot be put above a lighter one. Consequently, we set $s_{ij} = 1$ if $w_i \leq w_j$, 0 otherwise. Figure 2 shows an optimal solution for SLP with BI $= 2$. On each item, the smaller number in the upper-left corner shows the index of the item. The left and right numbers are respectively the retrieval time and the weight. Shadowed items represent blocking items in the final configuration. An optimal solution for this instance of SLP is $(1, 2, 2, 1, 3, 2, 3, 3)$.

f: Conflict graphs



(a) s-conflict graph (b) rs-conflict graph

FIGURE 3: Conflict graphs (numbers are item indices)

One can visually represent hard stacking constraints of SLP as an undirected graph $G_s = (V, E_s)$ called s-conflict graph. The latter is constructed as follows. A vertex is created in V for each item in I. Without loss of generality, assume that $i < j$. Two distinct vertices i and j are adjacent if items i and j cannot be placed in the same stack, i.e. $s_{ji} = 0$. Similarly, we construct a r-conflict graph, where vertices i and j are adjacent if $r_{ji} = 0$. We also introduce the undirected graph $G_{rs} = (V, E_{rs})$ called rs-conflict graph, where two vertices i and j are adjacent in G_{rs} if their corresponding items cannot be stacked together ($s_{ji} = 0$), or one is going to block the other if put in the same stack ($r_{ji} = 1$). Figure 3 illustrates a s-conflict graph and a rs-conflict graph built from the previous example. Such representations are helpful for the implementation of the algorithm presented in Section IV to compute the degree of the nodes.

Lemma II.1. *SLP is strongly NP-hard.*

SLP without hard stacking constraints has been proven strongly NP-hard in [1]. Using the latter fact, the proof for Lemma II.1 is trivial.

Lemma II.2. *Let C be the largest clique in G_s. If the number of stacks $m < |C|$, then SLP is infeasible.*

Proof. Suppose that $C = \{i_1, \ldots, i_{m+1}\}$ of size $m + 1$. By definition of $G_s = (V, E_s)$, for any pair $(i_\ell, i_{\ell'}) \in E_s$, items i_ℓ and $i_{\ell'}$ cannot be stacked together. Therefore, the $m + 1$ items contained in C must be placed in distinct stacks. As we have only m stacks, one item cannot be placed without violating stacking constraints. □

Given the s-conflict graph from Figure 3, the size of the largest clique is 3. Thus, our example requires at least 3 stacks to admit a feasible solution. Note that the largest clique can be found in polynomial time on perfect graphs [12]. When hard stacking constraints are defined by comparison of weights, they produce a comparability graph, which is also a perfect graph.

Lemma II.3. *Let C be a clique in G_{rs} containing $|C| > m$ vertices. Then $|C| - m$ is a lower bound on the number of blocking items.*

Proof. Consider a clique C of G_{rs} of size greater than m. Without loss of generality, we assume that $i < j$. By definition of G_{rs}, any pair (i, j) of items belonging to C are *incompatible*, i.e. cannot be stacked together ($s_{ji} = 0$), or one must block the other when put in the same stack ($r_{ji} = 1$). Any subset $S \subseteq C$ put in the same stack, either is infeasible (at least one pair satisfies $s_{ji} = 0$), or causes at least $|S| - 1$ blocking items (all the items in S except the bottommost one must be blocking). Suppose that a partition of $C = \{S_1, S_2, \ldots, S_m\}$ exists such that every subset S_k is feasible. Then the number of blocking items is at least $\sum_{k=1}^{m}(|S_k| - 1) = |C| - m$. □

Given the rs-conflict graph from Figure 3, one can observe a clique of size 5, composed of items 2, 3, 4, 5, and 6. Thus, a lower bound on BI is $5 - 3 = 2$. Lemma II.3 can be generalized by considering multiple independent largest cliques instead of one.

Lemma II.4. *Let $C_1, C_2, \dots, C_q$ be q cliques in G_{rs}, such that $\forall u \in \{1, \dots, q\}$, $|C_u| > m$, and $\forall v \in \{1, \dots, q\} \setminus \{u\}$, $C_u \cap C_v = \emptyset$. Then $\sum_{u=1}^{q} |C_u| - qm$ is a lower bound on the number of blocking items.*

Proof. From Lemma II.3, we know that for each $u \in \{1, \dots, q\}$, $|C_u| - m$ is a lower bound on the number of blocking items. All cliques C_u are independent, they do not share any item in common. Therefore, the sum $\sum_{u=1}^{q} (|C_u| - m)$ is a lower bound on the number of blocking items. □

III. MATHEMATICAL MODEL

In this section, we present a 0-1 linear programming model for SLP. In some contexts such as container terminals, the decision-maker may require a way to stack containers, i.e. a sequence of placements, even if there exists no feasible solution. To do so, we propose a model allowing hard constraint violations in the case of infeasibility. However, whereas this model gives a solution even in the case of infeasibility, the optimal solutions are preserved when the instance is feasible. An item i is said to be violating if it is stacked above another item j such that $s_{ij} = 0$. When an instance is infeasible, solving the model SLP results in a sequence of moves minimizing the number of violating items first, then the number of blocking items. The proposed model, called SLP, is defined by equations (1)–(7) and includes the following binary variables:

$$x_{ik} = \begin{cases} 1 & \text{if item } i \text{ is located in stack } k \\ 0 & \text{otherwise} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{if item } i \text{ is a violating item} \\ 0 & \text{otherwise} \end{cases}$$

$$z_i = \begin{cases} 1 & \text{if item } i \text{ is a blocking item} \\ 0 & \text{otherwise} \end{cases}$$

a: SLP

$$\min \sum_{i \in I} z_i + n \sum_{i \in I} y_i \tag{1}$$

s.t.

$$\sum_{k \in M} x_{ik} = 1 \quad \forall i \in I \tag{2}$$

$$\sum_{i \in I} x_{ik} \leq b \quad \forall k \in M \tag{3}$$

$$x_{ik} + x_{jk} \leq 1 + z_i \tag{4}$$
$$\forall i \in I, j \in I, k \in M : i > j, s_{ij} = 1, r_{ij} = 1$$

$$x_{ik} + x_{jk} \leq 1 + y_i \tag{5}$$
$$\forall i \in I, j \in I, k \in M : i > j, s_{ij} = 0$$

$$x_{ik} \in \{0, 1\} \quad \forall i \in I, k \in M \tag{6}$$

$$y_i, z_i \geq 0 \quad \forall i \in I \tag{7}$$

The objective is to minimize the number of violating items first, then the number of blocking items. Since the latter is upper-bounded by $n - m$ when $n \geq m$ (bottommost items are non-blocking), multiplying the former by n guarantees that the number of violating items is minimized in priority. The purpose of this additional objective is to penalize infeasibility. Thus, a feasible solution will always dominate any solution having violating items. Note that to forbid returning a configuration in case of infeasibility, one can force $y_i = 0$. Constraint (2) ensures that each item belongs to exactly one stack. Constraint (3) guarantees that the number of items in a stack does not exceed the maximum capacity b. Constraint (4) enforces $z_i = 1$ if the item i is blocking another item j. Constraint (5) ensures that hard stacking restrictions are satisfied or enforces $y_i = 1$ if item i is a violating item. Variables y_i and z_i can be set as continuous since they are minimized and bounded by binary variables. The number of variables is $mn + 2n$ and the number of constraints is at most $n + m + mn^2$. In case $I^{\text{fix}} = \emptyset$, we can enforce $x_{ik} = 0$ for each $i > k$ to reduce the search space. Indeed, when there are several empty stacks, there is no difference in choosing one or another of them since they are equivalent choices. When $I^{\text{fix}} \neq \emptyset$, the values of x_{ik} are enforced for all $i \in I^{\text{fix}}$ and $k \in M$, i.e. $x_{ik} = 1$ if $k = k_i^{\text{fix}}$, $x_{ik} = 0$ otherwise.

Since reshuffles are not allowed at loading time, items are stacked by their order of arrival, so the ordering is implicitly defined by item indices. In particular, if items i and j are in the same stack, then item i is located above item j if $i > j$.

IV. HEURISTIC FRAMEWORK

In this section, we define the framework for solving SLP. This is an iterative method, where each iteration consists of two phases: a construction phase and an improvement phase. This intuitive design, illustrated by Algorithm 1, is commonly proposed in metaheuristics, such as GRASP [9]. Our method generalizes the method presented in [15] and the First Fit rule from [2] by using a sorting rule, a parameterizable rule and taking into account hard stacking constraints as well as a maximum stack height. It terminates when a stopping criterion is met, such as a maximum number of iterations N or a time limit.

Algorithm 1: Framework

```
s* ← ∅
while stopping criterion not met do
    s ← Construct()
    if s is feasible then
        s ← Improve(s)
        if s* = ∅ or BI(s) < BI(s*) then
            s* ← s
return s*
```

A. CONSTRUCTION PHASE

The construction phase, formalized in Algorithm 2, builds a feasible solution for SLP in two steps: a sorting step and a selection step. First, incoming items are sorted by a specified criterion to determine in which order we assign them to stacks. Second, we select a stack for each item, one after another, according to a given rule. Moreover, if there is no feasible stack available for a given item, we attempt to repair the solution by moving incompatible items. Consequently,

Algorithm 2: Construct

$s_i \leftarrow \varnothing, \forall i \in I$
$J \leftarrow$ Sort (I)
foreach $i \in J$ **do**
  $k \leftarrow$ Select (i,s)
  if $k = \emptyset$ **then**
    $k \leftarrow$ Repair (i,s)
  $s_i \leftarrow k$
return s

the solution is not necessarily built in the so-called first-in last-out manner, where items are assigned to stacks in the order of arrival. Figure 4 illustrates how to assign items to stacks in an arbitrary order while respecting the validity of the configuration. In this example, six items numbered by arrival time have already been assigned to stacks by the construction algorithm. To respect the arrival order, the next items must be located above items arriving earlier and below items arriving later. Thus, the only candidate locations for item 4 are the red insertion points shown in Figure 4. Note that when there exists more than one empty stack, only the one with the lowest index is considered as a candidate and others are ignored.

Our algorithm can easily take into account the case $I^{\text{fix}} \neq \emptyset$ by setting in advance all the values of s_i where $i \in \{1, \ldots, |I^{\text{fix}}|\}$ i.e. are already placed items. Then in the following steps, the latter values of s_i must be fixed.

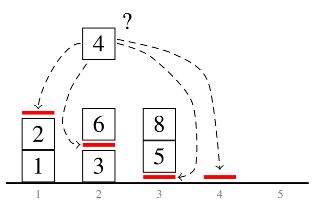


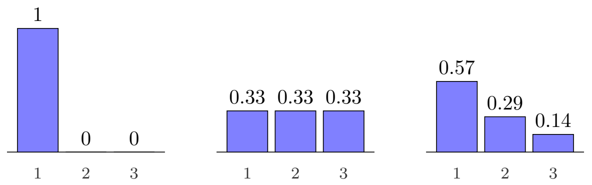
FIGURE 4: Insertion points (numbers are arrival times)

1) Sorting step

The order of items can heavily impact the decisions made during the selection step. In this paper, we study three different orders:

- LIFO: by increasing arrival time
- FIFO: by decreasing arrival time
- DEG: by decreasing degree in the conflict graphs

The LIFO order is equivalent to the common last-in first-out construction. The FIFO order is equivalent to the first-in first-out construction, i.e. appending every next item at the bottom of the stacks, like in a queue. The idea behind the DEG order is to increase the chance of obtaining a feasible solution by treating the most conflicting items first. To do so, we order the items by decreasing degree in the s-conflict graph (described in Section II). Items that have the same degree in the latter graph are ordered by decreasing degrees in the r-conflict graph. When two items have the same degree in both graphs, we choose first the one with the earliest arrival time.



(a) FIRST (b) UNIFORM (c) GEO ($q = 0.5$)

FIGURE 5: Selection probabilities

2) Selection step

During the selection step, we assign a stack to each item, one after another, according to a specified rule. Note that the latter rule must select a feasible stack, i.e. satisfying hard stacking and maximum stack height constraints. We study three rules based on the same principle: select a feasible stack in such a way that the number of additional blocking items is minimized. Such a stack is called a candidate stack. Though, there may be several candidate stacks. Assume that set of candidate stacks C is arranged from left to right, the leftmost having the index 1 and the rightmost having the index $|C|$. To break ties, we propose these three selection rules illustrated in Figure 5:

- FIRST: always choose the leftmost stack. This is identical to the First Fit rule from [2].
- UNIFORM: choose a stack randomly with equal probabilities (discrete uniform distribution).
- GEO: choose a stack randomly with decreasing probabilities from leftmost stack to rightmost stack. To do so, we define a geometric distribution with finite support.

The purpose of GEO is to provide a tradeoff between FIRST and UNIFORM to control the randomness of the selection while selecting leftmost stacks in priority. Since C has a finite size, we define a geometric distribution with finite support as follows. Let $q \in [0, 1]$ be a user-defined parameter, the probability of selecting $\ell \in C$ is:

$$\mathbb{P}(X = \ell) = pq^{\ell-1} \qquad \forall \ell \in C \tag{8}$$

To obtain a valid probability distribution, we need to define:

$$p = \frac{1-q}{1-q^{|C|}}$$

The value of q determines how the selection probability decreases from a stack to its right neighbor. When q gets closer to 0, then the chances to select the leftmost stacks are higher.

Inversely, when q gets closer to 1, the probability distribution is closer to a uniform distribution. For the particular cases $q = 0$ and $q = 1$, we assume that GEO is equivalent to FIRST and UNIFORM, respectively.

The efficiency of the construction phase is crucial since it may significantly impact the overall computational time. Indeed, solutions that are far from a local optimum may require a significant effort during the improvement phase.

3) Repair mechanism

In some cases, the selection step fails, because every stack is full or contains at least one incompatible item. Our algorithm solves this issue by running a repair mechanism. The goal of the `Repair` function (in Algorithm 2) is to make a stack available for item i by moving items causing infeasibility. Let i be the current item to assign. For each stack k, we get the set of items $\mathcal{I}_i(k)$ incompatible with i. Next, we try to move items of $\mathcal{I}_i(k)$ altogether in every feasible stack $\ell \neq k$. We select the stack leading to the minimum number of blocking items. In case of ties, we select the leftmost stack. When all feasible pairs (k, ℓ) have been enumerated, the `Repair` function chooses the first pair (k^*, ℓ^*) having the minimum number of blocking items and blocked items, lexicographically. Finally, items of $\mathcal{I}_i(k^*)$ are moved to stack ℓ^*, so item i can be assigned to stack k^*.

B. IMPROVEMENT PHASE

A feasible solution obtained from the construction phase might be further improved by local search. This procedure starts from a given solution s and attempts to move to a neighbor solution iteratively. The neighborhood $N(s)$ determines the search space reachable from s. In this paper, we define $N(s)$ as the set of feasible solutions that can be obtained by applying a k-reassignment ($k \geq 1$) on s, i.e. a reassignment of k distinct items to different stacks. When there exists no reassignment able to improve the solution, then it is a local optimum. Kim et al. [15] suggested two neighborhoods, denoted by one-opt and exchange in this paper.

A one-opt search attempts to apply a 1-reassignment in such a way that the number of blocking items BI is reduced. To do so, it explores all the feasible 1-reassignments and chooses the one that results in the best improvement. Among several equal best candidates, the stack is randomly selected with equal probabilities.

We extend this one-opt search by considering an additional objective: the number of blocked items bi. Then BI and bi are minimized lexicographically. When two reassignments result in the same value of BI, the one with the smallest bi is preferred. In addition, a solution is considered as a local optimum only when neither BI nor bi can be improved. This extended version of one-opt is called one-opt+.

Similarly, a two-opt search attempts to apply improving 2-reassignments. In this paper, the 2-reassignments are not limited to exchanges of items. For example, a first item located in the stack k may be reassigned to a stack ℓ, and a second item located in the stack ℓ may be reassigned to a stack $\ell' \neq k$. The extended version of two-opt considering bi as a secondary objective is denoted by two-opt+.

An exchange search is a restricted version of two-opt that only attempts to swap items. We denote it by exchange+ when considering bi as a secondary objective.

All the above search procedures break ties by random selection with equal probabilities.

Algorithm 3: Local search

```
repeat
    s ← One-opt (s)
    if no improvement then
        s ← Two-opt (s)
until no improvement
return s
```

The local search procedure described in Algorithm 3 applies one-opt until no more improvement is found. In this case, it attempts to perform a two-opt search. If an improvement is found, it retries to perform a one-opt search again, and so on. The algorithm stops when the current solution cannot be improved by either one-opt or two-opt.

C. IMPLEMENTATION

In practice, a naive implementation of the local search leads to significantly higher computational times than necessary. We identified two ways to reduce effort without missing solutions:

- Skip redundant 2-reassignments.
- Store additional information with the current solution.

1) Skipping redundant 2-reassignments

During the two-opt search, it is not necessary to check all the 2-reassignments. Indeed, it is easy to see that one 2-reassignment equivalent to two improving 1-reassignments can be skipped since such a reassignment should be found during a one-opt search. In fact, only the 2-reassignments in which items share common (origin or destination) stacks are non-redundant.

Let s be the current solution where s_i denotes the stack assigned to item i. Let i_1 and i_2 be a pair of distinct items to be reassigned to stacks k_1 and k_2 respectively. We assume $k_1 \neq s_{i_1}$ and $k_2 \neq s_{i_2}$. A 2-reassignment $\{(i_1, k_1), (i_2, k_2)\}$ is said non-redundant if it satisfies at least one of these equations:

- $s_{i_1} = s_{i_2}$ (same origin)
- $k_1 = k_2$ (same destination)
- $k_2 = s_{i_1}$ (destination of i_2 = origin of i_1)
- $k_1 = s_{i_2}$ (destination of i_1 = origin of i_2)

Whereas a naive two-opt search would explore up to $(m-1)^2$ choices for each pair (i_1, i_2), the number of non-redundant choices can be significantly smaller. Lemma IV.1 shows that non-redundant 2-reassignments for a given pair of items can be explored in linear time by the two-opt search.

Lemma IV.1. *When $s_{i_1} \neq s_{i_2}$, the number of non-redundant 2-reassignments of items i_1 and i_2 is at most $3m - 5$.*

Proof. There exist a total of $m - 1$ destination stacks k_1 for item i_1, since an item is not reassigned to its origin stack. Then we distinguish two cases. If $k_1 = s_{i_2}$ (the destination of i_1 is the origin of i_2), then there are $m - 1$ non-redundant possibilities for k_2. Otherwise, if $k_1 \neq s_{i_2}$, there exist $m - 2$ possibilities for k_1, and only two non-redundant possibilities for k_2: either $k_2 = k_1$ or $k_2 = s_{i_1}$. Therefore, the number of non-redundant moves is $1 \times (m - 1) + (m - 2) \times 2 = 3m - 5$. □

Algorithm 4: Remove an item i from a stack k

$J \leftarrow \{j \in I | (i > j$ and $s_{ij} = 1$ and $r_{ij} = 1)$ or $(i < j$ and $s_{ji} = 1$ and $r_{ji} = 1)\}$
foreach $j \in J$: $S_j = k$ **do**
  if $i > j$ **then**
    $u_i \leftarrow u_i - 1$
    $v_j \leftarrow v_j - 1$
    if $u_i = 0$ **then**
      $\text{BI} \leftarrow \text{BI} - 1$
    if $v_j = 0$ **then**
      $\text{bi} \leftarrow \text{bi} - 1$
  else
    $u_j \leftarrow u_j - 1$
    $v_i \leftarrow v_i - 1$
    if $u_j = 0$ **then**
      $\text{BI} \leftarrow \text{BI} - 1$
    if $v_i = 0$ **then**
      $\text{bi} \leftarrow \text{bi} - 1$

2) Storing additional information

The number of blocking items in a solution $(k_1, \ldots, k_n)$, without any additional information, can be evaluated in $\mathcal{O}(n^2)$ iterations. Since the number of evaluations may be large during the local search, it may be convenient to evaluate a solution in $\mathcal{O}(1)$ iterations. So we suggest to store BI and bi as variables as well as two vectors u and v of size n. We denote by u_i the number of items blocked by item i, and v_i the number of items blocking item i. For each item placement or reassignment, BI, bi, u and v need to be updated. This requires $\mathcal{O}(n)$ iterations (instead of $\mathcal{O}(1)$ previously), as shown in Algorithms 4 and 5. In our implementation, we observed empirically that the number of item placements/reassignments was approximately twice the number of evaluations, considering one-opt and two-opt searches. However, the overall complexity is still significantly reduced since the solutions are now evaluated in $\mathcal{O}(1)$ instead of $\mathcal{O}(n^2)$.

V. EXPERIMENTAL RESULTS

Algorithm 5: Insert an item i in a stack k

$J \leftarrow \{j \in I | (i > j$ and $s_{ij} = 1$ and $r_{ij} = 1)$ or $(i < j$ and $s_{ji} = 1$ and $r_{ji} = 1)\}$
foreach $j \in J$: $S_j = k$ **do**
  if $i > j$ **then**
    if $u_i = 0$ **then**
      $\text{BI} \leftarrow \text{BI} + 1$
    if $v_j = 0$ **then**
      $\text{bi} \leftarrow \text{bi} + 1$
    $u_i \leftarrow u_i + 1$
    $v_j \leftarrow v_j + 1$
  else
    if $u_j = 0$ **then**
      $\text{BI} \leftarrow \text{BI} + 1$
    if $v_i = 0$ **then**
      $\text{bi} \leftarrow \text{bi} + 1$
    $u_j \leftarrow u_j + 1$
    $v_i \leftarrow v_i + 1$

A. PRELIMINARY ANALYSIS

For a preliminary experiment, it is interesting to see how the number of items and the number of stacks of random instances can influence computational times. In order to obtain a landscape of random instances, we generated the heatmap of Figure 6 as follows. Given a number of items n and a number of stacks m, we generated 20 instances on the fly with retrieval times and weights defined as a random permutation in $\{1, \ldots, n\}$, and no maximum stack height. The SLP model was run on these instances with

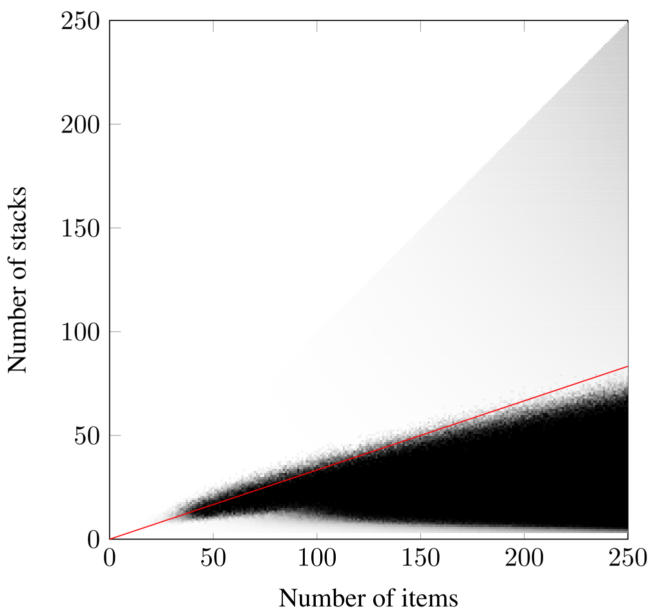


FIGURE 6: Heatmap of computational times

CPLEX 12.9.0 configured with a time limit of 5 seconds. This process has been done for all $n \in \{4, \ldots, 250\}$ and $m \in \{3, \ldots, n-1\}$. In Figure 6, the color of each pixel represents the total computational time obtained for the 20 instances corresponding to a given (n, m) pair. A white pixel means that the computational time was close to zero, whereas a black pixel means that the time limit of 5 seconds was reached for all the 20 instances. The heatmap suggests that almost all the instances above the red line of Equation $n = 3m$ were trivial. Indeed, a larger number of stacks allows smaller stacks and therefore a smaller probability of having blocking items and violating stacking constraints. On the other hand, finding a feasible solution is likely to be more difficult for instances having fewer stacks.

B. METHODOLOGY

We performed our next experiments on instances from two data sets.

The first data set (random) is made of randomly generated instances. The random dataset is itself split into two subsets: (T) instances having stacking constraints following a total order, and (A) instances having an arbitrary structure. We generated instances with $n \in \{100, 200\}$ items, $m \in \{25, 30, 35\}$ stacks for $n = 100$ and $m \in \{50, 55, 60\}$ stacks for $n = 200$; $b = 5$ and $I^{\text{fix}} = \emptyset$. Note that these parameters were chosen to cover the gap between easy and hard instances according to the heatmap of Figure 6, while avoiding infeasible instances. We created 10 instances for a selection of combinations of parameters, totalizing 120 instances, as follows. In (T) instances, each item i is associated with a retrieval time $d_i \in \{1, \ldots, n\}$ and a weight $w_i \in \{1, \ldots, n\}$ randomly permutated in $\{1, \ldots, n\}$. The retrieval order (r_{ij}) is defined by $r_{ij} = 1$ if $d_i > d_j$, 0 otherwise. Hard constraints (s_{ij}) are defined by $s_{ij} = 1$ if $w_i \leq w_j$, 0 otherwise. In (A) instances, (r_{ij}) and (s_{ij}) are randomly generated matrices where each cell is either 0 or 1 with both probabilities of $\frac{1}{2}$. Note that setting s_{ij} to 1 with a probability close to 1 would lead in significantly easier instances. On the other hand, a probability close to 0 would make instances infeasible most of the time. Similarly, too homogeneous r_{ij} values may not be relevant. Thus, we choose a probability of $\frac{1}{2}$ as a reasonable tradeoff.

The second data set (real) was produced from the real data courtesy of a port in Asia. The port's yard is organized into independent sets of stacks called blocks, each is served by one gantry crane. For compatibility reasons, we selected blocks that hosted more than 96 % of containers of the same size (either $20'$ or $40'$) for the experiments, since the case where both $20'$ and $40'$ containers are stored in the same block is not supported by our models. For each selected block, we obtained historical data covering one year and a half, which includes arrival times, retrieval times, weights, and the chosen stack. The whole period was partitioned into alternating loading and unloading sessions, in which only consecutive arrivals or retrievals occurred, respectively. After each session, items remaining in the stacks become the initial items of the next session. Taking the configuration of the previous retrieval session as inputs, each loading session is solved by the models to find the optimal configuration, and the process goes on. The sizes of the instances (in terms of the number of incoming items) are grouped by range in Table 2.

TABLE 2: Size of instances in each block (real dataset)

n^{in}	A45	A7	A8	B8	Total
1–9	713	545	512	1,025	2,795
10–49	191	247	182	346	966
50–99	35	25	21	24	105
100+	6	1	1	2	10
Total	945	818	716	1,397	3,876

The SLP model was implemented with the CPLEX C++ library version 12.9.0. The heuristic algorithms were implemented in C++. The real dataset and the random dataset with (T) and (A) instances were executed on a processor Intel Core i3-8121U with 8 GB RAM under Linux Ubuntu 20.04. In CPLEX, the time limit was set to 3600 seconds per instance.

C. HEURISTIC FRAMEWORK RESULTS

We first focus on the results of the heuristic framework on the random data set. For the sake of clarity, we treat the FIRST and UNIFORM selection rules as special cases of GEO with the parameter $q = 0$ and $q = 1$ respectively. For each variant, we ran 2000 iterations with different random seeds and saved the solution at each iteration. We analyze the impact of the sorting rule, the value of q, the repair mechanism, and the local search.

a: Sorting rule

The feasibility rate defines the percentage of iterations for which a feasible solution was found. Table 3 compares the feasibility rate among the sorting rules with the repair mechanism enabled and different values of the parameter q, for all random datasets. We observe that the DEG order always obtains a 100 % feasibility rate, regardless of the value of q, whereas LIFO and FIFO reach a 95 % feasibility rate in the best case. This suggests that treating the most conflicting items in priority decreases the chance of being stuck during the construction phase. We suppose that the most conflicting items are likely to require empty or nearly empty stacks at placement to avoid infeasibility. If one of these items is treated later, then more stacks may be occupied by incompatible items, reducing the number of candidate stacks. Since the feasibility rates of LIFO and FIFO are below our requirements, we adopt DEG as the sorting rule in the following part.

TABLE 3: Feasibility rate (in %), repair enabled

$q =$	1.0	0.5	0.2	0.15	0.1	0.05	0.0
LIFO	91.9	93.5	94.8	95	95.1	95.1	95
FIFO	91.7	92.8	93.8	94	94.2	94.5	95
DEG	100	100	100	100	100	100	100

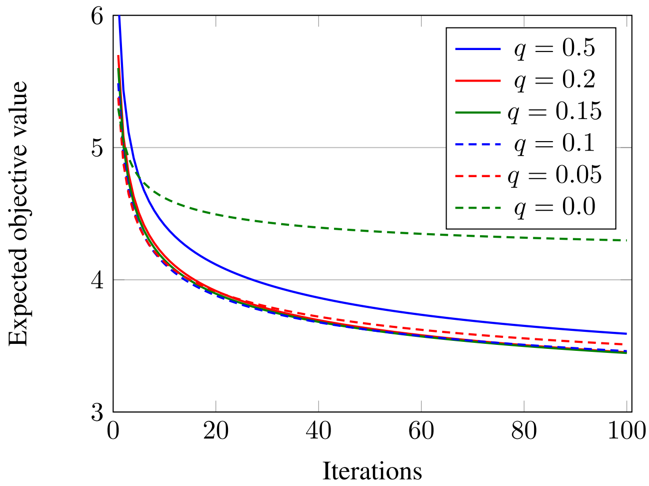


FIGURE 7: Average expected objective value with two-opt+

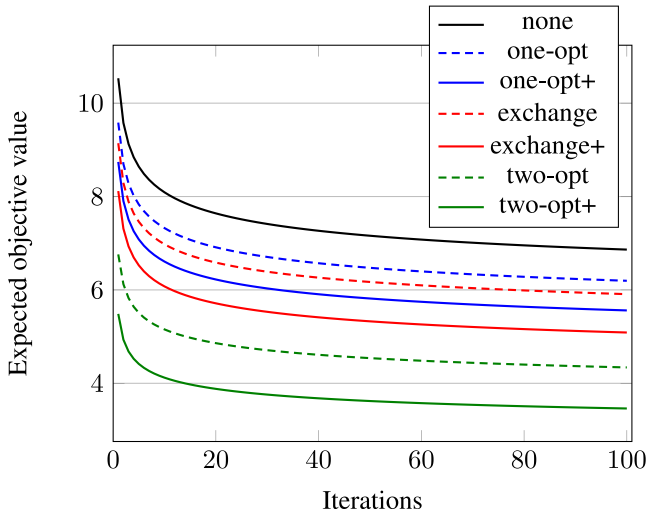


FIGURE 8: Average expected objective value with $q = 0.1$

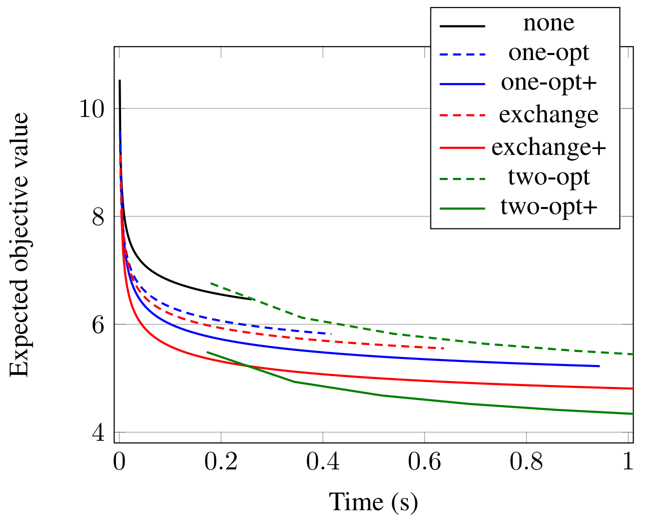


FIGURE 9: Average expected objective value over time with $q = 0.1$ (until 300 iterations)

TABLE 4: Average objective value at $N = 1$ iteration

$q =$	0.5	0.2	0.15	0.1	0.05	0.0
none	15.3	11.75	11.15	10.53	9.92	9.19
one-opt	13.39	10.56	10.08	9.58	9.05	8.39
one-opt+	11.65	9.53	9.15	8.74	8.33	7.75
exchange	12.41	10.03	9.6	9.14	8.65	7.96
exchange+	10.5	8.81	8.47	8.12	7.74	7.19
two-opt	8.12	7.16	6.96	6.76	6.54	6.32
two-opt+	6.13	5.7	5.6	5.49	5.38	5.3

TABLE 5: Average expected objective value at $N = 100$ iterations

$q =$	0.5	0.2	0.15	0.1	0.05	0.0
none	9.65	7.32	7.06	6.86	6.86	9.19
one-opt	8.25	6.53	6.37	6.2	6.2	8.35
one-opt+	6.96	5.78	5.64	5.56	5.55	7.28
exchange	7.59	6.17	6.02	5.91	5.91	7.9
exchange+	6.15	5.29	5.19	5.09	5.09	6.61
two-opt	4.83	4.41	4.35	4.34	4.4	5.85
two-opt+	3.59	3.45	3.45	3.46	3.51	4.3

TABLE 6: Average computational time of $N = 100$ iterations (in seconds)

$q =$	0.5	0.2	0.15	0.1	0.05	0.0
none	0.1	0.1	0.1	0.1	0.1	0.1
one-opt	0.2	0.2	0.1	0.1	0.1	0.1
one-opt+	0.5	0.4	0.3	0.3	0.3	0.3
exchange	0.4	0.2	0.2	0.2	0.2	0.2
exchange+	0.7	0.4	0.4	0.4	0.4	0.3
two-opt	40.3	23.1	20.5	17.9	15.4	12.2
two-opt+	31.4	20.8	19	17.2	15.3	12.5

b: Value of q

Assuming that the stopping criterion is a limit of N iterations, we compute the expected average objective value denoted by E_N. The method to compute E_N is described in Appendix A. Tables 4 and 5 show the average expected objective value at $N = 1$ and $N = 100$ iteration(s) respectively, according to the local search and the parameter q. Figure 7 shows the evolution of the average expected objective value with a two-opt+ local search. We observe that the most deterministic version ($q = 0$) of our algorithm finds better solutions from the first iterations, but it is not able to improve further the objective value because of a lack of diversity in the search space. On the other hand, a more randomized version slows down the convergence but may reach solutions of better quality after a very large number of iterations, as suggested by the promising trajectory of the curve with $q = 0.5$. Therefore, we suggest setting the value of q according to the computational time limits of the decision-maker.

c: Repair mechanism

We analyze the impact of the repair mechanism on the ability to find feasible solutions, assuming $q = 0.1$. Without the repair mechanism, LIFO, FIFO, and DEG failed to find at least one feasible solution on respectively 35, 37, and 4 instances. The feasibility rates are respectively 49.6 %,

TABLE 7: Results for $n = 100$
Sorting rule: DEG, local search: 2-opt+, $q = 0.1$

				BI				
				CPLEX			Time (s)	
Inst.	n	m	LB	10m	1h	H	CPLEX	H
A01	100	25	0	12	**4**	4.7	>3,600	**6.9**
A02	100	25	0	14	5	5.0	>3,600	**7.1**
A03	100	25	0	12	**4**	4.1	>3,600	**7.1**
A04	100	25	0	8	5	**4.1**	>3,600	**6.5**
A05	100	25	0	7	6	**3.4**	>3,600	**5.4**
A06	100	25	0	11	5	**4.3**	>3,600	**7.2**
A07	100	25	0	11	**4**	4.9	>3,600	**6.4**
A08	100	25	0	12	12	**4.2**	>3,600	**5.3**
A09	100	25	0	9	5	**3.8**	>3,600	**5.4**
A10	100	25	0	12	6	**3.6**	>3,600	**7.2**
A11	100	30	0	0	0	0.0	360.0	**0.7**
A12	100	30	0	0	0	0.0	367.3	**0.9**
A13	100	30	0	0	0	0.0	61.9	**0.6**
A14	100	30	0	0	0	0.0	25.9	**0.7**
A15	100	30	0	0	0	0.0	442.7	**1.7**
A16	100	30	0	0	0	0.0	332.2	**0.5**
A17	100	30	0	0	0	0.0	272.8	**0.8**
A18	100	30	0	0	0	0.0	260.8	**1.5**
A19	100	30	0	0	0	0.0	281.8	**0.6**
A20	100	30	0	0	0	0.0	513.4	**1.0**
A21	100	35	0	0	0	0.0	<0.1	<0.1
A22	100	35	0	0	0	0.0	<0.1	<0.1
A23	100	35	0	0	0	0.0	<0.1	<0.1
A24	100	35	0	0	0	0.0	<0.1	<0.1
A25	100	35	0	0	0	0.0	<0.1	<0.1
A26	100	35	0	0	0	0.0	<0.1	<0.1
A27	100	35	0	0	0	0.0	<0.1	<0.1
A28	100	35	0	0	0	0.0	<0.1	<0.1
A29	100	35	0	0	0	0.0	<0.1	<0.1
A30	100	35	0	0	0	0.0	<0.1	<0.1
T01	100	25	14	25	24	**19.6**	>3,600	**13.7**
T02	100	25	6	13	**10**	11.7	>3,600	**8.9**
T03	100	25	9	14	**9**	10.2	>3,600	**4.4**
T04	100	25	14	39	26	**22.3**	>3,600	**12.9**
T05	100	25	8	14	14	**11.7**	>3,600	**7.5**
T06	100	25	12	18	**16**	17.2	>3,600	**11.3**
T07	100	25	11	19	**13**	13.8	>3,600	**9.8**
T08	100	25	10	20	14	**13.8**	>3,600	**8.1**
T09	100	25	11	16	**14**	14.7	>3,600	**6.7**
T10	100	25	8	18	**11**	11.9	>3,600	**10.2**
T11	100	30	8	11	**8**	9.3	>3,600	**9.6**
T12	100	30	1	3	**1**	2.0	>3,600	**3.1**
T13	100	30	12	15	**12**	13.4	>3,600	**14.3**
T14	100	30	6	8	6	6.0	>3,600	**6.1**
T15	100	30	3	4	**3**	3.7	>3,600	**5.8**
T16	100	30	7	10	7	7.1	>3,600	**5.3**
T17	100	30	2	4	**2**	2.3	>3,600	**2.7**
T18	100	30	6	7	**6**	6.7	>3,600	**6.4**
T19	100	30	1	2	**1**	2.1	>3,600	**3.9**
T20	100	30	1	5	**1**	2.4	>3,600	**4.2**
T21	100	35	1	2	1	1.0	>3,600	**2.3**
T22	100	35	1	1	1	1.0	>3,600	**2.3**
T23	100	35	0	0	0	0.0	**<0.1**	1.2
T24	100	35	2	5	**2**	2.5	>3,600	**1.8**
T25	100	35	0	2	**0**	0.1	1,084.7	**1.3**
T26	100	35	2	6	**2**	3.0	>3,600	**6.8**
T27	100	35	0	0	0	0.0	**<0.1**	0.3
T28	100	35	0	0	0	0.0	<0.1	<0.1
T29	100	35	2	**2**	**2**	2.3	2,146.6	**4.9**
T30	100	35	0	0	0	0.0	**<0.1**	0.2

TABLE 8: Results for $n = 200$
Sorting rule: DEG, local search: 2-opt+, $q = 0.1$

				BI				
				CPLEX			Time (s)	
Inst.	n	m	LB	10m	1h	H	CPLEX	H
A31	200	50	0	16	16	**0.0**	>3,600	**25.5**
A32	200	50	0	15	15	**0.0**	>3,600	**29.6**
A33	200	50	0	17	17	**0.1**	>3,600	**31.4**
A34	200	50	0	16	16	**0.0**	>3,600	**27.1**
A35	200	50	0	25	25	**0.0**	>3,600	**23.2**
A36	200	50	0	22	22	**0.0**	>3,600	**27.4**
A37	200	50	0	16	16	**0.2**	>3,600	**32.3**
A38	200	50	0	16	16	**0.2**	>3,600	**33.4**
A39	200	50	0	24	24	**0.1**	>3,600	**29.9**
A40	200	50	0	21	21	**0.1**	>3,600	**29.2**
A41	200	55	0	0	0	0.0	27.2	**2.5**
A42	200	55	0	4	0	0.0	1,667.2	**1.7**
A43	200	55	0	0	0	0.0	**0.1**	1.6
A44	200	55	0	7	0	0.0	1,681.2	**2.4**
A45	200	55	0	0	0	0.0	**0.1**	3.2
A46	200	55	0	3	0	0.0	1,893.6	**3.0**
A47	200	55	0	5	0	0.0	1,359.1	**2.1**
A48	200	55	0	7	0	0.0	2,215.7	**3.5**
A49	200	55	0	2	2	**0.0**	>3,600	**3.8**
A50	200	55	0	11	9	**0.0**	>3,600	**2.3**
A51	200	60	0	0	0	0.0	0.1	<0.1
A52	200	60	0	0	0	0.0	0.1	<0.1
A53	200	60	0	0	0	0.0	0.1	<0.1
A54	200	60	0	0	0	0.0	0.1	<0.1
A55	200	60	0	0	0	0.0	0.1	<0.1
A56	200	60	0	0	0	0.0	0.1	<0.1
A57	200	60	0	0	0	0.0	0.1	<0.1
A58	200	60	0	0	0	0.0	0.1	<0.1
A59	200	60	0	0	0	0.0	0.1	0.1
A60	200	60	0	0	0	0.0	0.1	0.1
T31	200	50	5	27	27	**9.6**	>3,600	**82.5**
T32	200	50	12	45	45	**19.3**	>3,600	**174.2**
T33	200	50	5	24	24	**9.9**	>3,600	**74.5**
T34	200	50	4	25	25	**6.5**	>3,600	**54.2**
T35	200	50	0	20	20	**3.6**	>3,600	**40.1**
T36	200	50	8	29	29	**10.6**	>3,600	**82.0**
T37	200	50	4	24	24	**6.7**	>3,600	**85.4**
T38	200	50	7	35	35	**9.2**	>3,600	**76.5**
T39	200	50	5	28	28	**8.5**	>3,600	**87.5**
T40	200	50	0	24	24	**3.8**	>3,600	**38.3**
T41	200	55	1	22	22	**2.2**	>3,600	**24.1**
T42	200	55	7	29	29	**12.0**	>3,600	**79.6**
T43	200	55	0	20	18	**0.8**	>3,600	**31.5**
T44	200	55	3	19	19	**5.1**	>3,600	**56.1**
T45	200	55	1	22	22	**4.0**	>3,600	**39.0**
T46	200	55	3	20	20	**5.4**	>3,600	**51.0**
T47	200	55	4	14	14	**6.7**	>3,600	**72.9**
T48	200	55	5	27	25	**8.6**	>3,600	**54.8**
T49	200	55	2	19	18	**3.0**	>3,600	**28.1**
T50	200	55	10	28	27	**10.4**	>3,600	**73.4**
T51	200	60	0	0	0	0.0	**0.1**	4.0
T52	200	60	0	17	17	**1.7**	>3,600	**25.7**
T53	200	60	0	12	11	**0.7**	>3,600	**31.0**
T54	200	60	3	20	18	**3.8**	>3,600	**35.5**
T55	200	60	0	0	0	0.0	**0.1**	1.9
T56	200	60	2	17	16	**3.5**	>3,600	**21.2**
T57	200	60	0	20	18	**2.1**	>3,600	**33.2**
T58	200	60	0	0	0	0.0	**0.1**	6.6
T59	200	60	0	0	0	0.0	**0.1**	3.3
T60	200	60	1	11	11	**3.0**	>3,600	**28.8**

TABLE 9: Average computational time of $N = 100$ iterations with DEG and $q = 0.1$ (in seconds)

n	m	none	one-opt	one-opt+	exchange	exchange+	two-opt	two-opt+
100	25	<0.1	<0.1	0.1	0.1	0.2	6.2	7.9
100	30	<0.1	<0.1	0.1	<0.1	0.1	3.5	3.5
100	35	<0.1	<0.1	<0.1	<0.1	<0.1	1.3	1.1
200	50	0.1	0.2	0.7	0.5	0.9	53.6	54.2
200	55	0.1	0.2	0.6	0.3	0.6	30.3	26.8
200	60	0.1	0.2	0.3	0.2	0.3	12.8	9.6

47.4 %, and 92.9 %, suggesting that DEG is significantly more reliable for finding feasible solutions. With the repair mechanism, LIFO, FIFO, and DEG failed on respectively 1, 1, and 0 instances. These results show that the relevance of repairing infeasible solutions and confirm our choice of DEG as our default sorting rule.

d: Local search

Assuming $q = 0.1$, Figure 8 compares the evolution of the average expected objective value according to the local search depth. Applying a two-opt+ local search reduced by at least 47 % the number of blocking items on average compared to no local search, regardless of the iteration between 1 and 100. Compared to exchange+, two-opt+ reduced by 32 % the number of blocking items. We also observe that the extended versions of the local searches reduced significantly the blocking items compared to the basic versions.

Although each iteration of two-opt+ was on average 43 times slower than exchange+ (as shown in Table 6), two-opt+ outperformed all the other local searches after a few iterations, as shown in Figure 9. Nevertheless, the latter result may significantly vary depending on the implementation details. We observe that the computational time increases as the value of q increases. We focus on the runs where two-opt+ was enabled. When $q = 0$, the average number of one-opt operations per iteration was 5.7, whereas it was 7.9 when $q = 0.2$, and 11.2 when $q = 0.5$. The average number of two-opt operations per iteration was respectively 2.7, 4.1, and 5.6. It means that when the value of q is greater, the chances that the constructed solution is far from a local optimum are greater, then the local search required more computational time.

Table 9 gives more detailed average computational times with $q = 0.1$, according to the number of items and the number of stacks. We observe a significant gap between instances having less than $\frac{n}{3}$ stacks and the others. This gap and the one observed around the red line in the heatmap of Figure 6 suggest the existence of a clear shortage between easy and hard instances.

D. SLP MODEL RESULTS

In the following part, we adopt DEG, $q = 0.1$, two-opt+ and $N = 100$ iterations as the default parameter set of our heuristic framework for a comparison with the CPLEX performance on the SLP model. In Tables 7 and 8, the column LB shows the lower bounds computed using the Lemma II.4. We used a modified Bron-Kerbosch algorithm [3] to iteratively search for largest cliques in G_{rs}. The column BI reports the objective value of CPLEX obtained after 10 minutes (10m) and after 1 hour (1h), as well as the expected objective value of our heuristic (H). The last two columns show the computational times of CPLEX and our heuristic. These results highlight again that the computation times of the SLP model on CPLEX are not necessarily related to the overall size of the instance, but the ratio between the number of items and the number of stacks. Indeed, most of the instances having high $\frac{n}{m}$ ratios reached the time limit with CPLEX, whereas instances having low $\frac{n}{m}$ ratios were more often solved in 0.1 seconds. We observe discrepancies in computational times for instances T21 to T30, and instances T51 to T60. Some instances were solved to optimality in 0.1 seconds, whereas the rest reached at least 1,000 seconds. This large gap suggests that instances having the same n and m values could be split into two distinct classes of difficulty.

E. APPLICATION TO THE REAL DATA SET

Table 10 shows the performance of SLP model on the real data set. The column #inst shows the total numbers of instances for each block of the port. The columns n and Time show the total number of items and the total computational time respectively. The columns BI and V show the number of blocking items and the number of violating items, respec-

TABLE 10: Results of the SLP model and the heuristic framework on the real dataset

Block	#inst.	n	m	b	BI			V			Time (s)	
					Port	CPLEX	H	Port	CPLEX	H	CPLEX	H
A7	818	7670	54	4	2933	**57**	71	3601	13	13	61.4	**8.2**
A8	716	6765	54	4	2417	**5**	6	2684	1	1	2.5	**1.6**
A45	945	8835	66	5	3702	148	**146**	3842	26	**25**	19.0	**9.5**
B8	1397	11986	130	4	6951	**71**	156	6187	9	**4**	3103.8	**184.4**
Total	3876	35256			16003	**281**	379	16314	49	**43**	3186.6	**203.7**
(%)					45 %	**0.8 %**	1.1 %	46.3 %	**0.1 %**	**0.1 %**		

tively, split into three subcolumns showing the number of blocking/violating items obtained by the current practice of the port, CPLEX, and the heuristic framework (H). The last line in Table 10 expresses the percentage of blocking items. The SLP model (on CPLEX) found the optimal solution in less than 10 seconds in almost all instances except for three instances in blocks A7 and B8. However, in both cases, CPLEX was able to find a feasible solution in a few seconds. In comparison to current practice in the port, the SLP model is able to reduce the number of blocking items from 45 % to 0.8 %, the number of violating items from 46 % to 0.1 %, and the number of mixed blocking/violating items from 62.6 % to 0.9 %.

VI. CONCLUSION

In this paper, we tackled a Stack Loading Problem (SLP). We also proved a sufficient condition of infeasibility that can be checked in polynomial time. In addition to the theoretical studies, we proposed a mathematical model. We provided a flexible heuristic framework with several variants in order to analyze them. The experiments showed that the heuristic framework with certain parameters was competitive compared to a commercial solver such as CPLEX. Experiments with CPLEX have shown that our SLP model was able to solve most of the tested real cases in less than 10 seconds.

In this work, we assumed that the retrieval times were all known in advance. However, this assumption may not be applicable in some contexts. In our container terminal, the retrieval time was unknown for approximatively 10 % of the containers at loading time. One can use the average stay time as a default value, but it might lead to solutions lacking robustness. Taking into account this uncertainty based on past statistics is a perspective of our future work. In container terminals, a storage area might accept simultaneously 20' and 40' containers. Therefore, another perspective is to solve the problem that allows stacking containers of different sizes (e.g. two 20' containers on top of a single 40' container or the opposite). We also consider designing exact methods for the most difficult instances of SLP, in which $n > 3m$. Another future research we consider is to find tighter lower bounds.

.

APPENDIX A EXPECTED OBJECTIVE VALUE

To compute the average objective value obtained at N iterations, one way is to perform a large number of runs, with a stopping criterion of N iterations. However, this experiment might be very long. Instead, we exploit the fact that iterations are independent. Running a large number of single iterations results in a distribution of objective values illustrated in Figure 10. It shows which objective values were obtained with their respective probabilities. Using this data, we compute the expected objective value at N iterations.

Given a number N of iterations, an algorithm $\mathcal{A}$ and an instance $\mathcal{I}$, the expected objective value is computed as follows. Let $X_1, X_2, \ldots, X_N$ be random variables following identical discrete probability distributions, each of them representing the objective value obtained by one iteration. Let $\mathcal{X} = \{x_1, \ldots, x_k\}$ be the set of all the possible outcomes of X_i ordered by increasing values, and $\mathcal{P} = \{p_1, \ldots, p_k\}$ their respective probabilities. Let $Y = \min(X_1, \ldots, X_N)$ the minimum objective value among all the N iterations.

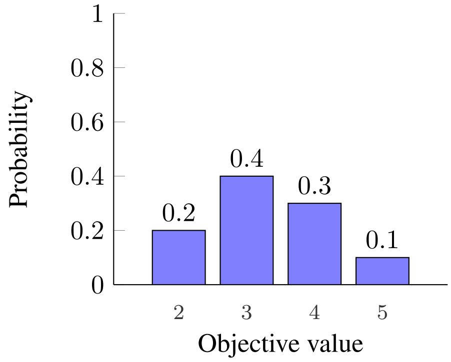


FIGURE 10: A distribution of objective values

The expected objective value at N iterations is defined by:

$$\mathbb{E}(Y) = \sum_{i=1}^{k} x_i \mathbb{P}(Y = x_i)$$

Given an outcome $x_i \in \mathcal{X}$,

$$\mathbb{P}(Y = x_i) = \mathbb{P}(Y \geq x_i) - \mathbb{P}(Y \geq x_{i+1})$$

$$\mathbb{P}(Y \geq x_i) = \mathbb{P}(X_1 \geq x_i, \ldots, X_k \geq x_i) = \prod_{j=1}^{N} \mathbb{P}(X_j \geq x_i)$$

The random variables are identical and independent:

$$\mathbb{P}(Y \geq x_i) = (\mathbb{P}(X_1 \geq x))^N$$

The values of x_i are ordered by increasing values, then we have $\mathbb{P}(X_1 \geq x_i) = \sum_{j=i}^{k} p_j$ and:

$$\mathbb{P}(Y \geq x_i) = \left(\sum_{j=i}^{k} p_j\right)^N$$

Consequently:

$$\mathbb{P}(Y = x_i) = \left(\sum_{j=i}^{k} p_j\right)^N - \left(\sum_{j=i+1}^{k} p_j\right)^N$$

Finally, the expected objective value at N iterations is defined by:

$$\mathbb{E}(Y) = \sum_{i=1}^{k} x_i \left(\left(\sum_{j=i}^{k} p_j\right)^N - \left(\sum_{j=i+1}^{k} p_j\right)^N\right)$$

In the example of Figure 10, the expected objective value at 10 iterations is:

$$2 \times (1^{10} - 0.8^{10}) + 3 \times (0.8^{10} - 0.4^{10}) + 4 \times (0.4^{10} - 0.1^{10}) + 5 \times (0.1^{10} - 0^{10}) = 2.10748$$

ACKNOWLEDGEMENTS

This research is supported by Newton Institutional Links grant no. 172734213 by the UK BEIS. We thank Mario Garza-Fabre for his very helpful remarks.

REFERENCES

[1] Sven Boge and Sigrid Knust. The parallel stack loading problem minimizing the number of reshuffles in the retrieval stage. European Journal of Operational Research, 280(3):940 – 952, 2020. ISSN 0377-2217. . URL http://www.sciencedirect.com/science/article/pii/S0377221719306587.

[2] Nils Boysen and Simon Emde. The parallel stack loading problem to minimize blockages. European Journal of Operational Research, 249(2):618–627, mar 2016. . URL https://doi.org/10.1016%2Fj.ejor.2015.09.033.

[3] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. Communications of the ACM, 16(9):575–577, sep 1973. . URL https://doi.org/10.1145%2F362342.362367.

[4] Florian Bruns, Sigrid Knust, and Natalia V. Shakhlevich. Complexity results for storage loading problems with stacking constraints. European Journal of Operational Research, 249(3):1074–1081, mar 2016. . URL https://doi.org/10.1016%2Fj.ejor.2015.09.036.

[5] Marco Caserta, Silvia Schwarze, and Stefan Voß. A mathematical formulation and complexity considerations for the blocks relocation problem. European Journal of Operational Research, 219(1):96–104, may 2012. . URL https://doi.org/10.1016%2Fj.ejor.2011.12.039.

[6] James Castonguay. International shipping: Globalization in crisis. Witness (online magazine), page 4, 2009. URL http://www.visionproject.org/images/img_magazine/pdfs/international_shipping.pdf.

[7] Lu Chen and Zhiqiang Lu. The storage location assignment problem for outbound containers in a maritime terminal. International Journal of Production Economics, 135(1):73–80, jan 2012. . URL https://doi.org/10.1016%2Fj.ijpe.2010.09.019.

[8] Alberto Delgado, Rune Møller Jensen, Kira Janstrup, Trine Høyer Rose, and Kent Høj Andersen. A constraint programming model for fast optimal stowage of container vessel bays. European Journal of Operational Research, 220(1):251–261, jul 2012. . URL https://doi.org/10.1016%2Fj.ejor.2012.01.028.

[9] Thomas A. Feo and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. Journal of Global Optimization, 6(2):109–133, mar 1995. . URL https://doi.org/10.1007%2Fbf01096763.

[10] V. Galle, S. Borjian Boroujeni, V.H. Manshadi, C. Barnhart, and P. Jaillet. An average-case asymptotic analysis of the container relocation problem. Operations Research Letters, 44(6):723–728, nov 2016. . URL https://doi.org/10.1016%2Fj.orl.2016.08.006.

[11] Marc Goerigk, Sigrid Knust, and Xuan Thanh Le. Robust storage loading problems with stacking and payload constraints. European Journal of Operational Research, 253(1):51–67, aug 2016. . URL https://doi.org/10.1016%2Fj.ejor.2016.02.019.

[12] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. Combinatorica, 1(2):169–197, jun 1981. . URL https://doi.org/10.1007%2Fbf02579273.

[13] Roberto Guerra-Olivares, Neale R. Smith, Rosa G. González-Ramírez, and Leopoldo Eduardo Cárdenas-Barrón. A study of the sensitivity of sequence stacking strategies for the storage location assignment problem for out-bound containers in a maritime terminal. International Journal of System Assurance Engineering and Management, 9(5):1057–1062, Oct 2018. ISSN 0976-4348. . URL https://doi.org/10.1007/s13198-018-0733-x.

[14] Jaeho Kang, Kwang Ryel Ryu, and Kap Hwan Kim. Deriving stacking strategies for export containers with uncertain weight information. Journal of Intelligent Manufacturing, 17(4):399–410, aug 2006. . URL https://doi.org/10.1007%2Fs10845-005-0013-x.

[15] Byung-In Kim, Jeongin Koo, and Hotkar Parshuram Sambhajirao. A simplified steel plate stacking problem. International Journal of Production Research, 49(17): 5133–5151, sep 2011. . URL https://doi.org/10.1080%2F00207543.2010.518998.

[16] Kap Hwan Kim. Evaluation of the number of rehandles in container yards. Computers & Industrial Engineering, 32(4):701–711, sep 1997. . URL https://doi.org/10.1016%2Fs0360-8352%2897%2900024-7.

[17] Kap Hwan Kim, Young Man Park, and Kwang-Ryul Ryu. Deriving decision rules to locate export containers in container yards. European Journal of Operational Research, 124(1):89–101, jul 2000. . URL https://doi.org/10.1016%2Fs0377-2217%2899%2900116-2.

[18] Xuan Thanh Le and Sigrid Knust. MIP-based approaches for robust storage loading problems with stacking constraints. Computers & Operations Research, 78:138–153, feb 2017. . URL https://doi.org/10.1016%2Fj.cor.2016.08.016.

[19] Jana Lehnfeld and Sigrid Knust. Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. European Journal of Operational Research, 239(2):297–312, dec 2014. . URL https://doi.org/10.1016%2Fj.ejor.2014.03.011.

[20] Northport (Malaysia) BHD. Tariff https://www.northport.com.my/npv2/containerservices.html, 2017. URL http://www.northport.com.my/npv2/tariff_ccd%20edit%20(final%2011.09.15)(2).pdf.

[21] Martin Olsen and Allan Gross. Probabilistic analysis of online stacking algorithms. In Lecture Notes in Com-

puter Science, pages 358–369. Springer Nature, 2015. . URL https://doi.org/10.1007%2F978-3-319-24264-4_25.

[22] Francisco Parreño, Dario Pacino, and Ramon Alvarez-Valdes. A GRASP algorithm for the container stowage slot planning problem. Transportation Research Part E: Logistics and Transportation Review, 94:141–157, oct 2016. . URL https://doi.org/10.1016%2Fj.tre.2016.07.011.

[23] Port of Portland. Terminal tariff no. 8 https://www2.portofportland.com/Marine/Tariff, 2017. URL https://popcdn.azureedge.net/pdfs/Marine%20Tariff%20No.%208%202017.pdf.

[24] The Mersey Docks and Horbour Company. Schedule of common user charges liverpool container terminals https://www.peelports.com/ports/liverpool, 2017. URL https://www.peelports.com/media/2317/2017-liverpool-container-terminals-charges.pdf.

[25] Canrong Zhang, Weiwei Chen, Leyuan Shi, and Li Zheng. A note on deriving decision rules to locate export containers in container yards. European Journal of Operational Research, 205(2):483–485, sep 2010. . URL https://doi.org/10.1016%2Fj.ejor.2009.12.016.

CHARLY LERSTEAU received the B.Sc and M.Sc degrees in Computer Science and Operational Research from the University of Nantes, France, in 2013 and the Ph.D. degree in computer science from the University of South Brittany, France, in 2016. From 2017 to 2019, he was a Research Fellow at Liverpool John Moores University, UK. Since 2019, he is a Research Fellow at Huazhong University of Science and Technology, in Wuhan, China.

His research interests span algorithms, graph theory, linear programming, metaheuristics, large-scale optimization, complexity theory. He has been involved in multiple projects with applications in military, maritime, and logistics domains, including one funded by DfT about rail transportation. His experience covers solving a range of optimization problems such as wireless sensor networks, facility location, container stacking, and vehicle routing problems.

TRUNG T. NGUYEN is a Reader in Operational Research (OR), Liverpool John Moores University and the co-director of the Liverpool Offshore and Marine Research Institute. He has an international standing in operational research for logistics/transport. He has led over 20 research projects in transport/logistics, most with close industry collaborations. He has published about 50 peer-reviewed papers. All of his journal papers are in leading journals (ranked 1st - 20th in their fields). He co-organized six leading conferences, was TPC member of more than 30 international conferences, edited eight books and gave speeches to many conferences/events.

TRI T. LE received the M.Sc degree in Information Technology in Department of Information Technology, Military Technical Academy (Le Qui Don Technical University), Hanoi, Vietnam, in 2010 and Bachelor of Information Technology at Faculty of Information Technology, Vietnam Maritime University, Haiphong, Vietnam, in 2004. He is currently pursuing the Ph.D. degree in mechanical engineering at VNU University of Science, Hanoi, Vietnam. From September 2016 to August 2017, he was a researcher in Liverpool John Moores University, Liverpool, UK. His research interests are optimization and simulation of maritime, transport and logistics problems.

HA N. NGUYEN (M'76) received his B.Sc in Information Technology from VNU-Hanoi University of Science and Technology in 1998, M.Sc in Computer Science from Chungwoon University, Korea in 2003, and Ph.D. in Software Applications from Korea Aerospace University, Korea in 2007. From 2007 to 2017, he worked for Department of Information Systems in the University of Engineering and Technology as a Senior Lecturer in Data Mining, Statistical Machine Learning, and Database. He is currently serving as Vice president of Information Technology Institute (ITI), Vietnam National University in Hanoi (VNU) since 2017. He is interested in financial risk analysis, behavior analysis, developing information systems and maritime logistics/transport using techniques from data analysis, modeling, and software engineering.

WEIMING SHEN (M'98–SM'02–F'12) received his Bachelor and Master's degrees from Northern (Beijing) Jiaotong University, China (in 1983 and 1986 respectively) and his Ph.D. degree from the University of Technology of Compiègne, France (in 1996). He is currently a Professor at Huazhong University of Science and Technology (HUST), China, and an Adjunct Professor at the University of Western Ontario, Canada. Prior to joining HUST, he was a Principal Research Officer at National Research Council Canada. His research interest includes collaborative intelligent technologies and systems, and their applications in industry. He is a Fellow of IEEE, a Fellow of the Canadian Academy of Engineering, a Fellow of the Engineering Institute of Canada, and a licensed Professional Engineer in Ontario, Canada.

• • •