A 3-Phase Randomized Constraint Based Local Search Algorithm for Stowing Under Deck Locations of Container Vessel Bays

Dario Pacino
Rune Møller Jensen

IT University Technical Report Series **TR-2010-123**

ISSN 1600–6100 **January 2010**

ISSN 1600–6100

ISBN 9788779492059

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: **+45 72 18 50 00**
Telefax: **+45 72 18 50 01**
Web **www.itu.dk**

A 3-Phase Randomized Constraint Based Local Search Algorithm for Stowing Under Deck Locations of Container Vessel Bays

Dario Pacino
Rune Møller Jensen, Dario Pacino*,[1], Rune Møller Jensen[1]

[a]*IT University of Copenhagen, Software Development Department, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark*

Abstract

Even though containerized shipping is an eco-friendly mode of transportation and millions of containers are stowed every week, container vessel stowage is an all but neglected combinatorial optimization problem. The currently most successful approaches use hierarchical decompositions of the problem. The sub-problems of these decompositions consist of assigning containers to slots in individual vessel bays and for automated stowage systems to be useful for stowage coordinators they each must be solved within a few seconds. In this article, we define to our knowledge the most accurate representative model to date of these problems that we have developed in close collaboration with a larger liner shipping company since 2005. We introduce a 3-phase randomized constraint based local search algorithm to solve the problems. The performance of our algorithm has been compared to a complete and highly competitive constraint programming approach that we have developed in a parallel project on a large benchmark suite extracted from real stow-plans from our industrial partner. Our experimental results show that our approach robustly finds optimal or near optimal solutions within a fraction of a second. Our results support the hypothesis that these sub-problems due to a high-level goal of clustering similar containers in a bay often are under-constrained and thus particularly suited for local search.

Key words: Container vessel stowage, hierarchical decomposition, local search, delta evaluation.

1. Introduction

The last two decades of growing demand for seaborne transportation has forced international trading companies to lower the cost of their services. To answer these demands, the liner shipping industries have tried to approach the problem by merging into larger alliances. This strategy however has directed the industry into using larger vessels which in turn has made vessel stowage a very challenging problem using the traditional manual stowage planning practices. For this reason there is an increasing interest in developing efficient automated decision support systems for container stowage planning. With today's ships carrying up to 14.000 containers, the time of loading and unloading of cargo contributes largely to the overall cost. The

*Corresponding author, tel. +45 7218 5049
Email addresses: dpacino@itu.dk (Dario Pacino), rmj@itu.dk (Rune Møller Jensen)

time required to load and unload the containers at any given port, is a function of the containers arrangement in the vessel. For that reason, both ship operators and port managers are interested in minimizing the loading and unloading time by finding an optimal cargo arrangement.

The planning of the cargo arrangement is computationally hard due to the set of constraints that limits the way cargo can be positioned. Cargo is represented as box formed containers that have different properties such as height, weight, power requirements (in case of refrigerated containers) and security restrains based on the type of cargo transported. Limitations to the planning also come from the vessel itself, where attention must be payed to height and weight limits of stacks as well as balance and visibility constraints. Moreover, one must also take into consideration the interaction between existing and future cargo since the ship never empties completely at each port.

Despite of the practical importance of stowage planning, the amount of previous work is surprisingly scarce with less than 30 scientific publications and three patents. The "flat" models among these introducing one variable for each possible container assignment or similar, have turned out to be intractable in practice (e.g., Ambrosino et al. (2004); Botter and Brinati (1992); Giemsch and Jellinghaus (2003)). Scalable approaches are either heuristic (e.g., Ambrosino et al. (2004); Dubrovsky et al. (2002); Avriel et al. (1998)) or based on a hierarchical decomposition of the problem (e.g., Ambrosino et al. (2006); Kang and Kim (2002); Wilson and Roach (1999); Ambrosino et al. (2009)). The latter methods build on a natural two-level decomposition of the problem used by the liner shipping stowage coordinators and are the currently most successful for solving the problem. At the first level, the coordinators assign containers to storage areas in bays (called locations in this article) such that overstowage is minimized, crane utility is maximized and high-level constraints such as balance and stress moments are satisfied. At the second level, the coordinators assign containers to specific slots in each location, satisfying dangerous-goods requirements and stacking rules such as power requirements and length and height limitations.

In order for stowage coordinators to use automated stowage systems efficiently, it is essential that the total computation time is in the order of 15 minutes. The reason is that several stow-plans often must be generated to adapt the plans either to specific requirements or last minute changes. Approaches that use hierarchical decompositions of the stowage planning problem, typically have to stow in the order of 100 locations at the low-level part of the decomposition. Since high quality solutions to the high-level part of the decomposition are hard to generate in less than half of the total computation time, a time limit of 15 minutes means that each location must be stowed within a few seconds.

In this article, we introduce a 3-phase randomized constraint based local search algorithm for assigning a set of containers to slots in a location. Our work is the result of a close collaboration with a larger liner shipping company since 2005. Our hypothesis is that the sub-problems of stowing locations in hierarchical decompositions of stowage planning are under-constrained since high quality solutions to the first level of the decomposition cluster similar containers in locations and assure that capacity limits for the different types of containers are met. Thus, we expect a large number of optimal or near optimal stow-plans to exist for each location. This situation is ideal for local search approaches. Complete methods like Integer Programming (IP) and Contraint Programming (CP), on the other hand, can be expected to spent too much time proving optimality due to the large number of good solutions.

Since there is a very large number of constraints and objectives involved in stowing containers in over and under deck locations and some of these are not fully understood, we have formulated a representative problem model in collaboration with the industry for stowing a

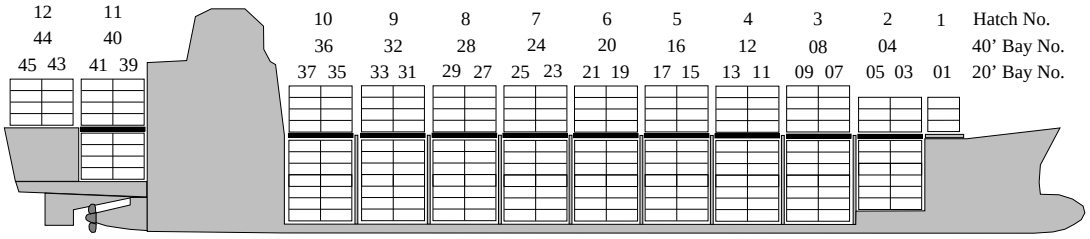


Figure 1: Vessel layout

given set of containers in an under deck location. To our knowledge, this is the most detailed and accurate model of this sub-problem published to date. The first phase of our 3-phase local search approach, is a construction heuristic that makes an initial assignment of containers to slots. The second and third phase use a local search (LS) algorithm with a swap neighborhood to find a feasible solution and a local optimum, respectively. A parallel race is performed between random restarts of the three phases, after which the best solution is selected. These LS algorithms use incremental update of objectives and constraints. They were originally implemented in COMET (Michel and Hentenryck, 2002), but an order of magnitude speedup could be achieved by reimplementing them in C++.

The LS algorithm has been experimentally evaluated by comparing its performance with a complete and highly competitive CP approach that we have developed in a parallel project (Delgado et al., 2009). The comparison has been carried out on a benchmark suite of 140 real stowage problems provided by our industrial collaborator. The experimental results support our hypothesis. The LS approach finds optimal or near optimal solutions within a fraction of a second (see Figure 5(a)). The complete CP approach when tuned with an efficient diving heuristic, symmetry breaking, and good lower bounds can often prove optimality quite fast. But as expected, there is a significant fraction of problems, where this is not the case (see Figure 5(b)).

The remainder of this article is organized as follows. In Section 2, we introduce the problem, for which a detailed mathematical model is presented in Section 3. The method is then presented in Section 4, and its results are analysed in Section 5. After the review of related work in Section 6, we draw conclusions in Section 7.

2. Background

A liner shipping vessel is a ship that transports box formed containers on a fixed cyclic route. Containers typically have a width of 8 feet, and a length of either 20 or 40-foot. There exists however longer containers such as 45 and 50-foot. Containers can be either 8 feet or 8 feet and 6 inches high, with the exception of some higher 40-foot containers called *high-cube* containers that are 1 foot taller. The weight limit of a container is about 34 tons, for a 40-foot, and 32 tons for a 20-foot. Some containers are refrigerated and as such require a connection to special power plugs. Other special type of containers are pallet-wide containers, where a standard European pallet can be stored, and IMO containers, which are used to store dangerous goods. Such containers must obey special stacking rules. In addition, there are out-of-gauge out-of-gate (OOG) containers with cargo sticking out in the top or at the side (e.g., a yacht) and non-containerized break-bulk like windmill wings.

The cargo space of a vessel is composed of a number of *bays*, which are a collection of container *stacks* along the length of the ship. Each bay is divided into an *upper deck* and *under*

deck part by a *hatch cover*, which is a flat water tight structure that prevents the vessel from taking in water. An overview of a vessel layout is show in Figure 1.

Figure 2 shows how each under deck stack is composed of two Twenty foot Equivalent Unit (TEU) stacks and one Forty foot Equivalent Unit (FEU) stack, which hold vertically arranged *cells* indexed by *tiers*. The TEU stack cells are composed of two *slots*, which are the physical positioning of a 20-foot container. The *aft* slot refer to the position toward the stern on the vessel, while *fore* slots are allocated on the bow side. Some of the cells have access to power plugs and are typically situated at the bottom of the bay.

The loading and unloading of containers are carried out by *quay cranes* that can access the stacks individually. Some cranes can lift two 20-foot containers at the same time, but they only have access to the container on top of the stack.

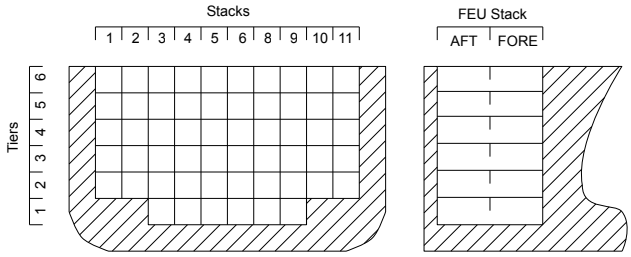


Figure 2: An under deck bay cell structure seen from behind (left) and from the side (right).

The primary objective of stowage planning is to minimize berth time, both because shipping companies pay for berth time and because shorter berth time gives air in the schedule that can prevent delays from rippling to downstream ports. Essentially berth time can be minimized in two ways: by minimizing the total number of quay crane moves and by distributing these moves evenly over the set of quay cranes assigned to the vessel (makespan minimization). The total number of quay crane moves can be reduced by avoiding *overstowage*. A container A is overstowing a container B, if A is above B in the stack but B must be discharged before A, such that A must be moved in order to unload B. An important secondary objective of stowage planning is to generate stow-plans that are robust to changes in the cargo forecast. The reason is that the number of containers of each type to load in downstream ports is only fairly accurately known three ports ahead.

When a set of containers to stow in a bay has been decided, the positioning of these containers has limited interference with containers in other bays. For this reason, it is natural to divide the constraints and objectives of the stowage planning problem into high-level inter bay constraints and objectives and low-level intra bay constraints and objectives.

High-level constraints mainly consider the stability of the vessel as defined by its trim, metacentric height, and stress moments such as shear, bending and torsion. In addition, any distribution of containers to bays must satisfy weight and volume capacity limits as well as capacity limits of the different container types.

High-level objectives include distributing moves evenly to cranes and avoid *lid-overstowage* which is overstowage between containers divided by the hatch cover. Lid-overstowage can be very costly, a single discharge from a storage area under a hatch-lid cover requires all containers resting on the hatch-lid and the hatch-lid itself to be unloaded. Finally, bays should cluster containers to the same discharge port, partly to avoid overstowage in individual stacks and partly to make the stow-plan more robust to changes in forecasted demands.

Low-level constraints are mainly stacking rules. They ensure that containers are arranged

in valid physical stacks satisfying the height and weight limits of the stacks, that reefer containers are positioned near power plugs, that 20-foot containers are not stacked over 40-foot containers (40-foot containers lack physical support points for 20-foot containers), that IMO containers are placed with the required separation, that a legal pattern of pallet-wide containers exist horizontally in the stacks, that OOG containers have sufficient spacing around them, and that break-bulk can be placed as required. Stacks may also have restrictions specifying which type of container can be stacked in them. Also special constraints apply over-deck, where containers must be stacked in specific configurations in order to resist wind forces, keeping the line-of-sight, and ensuring access to special containers or lashing rods (e.g. to water live plants or tighten lashing rods).

Low-level objectives reflect rules of thumb used by stowage coordinators in order to get stow-plans that are robust to changes in forecasted demands. The objectives include maximazing the number of unused stacks, clustering of containers with the same discharge port in stacks, minimizing the number of reefer slots used for non-reefer containers, and minimizing overstowage between containers in the same stack.

3. The Container Stowage Problem for an Under-Deck Location

As mentioned in the introduction, our hypothesis is that the sub-problems of stowing locations in hierarchical decompositions of stowage planning are under-constrained since good solutions to the high-level problem of distributing containers to bays cluster similar containers in locations and assure that capacity limits for the different types of containers are met. For that reason, we assume that these problems can be solved efficiently using local search rather than complete methods like constraint programming and integer programming that may spent prohibitively long time searching through a large number of near optimal solutions to prove optimality. Since stow-plans must be generated within 15 minutes to be of practical value for stowage coordinators and there are in the order of 100 locations in a vessel, our goal is to stow a single location for a given set of containers within a few seconds.

Due to the very large number of constraints and objectives involved in stowing containers in over and under deck locations, we decided to make our investigation manageable by formulating a representative problem called the Container Stowage Problem for an Under-Deck Location (CSPUDL) for stowing containers in under deck locations. Our problem model is the result of a close collaboration with a larger liner shipping company since 2005 and it is to our knowledge the most accurate description of this problem to date. The CSPUDL covers all constraint and objective classes of the problem and we assess that it has a high correlation with the complete problem model in terms of solution algorithm performance. Specifically, the CSPUDL includes stacking rules for 20 and 40-foot containers, FEU and TEU stack overlapping, reefer containers, pre-placed containers, and weight and height constraints. The objectives include overstowage and three rules of the thumb used by stowage coordinators to achieve robustness. The CSPUDL excludes break-bulk cargo, OOG containers, and odd slots (i.e., cells that can only hold a single 20-foot container). In addition, we do not consider IMO and pallet-wide containers since these are often placed in special locations.

Formally, let $\mathcal{C}$ denote the set of containers to stow in the location. A subset of these $\mathcal{C}^P$ defines containers that are pre-placed in the location. Slots that do not hold a container are assigned the *null container* $\perp$. Each slot in stack $s \in \mathcal{S}$, tier $t \in \mathcal{T}_s$ and position $p \in \mathcal{P}$ (the AFT ($p = 1$) and FORE ($p = 2$) part of a cell) is represented by a decision variable x_{stp} with domain $\mathcal{C} \cup \{\perp\}$. Since 40-foot containers take the space of two 20-foot containers, the physical space assigned by the decision variables overlap. For this reason, we use the

convention to assign 40-foot containers to the AFT position variable only and require that the associated FORE position variable is assigned to $\perp$.

For a stack s, a tier t and a position p, the Boolean constant A^R_{stp} indicates if the slot can hold a reefer container, while the capacity of a cell to hold a 40-foot and 20-foot container is represented by the Boolean constant A^{40}_{st} and A^{20}_{stp}, respectively. Each stack s in the location has a maximum height H_s and weight W_s. Attributes for a container $c \in \mathcal{C}$ are defined by the functions $w(c)$ for the weight, $h(c)$ for the height and $d(c)$ for the discharge port. The function $r(c)$ and $\perp(c)$ respectively indicate if a container c is a reefer container or a null container, while the functions $f(c)$ and $t(c)$ identify 40-foot and 20-foot containers. Finally, a pre-placed container $c \in \mathcal{C}^{\mathcal{P}}$ is assumed to be stored in the slot defined by the tuple (s_c, t_c, p_c). Table 1 offers an overview of the model parameters.

Sets	
$\mathcal{S} \in \{1, ..., N^S\}$	The index set of stacks in the location, where N^S is the number of stacks.
$\mathcal{T}_s \in \{t_s, ..., N^T\}$	The index set of tiers for stack s, where t_s is the bottom tier of s and N^T is the number of tiers of the location.
$\mathcal{P} \in \{1, 2\}$	The index set representation of the AFT ($p = 1$) and FORE ($p = 2$) position of a cell.
$\mathcal{C} \in \{1, ..., N^C\}$	The index set of containers to place in the location, where N^C is the number of containers.
$\mathcal{C}^P \subset \mathcal{C}$	The subset of containers pre-placed in the location.
Constants	
$\perp$	The null container.
$A^{20}_{stp} \in \mathbb{B}$	True iff the slot in stack $s \in \mathcal{S}$, tier $t \in \mathcal{T}_s$ and position $p \in \mathcal{P}$ can hold a 20-foot container.
$A^{40}_{st} \in \mathbb{B}$	True iff the cell in stack $s \in \mathcal{S}$ and tier $t \in \mathcal{T}_s$ can hold a 40-foot container.
$W_s \in \mathbb{R}_+$	The maximum weight of the stack $s \in \mathcal{S}$.
$H_s \in \mathbb{R}_+$	The maximum height of the stack $s \in \mathcal{S}$.
Attribute functions	
$w(c) : \mathcal{C} \cup \{\perp\} \mapsto \mathbb{R}_+$	The weight of the container $c \in \mathcal{C}$ or 0 if $c = \perp$.
$h(c) : \mathcal{C} \cup \{\perp\} \mapsto \mathbb{R}_+$	The height of the container $c \in \mathcal{C}$ or 0 if $c = \perp$.
$r(c) : \mathcal{C} \cup \{\perp\} \mapsto \mathbb{B}$	True iff the container $c \in \mathcal{C}$ is a reefer.
$\perp(c) : \mathcal{C} \cup \{\perp\} \mapsto \mathbb{B}$	True iff the container $c = \perp$.
$d(c) : \mathcal{C} \mapsto \mathbb{N}$	The discharge port of the container $c \in \mathcal{C}$.
$f(c) : \mathcal{C} \cup \{\perp\} \mapsto \mathbb{B}$	True iff the container $c \in \mathcal{C}$ is a 40-foot container.
$t(c) : \mathcal{C} \cup \{\perp\} \mapsto \mathbb{B}$	True iff the container $c \in \mathcal{C}$ is a 20-foot container.
Variables	
$x_{stp} \in \mathcal{C} \cup \{\perp\}$	The container placed in stack $s \in \mathcal{S}$,tier $t \in \mathcal{T}_s$ and position $p \in \mathcal{P}$.

Table 1: Constants, attribute functions, and variables of the CSPUDL.

3.1. Constraints

The constraints of the CSPUDL are:

$$\forall s \in \mathcal{S}, t \in \mathcal{T}_s \,.\, \neg f(x_{st2}) \wedge (f(x_{st1}) \Rightarrow \perp(x_{st2})) \tag{1}$$

$$\forall s \in \mathcal{S}, t \in \mathcal{T}_s \setminus \{t_s\}, p \in \mathcal{P} \,.\, \neg\bot(x_{stp}) \Rightarrow \big(t(x_{s(t-1)1}) \wedge t(x_{s(t-1)2})\big) \vee f(x_{s(t-1)1}) \quad (2)$$

$$\forall s \in \mathcal{S}, t \in \mathcal{T}_s, p \in \mathcal{P} \,.\, t(x_{stp}) \Rightarrow A^{20}_{stp} \quad (3)$$

$$\forall s \in \mathcal{S}, t \in \mathcal{T}_s \,.\, f(x_{st1}) \Rightarrow A^{40}_{st} \quad (4)$$

$$\forall s \in \mathcal{S}, t \in \mathcal{T}_s \setminus \{N^T\}, p \in \mathcal{P} \,.\, f(x_{st1}) \Rightarrow \neg t(x_{s(t+1)p}) \quad (5)$$

$$\forall c \in \mathcal{C} \,.\, |\{x_{stp} = c \,|\, s \in \mathcal{S}, t \in \mathcal{T}_s, p \in \mathcal{P}\}| = 1 \quad (6)$$

$$\forall c \in \mathcal{C}^{\mathcal{P}} \,.\, x_{s_c t_c p_c} = c \quad (7)$$

$$\forall s \in \mathcal{S}, t \in \mathcal{T}_s, p \in \mathcal{P} \,.\, r(x_{stp}) \wedge t(x_{stp}) \Rightarrow A^{R}_{stp} \quad (8)$$

$$\forall s \in \mathcal{S}, t \in \mathcal{T}_s \,.\, r(x_{st1}) \wedge f(x_{st1}) \Rightarrow A^{R}_{st1} \vee A^{R}_{st2} \quad (9)$$

$$\forall s \in \mathcal{S} \,.\, \sum_{t \in \mathcal{T}_s} (w(x_{st1}) + w(x_{st2})) \leq W_s \quad (10)$$

$$\forall s \in \mathcal{S} \,.\, \sum_{t \in \mathcal{T}_s} max(h(x_{st1}), h(x_{st2})) \leq H_s \quad (11)$$

Constraint (1) ensures that our convention for assigning 40-foot containers is maintained. All containers are guaranteed physical support from below by (2). 20-foot and 40-foot container restrictions are fulfilled by constraint (3) and (4). 20-foot containers are not allowed to be stored on top of 40-foot containers by constraint (5). Each container is assigned to exactly one slot (6). Constraint (7) ensures that pre-placed containers are given their assigned position. 20-foot reefer containers are stowed in reefer slots by constraint (8), while (9) ensures that 40-foot reefer containers are stowed in cells where either one of the two slots has a power-plug. Breaking weight and height maximum levels of the stacks is avoided with constraint (10) and (11).

3.2. Objectives

The main objective of the CSPUDL is to minimize overstowage in individual stacks. Other objectives, such as minimizing the number of used stacks and grouping containers with the same discharge port, help achieving a container arrangement that can minimize costs at later ports.

$$O_{os} = \sum_{s \in \mathcal{S}} \sum_{t \in \mathcal{T}_s} \sum_{p \in \mathcal{P}} o_{stp} \quad (12)$$

$$O_{ur} = \sum_{s \in \mathcal{S}} \sum_{t \in \mathcal{T}_s} \sum_{p \in \mathcal{P}} ur_{stp} \quad (13)$$

$$O_{ps} = \sum_{s \in \mathcal{S}} |\,\{d(x_{stp}) \,|\, t \in \mathcal{T}_s, p \in \mathcal{P}, \neg\bot(x_{stp})\}\,| \quad (14)$$

$$O_{us} = \sum_{s \in \mathcal{S}} us_s \quad (15)$$

In (12) one unit cost is counted for each container that is overstowing another one below in the stack, where o_{stp} defines whether a container c in stack s, tier t and slot position p overstows another container in the stack. Thus, $o_{stp} = 1$ if $\neg\bot(x_{stp})$ and there exists a tier $t' \in \{t_s, ..., t-1\}$ below t with an overstowed container $d(x_{st'p}) < d(x_{stp})$. Otherwise $o_{stp} = 0$. Objective (13) counts one unit cost for each misused reefer slot, where $ur_{stp} = 1$ if A^R_{stp} and $(f(x_{st1}) \wedge \neg r(x_{st1}) \vee t(x_{stp}) \wedge \neg r(x_{stp}))$, and 0 otherwise. Notice that a 40-foot non-reefer container will add a unit cost for each reefer slot it covers. In order to favor stacks stowing

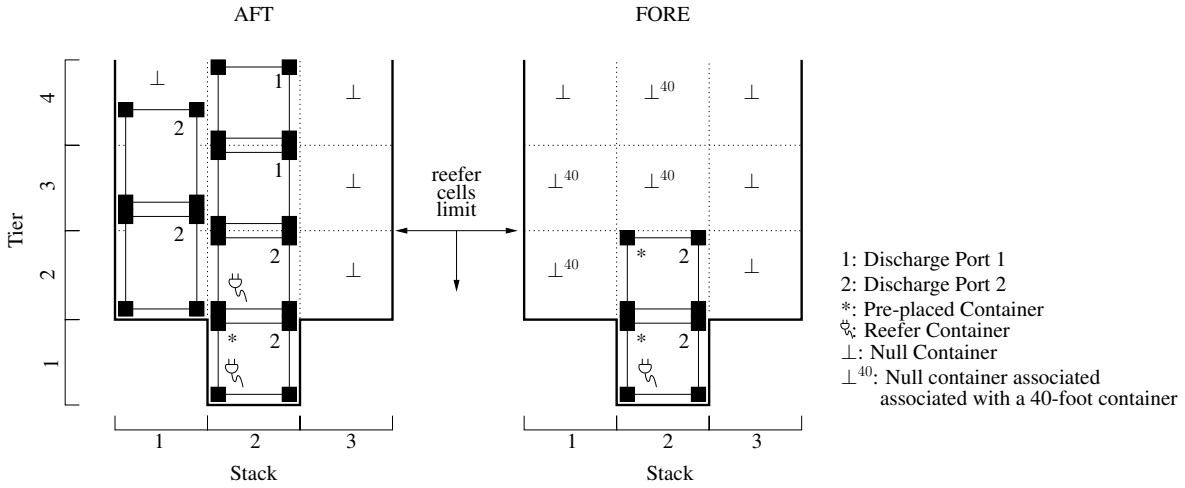


Figure 3: A stow-plan for a small under deck location. Notice that the two 40-foot containers stored in stack one are high-cube containers.

containers with the same port of destination, one unit cost is counted for each discharge port present in the stack (14). In order to minimize the number of used stacks, one unit cost for each stack used is counted by (15) where $us_s = 1$ if there exists a $t \in \mathcal{T}_s$ and $p \in \mathcal{P}$ such that $\neg\bot(x_{stp})$, and 0 otherwise. The objective function of the CSPUDL is a weighted sum of the above objectives reflecting their importance

$$O = 100O_{os} + 5O_{ur} + 20O_{ps} + 10O_{us}. \tag{16}$$

As an example, Figure 3 shows a feasible stow-plan for a small under-deck location. In this particular stow-plan there is no overstowage ($O_{os} = 0$), and since all the slots below the reefer cell limit are reefer slots, there are three reefer slots with containers violating the free reefer objective ($O_{ur} = 3$). Only one stack of the two used is pure, thus giving three unit costs for the pure stack objective ($O_{ps} = 3$) and two for the free stack objective ($O_{us} = 2$). In summary the sample stow-plan has a total objective value of 95.

4. 3-Phase Randomised Constraint Based Local Search (3-RCLS)

Our approach to solve the CSPUDL is a three phase search procedure where the result is improved by racing parallel randomized restarts. The main idea is to create a procedure that is able to to find a good solution to the CSPUDL in a very short time and exploit the potential of multi-core computing by running a race between parallel random restarts of the algorithm. The main search procedure is composed of an initialization phase, a feasibility phase, and an optimization phase. The initialization phase is a placement heuristic that tries to arrange containers in the location in a way that minimizes the objectives and the number of violated constraints. Using this initial container assignment, the feasibility phase performs a local search based on a *min-conflict heuristic* (Minton et al., 1992) that finds a feasible solution. When the feasibility phase terminates, it is possible to reduce the search space to feasible solutions. It is within this search space that the optimality phase performs a local search and finds a local cost minimum. The aim is to gradually reduce the complexity of the problem in such a way that a feasible solution is found fast such that the algorithm can use more time searching for an optimal solution. In order for the concurrent algorithms to pursue different search paths, randomization is used whenever possible.

The local search for both the feasibility and optimality phase uses a neighbourhood generated by swapping containers within the location. A swap is an exchange of some containers between a pair of cells. Formally, a swap γ is a pair of tuples $\gamma = (\langle s, t, c\rangle, \langle s', t', c'\rangle)$ where the containers c in the cell at stack s and tier t exchange position with the containers c' in the cell at stack s' and tier t'. The sets c and c' can contain at most two containers. Swaps are implemented with two functions $swap^{20}$ for exchanging position of 20-foot containers, and $swap^{40}$ for exchanging position of 40-foot containers.

- $swap^{20}(x_{stp}, x_{s't'p'})$ where $p \neq p'$ if $(s, t) = (s', t')$, swaps the value of x_{stp} and $x_{s't'p'}$ and covers the following swap types 1) $20' \leftrightarrow 20'$ and 2) $\perp \leftrightarrow 20'$, where $20'$ indicates a 20-foot container.

- $swap^{40}((x_{st1}, x_{st2}), (x_{s't'1}, x_{s't'2}))$ where $(s, t) \neq (s', t')$, pairwise swaps the values of the cell variables (x_{st1}, x_{st2}) and $(x_{s't'1}, x_{s't'2})$ and covers the following swap types 1) $(40', \perp) \leftrightarrow (40', \perp)$, 2) $(\perp, 20') \leftrightarrow (40', \perp)$, 3) $(20', \perp) \leftrightarrow (40', \perp)$, 4) $(20', 20') \leftrightarrow (40', \perp)$ and, 5) $(\perp, \perp) \leftrightarrow (40', \perp)$, where $40'$ indicates a 40-foot container.

Proposition 1. *The swap neighborhood Γ defined by the union of $swap^{20}$ and $swap^{40}$ swaps for a CSPUDL problem P is complete.*

Proof. We prove the claim by showing that any current assignment π of the variables of P can be changed to an arbitrary assignment π' via a sequence of swaps. Let x^{π}_{stp} denote the value of variable x_{stp} in assignment π. Assume without loss of generality that the variables of P are re-assigned from π to π' according to some total ordering of the cells $\prec$. Consider assigning the variables of cell (s, t) at some point in this ordered re-assignment. We have two cases:

1. $f(x^{\pi'}_{st1})$ (i.e., a 40-foot container must be placed in the cell). If $x^{\pi'}_{st1} = x^{\pi}_{st1}$ the cell assignment is correct. Otherwise find the container $x^{\pi'}_{st1}$ in a cell (s', t') and use $swap^{40}$ of type 1 to 5 to assign it to x_{st1}.
2. $\neg f(x^{\pi'}_{st1})$ (i.e., 20-foot containers or empties must be placed in the cell). If $x^{\pi'}_{st1} = x^{\pi}_{st1}$ then the AFT slot of the cell is assigned correctly. Otherwise find the container (including $\perp$) in the FORE slot of the cell or in another cell (s', t') and use $swap^{20}$ type 1 or 2 to assign it to x_{st1}. Do the same for x_{st2}, except only look for the container in another cell (s', t').

Because all variables of cells previous to (s, t) have already been assigned, the cells (s', t') with containers to swap into (s, t) must come after (s, t) (i.e. $(s, t) \prec (s', t')$). Since the cell (s, t) is arbitrarily chosen, we have that all cells can be re-assigned from π to π' using the resulting sequence of swaps. □

4.1. Constraint Violations

In constraint based local search it is common to evaluate and represent a constraint in terms of the set of variables that violate it. When the violation of a constraint is 0 the constraint is satisfied. When a variable violates a constraint, its violation degree is either fixed (e.g, 1) or reflects to which extent the variable breaks the constraint. This representation is ideal when using the min-conflict heuristic, since it makes it possible to identify which variable violates most constraints, and to which degree.

Let x^{π}_{stp} denote the value of variable x_{stp} in assignment π. Each constraint and objective is then defined in terms of a function $\sigma(\pi)$ on the assignment π of variables over the violations of each slot variable defined by $\nu(\pi, s, t, p)$, where s is the stack, t is the tier, and p is the position

of the slot. To ease the readability, we often interpret the Boolean values $false$ and $true$ as the numerical values 0 and 1, respectively.

The 20 and 40-foot capacity constraint (3) and (4) are represented by

$$\nu_1(\pi, s, t, p) = \neg(t(x^\pi_{stp}) \Rightarrow A^{20}_{stp})$$
$$\sigma_1(\pi) = \sum_{s\in\mathcal{S}} \sum_{t\in\mathcal{T}_s} \sum_{p\in\mathcal{P}} \nu_1(\pi, s, t, p)$$

and

$$\nu_2(\pi, s, t, p) = \neg(f(x^\pi_{stp}) \Rightarrow A^{40}_{st})$$
$$\sigma_2(\pi) = \sum_{s\in\mathcal{S}} \sum_{t\in\mathcal{T}_s} \sum_{p\in\mathcal{P}} \nu_1(\pi, s, t, p).$$

The "no hanging containers" constraint for 20-foot and 40-foot containers (2) is represented by

$$\nu_3(\pi, s, t, p) = \sum_{t'=t_s}^{t-1} \neg\left(\neg\bot(x^\pi_{stp}) \Rightarrow (t(x^\pi_{st'1}) \wedge t(x^\pi_{st'2})) \vee f(x^\pi_{st'1})\right)$$
$$\sigma_3(\pi) = \sum_{s\in\mathcal{S}} \sum_{t\in\mathcal{T}_s\setminus\{t_s\}} \sum_{p\in\mathcal{P}} \nu_3(\pi, s, t, p).$$

Thus for some stored container, the degree of violation is defined as the number of slots in cells under it with insufficient support. The reefer constraints (8) and (9) are represented by

$$\nu_4(\pi, s, t, p) = \neg\left(r(x^\pi_{xtp}) \Rightarrow A^R_{stp} \vee \left(f(x^\pi_{stp}) \wedge (A^R_{st1} \vee A^R_{st2})\right)\right)$$
$$\sigma_4(\pi) = \sum_{s\in\mathcal{S}} \sum_{t\in\mathcal{T}_s} \sum_{p\in\mathcal{P}} \nu_4(\pi, s, t, p).$$

The maximum stack height constraint (11) and the maximum stack weight constraint (10) are represented by

$$\nu_5(\pi, s, t, p) = \begin{cases} max\left(0, \frac{\vartheta^{H\pi}_s - H_s}{|\vartheta^{C\pi}_s|}\right) & : \neg\bot(x^\pi_{stp}) \\ 0 & : \text{ otherwise} \end{cases}$$
$$\sigma_5(\pi) = \sum_{s\in\mathcal{S}} \sum_{t\in\mathcal{T}_s} max\left(\nu_5(\pi, s, t, 1), \nu_5(\pi, s, t, 2)\right)$$

$$\nu_6(\pi, s, t, p) = \begin{cases} max\left(0, \frac{\vartheta^{W\pi}_s - W_s}{|\vartheta^{C\pi}_s|}\right) & : \neg\bot(x^\pi_{stp}) \\ 0 & : \text{ otherwise} \end{cases}$$
$$\sigma_6 = \sum_{s\in\mathcal{S}} \sum_{t\in\mathcal{T}_s} \sum_{p\in\mathcal{P}} \nu_6(\pi, s, t, p).$$

where $\vartheta^{W\pi}_s = \sum_{t\in\mathcal{T}_s} \sum_{p\in\mathcal{P}} w\left(x^\pi_{stp}\right)$ is the current weight, $\vartheta^{H\pi}_s = \sum_{t\in\mathcal{T}_S} max\left(h\left(x^\pi_{st1}\right), h\left(x^\pi_{st2}\right)\right)$ is the current height of stack s and $\vartheta^{C\pi}_s = \sum_{t\in\mathcal{T}_s} \sum_{p\in\mathcal{P}} \neg\bot(x^\pi_{stp})$ is the current number of containers stowed in stack s for assignment π. For these constraints the violation degree is distributed equally over variables holding containers since it is not possible to identify which one

of them will have the largest influence. Finally, the no 20-foot over 40-foot container constraint (5) is represented by

$$\nu_7(\pi, s, t, p) = \sum_{t'=t_s}^{t-1} \neg \left(t(x^\pi_{stp}) \Rightarrow \neg f(x^\pi_{st'1}) \right)$$

$$\sigma_7 = \sum_{s \in \mathcal{S}} \sum_{t=t_s+1}^{N^T} \sum_{p \in \mathcal{P}} \nu_7(\pi, s, t, p).$$

Similar to ν_3, the violation degree of a 20-foot container is the number of 40-foot containers stored below.

The remaining constraints are implicitly satisfied by the algorithm. By first assigning pre-placed containers to their given position, and by heuristically placing the remainder of the containers to variables at the beginning of the algorithm according to the convention of assigning 40-foot containers, constraint (1), the pre-placed constraint (7), and the all-loaded constraint (6) are also satisfied.

4.2. Objective Violations

The objectives have been defined in the form of soft-constraints, where the number of violations is the actual objective value. The overstowage objective (12) is represented by

$$\nu_8(\pi, s, t, p) = \exists t' \in \{t - s, \ldots, t - 1\}, p' \in \mathcal{P} \,.\, \neg\bot(x^\pi_{st'p'}) \wedge d(x^\pi_{stp}) > d(x^\pi_{st'p'})$$

$$\sigma_8(\pi) = \sum_{s \in \mathcal{S}} \sum_{t \in \mathcal{T}_s} \sum_{p \in \mathcal{P}} \nu_8(\pi, s, t, p).$$

The free reefer slots objective (13) is represented by

$$\nu_9(\pi, s, t, p) = A^R_{stp} \wedge \left(f(x^\pi_{st1}) \wedge \neg r(x^\pi_{stp}) \vee t(x^\pi_{stp}) \wedge \neg r(x^\pi_{stp}) \right)$$

$$\sigma_9 = \sum_{s \in \mathcal{S}} \sum_{t \in \mathcal{T}_s} \sum_{p \in \mathcal{P}} \nu_9(\pi, s, t, p).$$

The free stack objective (15) is represented by

$$\nu_{10}(\pi, s, t, p) = \begin{cases} \frac{1}{\vartheta^C_s} & : \neg\bot(x^\pi_{stp}) \\ 0 & : \text{ otherwise} \end{cases}$$

$$\sigma_{10} = \sum_{s \in \mathcal{S}} \sum_{t \in \mathcal{T}_s} \sum_{p \in \mathcal{P}} \nu_{10}(\pi, s, t, p).$$

The last objective, pure stacks (14) is represented by

$$\nu_{11}(\pi, s, t, p) = \begin{cases} \frac{|\{d(x^\pi_{s't_p}) \,|\, t' \in \mathcal{T}_s \,, p' \in \mathcal{P} \,, \neg\bot(x^\pi_{st'p'})\}|}{\vartheta^{C\pi}_s} & : \quad \neg\bot(x^\pi_{stp}) \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$\sigma_{11} = \sum_{s \in \mathcal{S}} \sum_{t \in \mathcal{T}_s} \sum_{p \in \mathcal{P}} \nu_{11}(\pi, s, t, p).$$

As for ν_5 and ν_6, we distribute the violation degree equally over variables holding containers since it is not possible to identify which one of them will have the largest influence.

4.3. The Local Search Strategy

The feasibility and optimality phase are based on the same underlying strategy. Candidate swap variables are selected based on their current violation degree; once the first variable is chosen, the algorithm evaluates the possibility of swapping the container related to this variable with the container of any other variable. The evaluation is performed using efficient algorithms, which are able to evaluate the delta change in objective value without actually performing the swap. The evaluation only recalculates the portion of objective or constraint violations which are affected by the swap. If a beneficial swap is found, the selected swap is now performed on the current assignment. It is at this point that the state of the model changes. The model is not recalculated entirely, only those parts that are influenced by the change are re-evaluated. Such partial re-evaluation is referred to as *incremental update*. Incrementally maintained variables in the model are the violation variables ($\nu_1 ... \nu_{11}$ and $\sigma_1 ... \sigma_{11}$) and the auxiliary variables (ϑ_s^W, ϑ_s^H and ϑ_s^C). The next session describes how partial evaluation is carried out when evaluating swap moves. Incremental updates are implemented using very similar principles and are therefore not discussed any further.

4.4. Delta Evaluation of Swaps

The partial evaluation of swap moves is performed using specialized algorithms for each constraint and objective. Evaluation of a move with those operations only involves a partial re-computation of the constraints or objectives that are affected by the change. Formally, let $swap(\pi, \gamma)$ be the assignment π' resulting from performing swap $\gamma \in \Gamma$ on the current assignment π. For a constraint or objective $\sigma \in \{\sigma_1, ..., \sigma_{11}\}$, we then define the delta change $\delta_\pi(\sigma, \gamma) = \sigma(\pi') - \sigma(\pi)$.

Taking the weight constraint (σ_6) as an example, the violation based on the swap can be recalculated by removing the weight of the swapping containers from the current weight of their respective stacks and then adding it to the current weight of the respective stacks of destination. The difference in violation between the new weight violation and the original one would be the result of the delta evaluation. The delta evaluations for constraints σ_1, σ_2, σ_4, and σ_5 are similar to the weight constraint and will not be described further. The σ_3 and σ_7 constraints on the other hand are more complex, since the swap may affect many containers in the involved stacks. Here we show how to define one of these constraints for a specific swap case. The remaining constraints can be implemented using similar ideas.

We consider the no 20-foot containers on top of 40-foot constraint (σ_7) for a swap $\gamma = (\langle s, t, c\rangle, \langle s, t', c'\rangle)$ within the same stack of a 40-foot container with two null containers ($swap^{40}$ type 5). The other cases can be defined in a similar fashion. If the 40-foot container is above the null containers, the value of the delta evaluation is equal to the number of 20-foot containers between the null containers and the 40-foot container. Otherwise it is simply the negative of this. Thus, if $tb_\pi(c, c')$ denote the number of 20-foot containers stored between container c and c', the delta evaluation for this swap case is defined by

$$\delta_\pi(\sigma_7, \gamma) = \begin{cases} tb_\pi(c, c') & : t > t' \\ -tb_\pi(c', c) & : \text{ otherwise} \end{cases}, \text{ where } tb_\pi(c, c') = \sum_{m=t}^{t'} \sum_{p \in \mathcal{P}} t(x_{stp}^\pi).$$

Similar to the constraints, the delta evaluation of the objectives can be divided into simple (σ_9 and σ_{10}) and complex (σ_8 and σ_{11}) cases. The delta evaluation for objectives σ_9 and σ_{10} are similar to the weight constraint (σ_6) and are therefore not described further. Here we show how to define the delta evaluation of the overstowage objective (σ_8). The delta evaluation of the pure stack objective (σ_{11}) can be computed in a similar way.

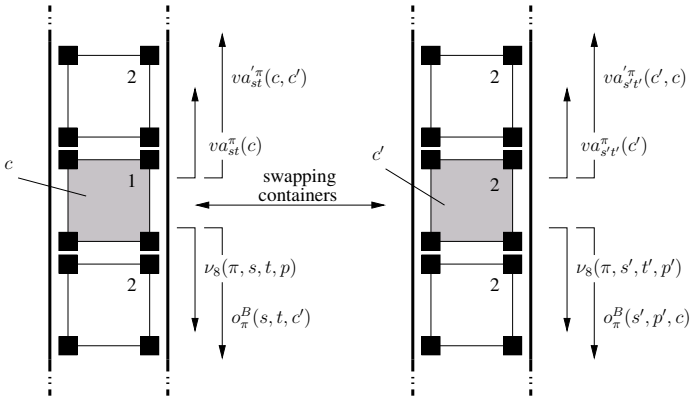


Figure 4: Graphical representation of the delta evaluation of the overstowage objective for a $20' \leftrightarrow 20'$ swap between two distinct stacks.

Consider a single 20-foot to 20-foot swap $\gamma = (\langle s,t,c\rangle, \langle s',t',c'\rangle)$ ($swap^{20}$ type 1) between two distinct stacks. The delta evaluation is computed using the following schema. For each container, we remove the violation it contributes, and we add the violation that it creates when swapped to its new position. A graphical representation of the algorithm is shown in figure 4. Consider for the moment the container c, following the evaluation schema. Its violation $\nu_8(\pi,s,t,p)$ must first be removed, but also the violations of the containers overstowing c. For this purpose we have defined the function $o_\pi(s,t,c)$, which given a cell (s,t) and a container c, counts the number of containers stowed under the specific cell which are overstowed by c. Formally

$$o_\pi(s,t,c) = \sum_{t'=1}^{t-1} \sum_{p' \in \mathcal{P}} ov_\pi(c, x^\pi_{st'p'}), \text{ where } ov_\pi(c,c') = \begin{cases} 0: & \bot(c) \vee \bot(c') \\ 1: & d(c) > d(x^\pi_{st'p'}) \\ 0: & \text{otherwise} \end{cases}.$$

One violation can now be subtracted for each container c^a above c which *only* overstow c (i.e., $d(c^a) > d(c) \wedge o_\pi(s,t,c^a) = 1$), since only in this case will the violation of c^a change to zero when c is removed. This is done using the function $va^\pi_{st}(c) = \sum_{m=t+1}^{N^T} \sum_{n \in \mathcal{P}} \big(\neg\bot(x^\pi_{smn}) \wedge o_\pi(s,m,x^\pi_{smn}) = 1 \wedge d(x^\pi_{smn}) > d(c)\big)$. The next step is adding the violations created by replacing c' with c in stack (s',t'). Here we can use $o_\pi(s',t',c)$ and add one violation if $o_\pi(s',t',c) > 0$. Counting the number of violations caused by c to the containers above c' is a similar to the $va^\pi_{st}(c)$ computation, but we must remember that the current violation is based on c' being present in stack (s',t'). The previous function is thus modified to be $va'^{\pi}_{s't'}(c,c') = \sum_{m=t'+1}^{N^T} \sum_{n \in \mathcal{P}} \neg\bot(x^\pi_{s'mn}) \wedge (o_\pi(s',m,x^\pi_{s'mn}) = 0 \wedge d(x^\pi_{s'mn}) > d(c)) \vee (o_\pi(s',m,x^\pi_{s'mn}) = 1 \wedge d(x^\pi_{s'mn}) > d$
Performing the same operations for c' the complete delta evaluation is given by

$$\begin{aligned} \delta_\pi(\sigma_8,\gamma) = &- \nu_8(\pi,s,t,p) - va^\pi_{st}(c) + o^B_\pi(s',t',c) + va'^{\pi}_{s't'}(c,c') \\ &- \nu_8(\pi,s',t',p') - va^\pi_{s't'}(c') + o^B_\pi(s,t,c') + va'^{\pi}_{st}(c',c), \end{aligned}$$

where $o^B_\pi(s,t,c) = 1$ if $o_\pi(s,t,c) > 0$ and 0 otherwise.

The delta evaluations must be highly efficient operations due to their extensive use in the search procedures. For most of the constraints and objectives (σ_1, σ_2, σ_4, σ_5, σ_6, σ_9, and σ_{10}), the delta swap function run in constant time. This is not the case for the more complex ones (σ_3, σ_7, σ_8, and σ_{11}), but it is easy to show that the complexity in these cases is at most linear in the number of tiers.

4.5. Placement heuristic

The aim of the placement heuristic is to find an initial container assignment. The procedure tries to minimize the objectives and satisfy as many of the constraints as possible. The heuristic is based on simple rules suggested by the nature of the constraints and objectives. Pre-placed containers are assigned to their original position in a pre-processing step, thus implicitly fulfilling constraint (7). Stacks are then filled bottom-up one container at a time, making sure that all containers have support (σ_3). Before the placement, all the containers to be loaded $\mathcal{C} \setminus \mathcal{C}^P$ are sorted using the ordering $\preceq_c$ defined by

$$\begin{aligned} c \preceq_c c' \Leftrightarrow\ & d(c) > d(c') \\ \vee\ & d(c) = d(c') \wedge t(c) \wedge \neg t(c') \\ \vee\ & d(c) = d(c') \wedge (t(c) \Leftrightarrow t(c')) \wedge r(c) \wedge \neg r(c') \\ \vee\ & d(c) = d(c') \wedge (t(c) \Leftrightarrow t(c')) \wedge (r(c) \Leftrightarrow r(c')) \end{aligned}$$

This ordering reduces overstowage by placing first containers with a later discharge port. 20-foot containers are placed before 40-foot containers in an attempt to avoid the placement of 20-foot containers on top of 40-foot containers (σ_7), and reefer containers are assigned before non-reefer since usually reefer slots are at the bottom of a bay (σ_4).

The placement procedure is shown in Algorithm 1. A critical point is the assignment of 20-foot containers, since it is possible to have two 20-foot containers with different discharge ports being assigned to two different stacks and generating odd cells (cells that only contain one 20-foot container). In order to avoid this behaviour, the placement heuristic uses the $oddSlot$ flag, which is raised when a 20-foot container is assigned to a slot in an empty cell. When the flag is raised, the heuristic is forced to place the next 20-foot container in the empty slot of the odd cell (line 3-4 and 15-16). This check ensures that as long as the number of containers to load in the location is consistent with the location capacity, the placement heuristic will always be able to assign all containers to a slot. The procedure places the containers one at a time (line 2), first trying to find space on a stack that contains containers with the same discharge port (line 5-6). If none is found, it tries to find a stack with containers that have a greater discharge port minimising overstowage (line 7-8). If it is not yet possible to find a placement for the container, then an empty stack is chosen (line 9-10). Should none of those cases find a suitable assignment, the container is pushed to the $WAIT\_STACK$ (line 11-12). Once a placement has been tried for each container, the $WAIT\_STACK$ is emptied and each container is placed sequentially on the first available stack (line 13-18). The selection of stacks (lines 5-10) is done in an order $\preceq_s$ based on the number of tiers: $s \preceq_s s' \Leftrightarrow |\mathcal{T}_s| \leq |\mathcal{T}_{s'}|$. This ordering helps the heuristic identify an assignment which uses as few stacks as possible.

4.6. Feasibility Phase

The goal of the feasibility phase is to find an assignment where all constraints are satisfied. This phase will then permit the optimization algorithm to limit the neighbourhood to swaps between feasible solutions. This algorithm is a hill-climbing constraint based local search based on the min-conflict heuristic over the constraint violations. To simplify the description of the algorithm, we define the following functions $\nu^\sigma(\pi, s, t, p) = \sum_{i=1}^{7} \nu_i(\pi, s, t, p)$ and $\delta_\pi^\sigma(\gamma) = \sum_{i=1}^{7} \delta_\pi(\sigma_i, \gamma)$.

The feasibility phase is shown in Algorithm 2. It is important to notice that the swap selection is done through the selection of two slots, where a slot is defined by the triple $\tau = \langle s, t, p \rangle$, where s is the stack, t is the tier, and p is the position of the slot. Let T denote the

Algorithm 1: Placement heuristic

1 **Require:** *all pre-placed containers assigned their given slot,* $\mathcal{C} \setminus \mathcal{C}^P = \{$*all containers to load*$\}$ *are sorted according to the ordering* $\preceq_c$
2 **forall** $c \in \mathcal{C} \setminus \mathcal{C}^P$ **do**
3 **if** $t(c) \wedge oddSlot$ **then**
4 *place* c *in the odd slot*
5 **else if** $\exists s \in \mathcal{S}, t \in \mathcal{T}_s, p \in \mathcal{P} \,.\, \neg\bot(x^{\pi}_{stp}) \wedge d(x^{\pi}_{stp}) = d(c)$ **then**
6 assign c to stack s
7 **else if** $\exists s \in \mathcal{S}, t \in \mathcal{T}_s, p \in \mathcal{P} \,.\, \neg\bot(x^{\pi}_{stp}) \wedge d(x^{\pi}_{stp}) > d(c)$ **then**
8 assign c to stack s
9 **else if** $\exists s \in \mathcal{S}\, \forall t \in \mathcal{T}_s, p \in \mathcal{P} \,.\, \bot(x^{\pi}_{stp})$ **then**
10 assign c to stack s
11 **else**
12 *push(c,WAIT_STACK)*
13 **while** *NOT empty(WAIT_STACK)* **do**
14 $c = pop(\,WAIT\_STACK\,)$
15 **if** $t(c) \wedge oddSlot$ **then**
16 *place* c *in the odd slot*
17 **else**
18 *place* c *in the first available stack*

set of all slot triples. The function $\Gamma(\tau, \tau') : T \times T \mapsto \Gamma$ then defines the swap based on the containers found in the two slots defined by τ and τ'.

Algorithm 2: Feasibility phase

1 sideMove $\leftarrow$ false
2 $\pi = placementHeuristic()$
3 **while** $\sum_{i=1}^{7} \sigma_i > 0$ **do**
4 $\pi' \leftarrow \pi$
5 **selectMax** $s \in \mathcal{S}, t \in \mathcal{T}_s, p \in \mathcal{P}$ ***on*** $\nu^{\sigma}(\pi, s, t, p)$ **do**
6 **selectMin** $s' \in \mathcal{S}, t' \in \mathcal{T}_s, p' \in \mathcal{P}$ ***on*** $\delta^{\sigma}_{\pi}(\gamma = \Gamma(\langle s, t, p\rangle, \langle s', t', p'\rangle))$ **do**
7 $\pi \leftarrow swap(\pi, \gamma)$
8 sideMove $\leftarrow \sigma(\pi') = \sigma(\pi)$
9 **return** π

The algorithm starts with the initial candidate assignment returned by the placement heuristic on line 2, and then performs a local search over the neighbourhood until all the constraints are satisfied (line 3-8). The step function, which selects a new assignment, makes the selection of the slots that will define the swap in two phases. It begins (line 5) by selecting a slot triple $\tau = \langle s, t, p\rangle$ preventing, however, the selection of slots holding pre-placed containers or null containers, and selects only those that actually violate some constraints. Between all the possible slots the one that contains the container with the maximum degree of violation is selected.

The algorithm proceeds (line 6) by selecting another slot triple $\tau' = \langle s', t', p' \rangle$ and prioritises swaps that improve the objective value. The selection in this case filters out all the slots holding pre-placed containers and make sure that the second slot is not the same as the first one. Moreover it selects only slots, which resulting swap actually minimize the violations the most. The *sideMove* flag has been included in the filter in such a way that if raised, it will allow the selection of swaps which will result in an assignment that does not improve the solution, in an attempt to escape the local minima. The $sideMove$ flag is initiated in line 1 and is raised in line 8 if no selection of swap is performed. Should a swap selection have been made, the swap is then actually performed in line 7 using the function $swap(\pi, \gamma)$.

Using a neighbourhood operator that checks all possible swaps given an initial selection, is clearly more expensive in terms of runtime than a more classical one where only a limited number of swaps is evaluated. However this selection has shown to perform moves that are highly valuable in terms of solution improvement, allowing the search to converge quickly.

4.7. Optimality Phase

Once a feasible solution has been found, the optimality phase performs a constraint based local search for the optimal value within the search space of feasible solutions. The optimality phase is a hill-climbing search that makes use of tie-breaking rules to avoid local minima.

For the objectives, in particular overstowage (σ_8), free stack (σ_{10}) and pure stack (σ_{11}), a single swap often does not lead to a change in the objective value. To address this problem, we have defined a tie-breaking function that can evaluate a swap that has no objective value improvement, but still causes a more desirable assignment. The function is called $eval_\pi(o, \gamma)$ and equals the evaluation value of the swap γ for the objective $o \in \{\sigma_8, \sigma_9, \sigma_{10}, \sigma_{11}\}$.

For the overstowage objective (σ_8) the evaluation function is defined as the number of containers overstowed by each container, which will make the algorithm choose a container that overstows many containers over one that only overstows a single container. For the free stack objective (σ_{10}) the swap is evaluated by the number of containers in the stack, so that the search rather swaps containers that are in almost empty stacks. The pure stack objective (σ_{11}) calculates the evaluation by summing the quadratic product of the number of containers with the same discharge port, which will favour those stacks that have the most containers with the same discharge port. The optimality phase uses the function $eval^o_\pi(\gamma) = \sum_{i=8}^{11} eval_\pi(\sigma_i, \gamma)$ as a tie-breaking rule, and similar to the feasibility phase the functions $\nu^o(\pi, s, t, p) = \sum_{i=8}^{11} \nu_i(\pi, s, t, p)$ and $\delta^o_\pi(\gamma) = \sum_{i=8}^{11} \delta_\pi(\sigma_i, \gamma)$ represent the total number of objective violations and the sum of all the delta evaluations of swaps, respectively.

The optimality phase is shown in Algorithm 3. The optimality phase starts with the feasible solution returned by the feasibility phase (line 1). Similarly to the feasibility algorithm, two slots are selected from which a swap then is generated. The first slot is selected between all the non-empty slots holding no pre-placed containers; the one with the maximal objective violation is chosen (line 6). The second slot is selected randomly among all the slots that generate swaps leading to feasible solutions with improved objective value (line 7). Should a swap not be selected, the tie-breaking rule comes into action at line 12, where a new second slot is selected using the tie-breaking rule as defined by the evaluating function on non-improving swaps. The counter $sideMove$ is used to limit the number of side moves that the algorithm may perform. The local search terminates once the maximum number of side moves is reached (line 4).

Algorithm 3: Optimality phase

```
1  π = feasibilityPhase()
2  changed = true
3  sideMove = 0
4  while changed ∧ sideMoveCount < MAX_SIDEMOVES do
5      changed = false
6      selectMax s ∈ S, t ∈ T_s, p ∈ P on ν^o(π, s, t, p) do
7          select s' ∈ S, t' ∈ T_s, p' ∈ P do
8              changed = true on δ^o_π(γ = Γ(⟨s, t, p⟩, ⟨s', t', p'⟩))
9              π = swap(π, γ)
10             sideMove = 0
11         if ¬changed then
12             select s' ∈ S, t' ∈ T_s, p' ∈ P on eval^o_π(γ = Γ(⟨s, t, p⟩, ⟨s', t', p'⟩)) do
13                 sideMove ← sideMove + 1
14                 changed = true
15                 π = swap(π, γ)
16 return π;
```

5. Experimental Evaluation

The algorithm has been implemented in C++ and all experiments have been conducted on a Linux system with 8 Gb RAM and 2 Opteron Quad Core CPUs, each running on 1.7 GHz and having 2Mb of cache. A preliminary implementation of this model with a similar algorithm has been done in in the COMET language (Pacino and Jensen, 2009). An order of magnitude speedup was achieved by the current re-implementation in C++.

The proposed algorithm has been evaluated on a set of instances derived from stowage jobs provided by our liner shipping industrial collaborator. The instances are generated from real stowage examples from a wide range of vessels. Thus, the number of containers and their type distribution correspond to the under deck stowage problems faced by the industry. We have divided the instances into a *test set* and a *training set*. The test set is composed of 133 instances, including locations between 6 TEUs and 220 TEUs. Table 2 gives a summarized overview of these instances.

The training set, has been used to tune the algorithm parameters. The parameter tuning session, which results can be seen in Table 3, used a combination of time, variance and solution quality to tune the parameters. This session included the tuning of the number of parallel restarts and the maximal number of side moves to end the optimality phase. The result of the parameter tuning session is that the algorithm performs best using as few as 50 side moves and 8 parallel restarts. This combination of parameters gives a small variance on high quality solutions within a short time.

The experimental results on the test data set, which can be seen in Table 4(a), show the percentage of solutions solved within a specific optimality gap (optimal solutions have been generated using a modification of the constraint programming algorithm described in Delgado et al. (2009)). It is easy to see that only few instances diverge from near optimality and that in 86% of the cases the algorithm actually reached the optimal solution.

Studying the algorithm performance closer, we were able to gain some insight on the quality

Class	40'	20'	Reefer	HC	DSP>1	Inst.
1	√					6
2		√				18
3	√	√				4
4	√			√		42
5	√	√		√		27
6	√		√	√		8
7	√	√	√	√		7
8	√			√	√	7
9	√	√		√	√	10
10	√		√	√	√	2
11	√	√	√	√	√	2

Table 2: *Test Set Characteristics*. The first column is an instance class ID. Column 2, 3, 4, and 5 indicate whether 40-foot, 20-foot, reefer, and high-cube containers are present. Column 6 indicates whether more than one discharge port is present. Finally, column 7 is the number of instances of the class.

of the different phases. The heuristic placement does not take the height and weight constraint into account, leaving space for improvements. However Table 4 (c) and (d) show that the solution found by the heuristic placement is not far from being feasible in most of the cases. This is supported by the fact that often feasibility is reached within 20 iterations and that 61% of the time, the objective value is not compromised. The quality of the objective value of the first feasible solution is also optimal in 74% of the cases (Table 4(b)), suggesting that the heuristic placement procedure is performing well. The results also support our initial hypothesis suggesting that the problem is under-constrained and that as such it is possible to heuristically find high quality solutions in short time. The results also point to the fact that the optimality phase improves only a limited number of instances, which however is important especially in the cases where the optimality gap after the feasibility phase is more than 20%.

The limited improvement of solutions in the optimality phase probably happens because the search does not allow for a large degree of diversification. Preliminary tests using tabu search have shown that local diversification often was unsuccessful to escape a local minimum due to large structural differences between the local minimum and an optimal solution. A possible approach to solve this problem could be to change the initial placement and the search procedures to follow the heuristic less closely. This method, however, would probably be more expensive in terms of runtime performance.

It is important to remark that two of the test instances were not solved to feasibility. This was also due to the strict following of the min-conflict heuristic during the feasibility phase. In an industrial implementation of the algorithm, this problem can be solved by addressing a number of special cases.

All the stow-plans for the locations have been calculated in an average time of 0.18 seconds, with a worst case of 0.65 seconds. Figure 5(a) shows the runtime of the algorithm as a function of the size of the instance measured in TEUs. As depicted, the execution time scales well with the instance size. Figure 5(b) compares the execution time between our algorithm and the complete constraint programming approach used for generating optimal solutions. When generating the instance set for investigating the optimality gap shown in Table 4, we excluded instances that were not solvable by the CP approach within 160 seconds. However, in a comparison between the two approaches these instances are particularly interesting.

As depicted, the CP approach is highly competitive within the set of instances that it can

Side moves	Parallel Restarts			
	1	**2**	**3**	**4**
25	(0.02 , 241.9 , 0.51)	(0.04 , 240.4 , 1.09)	(0.05 , 237.8 , 0.46)	(0.05 , 237.0 , 0.32)
50	(0.03 , 241.5 , 0.00)	(0.07 , 237.0 , 1.42)	(0.09 , 235.7 , 0.09)	(0.11 , 236.3 , 0.27)
75	(0.06 , 239.2 , 0.57)	(0.11 , 236.0 , 0.02)	(0.14 , 235.9 , 0.03)	(0.18 , 235.8 , 0.06)
100	(0.08 , 238.0 , 0.00)	(0.13 , 236.0 , 0.00)	(0.18 , 235.3 , 0.90)	(0.24 , 235.5 , 0.14)
150	(0.10 , 236.4 , 0.51)	(0.19 , 235.6 , 0.13)	(0.25 , 235.5 , 0.16)	(0.34 , 234.6 , 0.44)
200	(0.12 , 236.3 , 0.54)	(0.25 , 235.3 , 0.22)	(0.32 , 235.8 , 0.11)	(0.42 , 235.9 , 0.02)
250	(0.16 , 238.0 , 0.63)	(0.30 , 235.8 , 0.09)	(0.41 , 234.9 , 0.77)	(0.53 , 235.5 , 0.16)
300	(0.17 , 268.0 , 0.65)	(0.36 , 235.1 , 0.27)	(0.48 , 235.6 , 0.11)	(0.63 , 235.6 , 0.98)
400	(0.22 , 235.8 , 0.06)	(0.46 , 236.0 , 0.13)	(0.64 , 234.9 , 0.35)	(0.83 , 235.3 , 0.05)
	5	**6**	**7**	**8**
25	(0.07 , 237.0 , 0.47)	(0.08 , 236.8 , 0.70)	(0.09 , 237.3 , 0.49)	(0.10 , 236.8 , 0.32)
50	(0.13 , 235.5 , 0.16)	(0.16 , 235.8 , 0.05)	(0.19 , 235.0 , 0.14)	(0.22 , 235.4 , 0.03)
75	(0.22 , 235.4 , 0.19)	(0.25 , 235.4 , 0.92)	(0.30 , 235.7 , 0.08)	(0.34 , 235.4 , 0.03)
100	(0.28 , 235.8 , 0.09)	(0.33 , 235.2 , 0.25)	(0.38 , 235.7 , 0.06)	(0.44 , 234.7 , 0.25)
150	(0.40 , 235.6 , 0.13)	(0.46 , 235.1 , 0.27)	(0.53 , 235.0 , 0.16)	(0.62 , 234.7 , 0.70)
200	(0.51 , 234.2 , 0.57)	(0.60 , 234.1 , 0.52)	(0.71 , 235.9 , 0.03)	(0.81 , 235.3 , 0.05)
250	(0.64 , 234.6 , 0.68)	(0.74 , 235.7 , 0.06)	(0.86 , 234.1 , 0.52)	(0.98 , 234.6 , 0.27)
300	(0.75 , 234.5 , 0.46)	(0.88 , 234.8 , 0.38)	(1.03 , 235.0 , 0.14)	(1.17 , 234.8 , 0.36)
400	(0.98 , 235.8 , 0.09)	(1.15 , 235.0 , 0.14)	(1.34 , 235.5 , 0.16)	(1.55 , 234.7 , 0.41)

Table 3: *Parameter Tuning*. Each cell holds a triple $\langle t, q, v \rangle$ where t is the average time, q the average solution cost and v the average cost variance. The highlighted cell indicates the parameter selection.

solve in 160 seconds. However our approach can solve the problematic instances for CP fast as well. This robustness is important in practice if we want to guarantee that "good" solutions can be returned within a second. Since a single vessel often contains in the order of 100 locations there would be a high probability of spending more than 100 seconds on a few locations if generating solutions solely with CP. A pragmatic solution would be to execute the two approaches in parallel and rely on the optimal CP solution for the locations where it can be generated fast.

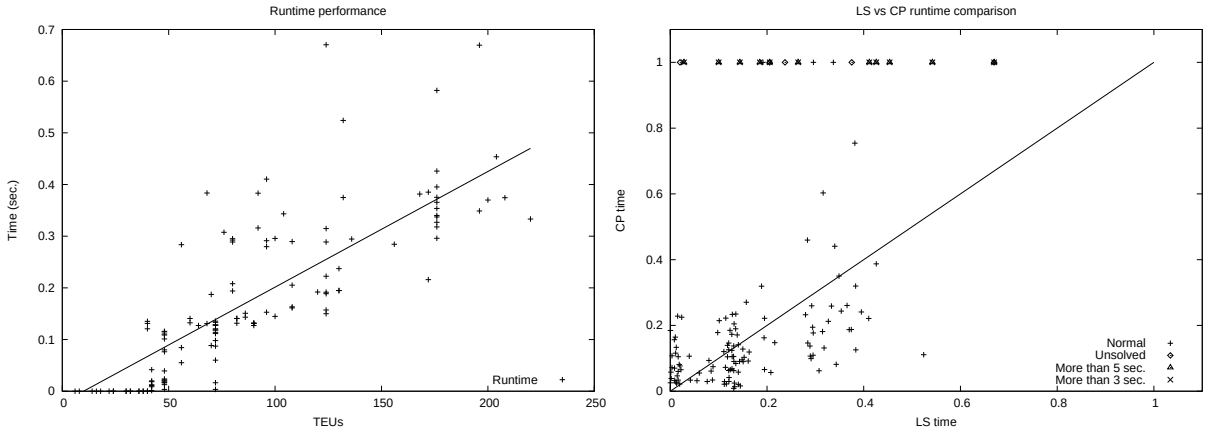
6. Related work

Both academic and commercial organisations have contributed to model and identify methods to solve the Container Stowage Problem (CSP). Modeling of the CSPUDL can be found embedded in larger models for the CSP in the early work of Botter and Brinati (1992). This model, however, does not consider high-cube containers and does not allow the allocation of 20-foot and 40-foot containers in the same stack. Pre-placed containers are not considered either since the vessel is assumed to be fully emptied at each port. Only overstowage is modelled as objective, which is also the case for all other work discussed here except Wilson et al. (2001) and Ambrosino et al. (2009) which we discuss below. Using an implicit enumeration Botter and Brinati (1992) claim to solve CSPs of 740 TEUs , however no experimental results have been provided. Avriel et al. (1998) has contributed a simplified model where only one type of containers was handled and where reefer and high-cube containers were not taken into consideration. Without considering the weight constraints, Avriel et al. (1998) designed a heuristic which was able to solve problems of 1700 TEUs in 30 seconds. This performance is, however, hard to evaluate if applied to the range of constraints and objectives of CSPUDL. Giemsch and Jellinghaus (2003) recently improved the solutions to the model defined by Avriel et al. (1998) using mixed-integer programming. But no concrete computational results were pub-

Opt. Gap		**Opt. Gap (Feas.)**		**Feas. Iter.**		**Feas. Worsening**	
Gap.	Freq.	Gap.	Freq.	Iter. Up to	%	Worsening %	%
0%	86%	0%	74%	0	29%	-20%	2%
1%	2%	5%	6%	5	23%	-10%	2%
2%	2%	10%	2%	10	18%	-5%	4%
3%	2%	15%	5%	15	8%	0%	61%
4%	1%	20%	3%	20	6%	5%	8%
10%	1%	25%	3%	25	3%	10%	2%
15%	4%	35%	3%	30	6%	20%	7%
20%	1%	40%	1%	35	1%	$> 20\%$	15%
30%	2%	$> 100\%$	3%	40	2%		
				45	2%		
				50	1%		
				65	2%		
(a)		(b)		(c)		(d)	

Table 4: *Algorithm Analysis* (a) Cost gap between returned solution and optimal solution. (b) Cost gap between feasibility phase solution and optimal solution. (c) Number of iterations needed to find a feasible solution in the feasibility phase. (d) Worsening of the cost of the heuristic placement when making it feasible in the feasibility phase.

lished. Wilson et al. (2001) use a hierarchical decomposition strategy for the CSP and solves a problem similar to the CSPUDL using tabu search. Their article does mention the constraints of the CSPUDL, but they are not formally definde. Compared to other work, Wilson et al. (2001) introduce the pure stack objective to complement the overstowage objective. But the free stack and reefer objective of CSPUDL are not considered. Moreover, the tabu search is not described in detail, probably due to a commercial nature of the author's industrial collaboration. Concrete results have also been omitted from the article. The authors mention that locations in the test cases are of 12-60 TEUs and that the entire CSP can be solved in 90 minutes. This is well beyond the limit of 15 minutes for automated stowage planning to be of practical value for stowage coordinators. Kang and Kim (2002) also proposes a decomposition approach where the container to holds (locations) assignment is solved using implicit enumeration. The model proposed, however, lacks the handling of reefer and high-cube containers and only 40-foot containers are taken into consideration. The authors claim to solve the entire CSP in 5 minutes considering a vessel with 20 bays (no size in TEUs of the vessel is given). The simplifications made to the model makes it hard to apply to industrial instances. The same year Dubrovsky et al. (2002) published a genetic algorithm approach again on a simplified version of the problem where reefer and high-cube containers are not handled and only one type of container is taken into consideration. The method was able to solve vessels of 1000 TEUs within 30 minutes, which is too extensive compared to the size of the vessel. Moreover, such simplified models cannot be used for industrial instances. Ambrosino et al. (2004) propose a heuristic procedure for a problem similar to the CSPUDL except that it does not handle reefer and pre-placed containers and only includes the overstow objective. They are only able to solve a CSP of 188 TEUs in 718 seconds. Based on the same model Ambrosino et al. (2006) improve the runtime of the problem using a 0/1 linear program for the bay assignment problem, which solves the CSP in 383 seconds, which is still too much for a vessel of only 188 TEUs. Successively Ambrosino et al. (2009) improved the runtime further by solving only a simplified version of the CSPUDL where each bay can only allocate one port of discharge and where



(a) (b)

Figure 5: *Algorithm Performance*. (a) Execution time as a function of instance size measured in TEUs. (b) Execution time comparison between our algorithm and the complete constraint programming approach used to generate optimal solutions.

weight groups of containers are assigned rather than single containers. All the bay assignments on a vessel of up to 1800 TEUs have been solved in 37.9 seconds. The model simplifications and the removal of the overstowage objective, however, makes this model unsuitable to be used on large industrial instances. Imai et al. (2006) propose a model which only handles one type of containers and where reefer and high-cube containers are not taken into consideration. Solutions to CSPs of 504 TEUs can be found with a genetic algorithm in 5883 seconds. The authors however take also into consideration the minimization of overstowage in yards which makes it hard to evaluate the runtime of the algorithm. Furthermore, they suggest that a simpler version of the algorithm should be used on industrial instances, but no results of its runtime are given. An IP approach was devised by Li et al. (2008) on a model that disregarded reefer, high-cubes and pre-placed containers. Constraints such as no 20-foot containers on top of 40-foot containers are not handled. The paper mentions test instances of 800 TEUs but no results have been published. Another decomposition approach for the CSP was published by Gumus et al. (2008) where a problem similar to the CSPUDL was solved. The model, however, is scars in details and seems to disregard high-cube containers. No results or details of the method have been published.

7. Conclusion

In this article, we have introduced a representative problem model called the CSPUDL for assigning containers to slots in under deck storage areas of container vessels. The CSPUDL has been developed in close collaboration with a larger liner shipping company since 2005 and is to our knowledge the currently most accurate of its kind. The CSPUDL is a sub-problem of hierarchical decompositions of stowage planning which are currently the most efficient. Due to the high-level constraints and objectives in stowage planning that cluster similar containers in bays, we hypothesise that the CSPUDLs solved by hierarchical algorithms are widely under-constrained. To test this hypothesis, we have introduced a 3-phase constraint based local search algorithm for solving the CSPUDL. Details of the algorithm in particular of incremental algorithms for fast neighbourhood evaluation have been presented. The approach has been tested on naturally distributed real instances from the industry and is able to find high quality solutions within an average runtime of 0.18 seconds. The experimental evaluation supports our

hypothesis and as expected a complete method based on constraint programming is unable to robustly find good solutions fast. Our results suggest that an industrial system should race local search and complete algorithms in parallel to achieve both near optimality and time robustness. Further research is needed to develop these algorithms and complete the CSPUDL definition (e.g., to handle IMO, OOG and palled-wide containers).

References

D. Ambrosino, A. Sciomachen, E. Tanfani, Stowing a containership: the master bay plan problem, Transportation Research Part A: Policy and Practice 38 (2) (2004) 81–99.

R. Botter, M. Brinati, Stowage Container Planning: a Model for Getting an Optimal Solution, Computer Application in the Automatio of Shipyard Operation and Ship Design VII (C) (1992) 217–229.

P. Giemsch, A. Jellinghaus, Optimization Models for the Containership Stowage Problem, Operations Research Proceedings .

O. Dubrovsky, G. Levitin, M. Penn, A Genetic Algorithm with Compact Solution Encoding for the Container ship Stowage Problem, Journal of Heuristics 8 (2002) 585–599.

M. Avriel, M. Penn, N. Shpirer, S. Witteboon, Stowage planning for container ships to reduce the number of shifts, Annals of Operations Research 76 (1998) 55–71.

D. Ambrosino, A. Sciomachen, E. Tanfani, A decomposition heuristics for the container ship stowage problem, Journal of Heuristics 12 (3) (2006) 211–233.

J. Kang, Y. Kim, Stowage Planning in Maritime Container Transportation, Journal of the Operational Research Society 53 (4) (2002) 415–426.

I. Wilson, P. Roach, Principles of Combinatorial Optimisation Applied to Container-ship Stowage Planning, Journal of Heuristics 5 (1999) 403–418.

D. Ambrosino, D. Anghinolfi, M. Paolucci, A. Sciomachen, A new three-step heuristic for the Master Bay Plan Problem, Maritime Economics and Logistics 11 (1) (2009) 98–120, URL `http://ideas.repec.org/a/pal/marecl/v11y2009i1p98-120.html`.

L. Michel, P. V. Hentenryck, A constraint-based architecture for local search, ACM SIGPLAN Notices 37 (11) (2002) 101–110, ISSN 0362-1340.

A. Delgado, R. M. Jensen, C. Schulte, Generating Optimal Stowage Plans for Container Vessel Bays, in: I. Gent (Ed.), Fifteenth International Conference on Principles and Practice of Constraint Programming, vol. 5732 of *Lecture Notes in Computer Science*, Springer-Verlag, 6–20, 2009.

S. Minton, M. D. Johnston, A. B. Philips, P. Laird, Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems, Artif. Intell. 58 (1-3) (1992) 161–205.

D. Pacino, R. M. Jensen, A Local Search Extended Placement Heuristic for Stowing Under Deck Bays of Container Vessels, in: Proceedings of ODYSSEUS2009, 2009.

I. Wilson, P. Roach, J. Ware, Container stowage pre-planning: using search to generate solutions, a case study, Knowledge-Based Systems 14 (2001) 135–145.

A. Imai, K. Sasaki, E. Nishimura, S. Papadimitriou, Multi-objective simultaneous stowage and load planning for a container ship with container rehandle in yard stacks, European Journal of Operational Research 171 (2) (2006) 373–389.

F. Li, C. Tian, R. Cao, W. Ding, Computational Science ICCS 2008, vol. 5101/2008 of *Lecture Notes in Computer Science*, chap. An Integer Linear Programming for Container Stowage Problem, Springer Berlin / Heidelberg, 853–862, 2008.

M. Gumus, P. Kaminsky, E. Tiemroth, M. Ayik, A Multi-stage Decomposition Heuristic for the Container Stowage Problem, in: Proceedings of the 2008 MSOM Conference, 2008.