CORE
Metadata, citation and similar papers at core.ac.uk
Provided by The IT University of Copenhagen's Repository

Matheuristics for Slot Planning of Container Vessel Bays[☆]

Aleksandra Korach[a], Berit Dangaard Brouer[b], Rune Møller Jensen[a,*]

[a] *IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark*
[b] *Optivation, Njalsgade 76, 2300 Copenhagen S, Denmark*

Abstract

Stowage planning is an NP-hard combinatorial problem concerned with loading a container vessel in a given port, such that a number of constraints regarding the physical layout of the vessel and its seaworthiness are satisfied, and a number of objectives with regard to the quality of the placement are optimized. State-of-the-art methods decompose the problem into phases, the latter of which, known as slot planning, involves loading the containers into slots of a bay. This article presents an efficient matheuristic for the slot planning problem. Matheuristics are algorithms using mathematical programming techniques within a heuristic framework. The method finds solutions for 96% of 236 instances based on real stowage plans, 90% of them optimally, with an average optimality gap of 4.34% given a limit of one second per instance. This is an improvement over the results provided by previous works.

Keywords: OR in Maritime Industry, Stowage Planning, Slot Planning, Matheuristics, Large Neighbourhood Search

1. Introduction

With approximately 90% of non-bulk cargo worldwide carried by container vessels, maritime transport remains the foundation of international trade. These vessels travel on specific trade routes calling at multiple ports, where some containers remain on board during the port calls.

By providing an assignment of containers loaded at a given port to specific slots within bays of a container ship, stowage plans aim to maximise capacity utilisation and minimise operational

[☆]This research has been partially funded by the IT University of Copenhagen and Optivation ApS.
[*]Corresponding author
Email addresses: akor@itu.dk (Aleksandra Korach), berit.dangaard.brouer@optivation.dk (Berit Dangaard Brouer), rmj@itu.dk (Rune Møller Jensen)

costs. One crucial factor in achieving these objectives is the minimisation of *overstowage*, i.e. a situation where a container to be unloaded is blocked by containers destined for later ports. Since stacks can only be accessed from the top, such containers need to be removed before a container underneath them can be retrieved. It is furthermore required that a good stowage plan satisfies a number of high-level objectives, pertaining to seaworthiness of the vessel and crane utilization in ports, as well as low-level objectives, concerned with stacking rules and cargo consolidation. All of these characteristics make generation of feasible and cost-efficient stowage plans a challenging endeavour and an NP-hard combinatorial problem (e.g., [7, 25]).

As noted by [17], it is often overlooked in the literature that the complete stowage process involves not only the shipping company but also the terminal. Typically, stowage plans are prepared by stowage planners of the shipping company in the process known as *stowage planning*. Most stowage plans are made manually with an aid of graphical tools. However, for the past few decades, the liner shipping industry has been confronted with a surge in demand for containerised transport of goods. The demands are being met by providing a higher frequency of service on the shipping routes and through deployment of ever larger vessels, now capable of carrying more than 20,000 TEUs (Twenty-foot Equivalent Units). Plans for such vessels are still generated within mere hours before they call at a port, and under these time constraints stowage planners must be able to evaluate different forecast scenarios. Consequently, stowage planning is gradually becoming too cumbersome for humans, drawing industrial interest in the automation of this process.

The final stowage plan is transferred to the terminal. The terminal planners decide in which sequence to load the containers. They may be allowed to diverge slightly from the plan typically by choosing containers with same *port of discharge* (POD) and type, but sightly different weight. For that reason they may refer to the plan received from the shipping company as the *pre-stow*, while the result of their work, which is send back to the shipping company is referred to as the *departure condition* of the vessel.

State-of-the-art algorithms employed to solve the stowage planning problem faced by the shipping company (e.g., [26, 18, 3]) generally apply a 2-phase approach, emulating the steps followed by stowage planners, as illustrated in Figure 1. Given the arrival stowage condition of the vessel and a list of containers to load in the current and a set of future ports of call, a *master planning* phase initially determines a *master plan* for each port. A master plan distributes containers over

subsections of bays known as *locations* without giving them specific positions. It focuses on high-level objectives such as maximizing crane utilization in the ports and minimizing fuel consumption on long legs. The multi-port outlook of master planning ensures that the master plan of the current port (Master Plan 1 in Figure 1) takes future cargo into account. The *slot planning* phase takes this plan as input and assigns the containers to slots in each location in accordance with the low-level objectives of stowage planning. The result is a complete stowage plan for the current port.

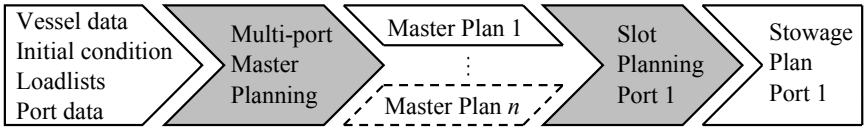


Figure 1: Decomposition of stowage planning into master planning and slot planning phases.

In this article, we build upon the approach presented in [18, 19, 10], focusing on the slot planning phase. Specifically, we will assume that a master plan for the current port which is needed as input for the slot planning phase is available and that it achieves the high-level objectives of stowage planning including vessel stability, operational stress forces, and high crane utilization. Although there exists fast, optimal methods for the slot planning problem [10], in the context of stowing the whole vessel with potentially as many as 100 locations this task needs to be performed as fast as possible, opening the path for even faster, albeit heuristic solutions. We contribute to this cause in this article by presenting an efficient ruin-and-recreate matheuristic using an IP model (as presented in [9]) and based on a heuristic search framework. It builds upon an idea proposed by Prescott-Gagnon et al. [23], where a mathematical model is iteratively solved through branch-and-price in the re-creation phase of Large Neighbourhood Search (LNS). To the best of our knowledge, no attempts at applying such a matheuristic to the problem at hand have been made to date.

Our matheuristic first applies the constructive heuristic introduced in [20]. It can often place all containers assigned to a location without breaking any stowage rules yielding a feasible initial slot plan. If it is unable to find a feasible solution for all containers, leftover containers are placed in free cells causing an infeasible inital slot plan. In each iteration of the LNS, it computes a neighbourhood of solutions by first partially ruining the current slot plan using a battery of destructors and then rebuilding by solving a slot planning IP on the reduced problem. The reconstruction maintains feasibility of feasible solutions and either improves or maintains their quality. We apply

the approach to solve the Container Stowage Problem for Below Deck Locations (*CSPBDL*) introduced in [10], and formulated by the authors in collaboration with a large liner shipping company. It is a representative, although slightly simplified, abstraction of the slot planning problem for bay locations found under deck. To provide a sound evaluation and a ground for comparison of methods, we have used the same set of instances as our predecessors [10], adapted from genuine industrial stowage plans. It is evident from the results, that given a set of instances and a limited time frame, the matheuristic yields more solutions, and of higher quality, than the IP formulation alone when solved by a general-use solver, in a significantly decreased time. For a limit of one second, applying the matheuristic results in an average optimality gap of 4.34% with instances solved within 106.16 seconds in total, as compared to a gap of 29.14% achieved by the general-use solver within 140.77 seconds.

The remainder of this article is organised as follows. Section 2 introduces background information and provides a detailed definition of the problem. It is followed by a literature review in Section 3. Section 4 presents the IP model used by the matheuristic during the re-creation procedure, while Section 5 describes the placement heuristic employed to generate initial solutions. The matheuristic framework is examined in Section 6, and Section 7 explores the designed ruin neighbourhoods. The details of experiments and their results can be found in Section 8, whereas Section 9 discusses conclusions and potential for further development.

2. Background and Problem Statement

A shipping container is a standardised piece of equipment universally used in international trade for transportation of non-bulk commodities, be it by land (trains, trucks) or water (container ships). Made of steel, they mostly come in three standard lengths of 20', 40', or 45' with a width of 8', although longer containers are also in use. Containers can be either 8'6" (*dry-cargo*, DC) or 9'6" high (*high-cube*, HC), HC 20' containers are, however, rare. At a maximum, 20' and 40' containers can have a weight of about 30 tons. Some cargo requires special considerations. *Reefer* containers are intended for goods in need of refrigeration and must be placed near power plugs. *DG* containers hold dangerous items and are bound by strict segregation rules. *Out-of-gauge* (OOG) containers are meant for goods exceeding their standard dimensions, and are usually stored in top of stacks under deck.

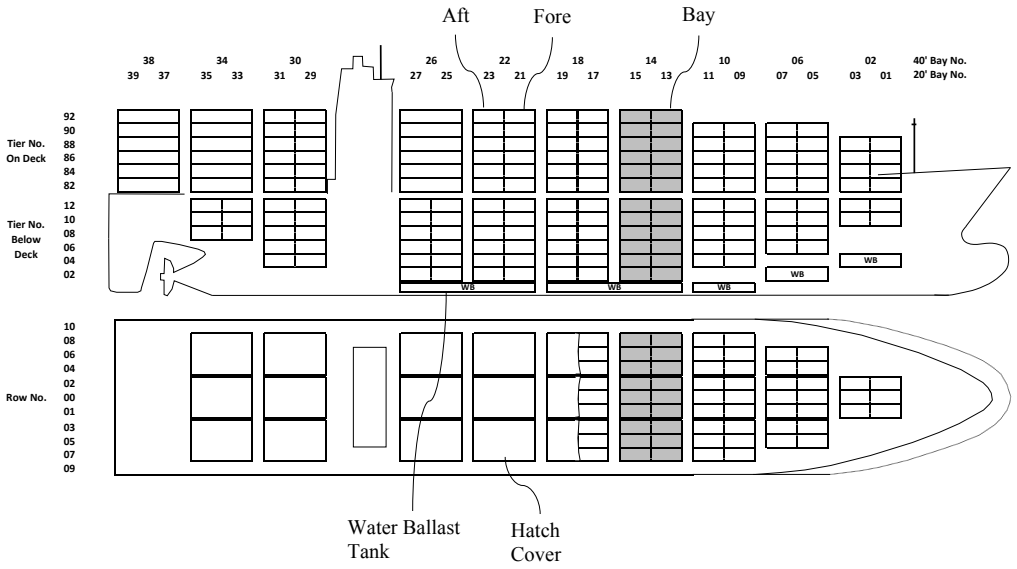
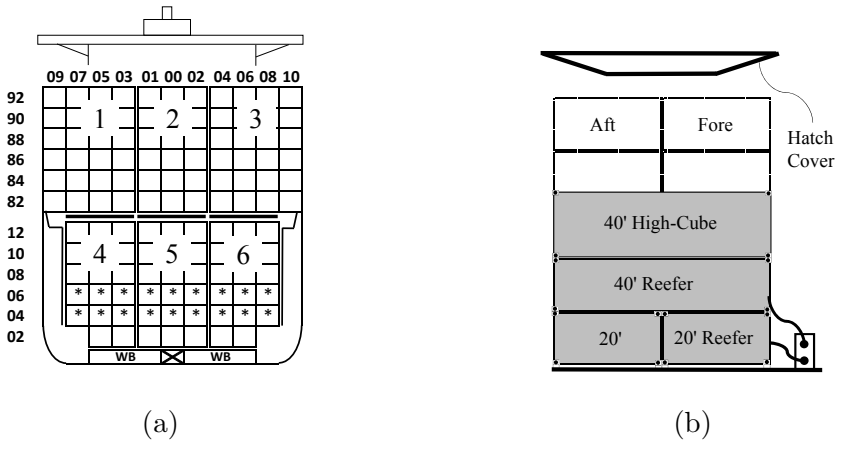


Figure 2: Side and top view of a container vessel with classical bay, tier, and row numbering.

Container vessels are ships tailor-made for transporting containers, and their nominal capacity is given in Twenty-foot Equivalent Units (TEUs). As shown in Figure 2, they are divided into sections called *bays*, which consist of *on-deck* and *below-deck* parts separated by a *hatch cover* – a flat, waterproof barrier. Within bays, containers are organised into numbered *stacks* comprised of *cells* indexed by *tiers*. Each cell may contain either a single 40', 45' container, or two 20' containers, an equivalent of two TEUs. 45' containers are usually kept on deck. Cells are composed of two *slots*, *fore* and *aft*, with the former situated towards the bow, and the latter to the stern of the ship. Traditionally, even numbers are used for 40' cells of a bay while odd numbers are used for its 20' slots. Some slots have access to power plugs to allow for stowage of reefers. Below deck, the safety of containers is guaranteed by *cell guides*, while *lashing rods* and *twist locks* keep them in place on deck.

The high-level seaworthiness requirements of a vessel includes: sufficient line of sight, transversal stability, vessel at even keel, trim and stress forces such as shear force, bending moment, and torsion moment within operational limits. For most weight distributions of cargo, water ballast can be used to fulfil these requirements.

Figure 3a shows the cells of a single bay with three hatch-covers. Each hatch-cover defines two locations one above deck and one below deck, respectively. Hence in this example there are six numbered locations. A star symbol indicates that the cell can hold a reefer container. Figure 3b



(a) (b)

Figure 3: (a) A front view of a bay. (b) A side view of a container stack with power plugs.

shows an example of containers stowed in a stack below deck with five cells. Notice the free space between the top cell and the hatch cover. This extra height may in this example allow one HC container to be stowed in the stack without killing a cell (i.e., without reducing the total number of 40' HC and DC containers that can be stowed in the stack). The containers rest on sockets on the tank top that each have maximum weight limits. A container within a stack rests on the corner fittings of the container below it. For that reason, it is not possible to stack 20' containers on top of 40' and 45' containers.

In ports, quay cranes retrieve containers from the pier and load them into designated bays on the vessel. Since many cranes can work in parallel along the vessel, it is a key high-level objective to spread crane moves along the vessel in each port. Containers are assigned to specific ports they are destined for, referred to as their *port of discharge* (POD). Containers that remain on board, waiting to be unstowed in further ports, are known as *remain-on-board* (ROB) containers, and need to be accounted for when preparing a stowage plan. Should a container bound for a later port be found above a container to be unloaded, the former is said to be *overstowing* the latter. The necessity to unload and restow such containers prolongs the costly crane operation time and should be avoided, thus making overstowage minimisation one of the most important stowage planning objectives. For an in-depth coverage of container vessel stowage, we refer the reader to a recent book on the topic [12].

This article explores the problem of slot planning. The input to a slot planner is a set of

containers to assign to specific slots in a location of the vessel, where some cells may already be occupied by ROB containers. Recall from Figure 3a that a location corresponds to a storage space on or below a hatch-cover. Moreover, we assume that the containers from the load list have been assigned to locations by a preceding master planning phase such that the high-level objectives of stowage planning have been met (i.e., these do not have to be considered further in the slot planning problem). To this end, we utilise the abstraction presented in [10], which was devised in collaboration with a liner shipping company – the *Container Stowage Problem for Below-Deck Locations* (*CSPBDL*). The problem is argued to be NP-hard [10]. As justified by the authors, it is representative of the constraints and objectives present in the real-life problem. Due to high similarity of the problem elements between on-deck and below-deck locations, the presented approach could also be applied to the former likely with similar results [10]. The *CSPBDL* formulation includes constraints pertaining to weight and height limits for stacks as well as stacking rules for 20' and 40' containers. The weight limits are simplified to a maximum total weight of containers in the stack rather than defining the weight limits for each of its bottom sockets. Among the types of containers taken into consideration there are 20' and 40' containers of both DC and HC height. They can also be reefer or non-reefer. ROB containers may be present. A feasible *CSPBDL* solution must stow all the containers assigned to the location and be subject to the following rules [10]:

(i) A container cannot hang in the air. It must have support from below.
(ii) 20' containers cannot be stacked on top of 40' containers.
(iii) A 20' reefer container must be placed in a reefer slot. A 40' reefer container must be placed in a cell with at least one reefer slot.
(iv) The length constraint of a cell must be satisfied.
(v) The sum of heights and weights of the containers stowed in a stack are within the stack limits.
(vi) All ROB containers must be stowed in the slots they occupy in the arrival condition.
(vii) A cell must be either empty or with both slots occupied.

Moreover, an optimal *CSPBDL* slot plan must minimise the sum of the following objectives [10]:

(a) Minimise overstows. A penalty of 1000 units is paid for each container overstowing containers below it in the stack.

(b) Avoid stacks where containers have many different PODs. A penalty of 200 units is paid for each POD present in a stack.

(c) Keep stacks empty if possible. A penalty of 100 units is paid for each used stack.

(d) Avoid loading non-reefer containers into reefer slots. A penalty of 50 units is paid for each non-reefer container stowed in a reefer slot.

Overstowage minimisation is of the greatest importance and hence incurs the highest cost. The objectives (b) through (d) are based on rules of thumb employed by stowage planners. Keeping containers clustered according to their POD and using as few stacks as possible helps increase capacity utilisation and decrease opportunities for overstowage in future ports. Minimising the number of non-reefers stowed in reefer slots allows for loading more reefer containers in later ports [10]. As reported in [10], the penalty costs were consulted with the liner shipping partner. They are applied in the cost function.

3. Literature Review

A systematic classification scheme for stowage planning problems was introduced in [17]. It is a four field notation, where the first field indicates whether it is a carrier or terminal planning problem and the second field characterizes its output. The last two fields detail the setting and objective of the problem.

According to this classification scheme, the *CSPBDL* considered in this article is a carrier planning problem (*line*), that produces a detailed plan (*detl*) for a single bay setting (*1-bay*) using an objective that minimizes the weighted sum of the number of shifts (*oshifts*), the number of stacks with different POD (*mixst*), the number of stacks used to stow containers (*nstack*), and the number of non-reefer containers in reefer slots.

In addition to its relevance for 2-phase stowage planning methods, we focus on *CSPBDL* since our slot planning method then is directly comparable with previous work. The reason is that it is the only stowage planning class that has been studied before using the exact same problem definition and test instances.

There exists two previous studies of *CSPBDL*. The first is the slot planning method introduced in [10]. It is a constraint programming (CP) model of slot planning that can be solved fast to optimality due to the application of advanced modelling techniques such as channelling. The

second is a GRASP algorithm developed by Parreño et al. [20]. It is the best-performing method for the *CSPBDL* to date. We use the constructive heuristic proposed in their paper.

The majority of previous work on carrier stowage planning studies the whole problem of generating a complete stowage plan for a container vessel rather than just the slot planning sub-problem considered in this article and the work described above. These methods must take the high-level objectives of stowage planning into account such as seaworthiness requirements and crane utilization in the current and future ports. For that reason they often are multi-phase approaches like the 2-phase approach (e.g.,[9, 18]) shown Figure 1, where the initial phases solve a multi-port master planning problem that addresses these high-level objectives (i.e., the output of these phases is a multi-port plan (*prot*) according to the classification scheme of [17]).

Many of these decomposition approaches stem from the seminal work by Wilson and Roach [26] that solves a master planning phase for multiple ports with a branch-and-bound algorithm and uses tabu search for slot planning. Zhang et al. [28] present a 2-phase approach with a Bin-Packing heuristic in the first one, and tabu search in the second. Min et al. [16] describes a stowage planning system consisting of three modules: a stowage plan generator, a stability module, and an optimisation engine. In their study, Ambrosino et al. [1, 2, 3] present variations on a 3-phase approach, combining a tailored heuristic for the first stage, an IP model or a constructive heuristic for the second, and either an iterative swapping heuristic or tabu search for the third phase. The work of Pacino et al. [18] puts forward IP models for master planning, with slot planning solved by local search approaches. Li [15] attempted a 2-phase approach utilising an IP model and tabu search. Other contributions include work by Ning et al. [27] who distinguish five phases in their formulation; Lee et al. [13] introduce a 2-phase heuristic with objectives changing between phases.

In addition, to multi-phase approaches to generate complete stowage plans, there also is a large body of work using a single optimization model to solve the problem. Ambrosino and Sciomachen [4] developed a CP model for the whole Master Bay Plan Problem. Botter and Brinati [8] and Li et al. [14] present 0-1 IP models. Many studies employ metaheuristics such as genetic algorithms or tabu search to tackle the problem, often in tandem with other heuristic approaches. Examples include an ant colony algorithm by Hernández et al. [11], a genetic algorithm by Zhang and Lee [29] or pareto clustering search by Araújo et al. [5]. Finally, some studies attempt to solve the stowage planning problem based on its connection to the 3D Bin Packing Problem, most notably

Sciomachen and Tanfani [24], where the relationship between the two is discussed.

Matheuristics are optimisation methods integrating mathematical programming (MP) techniques into a heuristic framework. To date they have been successfully applied to a variety of problems, including vehicle routing problems, as presented in the survey by Archetti and Speranza [6]. The survey also identifies various matheuristic paradigms. *Decomposition approaches*, may divide the problem into parts, some of which may be solved with MP methods, or use MP to optimise a subset of objectives. *Improvement heuristics* include *one-shot methods*, where MP models are used to improve solutions found by other heuristics, as well as *approaches with MILP models for local optimisation*. The latter type embeds MP techniques as operators within a search algorithm, which can be used to explore a neighbourhood, as intensification tools, or as operators to complete a partial solution. Finally, some matheuristic approaches use *branch-and-price/column generation* procedures to solve MILP models. The MIP model by Ambrosino et al. [3] is iteratively solved with a 2-step relax-and-fix heuristic, making it the only matheuristic in this survey.

4. Integer Programming Model

Below we present the IP model for the *CSPBDL* problem, as introduced in [10, 9]. Although the considered objectives and constraints are the same as those presented in these studies, our application is different. We use the IP model in our matheuristic to rebuild solutions by assigning containers to parts of a solution unfixed by our destructors. This approach has a two key advantages. First, since the destructors usually only unfix a small fraction of the slots, the average time needed to find an optimal solution to this partial problem by an IP solver is much smaller than the average time it needs to solve complete *CSPBDL* problems. Second, since the IP model re-assigns containers optimally, the reconstruction maintains feasibility and leads to solutions with lower or equal cost. Notice, however, that such hill climbing steps are only guaranteed if the parent solution is feasible. For instance, if the parent solution is infeasible due to some containers violating stowage rules and the destructors do not unfix all these containers, the rebuild solution will also be infeasible. [1]

Table 1 contains the explanation of symbols for sets and constants, as well as variables used

[1] Even if unfixing all infeasibly placed containers, the fixed part of the partial solution may not allow a feasible assignment of all unfixed containers into the available open slots.

	Sets		***Constants***
Q	Containers to stow.	R_i^C	Indicates whether container i is reefer.
$Q^{\{20,40\}}$	20' and 40' containers in Q.	C^r	Penalty for stowing a non-reefer container in a reefer slot.
M	ROB containers.	C^p	Penalty for each POD used in a stack.
J	Stacks.	C^v	Penalty for a container overstowing containers below.
K_j	Cells in stack j.	C^u	Penalty for the use of a stack.
P	Discharge ports.	A_{ip}	Indicates whether container i is discharged at port p. If so, the value is 1; 0 otherwise.

	Constants		***Variables***
H_i^C	Height of container i in metres.	c_{jki}	Container i is stowed in cell k, stack j.
H_j^S	Height of stack j in metres.	o_i	Container i is overstowing other containers.
W_i^C	Weight of container i in tons.	d_{jp}	At least one container in stack j is discharged at port p.
W_j^S	Weight capacity of stack j in tons.	e_j	Stack j is in use (i.e., stack j is non-empty).
R_{jk}	Number of reefer plugs in cell k of stack j.	δ_{jkp}	Container below cell k, stack j is discharged before port p.

Table 1: Sets, constants, and variables used in the IP model.

in the model. All variables are binary. Stacks j in a given location are numbered from 1 to $|J|$, and each stack consists of $|K_j|$ cells k, numbered from 1. Similarly, containers i are also numbered from 1, and PODs p are numbered from 1 to $|P|$, ordered from the earliest to the latest. Penalties in the constants table take the values specified in the problem objectives (a) through (d). The c variables are decision variables representing the slot plan, while the auxiliary variables are utilised for calculating the objective components. Based on these elements, the following IP model is defined:

$$\begin{aligned} \min \quad & C^v \sum_{i \in Q} o_i + C^p \sum_{j \in J} \sum_{p \in P - \{1\}} d_{jp} + C^u \sum_{j \in J} e_j \\ & + C^r \sum_{j \in J} \sum_{k \in K_j} (R_{jk} \sum_{i \in Q^{40}} c_{jki}(1 - R_i^C) + \sum_{i \in Q^{20}} c_{jki}(\frac{1}{2} R_{jk} - R_i^C)) \end{aligned} \tag{1}$$

s.t.

$$\frac{1}{2} \sum_{i \in Q^{20}} c_{j(k-1)i} + \sum_{i \in Q^{40}} c_{j(k-1)i} - \sum_{i \in Q^{40}} c_{jki} \geq 0 \quad \forall j \in J, \;\; k \in K_j - \{1\} \tag{2}$$

$$\sum_{i \in Q^{20}} c_{jki} - \sum_{i \in Q^{20}} c_{j(k-1)i} \leq 0 \qquad \forall j \in J, \;\; k \in K_j - \{1\} \tag{3}$$

$$\frac{1}{2}\sum_{i\in Q^{20}} c_{jki} + \sum_{i\in Q^{40}} c_{jki} \leq 1 \qquad \forall j \in J,\ k \in K_j \tag{4}$$

$$\sum_{j\in J}\sum_{k\in K_j} c_{jki} = 1 \qquad \forall i \in Q \tag{5}$$

$$\sum_{i'\in Q^{20}} c_{jki'} - 2c_{jki} \geq 0 \qquad \forall j \in J,\ k \in K_j,\ i \in Q^{20} \tag{6}$$

$$\sum_{i\in Q} R_i^C c_{jki} - R_{jk} \leq 0 \qquad \forall j \in J,\ k \in K_j \tag{7}$$

$$\sum_{k\in K_j}\sum_{i\in Q} W_i^C c_{jki} \leq W_j^S \qquad \forall j \in J \tag{8}$$

$$\sum_{k\in K_j}(\frac{1}{2}\sum_{i\in Q^{20}} H_i^C c_{jki} + \sum_{i\in Q^{40}} H_i^C c_{jki}) \leq H_j^S \qquad \forall j \in J \tag{9}$$

$$\sum_{k'=1}^{k-1}\sum_{p'=2}^{p-1}\sum_{i\in Q} A_{ip'} c_{jk'i} - 2(k-1)\delta_{jkp} \leq 0 \qquad \forall j \in J,\ k \in K_j - \{1\},\ p \in P - \{1,2\} \tag{10}$$

$$A_{ip}c_{jki} + \delta_{jkp} - o_i \leq 1 \qquad \forall j \in J,\ k \in K_j,\ p \in P,\ i \in Q \tag{11}$$

$$e_j - c_{jki} \geq 0 \qquad \forall j \in J,\ k \in K_j\ i \in Q \tag{12}$$

$$d_{jp} - A_{ip}c_{jki} \geq 0 \qquad \forall j \in J,\ k \in K_j,\ p \in P,\ i \in Q \tag{13}$$

$$c_{jki} = 1 \qquad \forall (j,k,i) \in M \tag{14}$$

The objective function is defined as a weighted sum of the *CSPBDL* problem objectives, with the o_i, d_{jp} and e_j variables summed and multiplied by their respective penalties. As for the reefer objective, the number of stowed non-reefer containers is calculated for each cell with reefer slots. Seeing as 40' containers require only a single reefer slot per cell, they need to be considered independent of 20' containers. Constraint (2) guarantees that 40' containers do not hang in the air (i), demanding either two 20' or one 40' container be present below a cell holding a 40' container, with the exception of the bottom cell. Similarly, inequality (3) ensures that two 20' containers are always present beneath a cell with 20' containers (ii). The fact that no more than two 20' or one 40' container can be stowed in a cell is enforced by constraint (4), while inequality (5) requires all containers to be stowed in exactly one cell. Since a cell must always be either fully occupied or empty (vii), constraint (6) demands the number of 20' containers in a cell to be either 0 or 2. Reefer containers are forced to be stowed in slots with reefer plugs (iii) by constraint (7). The weight

and height stack limits (v) are constrained by inequalities (8) and (9) respectively. In constraint (10), the δ_{jkp} variables are defined, and then used in inequality (11) to assign correct values to the overstowage variables o_i. Variables e_j and d_{jp} require lower bounds to be set for them, which is accomplished by constraints (12) and (13) in the order given. Finally, constraint (14) assigns ROB containers to their original positions (vi).

Additionally, the model comprises linear cuts to remove non-integer values assigned to δ variables from consideration, which is attained through the use of a Linear Programming relaxation [10]:

$$A_{ip'}c_{jk'i} \leq \delta_{jkp} \qquad \forall j \in J,\ k \in K_j,\ p \in P,\ i \in Q,\ k' \in K',\ p' \in P' \tag{15}$$

$$\frac{1}{2}\sum_{i \in Q_p'^{20}} c_{jk'i} \leq \delta_{jkp} \quad \forall j \in J,\ k \in K_j,\ p \in P,\ k' \in K' \tag{16}$$

$$\sum_{i \in Q_p'^{40}} c_{jk'i} \leq \delta_{jkp} \qquad \forall j \in J,\ k \in K_j,\ p \in P,\ k' \in K' \tag{17}$$

$$\sum_{k'=1}^{k-1} A_{ip'}c_{jk'i} \leq \delta_{jkp} \quad \forall j \in J,\ k \in K_j,\ i \in Q,\ p \in P,\ p' \in P' \tag{18}$$

where $K' = \{k' | k' \in \{1, \ldots, k-1\}\}$, $P' = \{p' | p' \in \{2, \ldots, p-1\}\}$, and $Q_p'^{20}$ with $Q_p'^{40}$ describing the sets of, accordingly, 20' and 40' containers with POD earlier than p. Constraint (10) is decomposed into a semantically equivalent set of inequalities (15). It is then expanded in cuts (16) through (18), by including considerations of containers unloaded before port p, separately for 20' (16) and 40' (17) containers, as well as all cells below cell k (18).

5. Constructive Heuristic

Recall that the presented matheuristic approach comprises an LNS algorithm, where in every iteration solutions are partially ruined and rebuilt using a general-use mathematical solver. The search algorithm requires an initial solution as input that needs to be generated fast. For this purpose we employ a constructive heuristic described by Parreño et al. [20]. Below we present the details of this algorithm.

The heuristic follows an iterative process consisting of three steps preceded by an initialisation step. It takes the following elements as input: a list of ROB containers C_R for the given instance, a list C of containers to stow and a list S of stacks. Additionally, each stack $s \in S$ is associated with a list K_s of cells it consists of. The output of the constructive heuristic is an initial stowage

plan. In each iteration, the heuristic selects a stack from S, and considers its lowest empty cell. Next, a container is chosen from among the C list, so that its assignment to the selected stack and cell is feasible. When there are no more containers to stow or the remaining containers cannot be feasibly stowed in the available cells, the process is terminated. Finally, the left-over containers are placed in the remaining empty cells.

During initialisation, ROB containers are placed into their positions, and the other aforementioned lists are sorted. Stacks in S are ordered first according to whether they are empty, in which case they appear last. Then they are sorted by the number of cells available for loading, in non-increasing order with empty stacks appearing last. The rationale is to minimise the number of stacks in use. As for containers, an ordering is imposed based on their port of discharge, length, whether they are reefers, and their height class. Containers with later PODs are considered first to avoid overstowage. With regards to length, 20' containers appear first as they cannot be stowed on 40' containers. Then, reefers are considered before others, and finally HC containers before DCs.

In the first step of the iterative process, the heuristic chooses the appropriate stack and cell. Initially, the first stack from S is selected. Then, if the stack is filled or there are currently no containers that can be stowed in it, the algorithm moves on to the next stack. The stacks are revisited in the same order, considering that, as containers are loaded, some limits preventing other containers from being stowed in a given stack may be lifted, now permitting their placement (see conditions 2 and 4 below).

The second step manages the choice of containers for stowing in a selected cell. In order for a container to be legally stowed in the cell, it must satisfy a few conditions:

1. **Weight Limit**: The total weight of the stack including the weight of candidate containers cannot exceed its weight limit.
2. **Height Limit**: The total height of the stack including the height of candidate containers cannot exceed its height limit. Furthermore, special rules apply when the candidate is an HC container. If assigning the candidate to the given cell reduces the number of DC containers that could otherwise be stowed in this stack, it cannot be loaded. However, the condition is only applicable if there are more standard-height containers with the same POD as the HC container, as it is more important to limit the number of PODs in a stack than to optimally utilise stack volume.

3. **Length Class**: Some cells only accept containers of certain length. Length of a container must agree with cell restrictions. Secondly, 20' containers cannot be stowed in the selected cell if there is a 40' container underneath it. Moreover, a 20' container can only be loaded into the cell if there is another 20' container in C that can also be feasibly stowed in that cell.
4. **Power Plugs**: Depending on the number of reefer slots in the cell,

   0 reefer slots: the candidate must be either a 40' or a 20' non-reefer container; in the latter case a matching *non-reefer* container must be found.

   1 reefer slot: the candidate must be either a 40' or a 20' reefer; in the latter case a matching *non-reefer* container must be found.

   2 reefer slots: the candidate must be either a 40' or a 20' reefer; in the latter case, a matching *reefer* container must be found.

   Nevertheless, if there are no more reefer containers in C, non-reefers can also be loaded.

In the last step, the algorithm updates the list states. The container(s) selected in the previous step are removed from C and the values of stowed stack weights and heights are altered as required.

As mentioned above, after the iterative process is terminated, the remaining containers are placed in the first empty cells found, regardless of other constraints, and marked as violations. In consequence, the resulting solution may be infeasible.

6. Matheuristic

While designing our approach, we took example from the study by Prescott-Gagnon et al. [23], to use an improvement matheuristic which utilises an IP model for search optimisation, with LNS [22] as the metaheuristic framework. The reason is that LNS search methods lend themselves very well to complex problems with structure akin to that of slot planning. LNS follows a two-step 'ruin-and-recreate' procedure, where parts of a solution x are iteratively destroyed (unfixed, ruined) and then rebuilt, implicitly defining its neighbourhood, $N(x)$. Ruining and reconstruction is accomplished by two sets of destroy Ω^- methods (*destructors, destroyers, ruiners*) and rebuild Ω^+ methods. By substituting the reconstructive methods with an IP model over the unfixed parts and solving it to optimality with a general-purpose IP solver, we turn the LNS into a matheuristic. An advantage of this reconstruction is that it optimizes all objectives at the same time and for a

feasible x generates a solution that is at least as good as x. Below we discuss the overall design of the algorithm.

6.1. Pre-search Initialisation

As a first step, the method creates an IP model of the given instance. It is a one-time procedure, and the same model is later reused within search. To provide a basis for improvements during search, the fast constructive heuristic presented in Section 5 is used to generate an initial solution. The cost of that solution is then calculated according to the model's cost function $c(x)$, as required by the search algorithm for comparison. The obtained solution is subsequently presented as input to the search method.

6.2. Matheuristic Large Neighbourhood Search

Given that the re-creation step could never produce a solution of worse quality than the destroyed one, the LNS framework can be viewed as a hill climber with complex neighbourhoods. The presented approach is an any-time method that is terminated after a time limit is reached. An iteration limit and a non-improving iteration limit further restrict the algorithm's runtime. In the course of each iteration, we first consider the feasibility of the input solution x. If the solution is infeasible, all containers that violate constraints are unstowed. Next, the method randomly selects a destructor from Ω^-, which then commences to unload whole stacks (or their parts) based on its underlying selection principle. How much is destroyed depends on the size of the input instance. Nevertheless, it should be noted that if many violations were present, a considerable part of the solution may have to be rebuilt. Thus, the matheuristic is strongly reliant on the quality of the constructive method. By adding auxiliary constraints to the model, the remaining containers are set as fixed, and re-creation is performed using an IP solver, producing a neighbouring solution x'. The IP problem solved at each iteration is a reduced problem from the original IP problem, as the set of decision variables contains only the variables unfixed by the destructor. As a result the reduced IP is often solved very efficiently. The advantage of using an IP solver to rebuild the solution is that the IP considers all constraints along with the objective for the set of unfixed decision variables. If the repaired solution is infeasible, the matheuristic will continue to unstow additional stacks to try to allow a feasible solution to be constructed.[2]

[2]In practice, if the final solution is infeasible, the objective of the last iteration can be changed to stow as many containers as possible and roll the remaining containers. Rolling containers has been applied in previous work (e.g.

After reconstruction, the auxiliary constraints are removed, and the solution quality is evaluated. If $c(x') \leq c(x)$ the update $x = x'$ is performed, and x' marked as the best solution x^b if the cost of x' is strictly lower. The process continues until a given time limit, an iteration limit or a non-improving iteration limit is reached. During the course of search the remaining time is constantly updated, so that the solver does not run overdue.

To restrict the degree of destruction, all destroyers are allowed to unfix a random number of stacks between a minimum of 2 and a maximum of 50% of all eligible stacks (the number of stacks in the instances varies from 2 to 10). If the number of stacks is 3, two stacks are destroyed. A stack is eligible for destruction, if it contains non-ROB containers. Preliminary tests revealed that a maximum of 50% is a good compromise for the trade-off between destruction size and search time. Setting the limit to 25% resulted in a slight speed gain and a drop in performance, while setting it to 75% caused longer running times without a clear gain in solution quality.

7. Destructors

Each ruin method tries to address specific issues within the current solution, attempting to minimise relevant elements of the objective function. For the purposes of the matheuristic, several destructors were designed and are presented in this section.

In an article about adaptive large neighbourhood search [21], Pisinger and Ropke distinguish a few destroyer types: random-removal, worst-removal and related-removal. When applied to our problem, *Random-removal* methods select stacks at random. *Worst-removal* ruiners choose stacks that are the worst according to some criterion, e.g. have the most killed space. Finally, *related-removal* destructors define a relationship between stacks, and unfix the ones that are related. We have designed and tested 2 random-removal, 3 worst-removal and 4 related-removal methods.

7.1. Destructors related to the 'keep stacks empty' objective, (c)

Starting with the most general objective, these ruin methods aim at maximising volume utilisation within stacks to accommodate as many containers as possible or necessary. The first three are general-use, applicable to all instances.

[10]) and may be necessary as instances from the master planner can be infeasible or take too long to solve (i.e., by running the IP solver on the whole problem).

Worst-removal: Most killed space. As the name indicates, this algorithm chooses to destroy the stacks where volume (height) is poorly utilised - the most space is killed. The stacks are sorted in descending order according to the difference between their maximum height and their current height. Thus, partially filled stacks are selected first, potentially allowing them to be consolidated.

Related-removal: Volume vs. weight. The following heuristic juxtaposes heavy stacks with space for more containers but limited by weight, with the ones that are already volume-efficient, but could accept heavier containers instead. This is accomplished by sorting. Afterwards, stacks are selected by alternating between both head and tail of the list.

Stacks are sorted with respect to their *current volume/weight ratio*, which is calculated by dividing the current weight of a stack by its current height. The *perfect ratio* is defined by dividing the maximum weight by the maximum height, and describes the ratio that a stack would have if it was both volume- and weight-efficient. The algorithm orders the stacks according to the difference between these two ratios, ascending. A positive value indicates that so far a stack has accepted more weight for its current volume than the perfect ratio would suggest, while a negative value signifies the opposite. Therefore, at the head of the sorted list of stacks we would have stacks skewed to the heavier side, and at the tail - to the lighter.

Seeing as the ratio calculation is irrespective of the number of containers loaded, it is evident that partially filled stacks can also be chosen, and thus possibly consolidated.

Related-removal: Volume vs. weight, version 2. The only difference between this ruin method and the previous one is the sorting principle. Stacks are sorted first by height in a descending order, then by weight in an ascending order. As this sorting includes stacks with low volume (height) utilisation, it also allows partially filled stacks to be selected.

Related-removal: HC/DC. This destructor investigates the distribution of HC vs. DC containers within a solution, and is applicable if both HC and DC containers are present in the cargo mix. Its behaviour varies based on which type is dominant. As with the previous destroyer, the algorithm sorts the stacks depending on some criterion, and then selects stacks from both ends.

Before proceeding, it is necessary to introduce a few concepts. **Free HC capacity** is the number of containers a stack can accept without the necessity to *kill* any DCs, i.e. waste space that could have been taken by an additional DC container. **Extra HC capacity** are additional

HC containers that can be stowed after killing one extra DC space. **Current HC capacity** is the number of HC containers that can still be loaded without killing any *more* DC containers. A **perfect HC/DC cargo mix** describes a situation, where the cargo contains a number of HC containers equal to the sum of free HC capacities of all stacks within a location and the rest of containers are DC (see Figure 4). This allows for a maximum volume utilisation.

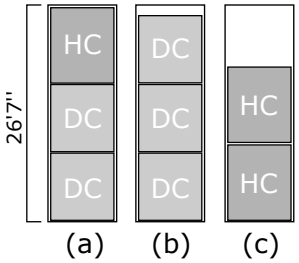


Figure 4: Illustration of perfect and imperfect HC/DC mix in stacks. Stack *(a)* holds 2 DCs and 1 HC container, a total 26'6" of height. It fully utilises its free HC capacity, and no space is wasted. Stack *(b)* does not, and thus some space is left unused. 2 HCs in stack *(c)* exceed its free HC capacity, causing one DC space to be killed. It follows, that an ideal HC/DC mix for these stacks is 3, although one HC from stack *(c)* should be swapped with a DC in *(b)*, allowing for an additional DC container to be stowed in *(c)*.

We consider three situations:

1. The cargo contains an ideal mix of HCs and DCs, i.e. the stacks can accept all of the HCs without the necessity to kill any DCs - their free HC capacity is enough.
2. The cargo has fewer HC containers than the stacks could accept without killing any DCs; as a result, some stacks won't need to stow any HCs - their free HC capacity is more than enough.
3. The cargo has more HCs than can be stowed without killing DCs; some have to be killed to accommodate for extra HCs - the free HC capacity of stacks is not enough.

For the perfect mix, the aim is to ensure the stacks contain exactly the number of HCs they can fit so that no space is killed. The idea in the second case is to put the HCs in as few stacks as possible. These two situations can be dealt with together, since the goal of both is essentially the same: ensuring full utilisation of the free HC capacity in stacks by moving containers from stacks exceeding this capacity or not utilising it fully. It can be accomplished by imposing the same ordering on the stacks. For each stack, we calculate the difference between their capacity to stow HCs freely and the number of currently stowed HCs. We use the result as a sorting principle.

Last but not least, if there are more HCs in the mix than the perfect number, a sound intention is to minimise the number of stacks with killed DC space, trying to fully utilise their HC capacity.

Stacks where no DC space is killed and that cannot accept more HCs without killing DCs are placed in a separate list. The algorithm sorts the remaining stacks according to 3 criteria. First, it checks whether there are any stacks with fully utilised HC capacity and DC kills. If so, they are pushed to the end. Next, the heuristic sorts the stacks by the quantity of killed DCs (ascending), and by their current HC capacity (ascending). This way stacks with almost perfectly utilised HC capacity are juxtaposed with stacks that have most killed DC containers and poorly utilised HC capacity. Removing an HC container from the latter can potentially result in restoring DC space.

We only select stacks from the auxiliary list if the algorithm chooses to destroy more stacks than available in the primary list. The order in the list is random.

7.2. Destructors related to the 'overstowage' (a) and 'multiple POD' (b) objectives

These destructors apply only to instances with more than one POD.

Worst-removal: Number of overstowing containers. The destructor unstows stacks with the highest quantity of containers that overstow others. It may be useful for situations where there is more than a single stack with overstowing containers. The order of stacks with the same number of overstowing containers is randomised.

Worst-removal: Stacks with multiple PODs. The algorithm unfixes the stacks with most varied PODs, as it might be possible to purify some stacks or at least reduce the number of different PODs they contain. In case of many stacks with the same number of PODs, we resolve the conflict by considering the stacks with the most significantly dominant POD first. The rationale is, that if there is a dominant POD in a stack, the algorithm may destroy it along with another one where a different POD is dominant, potentially allowing both, or one of them, to be purified by getting rid of the less numerous PODs.

7.3. Destructor related to the 'reefer' objective, (d)

Related-removal: This destroyer is only relevant in situations where there are more reefer slots than reefer containers.

In some cases, we might want to fully utilise reefer plugs of a stack instead of having reefers distributed unevenly between stacks. This could allow non-reefer containers to be stowed above the tiers with reefer plugs, imposing no penalty, moving them from other stacks and potentially saving a stack. The destruction is performed so that stacks with most, but not all, reefer slots

taken are juxtaposed with those, where reefer plugs are so far poorly utilised. This is done by sorting the stacks according to a ratio of plugs taken to all plugs in a stack. Stacks with no reefer slots or all reefer slots in use are pushed to an auxiliary list.

Just like with the HC vs. DC destroyer, if the algorithm chooses to destroy more stacks than present in the main list, the remaining stacks are chosen from the separate list. The ordering of these stacks is random.

7.4. Random destroyers

Full-Stack Random. The destroyer randomly chooses a set of eligible stacks to unfix.

Two stacks with partial destruction. This destructor ruins only two stacks at a time. If the number of stacks eligible for unstowing equals 2, random halves (upper or lower) of both stacks are unfixed.

8. Experiments

All experiments were performed on a Windows 10 machine with a 2-core 2.0 GHz Intel Core i7-4510U processor and 8.0 GB RAM. The matheuristic was implemented in C++, with the IP model implemented in Gurobi 6.5.1. It was evaluated on a set of 236 instances, each representing a location with a load list generated by unstowing a real stowage plan. These are the same instances as the ones used in [10]. Since the instances are derived from real stowage plans, they are feasible and have realistic cargo mixes in terms of container types and number of different PODs. In the intended application of our matheuristic, however, it is the second phase of a 2-phase decomposition of the stowage planning problem. The first phase is a master planner that produces the slot planning problems solved by the matheuristic. It is reasonable to assume that a master planner generates instances with cargo mixes similar to real plans, but it makes little sense to assume that it guarantees feasible instances as this in general entails solving the slot planning problem itself. For that reason when combined with a master planner, the matheuristic can be changed to roll containers in the last iteration to ensure feasibility as described in the footnote of Section 6.2.

To facilitate a comparison with the results of previous studies, the instances were grouped into classes according to their features, as presented in [10] and [20]. Table 2 summarises the groups, giving details about the number of instances, the TEU-capacity of locations, the number

of containers to stow, and the presence of certain features - 20', 40', reefer and high-cube containers, as well as the number of instances with either 1, 2, or 3 and more PODs. Notice that plans seldom have more than 3 PODs per location. This is natural as POD mixing reduces crane efficiency and makes the plans less robust to uncertain future cargo [12]. Moreover, since the number of hatch-covers grow with the size of the bay, a single location seldom holds more than 200 TEU even on large 20,000 TEU vessels. [3]

Class	#	Cap. (TEUs)			#Cont. (TEUs)			40's	20's	R	HC	#POD		
		min	avg	max	min	avg	max					1	2	≥ 3
1	13	16	63	116	8	54	116	×				13		
2	22	8	68	168	8	52	136		×			22		
3	13	30	74	124	8	68	124	×	×			13		
4	78	6	79	208	2	63	202	×			×	78		
5	36	38	97	176	8	81	170	×	×		×	36		
6	15	42	73	172	16	46	74	×		×		15		
7	14	72	147	204	24	117	202	×	×	×	×	14		
8	14	40	96	148	40	87	136	×		×	×		14	
9	17	44	124	220	36	111	200	×	×	×	×		15	2
10	8	72	122	176	10	93	156	×		×	×		6	2
11	6	48	101	176	28	84	148	×	×	×	×		3	3

Table 2: Instance classes.

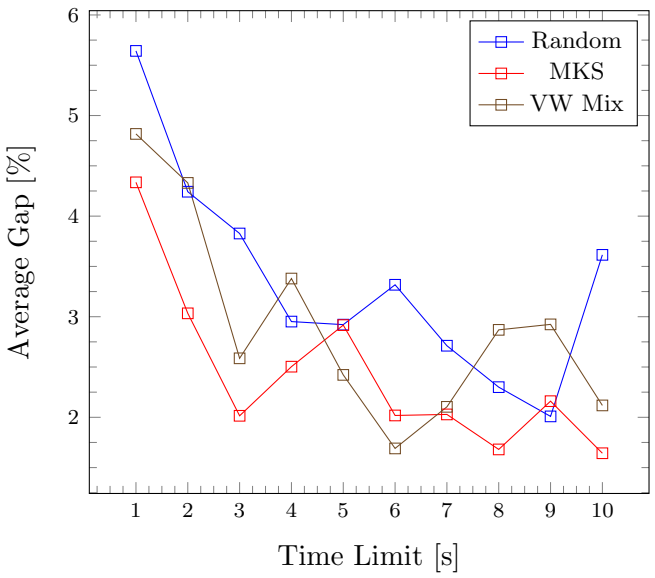
A series of tests with increasing time limits was performed on the algorithm, measuring time performance, the average gap, the percentage of solutions found, and the number of instances solved to optimality. Time limit was gradually increased from 1 to 10 seconds, with 1-second steps. A maximum of 15 iterations and 5 non-improving iterations were chosen as additional stopping criteria.

8.1. Comparison of different ruin methods

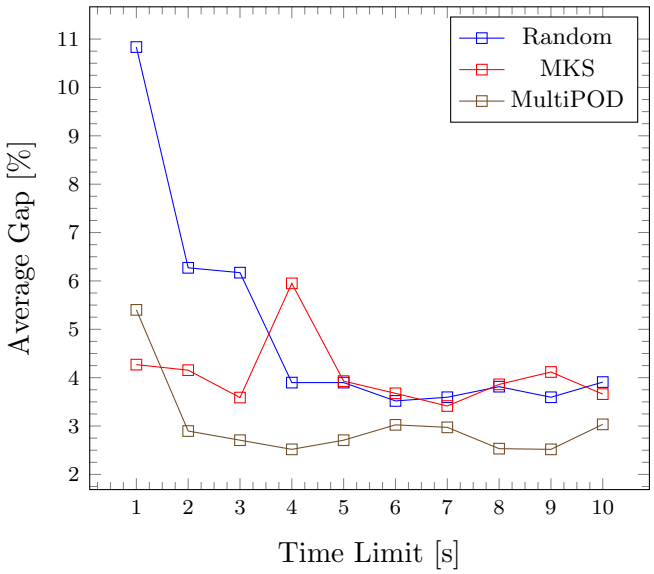
We first examine the performance of different destructors and their mixes presented in Section 7, considering the general and feature-specific methods separately.

To begin with, let us focus on the general destroyers, which include the Random and Two-Stack Random (*2-Random*) methods as well as the three general-use destructors related to objective (c):

[3] The instance set from [10] has relatively large locations due to its industrial origin, where outboard side locations are pairwise merged.



(a) Performance of general destructors, Random, MKS and VW Mix, on all 236 instances.



(b) Peformance of Random, MKS and MultiPOD destructors on 45 instances with multiple PODs.

Figure 5: Performance of general and multiple POD destructors.

Most Killed Space (*MKS*) and the ratio and non-ratio version of Volume vs. Weight (*VW* and *VWR*) methods. In addition, two mixed sets were tested, containing 3 destructors each. The first mix pool included the VWR, MKS and Random ruiners (VWR Mix), while the other one contained VW instead of VWR (VW Mix). The results show, that neither of them is clearly better than the other. Table 3 presents the results for all of the variants, while Figure 5a shows a performance graph of the best random, the best non-random destroyer and the VW mix of destroyers for the range of 1 - 10s time limits. Notice that we in this and several subsequent tables use bold font for entries with the lowest time and gap for the corresponding row.

Although somewhat slower, the Random destructor consistently achieved a lower, albeit slightly, average gap for nearly all time limits compared to the Two-Stack Random method. Specifically, for the one-second limit it yields a gap lower by 0.8 percentage points. All non-random general ruiners displayed a similar time performance to the full-stack random, with Most Killed Space and Ratio Volume/Weight producing solutions of moderately better quality than Random (on average, an impovement of 0.92 and 0.72 percentage points respectively). MKS is used for further comparisons in this study. The other VW destructor did not perform better than the random. The mixes also

took a comparable amount of time to solve the instances, although did not show any improvement in quality over the random.

Limit(s)	Random		2-Random		MKS		VWR		VW		VWR Mix		VW Mix	
	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time
1	5.64	108.15	6.46	**84.65**	4.34	106.16	**4.28**	107.76	4.84	103.68	4.43	105.97	4.82	105.54
2	4.24	157.90	4.04	**119.03**	**3.03**	154.76	3.07	163.17	4.37	155.67	4.19	157.61	4.33	157.00
3	3.83	193.80	4.47	**137.90**	**2.01**	197.57	2.55	205.16	2.54	192.69	3.11	202.72	2.59	196.24
4	2.95	223.87	3.51	**157.57**	**2.50**	234.52	3.41	231.69	3.35	223.12	2.55	229.94	3.38	230.82
5	2.92	253.04	3.86	**180.56**	2.92	245.28	**2.21**	243.35	3.37	253.36	2.88	268.57	2.42	255.77
6	3.32	279.72	5.25	**198.11**	2.02	265.43	2.14	281.87	2.92	255.40	3.34	266.35	**1.69**	261.86
7	2.71	288.87	4.37	**211.47**	2.03	293.77	**1.77**	298.41	2.51	284.12	2.51	307.40	2.10	291.87
8	2.30	297.65	3.82	**213.52**	**1.68**	335.19	2.63	302.18	2.57	286.62	2.55	309.53	2.87	330.04
9	**2.01**	317.20	3.21	**207.66**	2.16	317.33	2.16	317.11	2.99	315.63	2.37	307.16	2.92	316.20
10	3.62	315.23	3.67	**235.59**	**1.64**	330.57	2.12	336.58	2.52	316.70	2.01	324.09	2.12	306.46

Table 3: Performance of the general-use destroyers and mixed destroyer sets on 236 instances. Gap is given in (%) and time in (s).

Table 4 shows the performance of the HC/DC ruin method and a single HC destroyer mix against the random and the MKS destructors. The mixed set is comprised of the HC destroyer, the Random, and the MKS (due to its marginally better performance than VWR). They were tested on all instances that include both DC and HC containers, in a number of 146 out of 236. Again, in both cases the time taken is similar to the other ruin methods. However, neither the single destructor nor the mix achieved better results than the random, which indicates the HC/DC destroyer is not a beneficial ruin method for instances with a mix of both HC and DC containers.

Limit(s)	HC/DC		HC/DC Mix		MKS		Random	
	Gap	Time	Gap	Time	Gap	Time	Gap	Time
1	7.27	75.26	5.90	**74.96**	**5.51**	76.76	6.92	77.99
2	5.09	121.59	4.12	118.36	**3.39**	**114.41**	5.21	119.06
3	5.20	147.91	4.47	151.81	**2.50**	**146.61**	4.68	146.66
4	4.31	**170.97**	**3.20**	175.05	3.26	180.71	3.95	174.36
5	4.39	202.59	3.89	**182.85**	3.93	184.82	**3.90**	185.70
6	4.24	211.20	2.53	205.93	**2.48**	**204.73**	3.85	206.90
7	3.11	241.93	3.22	**209.45**	**2.47**	216.23	3.56	229.81
8	2.84	**220.09**	3.47	252.10	**1.83**	260.39	2.90	228.55
9	3.70	251.66	2.99	263.01	2.58	**246.70**	**2.43**	255.62
10	2.81	274.94	2.66	245.51	**1.90**	251.18	4.38	**240.95**

Table 4: Performance of the HC/DC destroyers and mixed destroyer sets on 146 instances with both HC and DC containers. Gap is given in (%) and time in (s).

As for the instances with multiple PODs (45 out of 236, spanning the last four groups), the two dedicated destroyers, 'no. of overstowing containers' (Overstows) and 'stacks with multiple PODs' (MultiplePOD), were tested along with two mixed sets. Both mixes include the random and the MKS ruiners, additionally with either the Overstows or the MultiplePOD destructor respectively. As shown in Table 5 and Figure 5b, the MultiplePOD method consistently yields better results than both random and the general method, with the exception of the one-second limit, where it fared worse by 1.13 percentage points. The performance of the other destroyers and mixes was comparable to that of random and MKS. Again, differences in time taken were insignificant.

Limit(s)	Overstows		Overstows Mix		MultiPOD		MultiPOD Mix		MKS		Random	
	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time
1	8.87	**23.38**	**4.27**	24.18	5.40	25.87	5.65	24.43	4.27	24.27	10.83	27.01
2	6.51	37.92	4.03	37.77	**2.90**	**32.69**	5.95	35.91	4.15	36.58	6.27	40.57
3	3.60	41.66	3.20	43.71	**2.71**	41.97	5.42	48.46	3.59	46.44	6.17	**41.57**
4	3.90	57.22	3.94	**45.35**	**2.52**	50.60	3.46	55.67	5.95	59.11	3.90	52.97
5	3.60	62.74	3.34	66.01	**2.71**	60.86	3.42	**54.69**	3.93	59.91	3.90	62.56
6	6.23	67.71	3.65	71.74	**3.02**	**54.28**	3.09	58.94	3.68	65.15	3.52	65.00
7	3.81	73.39	3.67	**55.42**	**2.97**	62.59	3.58	65.42	3.41	84.45	3.60	74.55
8	3.60	73.81	3.68	67.47	**2.53**	**59.73**	3.48	69.30	3.86	79.78	3.81	70.52
9	3.63	78.34	3.90	76.27	**2.52**	65.77	3.74	80.35	4.12	76.26	3.60	**63.31**
10	3.90	88.84	3.72	77.79	**3.03**	70.21	3.41	76.41	3.66	77.98	3.91	**64.51**

Table 5: Performance of destroyers and mixed destroyer sets related to the 'overstowage' objective *(a)* on 45 instances with multiple PODs. Gap is given in (%) and time in (s).

Finally, the reefer destroyer was tested on all instances featuring reefers, along with a mixed set of the reefer, the MKS, and the random destructors. However, there are only 35 such instances in the set, containing mere 9.8 reefers to load on average (minimum 1, maximum 35), making it difficult to properly evaluate the quality of the ruiners. Nevertheless, neither the standalone method nor the mix provided any improvement on the instances with time, and were no better than random and MKS. Neither of the methods dominated the others.

As the results yielded by the MKS destructor were the most satisfactory, they were chosen for further comparison against other methods.

Class	#	Matheuristic			IP			CH		
		%Sol	%Opt	Time	%Sol	%Opt	Time	%Sol	%Opt	Time
1	13	100	100	1.3	100	100	5.6	100	100	0.01
2	22	95	86	12.1	77	73	12.1	91	82	0.04
3	13	92	92	8.0	85	85	8.1	92	85	0.02
4	78	100	100	17.7	86	82	37.1	97	91	0.10
5	36	89	83	29.4	53	47	30.1	72	53	0.07
6	15	100	100	2.5	100	100	3.9	73	53	0.01
7	14	86	71	11	29	29	11.2	79	50	0.04
8	14	100	79	5.4	57	50	11.2	86	57	0.02
9	17	94	76	13.0	53	35	13.7	82	59	0.04
10	8	100	88	2.5	75	50	4.5	75	25	0.01
11	6	83	83	3.3	67	67	3.2	50	33	0.01
Total	236	**95.6**	**90.3**	106.2	73.3	68.6	140.8	86	71	**0.37**

Table 6: Comparison of the matheuristic against the IP model [10] and the constructive heuristic (CH) [20]. The *Most Killed Space* destructor was used.

8.2. Comparison of the matheuristic with the IP model and the constructive heuristic

Below follows the performance comparison between the matheuristic, the IP model, and the constructive heuristic, all of which have been tested on the same machine. As with destructor tests, the algorithms were run on the 236 instances, with time limits increasing from 1 to 10s per instance. Graphs in Figure 6 present the total time taken by the methods to solve all instances, as well as the average optimality gaps achieved. When no solution was returned, a gap of 100% was assigned for the instance in question. The general-use solver used to solve the IP model reported gaps over 100% in 4 cases, the highest being 4165%. In order to prevent these outliers from skewing the average, we capped the gap at 100%, essentially discounting these solutions as invalid, as if none was returned. This corrected average gap for the IP model is shown in the right graph in Figure 6 as 'IP Corrected'. Table 6 presents the performance of the algorithms in each class, when the time limit is set to 1s.

Evidently, the matheuristic both runs faster and attains significantly lower gaps than the IP model alone. For the limit of 1s it yields an average gap of 4.34% within 106s, compared to 29.14% (corrected) returned by the solver after 141s. Only when the limit is extended to 7s per instance does the IP model start performing as well as the constructive heuristic, to which much of the success of the matheuristic must be attributed. Being a simple placement method, the constructive

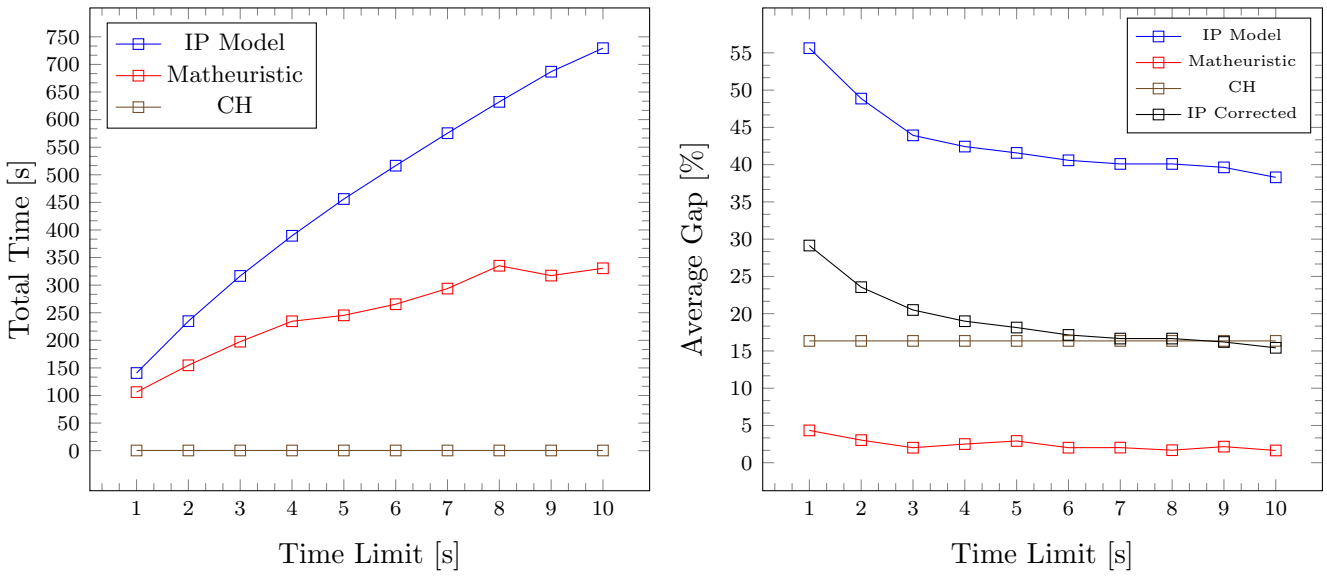


Figure 6: Comparison of the total time taken (left) by the solver, the constructive heuristic and the matheuristic to solve the 236 instances and the average optimality gap achieved (right), at different time limits.

heuristic is extremely fast, delivering results in below 1s in total. Still, the search process employed by the matheuristic is very effective, reducing the consistent average gap of 16.35% by 73.47% at 1s time limit.

Moreover, additional experiments were conducted with increased time limits per instance (from 10 seconds up to 5 minutes) to determine at which point the IP model – an optimal method – starts to perform better than the matheuristic. The IP model overtakes the heuristic after the time limit is extended to 120s, solving 97% of all instances optimally, achieving an average gap of 0.54%, continuing to increase the advantage as more time is afforded, and also using vastly more of it. However, allowing for such lengthy times is impractical for daily use in the industry, and since the matheuristic produces sufficiently good results in a significantly shorter time, it is also unnecessary to wait so long for optimal solutions.

Overall, the matheuristic improves on 50 out of 69 instances unsolved or not optimally solved by the constructive heuristic, finding optimal solutions for 44 of them. On average it takes 1.27 iterations to find the final solution. With no iteration restrictions imposed on the LNS, the search method continues for an average of 64 iterations on the improvable instances. An analysis of the 19 problematic instances (either unsolved or not solved to optimality by the matheuristic) revealed that the difficulties are varied and instance-specific. They may pertain to stack weight and height

limits being exceeded, many PODs stowed in a single stack, an unfavourable distribution of reefer containers in reefer slots, or the constructive heuristic failing to feasibly stow a large number of containers, which could result in a large portion of a solution not being fixed, posing difficulties for the solver to rebuild the solution within the given time constraints. In some cases, a combination of the aforementioned causes occurred. The latter cause was not often the suspect – among the instances solved by the constructive heuristic infeasibly, the observed rate of violations amounted to 22.25% of all cells on average (standard deviation 28.9%, median 5.56%), which did not pose a problem in most cases.

8.3. Comparison of the matheuristic with previous CSPBDL methods

This section compares the performance of the matheuristic against the previous constraint programming algorithm [10] and the state-of-the-art GRASP heuristic [20] methods to solve the *CSPBDL*. Having no possibility of running the methods on the same machine, we look at their results as reported in the papers. It should be noted that tests for the CP model were conducted on a slightly weaker, although still comparable machine. On the other hand, according to [20], the GRASP algorithm was run on an Intel Core Duo 2.93 GHz with 4 GB of RAM, also slower than the one used for testing the matheuristic. According to the reported values, the matheuristic offers moderately better performance than the CP algorithm - although more time is taken, more solutions are found, including just as many optimal ones. However, it falls short of the recently developed GRASP heuristic with 4.02% fewer solutions and 6.52% fewer optimal solutions found, albeit in a shorter time. Both the matheuristic and GRASP build upon the same constructive heuristic, although the GRASP algorithm utilises further randomisation.

9. Conclusion

In this study we developed a ruin-and-recreate matheuristic for the slot planning problem formulated as Container Stowage Problem for Below-Deck Locations. We designed and tested a number of ruin methods for use within Large Neighbourhood Search. Out of all devised destructors, three of them merit attention - two of the general ruin methods (MKS and VWR) turned out to produce solutions better than the random destroyer, while one destructor for instances with multiple PODs (MultiPOD) performed better overall than both random and the two general methods. The mixed method sets, however, did not offer any improvements.

Computational results show that the heuristic performs much better than just solving the IP model for all tested time limits, allowing to solve 95.6% of instances, including 90.3% solved to optimality within a strict limit of 1s per instance. The total time required for solving the instances is also shorter. Moreover, it achieves significantly lower optimality gaps and solves more instances than the constructive heuristic alone, providing an improvement of 11.16% more solved, and 27.18% more optimally solved instances, albeit in a longer time.

When compared to other methods, it can successfully compete with the CP algorithm by Delgado et al. [10]. Although its results are slightly worse than those of the GRASP heuristic, considering that the matheuristic relies heavily on the quality of the constructive method, one can speculate that improving on the placement heuristic could greatly improve the performance of the matheuristic, making it more competitive with GRASP.

Future work could involve further development of the constructive heuristic, as well as more research into destructive methods. For instance a heuristic could be employed to decide a selection of applicable destructors per iteration or per instance. The IP model could also be swapped for a constraint programming model, and solved with a CP solver in the repair step. In case of promising results, the matheuristic could be applied to more realistic problem extensions such as the one introduced by Parreño et al. [20]. Furthermore, the algorithm could be adapted for use across multiple locations sharing the same set of PODs, to allow for optimisation at the vessel level. However, some stability considerations would need to be included within the IP model.

References

[1] Ambrosino, D., Anghinolfi, D., Paolucci, M., Sciomachen, A., 2009. A new three-step heuristic for the Master Bay Plan Problem. Maritime Economics & Logistics 11 (1), 98–120.

[2] Ambrosino, D., Anghinolfi, D., Paolucci, M., Sciomachen, A., 2010. An experimental comparison of different heuristics for the Master Bay Plan Problem. In: Proceedings of SEA'2010, LNCS. Vol. 6049. pp. 314–325.

[3] Ambrosino, D., Paolucci, M., Sciomachen, A., 2015. A MIP heuristic for Multi-Port Stowage Planning. Transportation Research Procedia 10, 725–734.

[4] Ambrosino, D., Sciomachen, A., 1998. A constraint satisfaction for master bay plans. Transactions on the Built Environment 36, 175–184.

[5] Araújo, E., Chaves, A., Salles, L., Azevedo, A., 2016. Pareto clustering search applied for 3D container ship loading plan problem. Expert Systems with Applications 44, 50–57.

[6] Archetti, C., Speranza, M. G., 2014. A survey of matheuristics for routing problems. EURO Journal on Computational Optimization 2 (4), 223–246.

[7] Avriel, M., Penn, M., Shpirer, N., 2000. Container ship stowage problem: complexity and connection to the coloring of circle graphs. Discrete Applied Mathematics 103, 271–279.

[8] Botter, R., Brinati, M. A., 1992. Stowage container planning: A model for getting an optimal solution. In: Proceedings of the IFIP TC5/WG5.6 Seventh International Conference on Computer Applications in the Automation of Shipyard Operation and Ship Design, VII. North-Holland Publishing, pp. 217–229.

[9] Delgado, A., 2013. Models and Algorithms for Container Vessel Stowage Optimization. Ph.D. thesis, IT University of Copenhagen.

[10] Delgado, A., Jensen, R. M., Janstrup, K., Rose, T. H., Andersen, K. H., 2012. A constraint programming model for fast optimal stowage of container vessel bays. Journal of Operational Research 220 (1), 251–261.

[11] Hernández, P., Cruz-Reyes, L., Melin, P., Mar-Ortiz, J., Huacuja, H., Soberanes, H., Barbosa, J., 2013. An ant colony algorithm for improving ship stability in the containership stowage problem. Advances in Soft Computing and Its Applications 8266, 93–104.

[12] Jensen, R., Pacino, D., Ajspur, M., Vesterdal, C., 2018. Container vessel stowage planning. Weilbach.

[13] Lee, Z. Q., Fan, R., Hsu, W. J., 2015. Optimizing constraint test ordering for efficient automated stowage planning. Computational Logistics 9335, 343–357.

[14] Li, F., Ch. Tian, Cao, R., Ding, W., 2008. An integer linear programming for container stowage problem. In: Proceedings of the 8th International Conference on Computational Science, Part I. ICCS '08. Springer-Verlag, Berlin, Heidelberg, pp. 853–862.

[15] Li, K., 2012. Modelling and tabu search heuristic for solving container stowage planning problem. In: Control and Decision Conference (CCDC), 2012 24th Chinese. IEEE, pp. 2676–2680.

[16] Min, Z., Low, M. Y. H., Jing, H. W., H. Sh. Ying, Fan, L., W. Ch. Aye, 2010. Improving ship stability in automated stowage planning for large containerships. In: Proceedings of the International Multi-Conference on Engineers and Computer Scientists, IMECS 2010. Vol. 3. pp. 1838–1843.

[17] Monaco, M. F., Sammarra, M., Sorrentino, G., 2014. The terminal-oriented ship stowage planning problem. European Journal of Operational Research 239 (1), 256–265.

[18] Pacino, D., Delgado, A., Jensen, R. M., Bebbington, T., 2011. Fast generation of near-optimal plans for eco-efficient stowage of large container vessels. In: Proceedings of the 2nd International Conference on Computational Logistics (ICCL'11). Vol. LNCS 6971. Springer, pp. 286–301.

[19] Pacino, D., Delgado, A., Jensen, R. M., Bebbington, T., 2012. An accurate model for seaworthy container vessel Stowage Planning with ballast tanks. In: Proceedings of the 3rd International Conference on Computational Logistics (ICCL'12). Vol. LNCS 7555. pp. 17–32.

[20] Parreño, F., Pacino, D., Alvarez-Valdes, R., 2016. A GRASP algorithm for the container stowage slot planning problem. Transportation Research Part E: Logistics and Transportation Review 94, 141–157.

[21] Pisinger, D., Ropke, S., 2007. A general heuristic for vehicle routing problems. Computers & Operations Research 34 (8), 2403–2435.

[22] Pisinger, D., Ropke, S., 2010. Large Neighborhood Search. In: Handbook of Metaheuristics. Springer US, Boston, MA, pp. 223–246.

[23] Prescott-Gagnon, E., Desaulniers, G., Rousseau, L. M., 2009. A branch-and-price-based large neighborhood

search algorithm for the vehicle routing problem with time windows. Networks 54 (4), 190–204.

[24] Sciomachen, A., Tanfani, E., 2003. The Master Bay Planning Problem: a solution method based on its connection to the three-dimensional Bin Packing Problem. IMA Journal of Management Mathematics 14, 251–269.

[25] Tierney, K., Pacino, D., Jensen, R. M., 2014. On the complexity of container stowage planning problems. Discrete Applied Mathematics 169, 225–230.

[26] Wilson, I. D., Roach, P. A., 2000. Container stowage planning: a methodology for generating computerised solutions. Journal of the Operational Research Society 51, 1248–1255.

[27] Zh. Ning, Sh. Yifan, Ch. Jiaqi, Chao, M., Weijian, M., 2013. Study on bay-filling problem in stowage planning of export containers. Information Technology Journal 12 (21), 5967–5974.

[28] Zhang, W. Y., Lin, Y., Zh. Sh. Ji, Sep. 2005. Model and algorithm for container ship stowage planning based on bin-packing problem. Journal of Marine Science and Application 4 (3), 30–36.

[29] Zhang, Z., Ch. Y. Lee, 2015. Multiobjective approaches for the ship stowage planning problem considering ship stability and container rehandles. IEEE Transactions on Systems, Man, and Cybernetics: Systems PP (99), 1–16.