Abstract

Low cost containerized shipping requires high-quality stowage plans. Scalable stowage planning optimization algorithms have been developed recently. All of these algorithms, however, produce monolithic solutions that are hard for stowage coordinators to modify, which is necessary in practice due to exceptions and operational disruptions. This article introduces an approach for modifying a stowage plan interactively without breaking its constraints. We focus on re-arranging the containers in a single bay section and show two approaches for providing complete and backtrack-free decision support using symbolic configuation techniques, one based on binary decision diagrams and one based on DPLL solvers. We show that binary decision diagrams can be used to solve real-world sized instances of a single bay, and that search based solvers can be used to solve simplified instances going beyond a single bay.

Symbolic Configuration for Interactive Container Ship Stowage Planning

Christian Kroer[†] Martin Kjær Svendsen[*]
Rune M. Jensen[*] Joseph Kiniry[‡] Eilif Leknes[* *†‡§]

April 19, 2014

1 Introduction

Efficient containerized shipping is essential for the global economy, and relies on high-quality stowage plans that maximizes cargo yield while reducing sailing speed and port fees by minimizing port stays. Generating such plans is a combinatorial optimization problem with several NP-hard components [e.g., 5, 31, 10]. Traditionally these problems have been solved manually using graphical tools with no optimization capabilities [e.g., 29, 18], but due to the rapidly growing size of vessels, there has been an increasing interest in developing stowage planning optimization algorithms to support automated stowage planning [e.g., 3, 41, 31]. All of these algorithms output a single plan that is hard for stowage coordinators (SCs) to alter without making it infeasible or undesirable. Our experience with one of the first deployed auto-stowage tools [15] is that this is a significant limitation in practice, as it is often necessary for SCs to make some changes to a produced plan. One reason for this is that it is difficult to make an accurate stochastic model of future cargo and represent highly non-linear physical constraints such as lashing forces. Moreover, there can be special circumstances caused by break-bulk, equipment failure, or agreements with customers that the optimization model cannot represent. Even if the optimization models eventually are sufficiently accurate and expressive, a typical stowage planning problem is highly symmetric with many equally good solutions that SCs should be able to choose freely among; for instance to achieve a preferred trade-off between multiple objectives of the problem.

In this article, we take the first steps beyond graphical support for user-driven modifications of stowage plans, applying two different interactive configuration

*† Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, Pennsylvania
†*IT University of Copenhagen, Rued Langgaards Vej 7, 2300, Copenhagen S, Denmark
‡‡ Galois, Inc., Portland, Oregon
§Address correspondence to ckroer@cs.cmu.edu
The article is an extension of a conference paper published at IMECS 2012 [22]

techniques, one [21] based on Reduced Ordered Binary Decision Diagrams (BDDs) [BDDs: 8] and the other based on fast boolean satisfiability (SAT) solving [7].

The choice of the two techniques was found in the literature. **(author?)** [34] showed that BBD-based techniques have superior performance to search based approaches. However, more recent work by **(author?)** [20] found that SAT-based configuration can outperform BDDs for a host of different configuration problems. These conflicting results indicate that the best performing approach is dependent on the specific structure of the problem at hand. In this article we evaluate both approaches, to determine whether either approach is sufficiently scalable to solve real-world container stowage instances.

In this initial study, we focus on the bay sections below each hatch-cover of the vessel, where containers can be re-arranged without breaking overall stability, stress moment, and crane activity properties of the stowage plan. The BDD approach compiles a compact representation of the container stacking rules offline, which is used online to make fast inferences about all valid container assignments simultaneously. For the search based approach, the problem is encoded as a SAT model offline, and the valid container assignments are identified by searching in the SAT model online, using a complete SAT solver.

Based on these approaches it is possible to give SCs interactive decision support with complete feedback about how to re-arrange containers and still maintain a valid assignment. The support is backtrack-free in the sense that even for a partial assignment, SCs will only be allowed to do assignments and re-arrangements that can be extended to a valid complete assignment. In this way, the configurator keeps track of all the constraints of the problem such that the SCs can concentrate on re-arranging the containers according to their preferences.

To make the investigation practically manageable, we limit the scope to an NP-hard representative problem of stowing containers in these bay sections that takes all major stacking rules below deck into account. The main contributions of our work are: 1) a formal model of the rules for stowing containers below deck that is suitable both for BDD compilation and SAT conversion, 2) a direct SAT encoding of the rules, 3) a type-based grouping of containers that enables the configurator to scale to real-sized instances, 4) a performance comparison of the configuration techniques, and 5) a configuration tool with a Graphical User Interface (GUI) using color codings to guide the stowage planning process.

Our computational results show that the BDD-based configuration technique can scale to bay sections of large deep sea vessels when breaking symmetries using type-based grouping of containers. With this approach, the offline compilation time of a configurator is close to independent of the number of containers in the bay. It only depends on the number of container types. Since a good stowage plan clusters similar containers in bays, the number of different container types in each bay is limited and makes it possible to quickly compile a configurator. The online response time of the configurator only depends on the size of the generated BDDs. It is below one second even for the largest BDDs compiled in our experiments. We also introduce a restricted, but still realistic, version of the interactive stowage problem, and show that SAT-based solvers are able to

scale to even larger restricted instances, in spite of the original problem being intractable for SAT-based techniques.

2 Background

A standard ISO container (DC) is 20 or 40 feet long and 8.6 feet high. 20' containers cannot be stacked on top of 40' containers due to the lack of structural support in the middle of the 40' container. In this article we focus on standard 20 and 40 foot containers, along with reefer containers (RF,WC), which have a refrigeration unit, meaning that they must be placed such that they can be connected to on-board power. All containers have a destination port and a vessel normally has a route spanning across several ports. A problem that might arise out of this is *overstowage* where a container is stowed over another container in the same stack with earlier discharge port, and thus must be removed to reach the *overstowed* container.

A container vessel normally has a 1.000-15.000 Twenty-Foot Equivalent Unit (TEU) capacity. A vessel is made up of a number of bays, which are collections of stacks of containers. Each stack has a height and weight limit that depends on the position of the stack. Each bay is divided into an on and below deck area separated by one or more hatch-lid covers. Each container stack is composed of vertically arranged groups of cells. Each cell is 40 feet long, 8 feet wide and 8.6 feet high and can hold one 40' container or two 20' containers. When holding two 20' containers, they are said to be placed in the *aft* and *fore* TEU slot. If instead a 40' container is placed in the cell, it is said to be placed in a Forty-Foot Equivalent Unit (FEU) slot. A *location* is a bay section below deck comprised of the slots under a hatch-lid cover. A bay has two to four of these depending on the number of hatch covers and for a typical large deep sea vessel, they have between 2 and 65 cells. Figure 1 shows a typical physical arrangement of containers in a small location.

The position of a container is given by its bay, stack, and tier number. The tier number defines its vertical position in its stack. It is normal to use even

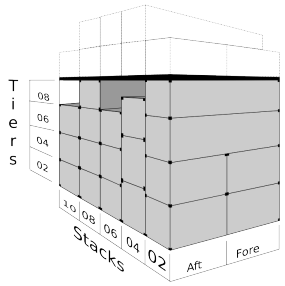

Figure 1: A typical arrangement of containers in a location.

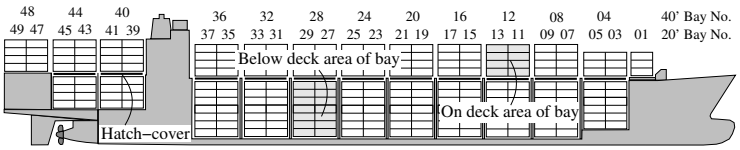

Figure 2: Layout and numbering of bays of a container vessel.

numbers for FEU stacks and odd numbers for TEU stacks. The bays and cells of a container vessel are shown in Figure 2.

3 Problem Statement

The representative problem we focus on is to stow or re-arrange a given set of containers in a single location. Moving containers between locations is outside the scope of this article but is a topic for future work. It requires that we also take stability and stress moment limits into account as well as potential increase in the port stay due to uneven distribution of container moves over bays. The subset of constraints that we model is described below.

1. One container must be placed in one slot, and in one slot only. Slots can only hold one container.
2. 40' containers cannot be placed in a TEU slot and 20' containers cannot be placed in a FEU slot.
3. The FEU slot of a cell occupies the same space as the TEU slots, so there can be no container in the TEU slots if a container is placed in the FEU slot and vice versa.
4. A 20' container cannot be placed on top of a 40' container.
5. A container must have support from below.
6. When placing 20' containers, the containers must be placed 'evenly', i.e. the number of containers in one TEU stack cannot exceed the number of containers in the other TEU stack by more than one.
7. For each stack, the sum of the weight of the containers must be within the weight limit of the stack.
8. Overstowage will not be allowed at all, so one container cannot be placed on top of a container having a lower (earlier) discharge port.
9. Reefer containers cannot be placed in non-reefer slots.

Notice that we do not consider the height of containers since height constraints to a large extent are similar to weight constraints. We also do not take into consideration special cargo such as OOG and IMO, and we exclude pallet-wide containers since the stacking rules associated with them are similar to the overstowage rule and the no 20' above 40' rule. Also notice that we model an objective of having no overstows indirectly as a constraint. Alternatively a limited number of overstows could be allowed. It is easy to show by a reduction from bin packing that determining feasibility of this problem is NP-complete [10]. Finding solutions without overstowage is also NP-complete by a reduction from coloring of circle graphs if stacks are uncapacitated [5]. Succinctly, our goal thus becomes to build an interactive stowage configurator to guide a stowage coordinator in placing a set of containers within a single location on the ship such that the configuration satisfies the above constraints. At each step in the process, we present the user with a set of assignments that are all part of some valid stowage plan given the assignments already chosen by the user. This ensures that the stowage coordinator never has to backtrack in their choices. This process is then repeated until all containers have been placed.

4 Configuration

We formally define a *configuration problem* $\mathcal{C}$ as a triple $\mathcal{C} = (X, D, F)$, where X is a set of variables $x_1, x_2, \ldots, x_n$, D is the Cartesian product of their finite domains $D_1 \times D_2 \times \cdots \times D_n$, and $F = f_1, f_2, \ldots, f_m$ is a set of formulas specifying conditions that the variable assignments must satisfy [17]. For each variable i, the domain D_i can, in general, be any type of values. In this work we will use integers to denote different options.

A solution to a configuration problem is an assignment of the variables $x_i = v_i$, where $v_i \in D_i$ for $1 \leq i \leq n$ that satisfies all the formulas in F. An *interactive configurator* is a decision support tool that efficiently guides users to a desirable assignment. The configurator takes the set of variables X, their finite domains D and a set of formulas F as input. The configurator initially has an empty set of assignments. In each iteration, the invalid values of the domains of unassigned variables are removed such that each value in the domain of a variable is part of at least one valid assignment. The user then selects a preferred value of any unassigned variable. The algorithm terminates when all the variables are assigned. Pseudo code of the configurator is shown in Algorithm 1. In this pseudo code, we implicitly assume that the problem is initially feasible. In practice, if the problem is not feasible initially we would immediately return an error to the user.

In line 1, the variable R is initialized. R is a data structure representing the problem. For BDD based configuration, this means compiling a BDD representation of $\mathcal{C}$. For SAT based solving, this means generating a CNF

Out of Gauge cargo, which is higher or wider than the size of a standard container, requiring special considerations

Dangerous Cargo, with extra constraints on possible placement

formula describing $\mathcal{C}$. Line 2 to 5 then repeatedly run until all variables have valid assignments. In Line 3, the function VALID-ASSIGNMENTS computes the whole set of valid assignments at the current stage in the configuration process, and the user is asked to pick an assignment among this set. Line 4 updates R to include the chosen assignment.

In some cases a partial assignment might already be known from some previous solution. In our case, this could be a previously computed stowage plan that became invalid due to unforeseen constraints. For such cases, it is easy to change the algorithm to start with a partial assignment instead of an empty assignment. The algorithm is complete since the user can choose freely between any of the valid assignments, and it is backtrack-free since the user cannot choose a variable assignment for which no valid complete assignment exists, and is therefore never forced to go back and choose differently.

```
1 R ←INITIALIZE(C);
2 while X has unassigned variables do
3 |   choose (x_i = v) ∈ VALID-ASSIGNMENTS(R);
4 |   R ← R ∧ (x_i = v);
5 end
```

Algorithm 1: Configurator.

4.1 BDD based configuration

BDD-based configuration [34] uses a BDD to represent R in order to compute valid assignments efficiently. In practice, the configurator algorithm is then a two phase approach. The first phase is an offline phase where the BDD is compiled and the second phase uses this BDD for fast complete inference. The advantage is that the worst case inference time in the online phase only grows polynomially in the size of the final BDD.

A BDD is a rooted directed acyclic graph that is a compact representation of the decision tree of a Boolean operation. It has one or two terminal nodes, 0 and 1, and a set of internal nodes associated with the variables of the operation. Each internal node has a *high* and a *low* edge. For a particular assignment of the variables, the value of the operation is determined by traversing the BDD from the root node to a terminal node by recursively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The value of the operation is *true* if the reached terminal node is 1, and *false* if the reached terminal node is 0. Figure 3 shows an example. Every path in the graph respects a linear ordering of the variables, which can be any linear ordering. To get small BDDs, it is important that variables that depend on each other are close in the ordering. This is done through domain specific knowledge and trial and error. Modern BDD solvers represent sets of BDDs compactly as a single multi-rooted BDD. All Boolean operations can be carried out in polynomial time. Thus, BDDs are well suited for building fast inference

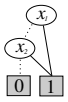

Figure 3: A BDD of the operation $f(x_1, x_2) = x_1 \vee x_2$ using order $x_1 \prec x_2$. High and low edges are drawn with solid and dashed lines, respectively.

systems for propositional logic. For a comprehensive introduction to BDDs and their compilation, we refer the reader to **(author?)** [8, 25, 40].

We now illustrate BDD-based configuration on the small example shown in Figure 4. The problem is to place three standard 40' containers in a FEU bay

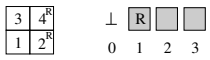

Figure 4: A bay stowage example with 4 slots (left) and 3 containers (right).

with two stacks each with two slots. We assume that one of the containers is a reefer container and two of the slots are reefer slots (both indicated by "R" in the figure). We also assume that containers and slots are numbered as shown in the figure and that slots assigned to 0 ($\perp$) do not hold any container.

We use four variables x_1, x_2, x_3, x_4, each with domain $\{0, 1, 2, 3\}$. The variable x_i denotes which container is placed in slot i. An assignment $x_i = v_i$ denotes that slot i holds a container from group v_i. The set of propositional formulas shown below defines the set of valid assignments.

$$x_2 = 1 \vee x_4 = 1 \tag{1}$$

$$(x_3 \neq 0 \Rightarrow x_1 \neq 0) \wedge (x_4 \neq 0 \Rightarrow x_2 \neq 0) \tag{2}$$

$$x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \wedge x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge x_3 \neq x_4 \tag{3}$$

Constraint (1) ensures that the reefer container is placed in a reefer slot, constraint (2) ensures that no container hangs in the air. Finally, constraint (3) ensures that each container is assigned to only one slot, and that only one slot is empty (0).

The Boolean representation of this problem uses a binary encoding of the finite domains of the variables. Since each variable has a domain of size four, we need two Boolean variables to represent each variable. Let $x_i^1, x_i^0 \in \{0, 1\}$ denote the two Boolean variables of x_i. In a binary encoding of the domain of x_i, x_i^1 and x_i^0 represent the value of bit 1 and 0 of x_i, respectively. Thus, the expression $x_i^1 \wedge \neg x_i^0$ represents setting $x_i = 2$, i.e. placing the second container in slot i, and so on. We can now translate constraint (1 - 3) to propositional formulas.

In the offline phase of the BDD-based configurator (line 1 of Algorithm 1), a single BDD is built equal to the conjunction of the propositional formulas.

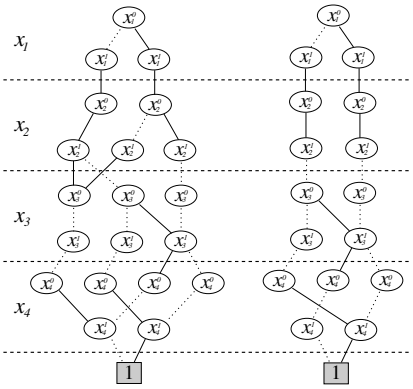

Figure 5: Left: BDD R of the conjunction of the constraints (1-3). Right: Reduced BDD $R \wedge (x_2 = 1)$.

The resulting BDD of our example is shown in Figure 5 (left). For clarity, we do not draw edges leading to terminal 0. This is known as a zero-suppressed BDD [26]. Each of the six distinct paths leading to terminal 1 corresponds to one of the six possible arrangements of the containers shown in Figure 6. Each

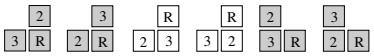

Figure 6: The six valid container configurations of the example shown in Figure 4. The four shaded configurations are represented by the BDD shown in Figure 5 (right) with $x_2 = 1$.

iteration of the online phase of the BDD-based configurator (line 2-5) starts with a computation of the valid assignments of the remaining configurations represented by the BDD R. Since the variables are ordered in layers from top to bottom in R, it is possible to perform the VALID-ASSIGNMENTS computation in Algorithm 1 by probing it in time $O(\sum_{i=1}^{n} |D_i|V_i)$, where V_i is the number of BDD nodes in the layer of variable x_i and $|D_i|$ is the size of the original domain of x_i [16]. Unless the BDD representing a configuration problem grows very large, the time needed to compute valid assignments is a fraction of a second and hardly noticed by the user. The result of the valid assignments computation is a reduced domain for each variable. If there is only a single value left for a variable there is no further decisions to make for it and it is assigned automatically by the system. The user then chooses one of the possible values from the reduced domains and R is restricted to this choice by $R \leftarrow R \wedge (x_i = v_i)$. Assume that the user chooses $x_2 = 1$ in the first iteration. The reduced BDD R is shown in Figure 5 (right).

Due to the speed of the valid assignments computation, BDD-based configu-

ration enables the user to navigate quickly in the configuration space without having to worry about the feasibility of the partial configuration. This allows SCs to focus on achieving their preferred stowing when re-arranging containers in a location.

In this paper, we focus on the log encoding for BDD based configuration. For BDDs to be effective in practice, it is important to keep the number of variables small, and thus the slew of encodings designed for good propagation power, at the expense of more variables, in SAT solvers are unlikely to be effective for BDDs. This changes with the introduction of ZDDs, where direct encodings can sometimes be efficient. Thus, a future research direction could be to investigate any potential performance gains from using ZDDs rather than BDDs.

4.2 SAT based configuration

SAT-based configuration uses a propositional logic formula to represent R, produced in the INITIALIZE function by conjoining the propositional formulas in F, changing representation of variables with domain sizes larger than 2 to a number of binary variables as demonstrated in the BDD approach, and converting the resulting formula to conjunctive normal form (CNF).

With this representation F_{CNF} built, deciding whether a variable assignment in X, represented by a number of binary variable assignments

$$V = (x_1 = v_1 \wedge x_2 = v_2 \wedge \ldots \wedge x_n = v_n)$$

is legal, reduces to determining whether $F_{CNF} \wedge V$ is satisfiable. This problem can be solved using a SAT solver. Notice that this requires a solver call for each variable assignment in X that we wish to test.

Transforming X to a set of binary variables can be done in several ways, with different implications for the efficiency of the SAT solver.

In a **direct encoding** [23], a Boolean variable is created for each domain value of each CSP variable, with the Boolean variable representing whether the given value is assigned to the CSP variable. This requires the introduction of an *at-least-one* constraint over the Boolean variables representing the domain of a given CSP variable, and an *at-most-one* constraint can optionally be added too [39]. The number of Boolean variables produced with this encoding, just for representing the CSP variables, will be $O(|X|d)$, where d is the largest domain of any variable.

In a **log encoding** [19], a logarithmic number of Boolean variables is used to represent the domain of a given CSP variable x. This is what is used in the BDD approach. Each domain value is represened by an unique pattern over a number of Boolean variables. Since it is possible to make 2^n patterns with n Boolean variables, the log encoding creates $O(x\lceil log(d)\rceil)$ variables to encode all the variables of the configuration problem. While the log encoding is more compact than the direct encoding, it has weaker propagation in the SAT solver [39, 14].

In an **order encoding** [36], as with the direct encoding, a Boolean variable is introduced for each domain value of each CSP variable. However, the semantics

of the Boolean variables are different. For the order encoding, Boolean variables represent whether a given CSP variable is less than or equal to some value.

Formally, consider a *finite linear CSP*, defined as a tuple (X, l, u, P, F) where X is a set of integer variables, $l(x)$ and $u(x)$ are the lower and upper bounds of each variable $x \in X$, P is a set of propositional variables and F is a set of formulas. We then let $p(x \leq c)$ denote the Boolean variable representing whether the value of x is less than some constant c.

Any domain value c can be specified by the unit clauses $\neg p(v \leq c-1) \wedge p(v \leq c)$. Complexity wise, this encoding is the same as the direct encoding, and requires $O(|X|d)$ Boolean variables, where d is the highest upper bound in X. The approach has been shown to outperform various other encodings in **(author?)** [36], and the CSP solver Sugar [35] used this encoding to win four categories at the 2008 CSP solving competition, and three categories at the 2009 competition [1, 2]. While Sugar has shown strong performance results at these competitions, we wish to employ more recent advances in CSP-to-SAT conversion described below. Notably, these advances include the conversion used by Sugar as a special case.

In the **compact order encoding** [37], every original CSP variable $x \in X$ is represented by a number of new variables, where the number of new variables is equal to the number of digits necessary to represent the largest domain of any $x \in X$ using some selected base B.

Let d define the maximum domain size, then $m = \lceil log_B d \rceil$ is the order of magnitude or required number of new variables for each original variable.

We denote the digits of x as $x_0, \ldots x_{m-1}$, with x_0 being the least significant digit. The digit x_i for any value $a < d$ can then be determined by $x_i = \lfloor \frac{a}{B^i} \rfloor \bmod B$. Each digit is encoded as binary variables using the order encoding. In a solution to this new digit encoded problem, the value for a variable x is the value of its digits $x_{m-1} \ldots x_0$. Consider some variable x taking on values in the range $[0 - 99]$. To encode this variable in base 10, the compact order encoding introduces two variables, $x_1, x_0 \in [0 - 9]$. To encode the specific value $x = 15$, x_1 would take on the value 1, and x_0 would take on the value 5.

Both order and log encoding are special cases of the compact encoding. When $B \geq d$, the compact order encoding is equivalent to the order encoding, as each variable is represented by a single digit, while $B = 2$ is equivalent to the log encoding, as all variables in X are then Boolean. Choosing a base value between these two extremes forms a trade-off between the size of the final CNF structure and the strength of propagation in the solver. By choosing a large value for B it is possible to achieve strong propagation, since few digits entail fewer inference steps to move carries through the digit structure, while smaller values will result in fewer Boolean variables and CNF clauses introduced when applying the order encoding to the compact CSP. Since the number of digits is central to this compromise, it is advantageous to specify this parameter (the order of magnitude) to the encoding algorithm instead of the base.

For every encoding mentioned above, we use the Tseitin transform [38] to convert the obtained boolean formula to CNF form, along with the standard optimizations for reducing the number of new variables.

The basic SAT implementation of the VALID-ASSIGNMENTS procedure is shown in Algorithm 2. Lines 3-10 iterate over all variables x that have not yet had values assigned to them. Lines 3-7 iterate over the domain of x, and Lines 4-6 prune all values in the domain that lead to unsatisfiability. Lines 8-10 force variables with only one domain value left to take on that value without asking the user. Since the user is never able to pick assignments that are not part of some satisfying solution, VALID-ASSIGNMENTS is guaranteed to always either *(a)* return a set of two or more potential assignments for the user to pick from, or *(b)* a final solution where the remaining variables had only one domain value left, and were thus forced to take on that value in Lines 8-10.

```
1  Input: R;
2  for x not assigned in R do
3  |  for val ∈ D_x do
4  |  |  if IsUnsatisfiable(R ∧ x = val) then
5  |  |  |  D_x = D_x \ val;
6  |  |  end
7  |  end
8  |  if |D_x| = 1 then
9  |  |  x = D_x;
10 |  end
11 end
12 return R;
```

Algorithm 2: The VALID-ASSIGNMENTS procedure for the SAT based solution. Determining the result of ISUNSATISFIABLE relies on a SAT solver.

To increase performance, different techniques were applied. In general, a search based configurator needs to search for a satisfying solution every time the ISUNSATISFIABLE procedure is invoked. However, there are several optimizations that can be applied for Algorithm 2, such that the amount of searches can be minimized. For clarity, these optimizations are not shown in our simple version of VALID-ASSIGNMENTS shown in Algorithm 2, but can be easily added. The simplest optimization is to remember falsifying assignments, such that if a given assignment was previously found to lead to unsatisfiability, this assignment will still lead to unsatisfiability if more constraints are added to the same problem. A call to the function ISUNSATISFIABLE can thus be avoided in Line 4 for such assignments.

When a solver returns satisfiable on a given configuration problem, it means that it has found a set of variable assignments that satisfy the constraints, a *witness*. This can be exploited, as this witness includes more information than just the fact that the problem is satisfiable with the currently selected assignments. It shows that all of the assignments selected in the solution can be part of some satisfying assignment. We utilize this fact by analyzing the whole witness, and remembering all the satisfying assignments, such that we can immediately return $false$ in the ISUNSATISFIABLE procedure on Line 4, when

asked about these assignments in later invocations. It is important to note here that the set of assignments known to be satisfying has to be cleared whenever some new variable assignment is made by the user, as this new constraint can lead to unsatisfiability for any number of other assignments [34].

Finally, we also try an optimization introduced by **(author?)** [20], which guides the solver towards solutions that provide witnesses with as much new information as possible. This is done by modifying the branching of the solver, such that whenever it branches on some variable v, it tries assigning the unknown value to v first, thereby preferring witnesses that give new information. This optimization is somewhat conflicting with the phase saving heuristic [32], which attempts to assign previous values to backjumped variables, in order to save search time. We briefly tested this both with and without phase saving, but found that it had negligible impact on the the amount of solver calls needed.

As an engine for the SAT based approach, we use the MiniSat solver, along with its assumption based incremental SAT interface. This allows us to build the model of a given configuration instance only once, and solve it incrementally, storing any clauses learnt. [11, 12].

5 Container Stowage Modeling

5.1 General model

We now present a general model of the container stowage problem, that will be the model that we implement both for the BDD and CSP-to-SAT based approach. Our stowage model is a simplification of the one presented in [10], where they solve the simpler problem of computing a single configuration, rather than performing interactive configuration.

We have a set of stacks $S = \{1, \ldots, s\}$. A set of tiers $T_\sigma = \{1, \ldots, t_\sigma\}$ of stack $\sigma \in S$. A set of containers $C = \{1, 2, \ldots, c\}$ where c is the number of containers, with two subsets C_F and C_T containing all 40' and 20' containers, repectively ($C = C_F \cup C_T$). To model the placement of a container in a location, we introduce decision variables $X = \{c^{\mathsf{FT}}_{\sigma,\tau}, c^{\mathsf{AT}}_{\sigma,\tau}, c^{\mathsf{F}}_{\sigma,\tau} \mid \sigma \in S, \tau \in T_\sigma\}$ where FT stands for Fore TEU, AT stands for Aft TEU and F stands for FEU. For example; $c^{\mathsf{FT}}_{2,4}$ is the fore TEU slot in stack 2 and tier 4. Each $c^{\mathsf{FT}}_{\sigma,\tau}$ and $c^{\mathsf{AT}}_{\sigma,\tau}$ has domain $D_{20} = C_T \cup \{0\}$, each $c^{\mathsf{F}}_{\sigma,\tau}$ has domain $D_{40} = C_F \cup \{0\}$, for both, 0 means no container is placed in the slot. The function $w(\chi)$ denotes the weight of container χ with $w(0) = 0$. The weight limit of a stack σ is denoted w_{lim_σ}, while $d(\chi)$ denotes the discharge port of container χ, where the ports are visited in ascending order.

5.2 Constraints

The constraints of the model are presented in the following.

For the alternative model that directly translates the stowage problem to a SAT problem, please refer to appendix 7

$$\bigwedge_{\sigma \in S} \bigwedge_{\tau \in T_\sigma} c^{\mathsf{F}}_{\sigma,\tau} = 0 \vee (c^{\mathsf{FT}}_{\sigma,\tau} = 0 \wedge c^{\mathsf{AT}}_{\sigma,\tau} = 0). \tag{4}$$

$$\bigwedge_{\sigma \in S} \bigwedge_{\tau \in T_\sigma \setminus \{t\}} c^{\mathsf{F}}_{\sigma,\tau} \neq 0 \Rightarrow (c^{\mathsf{FT}}_{\sigma,\tau+1} = 0 \wedge c^{\mathsf{AT}}_{\sigma,\tau+1} = 0). \tag{5}$$

$$\bigwedge_{\sigma \in S} \bigwedge_{\tau \in T_\sigma \setminus \{t\}} \begin{array}{l} (c^{\mathsf{FT}}_{\sigma,\tau} = 0 \Rightarrow c^{\mathsf{AT}}_{\sigma,\tau+1} = 0) \wedge \\ (c^{\mathsf{AT}}_{\sigma,\tau} = 0 \Rightarrow c^{\mathsf{FT}}_{\sigma,\tau+1} = 0). \end{array} \tag{6}$$

$$\bigwedge_{\sigma \in S} \bigwedge_{\tau \in T_\sigma \setminus \{t\}} \begin{array}{l} (c^{\mathsf{FT}}_{\sigma,\tau} = 0 \Rightarrow c^{\mathsf{FT}}_{\sigma,\tau+1} = 0) \wedge \\ (c^{\mathsf{AT}}_{\sigma,\tau} = 0 \Rightarrow c^{\mathsf{AT}}_{\sigma,\tau+1} = 0) \wedge \\ \Big(c^{\mathsf{F}}_{\sigma,\tau} = 0 \wedge (c^{\mathsf{FT}}_{\sigma,\tau} = 0 \vee c^{\mathsf{AT}}_{\sigma,\tau} = 0) \Rightarrow \\ c^{\mathsf{F}}_{\sigma,\tau+1} = 0\Big). \end{array} \tag{7}$$

$$\bigwedge_{\chi \in C} \Big(\sum_{\sigma \in S} \sum_{\tau \in T_\sigma} (c^{\mathsf{FT}}_{\sigma,\tau} = \chi) + (c^{\mathsf{AT}}_{\sigma,\tau} = \chi) + (c^{\mathsf{F}}_{\sigma,\tau} = \chi) \Big) = 1, \tag{8}$$

$$\bigwedge_{\sigma \in S} \Big(\sum_{\tau \in T_\sigma} w(c^{\mathsf{FT}}_{\sigma,\tau}) + w(c^{\mathsf{AT}}_{\sigma,\tau}) + w(c^{\mathsf{F}}_{\sigma,\tau}) \Big) \leq w_{lim_\sigma}. \tag{9}$$

$$\bigwedge_{\sigma \in S} \bigwedge_{\tau \in T_\sigma \setminus \{t\}} \begin{array}{l} min\{d(c^{\mathsf{FT}}_{\sigma,\tau}), d(c^{\mathsf{AT}}_{\sigma,\tau}), d(c^{\mathsf{F}}_{\sigma,\tau})\} \geq \\ max\{d(c^{\mathsf{FT}}_{\sigma,\tau+1}), d(c^{\mathsf{AT}}_{\sigma,\tau+1}), d(c^{\mathsf{F}}_{\sigma,\tau+1})\}. \end{array} \tag{10}$$

5.3 Constraint descriptions

First of all, note that some constraints are modeled by reducing the domains of the slot variables. These include that TEU slots cannot contain 40' containers, and FEU slots cannot contain 20' containers. Also, that reefer containers cannot be placed in non-reefer slots. The formula for each constraint is described in the following:

(4) Prevents placement in a FEU slot if the corresponding fore/aft TEU slot is occupied, and vice versa, preventing two containers from occupying the same space.

(5) A 20' container cannot stand on top of a 40' container

(6) When placing 20' containers, the containers must be placed 'evenly'

(7) Containers must have support from below

(8) One container must be placed in one slot, and in one slot only

(9) Enforces the weight constraints

(10) Requires the discharge port of the containers in each stack to be ordered ascendingly from top to bottom. This ensures that no container can be placed on top of another container that has an earlier discharge port

Note that the model can be simplified by rewriting (5), (6), and (7) as

$$\bigwedge_{\sigma\in S}\ \bigwedge_{\tau\in T_\sigma\setminus\{t\}}\ \begin{array}{l}\Big((c^{\mathsf{FT}}_{\sigma,\tau}=0\vee c^{\mathsf{AT}}_{\sigma,\tau}=0)\Rightarrow\\ (c^{\mathsf{FT}}_{\sigma,\tau+1}=0\wedge c^{\mathsf{AT}}_{\sigma,\tau+1}=0)\Big)\wedge\\ \Big(c^{\mathsf{F}}_{\sigma,\tau}=0\wedge(c^{\mathsf{FT}}_{\sigma,\tau}=0\vee c^{\mathsf{AT}}_{\sigma,\tau}=0)\Rightarrow\\ c^{\mathsf{F}}_{\sigma,\tau+1}=0\Big),\end{array} \tag{11}$$

or in plain words: If either of the TEU slots are empty, both of the TEU slots above must be empty (this takes care of the 'no containers hanging in the air' and 'evenly stacking' constraints, and given (4), also the 'no 20' on top of 40' constraint). If the FEU slot is also empty, the FEU slot above must be empty.

6 Application and Results

The two general configurators were implemented using the BDD-based configuration engine CLab [21], which we extended to also include SAT-based configuration, while the direct SAT configurator was built from scratch. Upon start-up, the program makes the configuration file for CLab, and CLab either compiles a BDD representation or creates a CNF encoding of the problem. Then a GUI, based on the Qt UI framework [30], emerges, as shown in Figure 7. The cells are divided into FTEU, ATEU and FEU slots. Long container buttons at the bottom of the screen represent 40' containers, short buttons represent 20' containers. The color shade of containers is determined by their discharge port or weight. Slots and containers with an 'R' are reefer slots and containers, respectively.

When the user clicks on a slot or a container, the color of the container or slot becomes red if the placement is illegal, or green otherwise, as shown in Figure 7(left). When the user clicks on a slot, the program returns the slot's domain (the legal containers). When the user clicks on a container, the program returns the variables (slots) having the container in its domain. By clicking on a green (legal) container or slot, the container is placed in the slot, and the color turns grey (the shade is still determined by the discharge port or weight of the container). This can be undone by clicking on the grey slot, container, or the 'undo' button.

Slots that cannot have any containers placed in them due to the constraints are colored white. If the reason for this is that there is (or has to be) a container in the overlapping FEU slot (if the slot is an FTEU or ATEU slot) or in the overlapping FTEU or ATEU slot (if the slot is an FEU slot), this is marked with a black dot.

After each user selection, the program searches for any new restrictions, container placements that are now either illegal or required, and enforces these. This will for example happen when the 20' container selected in Figure 7(left) is placed in $c^{\mathsf{FT}}_{2,2}$. The result is shown in Figure 7(right).

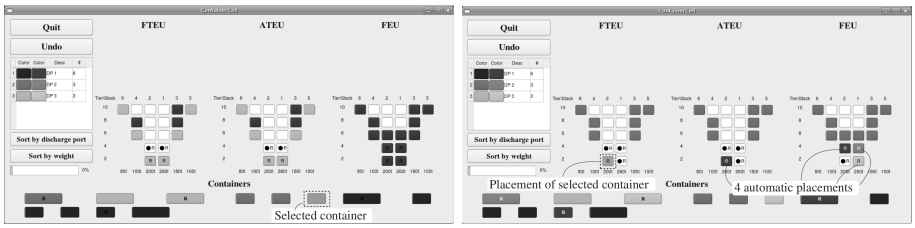

Figure 7: Left: A 25 ton 20' non-reefer container with discharge port 3 selected. GUI shows valid (green / light shaded) and invalid (red / dark shaded) slots. Right: the container placed in reefer slot $c^{\mathsf{FT}}_{2,2}$. GUI has made 4 automatic placements.

Notice that a partial plan does not have to satisfy all constraints (e.g., that the containers form physical stacks). The requirement is that a partial plan can be extended to a complete plan that does satisfy them.

The purpose of the stowage configurator is to support re-assignment of containers in individual locations in the last phase of the stowage planning process where the SC is either checking low-level stacking rules of his or her own plan or is correcting a stowage plan generated offline by an optimization algorithm. In each case, the plan will typically be correct wrt. high-level constraints and objectives such as vessel stability, lid-overstowage, and minimization of crane makespan. For that reason any low-level issues should be solved without moving containers between locations, since this may break the achieved high-level solution.

A configurator should be built for each location. This can either be done online by the SC high-lighting a storage area that a configurator is then built over, or offline as part of running an optimization algorithm. For this to be practical, it is necessary that an initial representation can be compiled in reasonable time, and that the valid assignment computation performed online can be done fast enough that the user experiences it as instantaneous.

Symmetry breaking Often there are containers that share the same characteristics and can exchange position within the location in a stowage plan without breaking any rules. Given such groups of similar containers, we want to change the domain of slot variables from single containers to container class indices. The value of a slot variable then denotes which container class is stored in the slot, rather than exactly which container it is. This breaks symmetry, since the model now ignores the ordering of the containers within each class.

Most of the characteristics of a container are discrete and finite, with only a small number of possible values. But the weight of a container is a continuous value. So, strictly speaking, two otherwise similar containers with a weight of 25.676 and 25.677 tons are not similar and can not be put in the same group. In order to be able to group containers at all, we have to relax the problem.

The relaxation we consider is to allow grouping of containers with 'almost

equal' weight. The containers in a group will get their weight $w(\chi)$ changed to the weight of the group the containers belong to, defined as $w_g(\chi)$. This value must be at least equal to the weight of the heaviest container in the group, to avoid underestimating the weight of a stack. We define 'weight loss'; i.e. the total weight overestimation to be $\sum_{\chi \in C} w_g(\chi) - w(\chi)$.

One could group containers according to pre-defined weight intervals, but this can lead to large losses. Consider a pre-defined grouping with an interval of 2 tons. A set of 4 containers having weights of 23.999, 24.001, 25.999 and 26.001 tons could lead to a solution that is far from optimal. Another approach is to find a solution with minimum weight loss given a limit on the number of container groups. This problem can be solved using dynamic programming. For more details on this, we refer to **(author?)** [22].

Experimental Setup The experiments were performed on a PC running Ubuntu 11.04, with a 12 core 3.33 GHz Intel i7 processor. Each experiment was allowed to use one CPU core and a maximum of 8 GBs of RAM. Each experiment was given a global timeout of 1 hour (CPU time), while a single call to the SAT solver was given a maximum of 5 minutes. Since a domain reduction consists of one solver call per value in the domain of each unset variable, each call to the solver must be extremely fast, and 5 minutes is thus a generous upper bound.

All stowage instances were hand-crafted on the basis of a real-world instance of a stowed bay from a large container vessel, supplied from a major company in the shipping industry. They were designed to keep proportions intact. For all instances, the number of 20' containers is about a fifth the amount of 40' containers, and about one in twenty containers are reefers.

In the experiments we focus on measuring online domain reductions, which is the time it takes to perform a full domain reduction following a user choice. The initial domain reduction, where any immediately invalid assignments are removed, is excluded. Thus, what we measure here is the waiting time the user experiences while using the configurator, without the initial precomputation time (since this can be done in advance). User interaction was simulated by making random variable assignments.

For all our shown results we use symmetry breaking, as we experienced very limited scalability without it, as shown by [22]. By grouping the containers, we only approximate container weights. All other attributes are still accurate. Moreover, the approximation is conservative such that valid relaxed configurations still satisfy the real weight constraints.

BDD Results In table 1 we present the results from the experiments with the BDD-based configurator. Each set of two columns describes the mean and max latencies using the specified maximum number of discrete container groups. Each row corresponds to a single container vessel stowage instance, with the number of cells and containers in the instance given in the two left-most columns. In the case of the BDD, "Time" refers to the BDD compilation process taking

more than one hour, whereas "Mem" refers to the compilation process requiring more than the maximum 8 GB of RAM.

Using 5 container groups, the BDD configurator performs well, solving all instances within reasonable time limits, although the average response time is above 1 second for the instance with 77 cells and 70 containers. Thus, we are able to stow bays in real-world instances, when using symmetry-breaking where only 5 container groups are allowed. As the number of container groups increases, timeouts start occurring on the biggest instances, and with 10 container groups it is only possible to compile the representation for up to 30 containers.

Building the BDD representation never took more than 10 minutes, even for the largest instances. It did, however, require significant amounts of memory, and indeed, all of the timeouts seen in the results are due to garbage collection taking large quantities of time once the memory barrier is hit.

Table 1: Online domain reductions for the BDD-based configurator. The median domain reduction time was generally in the range 0.0 - 0.4, with very few runs going higher. The highest median observed was 0.8 seconds.

Size		Number of Container Groups											
#	#	5		6		7		8		9		10	
Cel.	Cnt.	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max
14	5	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
14	10	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
24	5	0.2	0.2	0.2	0.2	0.1	0.1	0.2	0.2	0.2	0.2	0.2	0.2
24	10	0.2	0.2	0.2	0.2	0.1	0.1	0.2	0.2	0.2	0.2	0.2	0.2
24	15	0.2	0.2	0.2	0.2	0.1	0.1	0.2	0.2	0.2	0.3	0.2	0.3
24	20	0.2	0.2	0.3	0.3	0.1	0.1	0.2	0.3	0.2	0.4	0.2	0.5
34	5	0.2	0.2	0.2	0.2	0.1	0.1	0.2	0.2	0.2	0.2	0.2	0.2
34	10	0.2	0.2	0.2	0.2	0.1	0.1	0.2	0.3	0.2	0.3	0.2	0.3
34	15	0.2	0.2	0.2	0.3	0.1	0.1	0.3	0.5	0.3	0.7	0.3	0.7
34	20	0.2	0.3	0.2	0.4	0.1	0.2	0.3	0.6	0.3	1.3	0.4	2.6
34	25	0.2	0.3	0.2	0.4	0.1	0.3	0.4	2.2	0.6	3.8	0.8	6.0
34	30	0.2	0.2	0.2	0.2	0.1	0.1	0.2	0.3	0.3	0.5	0.3	0.7
44	10	0.2	0.2	0.2	0.2	0.1	0.1	0.2	0.3	0.2	0.3	0.3	0.4
44	20	0.2	0.4	0.2	0.5	0.1	0.3	0.3	1.0	0.4	2.5	0.6	5.3
44	30	0.3	0.4	0.2	0.6	0.1	0.6	0.5	2.5	Time		Time	
44	40	0.2	0.3	0.3	0.7	0.2	1.4	0.7	6.0	Time		Time	
55	10	0.2	0.2	0.3	0.3	0.1	0.1	0.2	0.3	0.2	0.3	0.2	0.3
55	20	0.2	0.4	0.3	0.5	0.1	0.2	0.3	0.8	0.3	2.0	0.4	2.9
55	30	0.3	0.8	0.4	1.4	0.5	2.4	0.9	6.2	Time		Time	
55	40	0.4	2.0	0.8	8.0	Time		Time		Time		Mem	
55	50	0.3	1.0	0.5	3.4	1.8	19.4	Time		Time		Mem	
66	10	0.2	0.2	0.2	0.2	0.2	0.3	0.2	0.2	0.2	0.3	0.2	0.3
66	20	0.3	0.6	0.3	1.0	0.4	1.2	0.4	2.3	0.8	6.8	1.7	14.1
66	30	0.3	1.0	0.5	2.2	0.8	6.7	1.1	9.9	Time		Mem	
66	40	0.5	3.8	2.2	18.1	Time		Time		Mem		Mem	
66	50	0.5	4.5	0.8	9.2	Time		Time		Mem		Mem	
66	60	0.4	1.8	1.7	22.1	Time		Time		Time		Time	
77	10	0.2	0.2	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.2	0.3
77	20	0.2	0.5	0.3	0.8	0.3	1.2	0.3	1.1	0.8	4.4	0.7	5.0
77	30	0.5	2.6	0.7	5.3	1.4	13.4	Time		Time		Mem	
77	40	0.8	6.2	2.0	27.0	Time		Mem		Mem		Mem	
77	50	0.7	9.5	Time		Time		Time		Time		Mem	
77	60	0.6	5.6	Time		Time		Time		Mem		Time	
77	70	1.7	6.6	5.0	58.0	Time		Time		Mem		Time	

SAT Results To get a clear picture of the performance impact of the different encodings, we have performed experiments using order encoding, log encoding as well as the compact order encoding with various orders of magnitude. In table 2 we show the results using the general SAT configurator with the best performing encoding, the compact encoding with m (order of magnitude) set to 3. This table is organized the same way as the BDD table, except that for the SAT configurator, "Time" refers to a timeout with single solver call taking more than 5 minutes, or the whole configuration process of choosing values for all decision variables taking more than 60 minutes. For all encodings the solvers give strong performance for up to 34 slots and 15 containers. However, even for the best encoding depicted here, we see severe scaling problems, as the instance with 34 slots and 20 containers times out with 10 container groups, and the instance with 25 containers times out on all container groupings except one, which has an unacceptable median runtime. Surprisingly, the SAT configurator with compact order encoding is able to solve the instance with 77 slots and 60 containers with 5 container groups for all choices of m, even when it has failed on most instances that are smaller than that. All our experiments with SAT encodings time out on most instances involving more than 20 containers, and thus we see that, in general, the SAT-based configurator using a wide array of encodings performs significantly worse than the BDD-based configurator.

To validate our findings on SAT performance, we took several steps to ensure that the lacking performance was not due to the choice of solver or encoding. First, we implemented an SMT-based solver, encoding straightforwardly from CP format. We used the SMT solver Z3 [27], which has won most categories in recent SMT competitions. This configurator scaled even worse than our SAT based approaches, with timeouts occuring at instances involving 20 containers, and on solved instances it was slower than the SAT based approaches. We also designed and implemented an alternative SAT encoding, which encodes the problem directly using dynamic programming, see appendix, however, this encoding also performed worse than the CSP based encoding.

Second, we compiled a set of 281 hard SAT instances encountered in searches and applied the following solvers to them: Lingeling [6], Glucose [4], Glueminisat [28] and Cryptominisat [33]. The results from this can be seen in Table 3. These were the top 4 solvers in the applications category at the 2011 SAT competition, and as such represent the state-of-the-art in SAT solvers for solving the sort of structured SAT instances that we are dealing with. They also employ some newer techniques which MiniSat does not, such as aggressive preprocessing. Out of the 281 instances, 147 of them were not solved by any of the solvers. Additionally, almost all solved instances took more than 10 seconds to solve. This shows that none of the state-of-the-art solvers are able to solve instances fast enough for interactive stowage.

Interestingly, the hard SAT instances generated were a fairly small fraction of all the solver calls. Even for large instances, most solver calls are easy, and only a few specific assignments lead to intractable problems. It has previously been shown in the literature that the hardness of SAT instances peaks at a certain phase transition related to either the satisfiability threshold as it shifts from

Table 2: Online domain reductions for the SAT-based configurator using compact encoding with order of magnitude = 3. The median domain reduction time was generally less than 0.3 seconds, only to be seen higher in very few of runs bordering on a timeout, such the one for 66 cells, 50 containers and 5 container groups with a median of 6.2 seconds, the highest seen for a non-timeouted run.

Size		Number of Container Groups											
#	#	5		6		7		8		9		10	
Cel.	Cnt.	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max
14	5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14	10	0.0	0.1	0.0	0.1	0.0	0.1	0.1	0.2	0.1	0.2	0.1	0.2
24	5	0.0	0.1	0.0	0.1	0.0	0.0	0.0	0.1	0.0	0.1	0.0	0.1
24	10	0.1	0.2	0.2	1.2	0.1	0.5	0.5	2.2	0.4	2.2	0.6	3.3
24	15	0.1	0.2	0.1	1.1	0.7	5.6	0.3	3.1	0.4	2.8	0.8	7.2
24	20	0.3	2.4	0.4	1.8	0.2	1.5	0.6	7.8	0.6	5.2	Time	
34	5	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
34	10	0.0	0.0	0.1	0.5	0.1	0.3	0.1	0.5	0.1	1.0	0.2	1.2
34	15	0.1	0.6	0.7	2.8	0.5	2.6	0.6	3.1	0.9	5.1	1.6	8.7
34	20	1.2	10.2	0.8	4.0	0.5	6.2	11.9	135.7	2.8	28.2	Time	
34	25	Time		25.6	282.3	Time		Time		Time		Time	
34	30	Time		Time		Time		Time		Time		Time	
44	10	0.2	0.5	0.7	4.7	0.3	2.0	0.9	5.8	1.5	11.3	2.1	14.1
44	20	7.0	100.9	7.8	128.5	4.5	61.5	Time		Time		Time	
44	30	Time		Time		Time		Time		Time		Time	
44	40	Time		Time		Time		Time		Time		Time	
55	10	0.3	0.7	1.3	6.8	0.6	3.1	1.6	8.5	3.4	22.5	4.0	26.1
55	20	16.6	220.7	17.9	257.4	7.6	86.1	Time		Time		Time	
55	30	Time		Time		Time		Time		Time		Time	
55	40	Time		Time		Time		Time		Time		Time	
55	50	Time		Time		Time		Time		Time		Time	
66	10	0.7	2.6	1.7	6.4	0.7	6.4	1.3	11.1	2.5	22.4	2.5	22.8
66	20	Time		18.5	273.0	13.9	199.2	Time		Time		Time	
66	30	Time		Time		Time		Time		Time		Time	
66	40	Time		Time		Time		Time		Time		Time	
66	50	4.5	43.3	Time		Time		Time		Time		Time	
66	60	Time		Time		Time		Time		Time		Time	
77	10	0.2	0.9	1.2	7.6	0.8	5.7	2.5	16.3	5.7	43.1	4.7	36.0
77	20	Time		26.6	1.1	283.3	109.0	Time		Time		Time	
77	30	Time		Time		Time		Time		Time		Time	
77	40	Time		Time		Time		Time		Time		Time	
77	50	Time		Time		Time		Time		Time		Time	
77	60	1.5	8.2	Time		Time		Time		Time		Time	
77	70	Time		4.0	24.1	Time		Time		Time		Time	

underconstrained to overconstrained [13], or the distribution of solutions and structure in instances [9]. As the configurator has to test all potential assignments after each user selected assignment, it inevitably encounters some that have a large impact on the distribution of solutions. For example, assigning a 20' container to the top of a stack will prevent all 40' containers from being placed in that stack. As the configurator tries all these assignments, it moves around in, and has to solve, a large space of possible problems, that have significant differences in how constrained they are and what the solution space looks like. Thus, as the container vessel stowage instances get larger, the chances of testing an assignment that generates a significantly harder SAT instance increases, as many more assignments end up being tried.

Table 3: Performance of various SAT solvers on 281 instances that were hard for MiniSat under different time limits. The last row shows the number of instances that none of the solvers could solve. As can be seen, 147 instances were unsolvable by all algorithms.

Solver	$< 10s$	$< 60s$	$< 300s$	Unsolved
Lingeling	4	67	48	162
GlueMiniSat	0	5	78	198
MiniSat	0	0	84	197
CryptoMiniSat	0	48	77	156
Glucose	0	27	82	172
Unsolvable by all solvers	277	201	147	147

For a more comprehensive treatment of the performance of all the encodings, and to see the complete set of experimental results we refer the reader to **(author?)** [24].

Simplified instances In an effort to get a clearer picture of which components of the container vessel stowage problem make it difficult, we created instance without the weight constraint, the no overstowage constraint, and the gravity constraint. Even this had little impact on the performance. We also performed experiments where the user is not allowed to pick that a cell must be empty. Removing this option along with the three constraints mentioned above led to the solver solving all instances. Since a solver call is needed for each potentially allowed container placement, we also introduce a simplified version of the problem, where users are only allowed to place containers immediately above already placed containers. Such placement actions might be more likely to occur in real-world usage. To test these assumptions, a modified version of the SAT based solution including this limitation was developed, and tested in the same way as previously, with random user choices respecting the new constraint. The results are presented in table 4. The order of magnitude in the compact order encoding has been increased to 4, since this provides better results than 3 for larger instances due to the reasons discussed in section 4.2. This change made the solver scale to much larger instances, solving a significant number of the instances involving 132 cells.

Summarizing, we see that the BDD-based approach is able to stow bays in real-world instances, and with restricted user choices the SAT based approach can scale even beyond a single bay. We also see that for SAT, the general problem is intractable, even when applying many different SAT solvers, and we see that even simplified versions of the problem are usually hard.

Table 4: Online domain reductions for the SAT-based configurator using compact encoding with order of magnitude = 4 where the user is only allowed to place containers immediately above already placed containers. Median solver times generally lie in the range 0.0 - 0.4, with a two outliers of 0.7 and 0.8.

Size		Number of Container Groups											
#	#	5		6		7		8		9		10	
Cel.	Cnt.	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max
24	10	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.1	0.0	0.1	0.0	0.1
24	20	0.0	0.3	0.0	0.4	0.0	0.5	0.0	0.3	0.2	3.7	0.6	10.4
34	10	0.0	0.1	0.0	0.1	0.0	0.1	0.1	0.2	0.1	0.2	0.1	0.2
34	20	0.2	2.8	0.0	0.1	0.2	2.8	0.1	0.2	0.1	0.3	0.1	0.3
34	30	Time		Time		Time		Time		Time		Time	
44	10	0.0	0.1	0.1	0.2	0.1	0.2	0.1	0.3	0.1	0.3	0.1	0.3
44	20	0.0	0.1	0.1	0.2	0.1	0.3	0.1	0.2	0.1	0.5	0.2	0.6
44	30	1.0	26.6	0.3	4.6	0.1	1.2	1.1	25.9	0.9	13.8	3.7	63.7
44	40	Time		Time		Time		Time		Time		Time	
55	10	0.1	0.2	0.1	0.3	0.1	0.3	0.1	0.4	0.2	0.5	0.2	0.6
55	30	0.1	0.2	Time		Time		0.1	0.5	0.2	0.1	0.3	0.8
55	50	Time		Time		Time		Time		Time		Time	
66	20	0.1	0.4	0.1	0.4	0.1	0.6	0.1	0.7	0.4	2.8	1.7	28.7
66	40	0.2	4.2	0.2	0.6	0.2	0.6	Time		Time		Time	
66	60	1.3	44.0	Time		Time		Time		Time		Time	
88	30	0.1	0.4	0.2	0.5	0.2	0.7	0.2	1.0	0.5	1.6	0.6	1.7
88	50	0.1	0.5	0.2	0.8	0.3	0.9	Time		Time		2.7	40.0
88	70	0.2	2.0	0.2	1.2	0.3	1.4	Time		0.6	2.6	0.7	4.1
88	90	Time		Time		Time		Time		Time		Time	
110	30	0.2	0.8	0.2	0.8	0.5	1.1	0.4	1.4	0.6	2.4	0.7	3.3
110	50	Time		0.5	5.0	Time		Time		3.9	114.0	Time	
110	70	0.2	0.8	0.4	1.4	Time		Time		Time		Time	
110	90	Time		Time		Time		Time		Time		Time	
110	110	Time		Time		Time		Time		Time		Time	
132	50	0.3	0.8	0.5	1.6	0.6	2.4	Time		Time		11.3	112.0
132	70	0.2	0.8	0.5	1.9	0.6	2.5	3.5	93.7	2.4	42.4	3.8	49.1
132	90	0.4	2.2	Time		3.4	187.0	0.6	3.8	Time		Time	
132	110	0.6	2.9	Time		Time		0.9	6.2	Time		Time	
132	130	Time		Time		Time		Time		Time		Time	
154	50	0.3	1.5	0.6	2.5	0.7	3.9	9.2	274.5	8.1	251.9	11.3	89.7
154	70	0.6	20.4	1.2	48.8	Time		Time		Time		Time	
154	90	Time		Time		Time		Time		1.8	21.9	Time	
154	110	Time		Time		Time		Time		1.5	7.6	Time	
154	130	Time		Time		Time		Time		1.6	8.3	Time	
154	150	Time		Time		Time		Time		Time		Time	

7 Conclusion

In this article, we set out to investigate whether computational approaches can be used to provide interactive decision support for container stowage. We introduced two general approaches to this problem, one using precompiled BDDs, and one using search methods to compute valid assignments at runtime. For search based stowage, we introduced several approaches, based on both SAT and SMT solvers.

We showed experimentally that using our techniques, real-world interactive stowage instances can be solved. The BDD-based approach solves instances with up to 70 containers when breaking symmetries among similar containers, thus satisfactorily stowing a bay.

We also showed that SAT and SMT based approaches perform significantly worse than BDDs. As the instances get larger, we found that for every stowage instance, the search-based approaches encounter a small number of extremely hard permutations of the problem. We ran a number of state-of-the-art SAT solvers on these hard SAT problems, showing that no solver could solve them. In particular, we found a large group of instances for which no solver could solve any of the instances. We then introduced a restricted version of the problem, which allowed the SAT-based approach to scale to much larger instances. It is an interesting future research direction to investigate other relaxations or restrictions, that could lead to better scalability while still providing a valuable tool for stowage coordinators.

Bibliography

[1] CSP competition 2008. `http://www.cril.univ-artois.fr/CPAI08/results/ranking.php?idev=15`.

[2] CSP competition 2009. `http://www.cril.univ-artois.fr/CPAI09/results/ranking.php?idev=30`.

[3] Daniela Ambrosino, Davide Anghinolfi, Massimo Paolucci, and Anna Sciomachen. An experimental comparison of different heuristics for the master bay plan problem. In *Proceedings of the 9th Int. Symposium on Experimental Algorithms*, pages 314–325, 2010.

[4] Gilles Audemard and Laurent Simon. glucose, solver description. In *SAT Competition*, 2011.

[5] Mordecai Avriel, Michal Penn, and Naomi Shpirer. Container ship stowage problem: complexity and connection to the coloring of circle graphs. *Discrete Applied Mathematics*, 103(1):271–279, 2000.

[6] Armin Biere. Lingeling and friends at the SAT competition 2011. In *SAT Competition*, 2011.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Handbook of satisfiability. frontiers in artificial intelligence and applications, vol. 185, 2009.

[8] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[9] Cristian Coarfa, Demetrios D. Demopoulos, Alfonso San Miguel Aguirre, Devika Subramanian, and Moshe Y. Vardi. *Random 3SAT: The Plot Thickens*. 2000.

[10] Alberto Delgado, Rune Møller Jensen, Kira Janstrup, Trine Høyer Rose, and Kent Høj Andersen. A constraint programming model for fast optimal

stowage of container vessel bays. *European Journal of Operational Research*, 220(1):251–261, 2012.

[11] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[12] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89, 2003.

[13] Ehud Friedgut. Necessary and sufficient conditions for sharp thresholds of graph properties and the k-sat problem. *Journal of the American Mathematical Society 12*, 1999.

[14] Marco Gavanelli. The log-support encoding of CSP into SAT. In *Principles and Practice of Constraint Programming*, pages 815–822, 2007.

[15] N. Guilbert and B. Paquin. Container vessel stowage planning, Patent Publication US2010/0145501, 2010.

[16] Tarik Hadzic, Rune Moller Jensen, and Henrik Reif Andersen. Calculating valid domains for BDD-based interactive configuration. *arXiv preprint arXiv:0704.1394*, 2007.

[17] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Mller, and Henrik Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO CONFERENCE*, pages 131–138. DTU-TRYK, 2004.

[18] Interschalt. Seacos. www.interschalt.de, 2011.

[19] Kazuo Iwama and Shuichi Miyazaki. SAT-variable complexity of hard combinatorial problems. In *IFIP Congress (1)*, pages 253–258, 1994.

[20] Mikolás Janota. *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin, 2010.

[21] Rune M. Jensen. CLab: a C++ library for fast backtrack-free interactive product configuration. In *In Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP-04)*, page 816. Springer, 2004.

[22] Rune Møller Jensen, Eilif Leknes, and Tom Bebbington. Fast interactive decision support for modifying stowage plans using binary decision diagrams. In *International MultiConference of Engineers and Computer Scientists*, 2012.

[23] Johan De Kleer. A comparison of ATMS and CSP techniques. In *International Joint Conference on Artificial Intelligence*, pages 290–296, 1989.

[24] Christian Kroer and Martin Kjær Svendsen. SAT and SMT based interactive configuration for container vessel stowage. Master's thesis, IT University of Copenhagen, 2012.

[25] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications*. Springer, 1998.

[26] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Design Automation, 1993. 30th Conference on*, pages 272–277. IEEE, 1993.

[27] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.

[28] Hidetomo Nabeshima. glueminisat, solver description. In *SAT Competition*, 2011.

[29] Navis. PowerStow. www.navis.com, 2011.

[30] Nokia Corporation. Qt. qt.nokia.com, 2010.

[31] Dario Pacino, Alberto Delgado, Rune Møller Jensen, and Tom Bebbington. Fast generation of near-optimal plans for eco-efficient stowage of large container vessels. In *Computational Logistics*, pages 286–301. Springer, 2011.

[32] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing*, pages 294–299, 2007.

[33] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing*, pages 244–257, 2009.

[34] Sathiamoorthy Subbarayan, Rune M Jensen, Tarik Hadzic, Henrik R Andersen, Henrik Hulgaard, and Jesper Møller. Comparing two implementations of a complete and backtrack-free interactive configurator. In *Proceedings of the CP-04 Workshop on CSP Techniques with Immediate Application*, pages 97–111, 2004.

[35] Naoyuki Tamura and Mutsunori Banbara. Sugar: A CSP to SAT translator based on order encoding. In *Proceedings of the Third International CSP Solver Competition1*, pages 815–822, 2008.

[36] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[37] Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. A compact and efficient sat-encoding of finite domain csp. In *Theory and Applications of Satisfiability Testing-SAT 2011*, pages 375–376. Springer, 2011.

[38] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Logic*, Part II, 1968.

[39] Toby Walsh. SAT v CSP. In *Principles and Practice of Constraint Programming*, pages 441–456, 2000.

[40] Rainer Weigel and Boi Faltings. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115(2):257–287, 1999.

[41] Malcolm Yoke, Hean Low, Xiantao Xiao, Fan Liu, Shell Ying Huang, Wen Jing Hsu, and Zhengping Li. An automated stowage planning system for large containerships. In *Proceedings of the 4th Virtual Int. Conference on Intelligent Production Machines and Systems*, 2009.

A

A.1 Alternative IP-inspired model for SAT

As an alternative to the more general encoding that can be converted into SAT form in a series of steps, we also try a direct encoding of the container vessel stowage problem as a SAT problem. The absence of a complicated conversion process renders this encoding more straightforward, possibly making the structure of the problem more transparent to a SAT solver. For this approach, we introduce a boolean variable for each cell and container combination, signifying whether the container is stored in the cell: b^F_{kji} signifies container $i \in C_F$ being stored in slot s^F_{kj}, b^{AT}_{kji} signifies container $i \in C_T$ being stored in slot s^{AT}_{kj}, b^{FT}_{kji} signifies container $i \in C_T$ being stored in slot s^{FT}_{kj}, and e^l_{kj} signifies slot S^l_{kj} being empty.

A.2 Constraints

All containers must be placed

For each container c we introduce a clause requiring that at least one of the container placement variables is true.

$$\bigvee_{\sigma \in S} \bigvee_{\tau \in T_\sigma} b^F_{\sigma\tau c} \qquad \text{if } c \in C_F \tag{12}$$

$$\bigvee_{\sigma \in S} \bigvee_{\tau \in T_\sigma} b^{FT}_{\sigma\tau c} \vee b^{AT}_{\sigma\tau c} \qquad \text{if } c \in C_T \tag{13}$$

A container can only be placed once

For each pair of Boolean variables (b^m_{ikc}, b^n_{jlc}) representing placement of the container c in slot s^m_{ik} and s^n_{jl}, we require one of them to be false.

$$\neg(b^m_{ikc} \wedge b^n_{jlc}) \tag{14}$$

A cell can only hold one 40' container or up to two 20' containers

For each set of cell slots $\{s^F_{kj}, s^{AT}_{kj}, s^{FT}_{kj}\}$, we introduce the following two sets of clauses, the first ensuring that we can only use the TEU or FEU slots, by requiring either the FEU slot to be empty, or both TEU slots.

$$\bigwedge_{l \in C_F} \bigwedge_{m \in C_T} (\neg b^F_{kjl} \vee \neg b^{FT}_{kjm}) \wedge (\neg b^F_{kjl} \vee \neg b^{AT}_{kjm}) \tag{15}$$

The second ensures that we use each slot only once, by requiring that at least one container placement in all container placement pairs for a given cell is not carried out.

$$\bigwedge_{l \in C_F} \bigwedge_{k \in C_F \setminus \{l\}} (\neg b^F_{ijk} \vee \neg b^F_{ijl}) \tag{16}$$

$$\bigwedge_{l \in C_T} \bigwedge_{k \in C_T \setminus \{l\}} (\neg b^{FT}_{ijk} \vee \neg b^{FT}_{ijl}) \wedge (\neg b^{AT}_{ijk} \vee \neg b^{AT}_{ijl}) \tag{17}$$

Containers must have support from below

For all 40' slots s^F_{kj}, where tier $j > 1$ and containers $l \in C_F$, we require that either the container is not placed in the slot, or there's a 40' container or two 20' containers placed below it.

$$(\neg b^F_{kjl} \vee (\bigvee_{c \neq l \in C_F} b^F_{kj-1c}) \vee (\bigvee_{c \neq l \in C_T} b^{FT}_{kj-1c})) \tag{18}$$

$$(\neg b^F_{kjl} \vee (\bigvee_{c \neq l \in C_F} b^F_{kj-1c}) \vee (\bigvee_{c \neq l \in C_T} b^{AT}_{kj-1c})) \tag{19}$$

And for all 20' slots s^x_{kj}, where $j > 1$ and containers $l \in C_T$, we introduce similar constraints, except we require two 20' containers to be placed below.

$$(\neg b^x_{kjl} \vee (\bigvee_{c \neq l} b^{FT}_{kj-1c})) \tag{20}$$

$$(\neg b^x_{kjl} \vee (\bigvee_{c \neq l} b^{AT}_{kj-1c})) \tag{21}$$

20' containers cannot be placed on top of 40' containers

This is implemented by the constraints (20) and (21), as the 20' foot container conjunctions require 20' containers to be placed beneath them.

20' containers must be placed evenly

This is also implemented by the constraints (20) and (21), as we require containers in both the aft and fore slot below, if a 20' container is placed in a given cell.

Reefer containers can only be placed in reefer slots

This is simply handled by only introducing variables for placement of reefer containers in slots that have reefer capacity. Since constraint (12) requires all containers to be placed, the appropriate amount of reefer slots will be forced to take on the reefer containers.

Overstowage is disallowed

For each container pair $i \in C_F, j \in C$ and each slot s^F_{mn} such that tier n is above 1, i.e. containers in the first tier cannot overstow, and the discharge port of i is later than the discharge port of j, we introduce the following.

$$\bigwedge_{k<n} (\neg b^F_{mni} \vee \neg b^{AT}_{mkj}) \wedge (\neg b^F_{mni} \vee \neg b^{FT}_{mkj}) \tag{22}$$

Similarly, For each 20' container pair $i, j \in C_T$ and each slot s^{FT}_{mn} and s^{AT}_{mn}, such that n is above 1 and $d(i) > d(j)$, we introduce:

$$\bigwedge_{k<n} (\neg b^{AT}_{mni} \vee \neg b^{AT}_{mkj}) \wedge (\neg b^{FT}_{mni} \vee \neg b^{FT}_{mkj}) \tag{23}$$

For each stack, the total weight of all containers in the stack must not exceed the stacks weight limit

To implement the stack weight limit constraint without using arithmetic we introduce a dynamic programming style encoding. One way to specify which container placements are legal for a given stack would be to specify each of the legal ways to place containers in that stack. This could be done with a disjunctive-normal-form sentence where each clause would be one of the legal configurations for the stack. This requires an exponential number of clauses, as there are $n!$ ways to order n containers, which is infeasible. However, as the weight limit of a stack is bounded by a constant, approximately 400, we can use a knapsack style dynamic programming representation to avoid the exponential blow-up.

At any given stack and tier, in order to specify the legal container placements in the stack from the current tier and above, we do not need to know exactly which containers have been placed in the lower tiers, we only need to know the sum of their weights. Note that for this approach to work, we need to disallow placing the same container twice, which is already implemented in constraint (14).

We use the above fact in our dynamic programming representation. For any given state (s, t, W), where s is the stack, t is the tier, and W is the remaining weight limit, we introduce a variable $Y[s, t, W]$ that represents all the legal placements in the stack, from tier t and upwards, with W as the weight limit. This variable is made equivalent to the statement specifying each of the possible legal ways to place containers in the state, which recursively uses variables representing the legal placements for later states $(s, t + 1, W')$. The formal description of this is given below.

$t > \text{MaxTier}(s)$:

$$Y[s,t,W] \Leftrightarrow true$$

$t \leq \text{MaxTier}(s)$:

$$Y[s,t,W] \Leftrightarrow (e^F_{st} \quad \vee$$

$$\bigvee_{c \in C_{40} | w(c) \leq W} b^F_{stc} \wedge Y[s,t+1,W-w(c)] \quad \vee$$

$$\bigvee_{c,k \in C_{20} | w(c)+w(k) \leq W} ((b^{AT}_{stc} \wedge b^{FT}_{stk}) \vee (b^{FT}_{stc} \wedge b^{AT}_{stk})) \wedge Y[s,t+1,W-w(c)-w(k)] \quad \vee$$

$$\bigvee_{c \in C_{20} | w(c) \leq W} (b^{AT}_{stc} \wedge e^{FT}_{st}) \vee (b^{FT}_{stc} \wedge e^{AT}_{st})) \tag{24}$$

The $t > \text{MaxTier}(s)$ case of equation (24) is the base case. MaxTier is a function that returns the highest tier in the stack s. The $t \leq \text{MaxTier}(s)$ case specifies the $Y[s,t,W]$ variable to be equivalent to the disjunction of the rest of the equation. Part 1 of the disjunction specifies the possibility of placing nothing in the cell, which would be the last placement in stack s. Part 2 specifies the next possibility, which is to place one of the 40' containers i in the cell, such that $w(i) \leq W$, and then using one of the legal placements for the following tiers, given the new weight limit $W - w(i)$. Part 3 specifies the same idea, but using two 20' containers, which can symmetrically be placed in either the aft or fore slot. Finally, part 4 allows the placement of a single 20' container, with the other TEU slot being empty. As with placing nothing in the cell, this option is required to be the last placement in stack s, as specified by the constraints 18, 19, 20, 21.