Generating Optimal Stowage Plans for Container Vessel Bays

Alberto Delgado[1], Rune Møller Jensen[1], and Christian Schulte[2]

[1] IT University of Copenhagen, Denmark
{alde,rmj}@itu.dk
[2] KTH - Royal Institute of Technology, Sweden
cschulte@kth.se

Abstract. Millions of containers are stowed every week with goods worth billions of dollars, but container vessel stowage is an all but neglected combinatorial optimization problem. In this paper, we introduce a model for stowing containers in a vessel bay which is the result of probably the longest collaboration to date with a liner shipping company on automated stowage planning. We then show how to solve this model efficiently in - to our knowledge - the first application of CP to stowage planning using state-of-the-art techniques such as extensive use of global constraints, viewpoints, static and dynamic symmetry breaking, decomposed branching strategies, and early failure detection. Our CP approach outperforms an integer programming and column generation approach in a preliminary study. Since a complete model of this problem includes even more logical constraints, we believe that stowage planning is a new application area for CP with a high impact potential.

1 Introduction

More than 60% of all international cargo is carried by liner shipping container vessels. To satisfy growing demands, the size vessel has increased dramatically over the last two decades. This in turn has made the traditional manual stowage planning of the vessels very challenging. A container vessel stowage plan assigns containers to slots on the vessel. It is hard to generate good stowage plans since containers cannot be stacked freely due to global constraints like stability and bending forces and many interfering local stacking rules over and under deck.

Despite of the importance of stowage planning, the amount of previous work is surprisingly scarce. In the last two decades, less than 25 scientific publications have been made on the topic and there only exists two patents. The early approaches were "flat" in the sense that they introduced a decision variable or similar for each possible slot assignment of the containers (e.g.,[1],[2]). None of these scale beyond small feeder vessels of a few hundred 20-foot containers. Approaches with some scalability are heuristic (e.g., [3],[4],[5]) in particularly by decomposing the problem hierarchically (e.g., [6],[7],[8],[9]). None of these techniques, though, have been commercialized. They are either too slow or neglect important aspects of the problem due to little contact with industry experts.

I.P. Gent (Ed.): CP 2009, LNCS 5732, pp. 6–20, 2009.

We have since 2005 collaborated closely with a large liner shipping company that has developed an efficient hierarchical stowage planning algorithm using a more accurate domain model than any published work. An important sub-problem of this algorithm and other hierarchical algorithms is to assign a set of containers in a vessel bay. One of our objectives has been to compare different optimization techniques for this sub-problem. To this end, we have first defined the complete set of constraints and objectives used by our industrial partner and then constructed a simplified problem with a representative subset of these for an under deck bay. We have investigated incomplete methods based on local search (e.g., [10]) and evaluated these using complete methods.

In this paper, we introduce an optimal CP approach which to our knowledge is the first application of CP to container vessel stowage planning, to solve the sub-problem mentioned above. We present our stowage model and show how to solve it efficiently using Gecode [11]. State-of-the-art modeling techniques are considered including: different viewpoints to achieve better propagation, extensive use of global constraints to avoid modeling with boolean variables, and static and dynamic symmetry breaking. In addition, we use a branching strategy that takes advantage of the structure of the problem and a set of early failure detection algorithms that determines whether a partial assignment is inconsistent. The CP approach presented in this paper has been successfully tested on industrial data. Our experimental evaluation shows that the modeling decisions we made, in particularly the ones related to early failure detection, improve computation times substantially. The definition of the problem and model introduced in this paper have been slightly modified in order to make them easy to understand. We consider, though, that this simplification does not make less relevant the results here presented. Interestingly, preliminary results on the original version of the problem shows that a less elaborated CP model outperforms two optimal approaches based on Integer Programming (IP) [12] and Column Generation (CG). We believe this to be due to the logical nature of local stacking rules and objectives of stowage planning that mathematical programming is unable to handle efficiently. Thus, we consider CP to be the most efficient general technique to solve these problems optimally and, it gave us the main motivation to upgrade the initial CP model to the one presented here.

The remainder of the paper is organized as follows. Section 2 describes stowage planning problems. Section 3 defines our CP model. Section 4 describes why we believe CP outperforms mathematical programming on this problem. Finally, Section 5 presents experimental results, and Section 6 draws conclusions and discusses directions for future work.

2 The Container Stowage Problem for an Under Deck Location

A container vessel is divided in sub-sections called *bays*, each bay of a container vessel consists of over and under deck *stacks* of containers. A *location* is a set of stacks that can be over or under deck. These stacks are not necessarily consecutive, but all stacks in the set are either over or under deck. Left figure in Fig.1

depicts a bay. The stacks 3, 4 and 5 under deck form a location, stacks 1, 2, 6 and 7 under deck form another location. The same stacks form two extra locations in the over deck section. This paper focuses on the under deck locations. The vertical alignments of cells in a location are called *tiers*. Left figure in Fig.1 shows how the tiers are enumerated for each section of the bay.

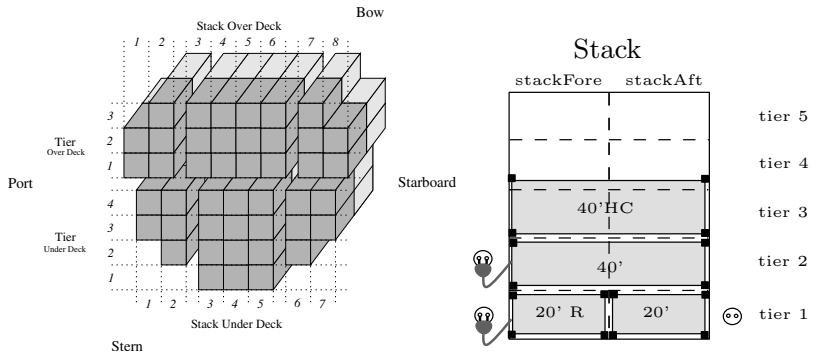

Fig. 1. To the left a back view of a bay, to the right a side view of a partially loaded stack. Each plug in the right figure represents a reefer slot, reefer containers are the ones with electric cords.

Each stack has a weight and height limit that must be satisfied by the containers allocated there. A stack can be seen as a set of piled *cells* one on top of the other. Each of these cells is divided in two *slots*, *Fore* and *Aft*. It is also possible to refer to the *Fore* and *Aft* part of a stack, i.e. *stackFore* refers to the *Fore* slots of all cells in a stack . Some slots have a plug to provide electricity to the containers, in case their cargo needs to be refrigerated. Such slots are called *reefer* slots. Right figure in Fig.1 shows the structure of a stack. As depicted in left figure in Fig.1, it is common for stacks not to have all slots physically available, they must fit into the layout of the vessel and some of the slots must be taken away to do so. These slots can be located either in the bottom or in the top of the stack and we refer to them as *Blocked slots*.

A *container* is a box where goods are stored. Each container has a weight, height, length, port where it has to be unloaded (discharge port) and indicates whether it11 needs to be provided with electric power (reefer). In an under deck location, containers have 20 or 40-foot length and 8'6" or 9'6" height. The weight is limited according to the length of the container and the discharge port depends on the route of the vessel. Containers that are 9'6" high are called *high-cube* containers, and according to the definition of the problem all high-cube containers are 40-foot long. Each cell in a stack can hold one 40-foot container or two 20-foot containers.

In order to generate stowage plans for complete vessels an efficient hierarchical algorithm has been developed. This algorithm decomposes the process of generating stowage plans into solving two derived problems: a master and a sub

problem. The master problem focuses on constraints over the complete vessel i.e. stability constraints, bending forces, etc. and distributes containers in the different locations of the vessel but it does not assign them to a specific slot. Here, all containers to be loaded in the actual port are considered, together with forecasting information of further ports on the route of the vessel.

The sub-problem finds stowage plans for the locations of the vessel according to the distribution of containers made by the master problem. The constraints here are mostly stack wise and each container is assigned to an specific slot. There are two main types of locations in a vessel, over and under deck. Besides their position on the vessel, they differ in the constraints that the containers allocated there must fulfilled. As mentioned before, this paper focuses on finding stowage plans for the under deck locations.

The Container Stowage Problem for an Under Deck Location (*CSPUDL* is defined by the following constraints and objectives. A feasible stowage plan for an under deck location must satisfy the following constraints:

1. Assigned cells must form stacks (containers stand of top of each other in the stacks. They can not hang in the air).
2. 20-foot containers can not be stacked on top of 40-foot containers.
3. A 20-foot reefer container must be placed in a reefer slot. A 40-foot reefer container must be placed in a cell with at least one reefer slot, either Fore or Aft.
4. The sum of the heights and weights of the containers allocated in a stack are within the stack limits.

Every allocation plan that satisfies these constraints is valid, but since the problem we are solving here is to find the best allocation plan possible, a set of objectives must be defined to evaluate the quality of the solutions:

1. **Minimize overstows.** A container is stored above another container if it is stored in a cell with a higher tier number. A container A overstows a container B in a stack, if A is stored above B and the discharge port of A is after the one of B, such that A must be removed in order to unload B. A cost is paid for each container overstowing any other containers below.
2. **Keep stacks empty if possible.** A cost is paid for every new stack used.
3. **Avoid loading non reefer container into reefer cells.** A cost is paid for each non reefer container allocated in a reefer cell.

The first objective is directly related to the economical costs of a stowage plan. The second and third are rules of thumb of the shipping industry with respect to generating allocation plans for further ports in the route of a vessel. Using as few stacks as possible increases the available space in a location and reduce the possibility of overstowage in further ports. Minimizing the reefer objective allows reefer containers to be loaded also in further ports.

A feasible solution to the *CSPUDL* satisfies the constraints above. An optimal solution is a feasible solution that has minimum cost.

As a requirement from the industry, generating a stowage plan for a vessel should not take more than 10 minutes. Since a big vessel can have an average

of one hundred locations, solving the *CSPUDL* as fast as possible is mandatory for this problem. An average of one second or less per location has been set as goal for solving the *CSPUDL*.

3 The Model

We present here a constraint programming model to find the optimal allocation plan for a set of containers $Containers$ in a location l. In order to make the description of the model clearer, some constants are defined: $Slots$ and $Stacks$ are the set of slots and stacks of location l. $stackFore_i$ and $stackAft_i$ are the set of Fore and Aft slots in stack i. The weight and height limit of a stack i are represented by $stack_i^w$ and $stack_i^h$. Without loss of generality and in order to simplify some of the constraints, $stack_i$ refers to the set of cells from stack i. Every time a constraint is posted over a cell of $stack_i$, the constraint is actually posted over the Aft and Fore slots of the cell.

The set of decision variables of the problem is defined first. To improve propagation, we implement two different viewpoints[13] and channel them together such that both sets of variables contain the same information all the time.

The first viewpoint is the set of variables S, where each variable corresponds to a $slot_i \in Slots$ from location l. The second one is the set of variables C, where each variable represents a container from $Containers$.

$$S = \{s_i | i \in Slots\} \wedge s_i \in \{Containers\}, \forall i \in Slots$$
$$C = \{c_i | i \in Containers\} \wedge c_i \in \{Slots\}, \forall i \in Containers$$

In order to connect the two viewpoints, it is necessary to define a set of channeling constraints. Since the number of containers is not guaranteed to be equal to the number of slots in location l, we consider two possible alternatives to modeling the channeling. The first one is to declare a new set of boolean variables $C \times S = \{c \times s_{ij} | i \in Slots, j \in Containers\}$, where $c \times s_{ij} \leftrightarrow c_j = i \wedge s_i = j$, that channels set S and C, and add to the domain of each variable in S the value 0 to represent an empty slot. The second one is to extend the number of containers to match the number of slots of l, and define a single global channeling constraint in order to propagate information from one model to the other.

The main difference between these two approaches is how and when the information is propagated among the two viewpoints. Since the first approach uses boolean variables to channel the two models, the flow of information is limited to reflect assignment of variables from one viewpoint to the other.

In the second approach, since there is a channeling constraint connecting the two viewpoints, any update in the domains of the variables are propagated among viewpoints as they occur, increasing the levels of propagation. An extra advantage of using the second approach is that the alldifferent constraint is implied here, all containers must be allocated in a different slot, and all slots must hold different containers.

Our model implements the second approach. Artificial containers are added to the original set of containers to match the number of slots in l. Since a 40-foot container occupies two slots, these containers are split into two smaller containers of the size of a slot each: Aft40 and Fore40. All 40-foot containers are removed from the original set of containers and replaced by the new Aft40 and Fore40 containers. Empty containers are also added, they will be allocated in valid slots where no container is placed. Finally, it is necessary to add some extra containers that will be allocated in the blocked slots. Once the number of containers matches the slots of l, a global channeling constraint is used to connect the two view points, i.e. $channeling(S, N)$.

The first two constraints from the previous section describe how containers can be stacked according to physical limitations of the problem. They define the valid patterns the containers in stacks can form. We assign a code to each type of container, i.e. 0 to blocked containers, 1 to 20-foot containers, 2 to 40-foot containers and, 3 to empty containers, and define a regular expression that recognizes all the well-formed stacks according to the first two constraints: $R = \{r|r \in 0^*1^*2^*3^*0^*\}$. Then we define a constraint just allowing stacks accepted by R. In order to do this, a new set of auxiliary variables must be defined. These variables will represent the type of the container allocated in a slot, and their domain is the set of possible types for the containers: $T = \{t_i|i \in Slots\}$, $t_i \in \{0, 1, 2, 3\}$. To bind this new set of variables with one of the viewpoints, it is necessary to declare an array of integers *types* representing the type associated to each container, and use element constraints such that: $types[s_i] = t_i, \forall i \in Slots$.

With the new set of auxiliary variables defined, we proceed to declare the constraints that will just allow well-formed stacks. A regular constraint [14] is declared for each Aft and Fore stack, together with the regular expression R that defines the well-formed stacks. In this constraint $stack_i$ refers to the subset of variables from T in stack i.

$$regular(stack_i, R), \forall\ i \in Stacks$$

For the reefer constraint two subsets of containers are defined: $\neg RC$ and $\neg 20RC$. $\neg RC$ is the subset of non-reefer containers and $\neg 20RC$ is a subset containing 40-foot, 40-foot reefer containers and 20-foot containers. The purpose of these two subsets is to restrict the domain of some of the slots of location l to allocate just the allowed containers. The first subset of slots is $\neg RS$, which are the slots that are non-reefer and that are not part of reefer cells. The second subset is RCS, which are slots that are non-reefer but that are part of a reefer cell. Then we remove the reefer containers from slots where it is not possible to allocate any reefer containers at all, and remove the 20-foot reefer containers from slots where it is possible to allocate part of a reefer container.

$$s_i \in \neg RC, \forall\ i \in \neg RS \wedge s_i \in \neg 20RC, \forall\ i \in RCS$$

Some extra sets of auxiliary variables are used in order to model the height and weight limit constraints for each stack in l. H is a set of variables where each h_i represents the height of the container allocated in s_i, W represents the same as

H but with respect to the weight of the container. Both sets of auxiliary variables are bound to S with element constraints, as it was previously explained for T. An extra set of variables is also declared here: $HS = \{hs_i | i \in Stacks\}$, $hs_i \in \{0, ..., stack_i^h\}, \forall i \in Stacks$, representing the height of each stack in location l. The constraints restricting the height and weight load of each stack in l are:

$$\sum_{j \in stackAft_i} h_j \leq hs_i, \forall i \in Stacks$$

$$\sum_{j \in stackFore_i} h_j \leq hs_i, \forall i \in Stacks$$

$$\sum_{j \in stack_i} w_j \leq stack_i^w, \forall i \in Stacks$$

3.1 Objectives

The first objective is overstowage. It is based on a feature of the containers that is not related to any previous constraint, the discharge port. A new set P of auxiliary variables is introduced here, where each p_i represents the discharge port of the container allocated in s_i. A new function is defined: $bottom : Stack \rightarrow Slots$, which associates each stack with its bottom slot. The finite domain variable Ov captures the number of overstows in location l.

$$Ov = \sum_{i \in Stacks} \sum_{j \in stack_i - \{bottom(i)\}} \Big(\sum_{k=bottom(i)}^{j-1} (P_j > P_k) > 0 \Big)$$

Since empty and blocked containers have discharge port 0, we use the previously declared set of auxiliary variables P to determine the number of stacks used in a solution. When a stack i is empty, the sum of the values assigned to the subset of P variables in i should be 0, otherwise the stack is been used. A finite domain variable Us captures the number of used stacks in location l.

$$Us = \sum_{i \in Stacks} \Big((\sum_{j \in stack_i} P_j) > 0 \Big)$$

A check over the reefer slots is performed, the reefer objective increases its value if a non-reefer container is allocated in a reefer slot. A finite domain variable Ro captures the number of non-reefer containers allocated in reefer slots.

$$Ro = \sum_{i \in RS} (s_i \in \neg RC)$$

3.2 Branch and Bound

The relevance of the objectives defined for this problem is given by the relation $Ov > Us > Ro$. A lexicographic order constraint is used for the branch and

bound search procedure to prune branches not leading to any better solution. Our model does not rely on an objective function to measure the quality of the solutions but on an order over the objective variables, which provides us with a stronger propagation. The branch and bound approach constraints the objective value of the next solution to be better than the one found so far. When this objective value is calculated by a mathematical function, the only constraint branch and bound posts is a relational constraint over this objective value, considerably reducing the amount of backwards propagation that can be achieved. In cases where an order determines the quality of the solution, lexicographic constraints can be used, which in most of the cases, propagate directly over each objective variable if necessary, increasing the level of backwards propagation achieved.

3.3 Branching Strategies

In our branching strategy we take advantage of the structure of the problem and the set of auxiliary variables defined in the model in order to find high-quality solutions as early as possible. We decompose the branching in three sub-branchings: the first one focuses on finding high-quality solutions, the second one in feasibility with respect to a problematic constraint and the third one finds a valid assignment for the decision variables.

Since two of the three objectives defined for this problem rely on the discharge port of the containers allocated in the slots of l, we start by branching over the set of variables P. Slots with discharge ports less or equal than the one assigned to the slot right below are selected, which decreases the probability of overstowage. The slots from stacks already used are considered first to reduce the used stack objective. When it is necessary to select a slot from an empty stack, the highest discharge port possible for the slot is selected.

After assigning all variables from P, we branch over a new set of auxiliary variables involved in one of the most problematic constraints: H. The height limits of the stacks are usually more strict than the weight limits and therefore, finding allocation plans that respect these limits become a difficult task in itself. No variable selection heuristic is involved for variables, we fill up stacks bottom-up and select the maximal height possible for a container to be allocated there.

At last, we branch over the set of variables S in order to generate an allocation plan. By this time, the discharge port and the height of the container to be allocated in each slot has been decided, and it is most likely that the objective value that any possible solution to be generated from this point is already known. Here we try to allocate slots from bottom-up in each stack, selecting the maximal possible container in the domain of the slot.

The decomposition of the branching plays along with the branch and bound strategy. The domain size of variables in P are considerable smaller than the ones from any of the viewpoints, making the process of finding valid assignments for P easier. Once the first valid allocation plan is found, most of the time the backtracking algorithm backtracks directly to the variables of the first branching in order to find a solution with a better objective value. Therefore, most of the

search is concentrated in a considerable smaller sub-problem, branching over the two other sets of variables just when the possibilities of finding a better solution are almost certain.

3.4 Improving the Model

Some Extra Constraints. Here some redundant constraints are declared in order to improve propagation and reduce search time. The first constraint is an *alldifferent* constraint over the set S of slots, reinforcing the fact that it is not possible to allocate one container in more than one slot. This constraint is forced in the model by the *channeling* constraint between S and C.

A second constraint deals with a sub-problem presented in the model. Since all containers have a height, they must be allocated in the stacks from l and each stack has a height limit, the problem of finding a stack for each container to be allocated without violating the height capacity can be seen as a bin-packing problem. A global constraint for this problem is introduced in [15], where the load of each bin and the position of each item are given as variables. Here a new set of auxiliary variables CS representing the stack where a container is allocated is necessary. We use element constraints to bind this new set of variables with the set C. A modified implementation of [15] is considered to model this sub-problem, in order to tighten the height limits of each stack and not allow unfeasible assignments of containers based on their height.

Symmetries on Containers. The weight of the containers make each of them almost unique, limiting the possibility of applying symmetry breaking constrains. It is possible, though, to use these constraints on the artificial containers that were added to the model. First we focus on the set of empty containers, this set is split into two equal subsets that become the empty containers to be allocated in each part of the location. By doing this we avoid any set of equivalent solutions where empty containers are swapped between Aft and Fore slots. Then, a non-increasing order is applied over each of the subsets mentioned before in order to avoid any symmetrical solution.

Splitting up all 40-foot containers into two smaller containers, Aft40 and Fore40, also provoke symmetrical solutions. All Fore40 containers are removed from Aft slots and all Aft40 from Fore slots in location l.

Symmetries on Slots. The first subset of slots that we consider for symmetry breaking is the cells: swapping the containers allocated in Aft and Fore slots of a cell generates equivalent solutions in several cases. Therefore, when the Aft and Fore slot of a cell can allocate containers with the same features, a non-increasing ordering constraint is used indicating that the id of the container allocated in the Aft slot of the cell has to be greater than the one allocated in the Fore slot. It is not possible to apply an order to the slots in a stack since the tier of the slot where a container is allocated is related to the overstowage objective. There are some cases, though, where ordering can be applied. The first case is when all containers to load in location l have the same discharge port. In this case,

it is possible to use a non-decreasing ordering constraint over all the slots in a stack that can allocate containers with the same features. In cases where there are different discharge ports it is not possible to sort the containers from the beginning. Here we take advantage of the different branching steps described in the previous section and select the subsets of slots in each stack where an ordering constraint can be used after the branching over the set of auxiliary variables P is finished. These subsets are defined by all the slots where containers with the same discharge port and the same features are allocated. Then a non-decreasing order constraint is used in each of this subsets.

At last, the possible symmetries between identical stacks are considered. In a pre-processing stage, stacks with the same features are grouped together: same slots capacity, reefer capacity, height limit and weight limit. When two stacks are in the same group, a lexicographic order is applied between them. The lexicographic order is also applied on the set of auxiliary variables P since this set of variables is assigned first than the set S.

Discussion on Symmetries. There is one relevant issue about the symmetry breaking constraints described in this section, more specifically on the lexicographic order constraints posted over stacks grouped together. Since these constraints are ordering the discharge ports, the height and the id of the containers in each stack, any assignment of values to these variables that does not follow such order will be considered invalid. It is necessary to sort the containers at a pre-processing stage to avoid any conflict among symmetry-breaking constraints over the different set of variables. The set of containers *Containers* is sorted such that containers with the highest discharge port have associated the highest id. This sorting avoids the lexicographic constraints posted over the set S and set P of variables in identical stacks to conflict with each other.

Estimators. Three estimators have been defined to determine whether a complete valid solution can be generated from a partial solution, leading to early pruning of branches from the search tree with no future. Two of the estimators are simple algorithms that compute lower bounds of objectives from relaxed versions of the problem, while the third estimator is an early termination detector for the height constraint.

The first estimator finds the minimum number of stacks necessary to allocate all containers in a location from a partial solution. It greedily solves a simplified version of the allocation problem, where the only constraint considered is the height limit of the stacks and all containers not yet allocated are considered as normal height containers. The estimator starts by assigning containers to used stacks, no new penalization is paid to do so. Once all used stacks have been totally filled up, the remaining containers are allocated in the empty stacks, which are sorted by capacity before the estimator fills them up. By sorting the empty stacks we guarantee that the number of stacks used to allocate the remaining containers will be the smallest possible.

Formally, let $\prec^\rho$ be a total pre-order defined over the set of stacks:

$$\begin{aligned} k \prec^\rho m \Leftrightarrow \quad & (k, m \in Stacks_{used}) \vee \\ & (k \in Stacks_{used} \wedge m \in Stacks_{empty}) \vee \\ & (k, m \in Stacks_{empty} \wedge cap(k) \geq cap(m)), \end{aligned}$$

where the capacity $cap(k)$ is the remaining number of free slots in stack k. Let C^N denote the number of containers not assigned yet in a partial solution

$$C^N = |\{C_i \,|\, i \in Containers, |C_i| > 1, i \notin Empty\}|.$$

A recursive function calculating a lower bound of the number of used stacks is then given by:

$$\mu_\rho(c, \prec^\rho_j, \sigma) = \begin{cases} j & : \text{ if } c = 0 \\ \mu_\rho(c-1, \prec^\rho_j, \sigma - 1) & : \text{ if } c > 0 \wedge \sigma > 0 \\ \mu_\rho(c, \prec^\rho_{j+1}, cap(\prec^\rho_{j+1})) & : \text{ if } c > 0 \wedge \sigma = 0 \end{cases}$$

where c is the number of remaining containers to be placed, $\prec^\rho_j$ is the jth stack in the ordering, and σ is the free capacity of this stack. The estimated number of used stacks for any partial solution is then given by:

$$ES^U = \mu_\rho(C^N, \prec^\rho_1, cap(\prec^\rho_1))$$

For the reefer objective, let s_R, c_R, and c_E denote the number of unassigned reefer slots, unassigned reefer containers, and unassigned empty containers. A lower bound of the number of non-reefer containers placed in reefer slots Ro is:

$$ES^R = \begin{cases} 0 & : \text{ if } s_R \leq c_R + c_E \\ s_R - c_R - c_E & : \text{ otherwise} \end{cases}$$

To achieve improved propagation, we restrict the reefer and used stack objective to be greater or equal to the estimated lower bound: $Ro \geq ES^R \wedge Us \geq ES^U$.

The third estimator detects inconsistency of the height constraint. Since stacks are filled bottom-up, a stack j for some partial solution p has some free height $h(j)$ at the top. Let M^N_j and M^{HC}_j denote the maximum number of normal and high-cube containers that can be placed in stack j, respectively. We have:

$$\begin{aligned} M^N_j &= \lfloor h(j)/h(N) \rfloor, \\ M^{HC}_j &= \lfloor h(j)/h(HC) \rfloor \end{aligned}$$

where $h(N)$ and $h(HC)$ denote the height of normal and high-cube containers. Let C^N and C^{HC} denote the number of unassigned normal and high-cube containers of p. For the height constraint to be consistent for p, we then must have:

$$\sum_{j \in Stacks} M^N_j \geq C^N \quad \wedge \quad \sum_{j \in Stacks} M^{HC}_j \geq C^{HC}.$$

4 Why CP

Despite the fact that several of the capacity constraints of the *CSPUDL* are linear, it is non-trivial to represent logical constraints and objectives like no 20-foot over 40-foot and overstowage using mathematical programming. Moreover, the under deck stowage problem considered in industry includes more rule-based constraints and may even affect containers in adjacent stacks. Two of these constraints are due to pallet-wide containers and IMO containers with hazardous goods. The former takes up the limited space between stacks and therefore can not be placed in adjacent stacks, while the latter may require packing patterns where no IMO containers are placed next to each other in any direction.

We have made a preliminary investigation of two optimal mathematical programming approaches for solving a slightly different version of the problem, where one extra objective related to clustering containers with the same discharge port and pre-placed containers in locations are considered, and the objective function is a linear inequality with different weights for each objective.

The first of these approaches is described in [12] and uses an IP model with binary decision variables c_{jki} indicating whether container i is placed in cell k in stack j. The results are shown in table 1. Despite adding several specialized cuts and exploiting the general optimization techniques of the CPLEX solver, this approach only performs significantly better than CP in two instances, 4 and 5, and slightly better (a matter of few milliseconds) in instances 11 and 12.

The second approach uses column generation. The idea is to let each variable of the master LP problem represent a particular packing of a stack. The dual variables of the master problem are used by the slave problem to find a packing with negative reduced price wrt. the current set of candidate packings. In our preliminary experiments, IP was used to solve the pricing problem. The approach was implemented in GAMS using CPLEX. As depicted in table 1 the results are much worse than for IP and CP. Moreover, the LP variables of the master problem become fractional, which actually means that lower bounds rather than optimal feasible solutions are found.

The CP model from table 1 was our first attempt to use constraint programming to tackle the *CSPUDL*. It heavily relies on boolean variables for modeling, the use of global constraints is limited and not all estimation algorithms were implemented. The results obtained with this model were promising enough to continue our work with CP. Four out of seventeen instances were notoriously performing over the time limit established as goal (one second), all instances were solved to optimality and, in just four of the instances IP outperformed CP.

Table 1. Preliminary results comparing three optimal approaches to a slightly different version of the *CSPUDL*. All the results are in seconds.

Method (time)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
CP(ms)	0.02	0.03	2	30.51	1.32	0.03	0.01	20	5.32	0.01	0.07	0.04	0.02	0.96	8.34	1.7	20.43
IP(ms)	0.06	0.15	-	1.75	0.160	0.12	0.19	0.15	-	0.01	0.01	0.01	0.1	1.17	21.89	4.94	40.05
CG(s)	14	22	1688	15588	-	17	22	22	352	7	12	28	14	27	18	11	37

Table 1 shows the results of solving to optimality seventeen instances of the *CSPUDL*. The first row shows the problem number. For these experiments, we used the same industrial problems as described in next section. The next three rows shows the computation time for the constraint programming (CP), integer programming (IP), and column generation (CG) approach. The dash in table 1 represents a timeout, the given time for this was thirty minutes.

5 Experimental Evaluation

A representative set of instances from different real-life vessels of different size, with different configurations of containers and discharge ports, with capacity location from 16 to 144 slots were gathered for the experiments here presented.

In table 2 the first eight columns of each row gives an overview of the features of each instance: The first column shows the instance number, the second one the total number of containers to be allocated, the third and fourth columns represent the number of 40 and 20-foot containers. The fifth and sixth columns are the number of reefer and high-cube containers, seventh column is the number of slots available and, the level of occupancy of the location is in the eight column. It is important to notice that a 40-foot container requires two slots to be allocated. All the experiments were run in a Ubuntu Linux machine with a Core 2 Duo 1.6 GHz and 1GB of RAM. The implementation of the constraint model took place in the C++ constraint library Gecode [11], version 3.0.2. The dash in table 2 represents the same as in table 1.

Table 2 shows the results of solving the set of locations using the CP model presented in this paper. Our main goal here is to present the impact of the estimator algorithms presented in section 3.4 and how they help to close the gap between the execution time of the CP approach and the time limit for this problem. The NS(s) and NS(nodes) columns show the response time in seconds and explored nodes of solving the instances without estimation algorithms. The E(s) and E(nodes) columns show the results of solving all instances to optimality including estimation algorithms in the model, time and explored nodes respectively. The branching strategy defined in section 3.3 was the one used.

From the results in table 2 it can be observed that estimation does have an important effect on the response time of the solver. Time is reduced in all instances but one, and in most of the cases the explored nodes also decrease. In instances where the explored nodes are equal but the time response has decreased, estimation algorithms are making nodes fail faster, avoiding unnecessary propagation. It is hard to determine the order of magnitude of the impact of the estimators, since it seems to vary from instance to instance, e.g. instance 4, 11, 13 and 16. The instance where it was not possible to prove optimality before has been solved in reasonable time, and instances 3 and 9 had a considerable reduction in their response time. It is also important to notice from the results in table 2, that our CP approach is getting closer to the goal described in the introduction, since just three instances out of seventeen remain with an execution time over one second.

With respect to the results presented in table 1, a small overhead can be seen in instances where finding the optimal solution usually takes less than 0.2 seconds, e.g. instances 1, 2, 6, etc. However, the substantial reductions in time response in instances 3, 4, 9 and 15 compensates the overhead.

Table 2. Problem instances and CP experimental results. Each row represents an instance. The first eight columns are general information of the instance, the extra four are the response time and explore nodes of the experiments described in this section.

Inst	Conts	C^{20}	C^{40}	C^{R}	C^{HC}	Slots	Full(%)	NS(s)	NS(nodes)	E(s)	E(nodes)
1	23	0	23	0	2	62	74	0.15	117	0.17	105
2	38	0	38	0	38	86	88	0.24	159	0.19	139
3	75	10	65	0	9	144	97	366.78	120461	0.59	213
4	40	0	40	34	34	90	89	14.14	4405	4.06	2391
5	28	0	28	24	28	90	62	0.79	271	0.31	187
6	28	0	28	0	10	60	93	0.28	119	0.07	61
7	35	0	35	0	8	72	97	0.39	153	0.19	153
8	34	0	34	0	7	70	97	0.34	147	0.17	147
9	53	0	53	0	5	108	98	37.19	9015	0.36	199
10	4	0	4	0	0	16	50	0.06	21	0.03	21
11	7	0	7	0	7	40	35	0.14	53	0.07	51
12	42	0	42	0	42	88	95	1.12	279	0.52	259
13	24	0	24	0	0	90	53	0.47	157	0.20	141
14	23	0	23	0	23	108	42	2.23	553	0.26	151
15	34	0	34	0	8	90	75	2.34	639	1.15	639
16	19	0	19	0	19	90	42	0.84	289	0.40	275
17	37	0	37	1	34	116	63	-	-	22.16	10153

6 Conclusion

In this paper we have introduced a model for stowing containers in an under deck storage area of a container vessel bay. We have shown how to solve this model efficiently using CP and compared our approach favorably with an integer programming and a column generation approach. CP is not widely used to solve problems to optimality. The estimation algorithms introduced in this paper, however, improves the performance of the branch and bound dramatically, good lower bounds are generated from partial solutions and unpromising branches are pruned in early stages without discarding any optimal solution.

We consider that the main reason of CP outperforming IP in most of the cases presented in this paper is the non-linear nature of some of the constraints and objectives of this problem, i.e. no 20-foot on top of 40-foot container, over-stowage. The logical nature of these constraints makes their linearization with 0-1 variables a non trivial task, and since further constraints to be included in this problem have the same logical nature as the ones mentioned before, i.e. IMO and pallet-wide containers, the CP approach will be most likely to keep outperforming an IP implementation.

An important objective of our future work is to make instances of stowage planning problems available to the CP community. We also plan to develop CP-based LNS stowage algorithms and investigate whether CP can be used to solve the pricing problem of column generation methods for this problem.

Acknowledgements. We would like to thank the anonymous reviewers and the following collaborators: Thomas Stidsen, Kent Hj Andersen, Trine Hyer Rose, Kira Janstrup, Nicolas Guilbert, Benoit Paquin and Mikael Lagerkvist. This research was partly funded by The Danish Council for Strategic Research, within the programme "Green IT".

References

1. Botter, R., Brinati, M.: Stowage container planning: A model for getting an optimal solution. In: Proceedings of the Seventh International Conference on Computer Applications in the Automation of Shipyard Operation and Ship Design (1992)
2. Giemesh, P., Jellinhaus, A.: Optimization models for the containership stowage problem. In: Proceedings of the International Conference of the German Operations Research Society (2003)
3. Ambrosino, D., Sciomachen, A., Tanfani, E.: Stowing a conteinership: the master bay plan problem. Transportation Research 38 (2004)
4. Avriel, M., Penn, M., Shpirer, N., Witteboon, S.: Stowage planning for container ships to reduce the number of shifts. Annals of Operations Research 76(55-71) (1998)
5. Dubrovsky, O., Penn, M.: A genetic algorithm with a compact solution encoding for the container ship stowage problem. Journal of Heuristics 8(585-599) (2002)
6. Ambrosino, D., Sciomachen, A., Tanfani, E.: A decomposition heuristics for the container ship stowage problem. Journal of Heuristics 12(3) (2006)
7. Kang, J., Kim, Y.: Stowage planning in maritime container transportation. Journal of the Operations Research society 53(4) (2002)
8. Wilson, I., Roach, P.: Principles of combinatorial optimisation applied to container-ship stowage planning. Journal Heuristics 1(5) (1999)
9. Gumus, M., Kaminsky, P., Tiemroth, E., Ayik, M.: A multi-stage decomposition heuristic for the container stowage problem. In: Proceedings of 2008 MSOM Conference (2008)
10. Pacino, D., Jensen, R.: A local search extended placement heuristic for stowing under deck bays of container vessels. In: Proceedings of ODYSSEUS 2009 (2009)
11. Gecode Team: Gecode: Generic constraint development environment (2006), `http://www.gecode.org`
12. Rose, H.T., Janstrup, K., Andersen, K.H.: The Container Stowage Problem. Technical report, IT University of Copenhagen (2008)
13. Smith, B.: Modelling. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
14. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004)
15. Paul, S.: A constraint for bin packing. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)