# SAT and SMT-based Interactive Configuration for Container Vessel Stowage Planning

**Master's Thesis**

Christian Kroer
Martin Kjær Svendsen

**Supervisors:**

Rune Møller Jensen
Joseph Roland Kiniry

IT University of Copenhagen
March 1, 2012

**Abstract**

The container vessel stowage problem is a hard combinatorial optimization problem concerned with the placement of containers on a container vessel, subject to various constraints. It is often the case that *stowage coordinators* need to modify existing stowage plans, while preserving their optimality. An *interactive configuration system* can guide the coordinator in this process. Such systems have previously been implemented using *binary decision diagrams* (BDDs), but this approach has been shown to have limited scalability. This thesis explores the usage of *Boolean satisfiability* (SAT) and *satisfiability modulo theory* (SMT) solvers as engines for such systems, to gain a better understanding of the scalability offered by using these approaches. Several methods for encoding the container vessel stowage problem as a SAT or SMT problem are introduced, and experimental results comparing the performance of BDD, SAT and SMT-based interactive configuration are presented, both on the container vessel stowage domain and on several other domains. The results show that SAT and SMT-based approaches suffer from scalability problems on the container vessel stowage domain, while providing strong performance on some other domains. In addition, experiments with simplified versions of the container vessel stowage problem are presented, giving insight on what makes the container vessel stowage problem hard for SAT and SMT-based interactive configuration systems.

# Contents

# Chapter 1

# Introduction

Since its takeoff in the 70's, containerized shipping has grown to become a cornerstone in the global economy. This form of standardized, low-cost, seaborne transportation of goods has facilitated an unprecedented growth in the international trade, and heightened the integration of the world's major economies.

## 1.1 The container vessel stowage problem

In the operation of container shipping services, the creation of high quality container stowage plans is a key factor in ensuring both the safety and effectiveness of transportation. A stowage plan describes where containers should be placed on a ship, subject to constraints derived from both the physical properties of the ship and the need for cost effective transportation. A good stowage plan leads to benefits such as shorter port stays and reduced sailing speed, which in turn leads to lower fuel consumption.

However, producing optimal stowage plans is a combinatorial problem with several NP-hard components [1–3], naturally becoming increasingly difficult to solve as the capacity of container ships continue to increase. Thus, as the prospect of solving this kind of problem optimally by hand has long faded, the need for effective techniques for stowage plan representation and optimization using computer algorithms has become increasingly manifest.

## 1.2 Interactive configuration

Providing useful tools for aiding stowage coordinators in carrying out their job efficiently does, however, not merely require the computation and presentation of ready-made optimal stowage plans. In their daily work, stowage coordinators are often faced with the need to change a stowage plan on the basis of unforeseen events such as changes in customer agreements, equipment failure or customers introducing special requirements that cannot be represented in the general model of the problem.

Thus, to be able to carry out their task effectively under such circumstances, the stowage coordinators need a flexible way of modifying stowage plans, without breaking their optimality. This goal can be accomplished using a *complete, backtrack-free interactive configurator*. In general, this is a tool that enables users to freely modify any configuration to suit their needs, without breaking its feasibility.

*Complete* means that all legal configurations are possible to reach from the starting point, such that the configurator only ever disallows assignments that would lead to invalid configurations.

*Backtrack-free* means that *all* assignments that result in an invalid configuration are removed at each step of the configuration process, such that the user can never make a selection that will result in an unfeasible solution, requiring backtracking of assignments [4].

In the container stowage domain, this means enabling the user to freely add and remove containers from the stow, allowing only container placements that are part of some optimal plan, while at the same time allowing the user to reach any existing optimal plan.

## 1.3 Interactive configuration approaches

Generally there are two approaches to the implementation of an interactive configurator, those based on precompilation of information, and those based on searching for solutions during the configuration process.

*Precompilation approaches* try to build a data structure up-front, representing the solution space in a *compressed form*. This gives access to quicker online computations when the structure has been built. The problems inherent in this approach are the difficulties of keeping both precompilation time and the size of the final structure within reasonable limits.

Reduced Ordered Binary Decision Diagrams (BDDs) are an example of such a data structure. They are the predominant precompilation approach used in interactive configuration [4–8], and have been applied to the container stowage problem, successfully providing interactive configuration for problems consisting of up to 60 containers [9]. It did, however, demonstrate limited scalability beyond that, since both the precompilation time and size of the BDD structure grew too fast as the problem size increased.

*Search-based approaches* employ a lazy strategy by postponing the computation of valid user choices until needed, when the user makes a change to the configuration. This avoids the need to precompile a large structure describing the solution space of the problem, out of which only a small fraction is touched in a concrete use session. The central problem of this approach is the worst case exponential time complexity of the search. However, different problem domains have different characteristics, and for many concrete domains the exponential worst case can be avoided entirely using state-of-the-art search technology.

One search-based approach involves encoding the instance as a constraint satisfaction problem (CSP) and searching for solutions using a CSP solver. This

has previously been shown to perform significantly worse than a BDD-based solution on several industrial configuration problems, some of which are part of the CLib package [10] (see also 4.1.2), despite the use of a state-of-the-art solver [4].

Another search-based approach is to encode the problem as a *Boolean satisfiability problem* (SAT). A study from 2004, which compared the search-based SAT solver zChaff [11] with a BDD-based approach on various problem domains, concluded that no approach dominates across all problem domains. Rather, they found that for some domains SAT solving is more efficient, whereas for others BDD-based solving is better [12]. One important difference between interactive configuration and these previous problem domains is that in interactive configuration the same problem is solved many times, but with different variable assignments. For BDD-based solving this is not a problem, as the initial compilation of the problem can be reused, whereas search-based solutions need to continually search for new satisfying assignments every time a new assignment is made.

However, SAT solving is a heavily studied research area, with steady improvements to solvers being introduced over the years, as evidenced in the SAT solver competition [13]. Due to this, there is reason to believe that SAT-based interactive configuration is becoming increasingly viable. Supporting this view, a 2010 thesis put forward the claim that modern SAT-based configurators do not suffer from the scalability problems of the BDD approach. This was based on empirical results showing that the response time of computing valid configurator choices using SAT grows linearly in the size of the selected problems, which consisted of feature models and example problems used in SAT competitions and benchmarking [14]. In addition, the *satisfiability modulo theories* (SMT) problem [15], which is a generalization of SAT capable of handling more complex constraints such as those found in CSPs, has been heavily researched in recent years, and solver performance has rapidly increased, as seen in the SMT solving competition [16].

### 1.3.1 Thesis question

Based on these promising results for SAT-based interactive configuration and the general advances in SAT and SMT solving, we want to investigate the following question:

*Can a SAT or SMT-based interactive stowage configurator outperform the BDD-based solution in terms of scalability?*

## 1.4 Thesis contributions

In this thesis we make the following contributions.

- An implementation of a general SAT-based interactive configurator, that accepts problems specified in the CSP style modeling language of CLab [17].

- An implementation of a general SMT-based interactive configurator, analogous to the above.
- A model and implementation of the container vessel stowage problem for interactive configuration, encoded directly in SAT form.
- Experiments on solving container vessel stowage with these three SAT and SMT-based implementations, and performance comparisons with a BDD-based configurator.
- Performance comparisons of the general SAT-based configurator with the BDD configurator on a set of problems from various other domains.

## 1.5 Organization

The rest of this thesis is organized as follows.

In chapter 2 we provide background information in the form of a formal description of the fundamentals of interactive configuration, as well as the basics of search-based and precompilation-based approaches to interactive configuration. For precompilation we focus on BDDs, and for search we focus on SAT and SMT-based search. This is followed by an introduction to the container stowage domain, with descriptions of the most common container types and the structure of a container transportation vessel. We also provide a brief overview over previous research in the container stowage planning domain. Lastly, we present the constraints we include in our model of the problem.

In chapter 3 we describe the three different search-based configurator approaches we have implemented: Two based on extensions to the general-purpose BDD-based configurator package CLab, enabling SAT and SMT-based configuration, and one based on a direct encoding of the container stowage problem to the CNF format for use with a SAT solver.

In chapter 4 we present and compare the results of the different container stowage configurator implementations, and present new experiments designed to probe into a major exponential complexity issue encountered in the initial results for all search-based configurator implementations. We also present results on a host of different problems from the CLib package.

Finally, we conclude on the work done and results obtained in chapter 5, and discuss directions and ideas for future work in chapter 6.

# Chapter 2

# Background

In this chapter we give an overview of what interactive configuration is and describe the most common solution techniques, along with a description of the current state-of-the-art in this field. We then give a formal definition of the problem and solution techniques. Lastly, we give an overview of current research in the area, followed by a formal description of the concrete container vessel stowage problem we are solving.

Interactive configuration is the process of helping users find a solution to a constraint satisfaction problem (CSP). Therefore, we start out by formally defining the structure of this problem, before moving on to defining interactive configuration.

## 2.1 CSP

A *constraint satisfaction problem* $\mathcal{C}$ is formally defined 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 $F = f_1, f_2, \ldots, f_m$, where each $f_i$ represents a constraint, and is a set of tuples from $D$, where each tuple specifies an assignment of values to all variables in $X$, such that they satisfy the constraint for $f_i$.

A solution to a CSP is an assignment $\alpha$ of the variables in $X$, where $\alpha \in \bigcap_{i=1}^{m} f_i$. Deciding whether a configuration problem is satisfiable is NP-complete. Since each $f \in F$ can be described by a propositional formula, this can be easily shown by reducing the Boolean satisfiability problem to it.

**Example 1.** Consider specifying a T-shirt by choosing the color (black, white, red, or blue), the size (small, medium, or large) and the print ("Men In Black" - MIB or "Save The Whales" - STW). There are two rules that we have to follow: If we choose the MIB print then the color black has to be chosen as well, and if we choose the small size then the STW print (including a big picture of a whale) cannot be selected as the large whale does not fit on the small shirt. The CSP $(X, D, F)$ of the T-shirt example consists of variables

| | | |
|---|---|---|
| $(black, small, MIB)$ | $(black, large, STW)$ | $(red, large, STW)$ |
| $(black, medium, MIB)$ | $(white, medium, STW)$ | $(blue, medium, STW)$ |
| $(black, medium, STW)$ | $(white, large, STW)$ | $(blue, large, STW)$ |
| $(black, large, MIB)$ | $(red, medium, STW)$ | |

Figure 2.1: Solution space for the T-shirt example.

$X = \{x_1, x_2, x_3\}$ representing color, size and print. Variable domains are $D_1 = \{black, white, red, blue\}$, $D_2 = \{small, medium, large\}$, and $D_3 = \{MIB, STW\}$. The two rules translate to $F = \{f_1, f_2\}$, where $f_1 = \{(x_1, x_2, x_3) \mid (x_3 = MIB) \Rightarrow (x_1 = black)\}$ and $f_2 = \{(x_1, x_2, x_3) \mid (x_3 = STW) \Rightarrow (x_2 \neq small)\}$.

There are $|D_1||D_2||D_3| = 24$ possible assignments. Eleven of these assignments are valid configurations and they form the solution space shown in Fig. 2.1. Example borrowed from [4]. $\Diamond$

## 2.2 Interactive configuration

As mentioned in the introduction, *interactive configuration* is the process of guiding the user through the configuration process, such that only legal assignments are allowed. We now give a formal description of how this is done.

The configurator takes as input a CSP with a set of variables $X$, their finite domains $D$ and a set of constraints $F$. Initially, all variables are unassigned, and the user is free to pick any variable assignment $x_i = v$ that can be part of a legal configuration, where $v \in D_i$. Once the user has chosen some assignment, the configurator removes all invalid domain values such that each leftover value is part of at least one valid assignment with the currently selected user assignments. The user is presented with this reduced set of free variables and corresponding domain values, from which a new preferred assignment is picked. This process repeats until all variables have been assigned values, at which point a valid configuration has been found. The algorithm for this is described in Algorithm 1.

```
1 R ←INITIALIZE(C)
2 while |R| > 1 do
3     choose (x_i = v) ∈ VALID-ASSIGNMENTS(R)
4     R ← UPDATE(R, x_i = v)
```

**Algorithm 1**: Configurator.

In line 1, $R$ is initialized to represent all legal variable assignments $\bigcap_{i=1}^{m} f_i$, before any user choices have been made. In line 3 the user selects some assignment from the currently valid assignments in $R$, and line 4 updates $R$ to reflect the user selection in line 3.

Note that the UPDATE procedure, which is described in Algorithm 2, not only removes invalid domain values, but also forces variables with only a single

```
1 Input: R, x_i = v
2 R ← {(x_1, x_2, ..., x_n) ∈ R | x_i = v}
3 for x ∈ SUPPORT(R) do
4     for val ∈ D_x do
5         if ISUNSATISFIABLE(R, x = val) then
6             D_x = D_x \ val
7     if |D_x| = 1 then
8         x = D_x
9 return R
```

**Algorithm 2**: The UPDATE procedure. SUPPORT($R$) returns all the variables in $R$ that have not yet been assigned values.

domain value left to take on that value, and removes the variable from selection by the user.

**Example 2.** For the T-shirt problem, the assignment $x_2 = small$ will, by the second rule, imply $x_3 \neq STW$ and since there is only one possibility left for variable $x_3$, it follows that $x_3 = MIB$. The first rule then implies $x_1 = black$. Unexpectedly, we have completely specified a T-shirt by just one assignment. $\Diamond$

Implementation techniques for interactive configuration can generally be split into two groups, those utilizing some sort of precompilation phase for fast look-up during the configuration process, and those utilizing search during the configuration process [4].

### 2.2.1 Precompiling

Precompilation is an eager approach that tries to do as much computation as possible up front, in order to compile a compact representation of the entire solution space of the problem, thereby allowing for faster look-up times during the interactive configuration process. In order to do this, a two phase approach is adopted. The first phase compiles a compact representation of the solution space, in Algorithm 1 this is line 1, where the procedure INITIALIZE is replaced with a COMPILE procedure, that builds the compact representation. This is done offline. The second phase is the interactive configuration process, line 2 to 4 in Algorithm 1, where the compact representation is used to implement an efficient UPDATE procedure, which only needs to do a look-up in the compiled data structure, to determine feasibility for any given assignment.

#### BDD-based precompilation

As mentioned in the introduction, the most widely used data structure for interactive configuration with precompilation is BDDs. A BDD is a directed acyclic graph (DAG) used for representing Boolean functions [18]. There is only one or two leaf nodes in a BDD, called *terminal nodes*, which represent the values *true* and *false*. All other nodes in the graph, called *non-terminal nodes*, represent

Boolean variables, and have two outgoing edges, called the *high* and *low* edges, representing the corresponding variable being assigned *true* or *false*, respectively. To represent a Boolean function as a BDD, a variable ordering must be selected. The variable ordering requires that the edges of a given non-terminal node can only point to nodes that represent variables placed later in the ordering, or the *true* and *false* nodes. For a given Boolean function and variable ordering, the BDD representation is canonical, because the BDD is reduced to its most compact form, given the variable ordering.

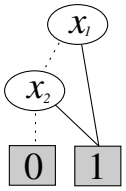


Figure 2.2: A BDD of the function $f(x_1, x_2) = x_1 \vee x_2$ using order $x_1 \prec x_2$. High and low edges are drawn with solid and dashed lines, respectively [9].

An example of a BDD is given in Figure 2.2. This BDD represents the function $f(x_1, x_2) = x_1 \vee x_2$ with the order $x_1 \prec x_2$. The function is satisfied if either variable is assigned the value *true*. This is reflected in the BDD with both high edges for the variable representing nodes leading to the terminal *true* node, while *low* for $x_1$ leads to the $x_2$ node, which also needs to select its *low* node in order for the BDD to return *false*.

If a Boolean function always returns *false*, it is represented by the single terminal *false* node, while a single *true* node means it always returns *true*. This means that checking whether a Boolean function in BDD form is unsatisfiable amounts to checking whether the BDD is equal to the single *false* node, and vice-versa. Moreover, checking whether a partial assignment leads to unsatisfiability can be done in time linear in the size of the BDD, by checking whether the *true* node can be reached, using only the edges that are valid given the current set of assignments.

The operations on the built BDD are all polynomially bounded. This includes extraction of valid domain values, which can be done using a specialized extract function. Generally, applying Boolean operators on two BDDs $f \otimes g$ can be done in $O(|f||g|)$. The construction procedure is however exponential in the worst case. Furthermore, the size of the BDD is also exponential in the worst case. In practice, BDDs for a range of real world configuration problems have been found to be both small and computable in reasonable time [4, 9], but specific constraints such as the alldifferent constraint are known to cause an exponential blowup in the size of the resulting BDD. This is important, as the alldifferent constraint is part of the container stowage problem.

### 2.2.2 Search-Based Interactive Configuration

Where precompilation approaches are eager, search-based approaches are lazy. If we look at Algorithm 1 again, the INITIALIZE procedure for a search-based approach just converts the configuration problem $\mathcal{C}$ into a suitable representation given the chosen search technique. The search for solutions happens in Algorithm 2, where solutions are searched for as they are needed in the ISUNSATISFIABLE procedure at line 4. Every time satisfiability needs to be decided for some assignment, the search procedure solves the configuration problem arising from adding the assignment to the current state of the configuration problem.

As mentioned previously, deciding satisfiability of a configuration problem is NP-complete. Since the search-based approach lazily tests for satisfiability every time it needs to test out a new assignment, it must solve NP-complete problems many times during a configuration process. While this is a discouraging fact, practical results have shown that most of these problems can be solved relatively fast [4, 14].

#### CP

Traditionally, search-based solutions for interactive configuration have been implemented with CP solvers [4, 19]. This is not surprising, as CP solvers will accept a configuration problem as it is, with no encoding required. Input for a CP solver consists of a CSP problem $(X, D, C)$, where $X$ is a set of variables, $D$ is a mapping from variables to finite sets of integer values, which are the domains of the variables, and $C$ is a set of constraints. The CP solver uses *consistency checking*, which means that inconsistent values are removed from the domains of the variables. This is coupled with search, where each step in the search consists of the solver assigning some value to a selected variable and performing consistency checks on the resulting problem. This process is continued until it either hits a satisfying solution, or the domain of some variable becomes empty, at which point the solver must *backtrack* and try other values for some or all of the previously assigned variables.

#### SAT

While CP solvers allow for a quite natural representation of the configuration problem, they do not have the strong propagation power that a SAT solver does. When a SAT solver discards a value, it can infer that the Boolean variable must necessarily be set to the only other remaining value. Recent results for SAT-based interactive configuration have shown promising performance characteristics [14], and SAT-based solving has previously been shown to perform well in other domains such as planning [20]. Also, recent CSP solving competitions have seen strong performance from SAT-based solvers [21].

In SAT-based solutions, the configuration problem is encoded as a Boolean satisfiability problem, which is a propositional logic formula. Modern SAT solvers are predominantly designed to work with SAT problems represented in conjunctive-normal-form (CNF), where problems are represented as a set

of *clauses*, each consisting of one or more *literals*, where at least one must be satisfied by evaluating to true, thereby making the whole clause satisfied. A SAT problem in CNF form can be described by the following expression.

$$\phi \equiv \phi \wedge clause \mid clause$$
$$clause \equiv clause \vee literal \mid literal$$
$$literal \equiv variable \mid \neg variable$$

Any SAT problem can be converted to a logically equivalent CNF form. However, converting some SAT problems to their CNF form can result in an exponential blowup in size. To mitigate this, there are transformations which preserve equisatisfiability as opposed to logical equivalence. These work by introducing new variables representing sub-problems of the SAT problem, and are linear in the size of the original problem [22].

Deciding whether a SAT problem is satisfiable is a topic that has received a great deal of attention over the years. In 1962 the DPLL algorithm was introduced [23], which laid the foundation for most of the modern complete solvers. DPLL is a depth-first search that branches on variables at each node, by assigning *true* or *false* to the variable, and recursively trying to find a solution with this new model. With a branching factor of two, this procedure is $O(2^n)$, but the DPLL algorithm uses some techniques to attempt to mitigate this bad runtime. The pseudo code is shown in Algorithm 3.

```
1  DPLL(φ)
2  if IsSatisfied(φ) then
3      return true
4  if HasEmptyClause(φ) then
5      return false
6  if HasPureSymbol(φ) then
7      φ = AssignPureSymbols(φ)
8      return DPLL(φ)
9  if HasUnitClause(φ) then
10     φ = UnitPropagate(φ)
11     return DPLL(φ)
12 v =PickVar(φ)
13 return DPLL(φ ∪ v = false) ∨ DPLL(φ ∪ v = false)
```

**Algorithm 3**: A recursive version of the DPLL algorithm. The UnitPropagate and AssignPureSymbols procedures are described in the text below.

First, the algorithm does not need a complete assignment. As long as every clause is satisfied it can conclude satisfiability. This can sometimes be achieved without assigning values to all variables, as every clause just needs one literal satisfied in order to be satisfied. As clauses are satisfied, they are removed from consideration.

Second, it uses the *pure symbol heuristic*, which watches for variables always appearing with the same sign in all clauses, and assigns the corresponding sign

to that variable without branching. For example, in the following sentence $(A \vee B) \wedge (A \vee \neg B)$, $A$ appears only positively, and hence the algorithm can assign *true* to A without branching. In Algorithm 3, this heuristic is used in line 6 to 8.

Third, it uses *unit clause propagation*. Whenever some clause has only one literal in it, it can be concluded that it is necessary to satisfy this literal, and hence the algorithm makes this assignment. This is propagated, such that all other clauses with the literal are removed, and all clauses with the negation of the literal have the negated literal removed, which can in turn lead to further unit clause propagation. As an example, consider the sentence $(A \vee B) \wedge (\neg A \vee C) \wedge (\neg C)$, where the DPLL algorithm will detect the unit clause $(\neg C)$, and assign *false* to $C$. This is propagated, and the literal $C$ is removed from the second clause, which in turn creates the unit clause $(\neg A)$, so unit propagation continues, assigning *false* to A and finally assigning *true* to $B$. In Algorithm 3 this heuristic is used in line 9 to 11.

In the last 15 years, several improvements to the DPLL algorithm have been proposed. We will briefly mention some of the most important ones used by the solver we deploy, which is a *conflict driven clause learning* (CDCL) solver.

*Watched literals* is a method for making unit propagation more efficient. It works by keeping track of only two literals from each clause, in order to know when unit propagation from the clause can occur. This works because a clause cannot propagate until all literals except one have become false, and it saves a lot of computation because backtracking does not require modification of the watched literals [24].

*Conflict clause learning* is another important aspect of modern SAT solvers. It keeps track of combinations of assignments that lead to unsatisfiability. Such combinations of assignments are called *conflict clauses*, and the basic idea is to remember these, such that the same conflicting assignments are not made in other branches of the search [25].

To guide the search, *activity heuristics* are used for deciding which variables to branch on. They measure how often variables occur in conflict clauses, and variables with high activity are preferred for branching [11].

Finally, we will also mention *backjumping*. This is a method for backtracking, where the backtrack immediately jumps to the last time a literal in the current conflict clause was assigned, thereby avoiding large parts of the search space that would lead to the same conflict [25]. Several other techniques have been introduced in recent years, such as intelligent restarting [26, 27] and phase saving [28].

One solver, MiniSat, is of particular interest to our work, as it deploys most of the modern techniques for SAT solving, while being implemented with the specific goal of easy extensibility and usage in mind. Furthermore, it supports incremental solving, which is highly relevant, as we are repeatedly solving the same SAT problem with extra constraints added. Incremental solving in MiniSat is implemented by allowing the calling code to specify a list of assumed literals when calling solve, which are then used as unit clauses, thereby specifying the incremental version of the problem. This approach allows for learned conflict

clauses to be reused in the incremental solving process, even when assumptions are removed. This technique has been found to substantially improve solving of incremental problems in [29].

In addition to DPLL-based solvers, a substantial amount of work has been devoted to local search-based (LS) solvers. It was shown in [30] that most satisfiable SAT problems can be solved quickly through LS. However, LS solvers are incomplete, since they can never prove that an instance is unsatisfiable. In this work we focus on DPLL solvers, since we require completeness. However, LS solvers are not wholly uninteresting for our work, as they could be used in parallel with DPLL solving, to speed up solving of satisfiable instances.

Another approach that has gained popularity in recent years is portfolio-based solving [31]. A portfolio is a collection of several solvers, with some technique for utilizing the different strengths of the solvers. This selection is usually done through machine learning techniques such as runtime prediction or clustering of instances [32, 33]. While this thesis focuses on using a single solver, it is a possible direction for future work to consider how portfolio-based solving can be used to speed up the solving process.

#### SMT

SMT problems are a generalization of SAT problems, that are much more expressive [15]. More precisely, an SMT problem is a SAT problem where some of the Boolean variables are replaced by Boolean predicates over non Boolean variables. As an example, consider the expression $a + b > c$, where $a, b, c \in \mathbb{Z}$. This inequality is either satisfied, in which case the expression evaluates to true, or not satisfied, in which case it evaluates to false. When specifying an SMT problem, a set of *theories* is selected. These theories specify which types of predicate expressions are legal. For the example above, we would need the theory of linear arithmetic on the integers. SMT problems are solved with a modified version of the DPLL algorithm, called *DPLL(T)* [34], where the T signifies a solver T for reasoning about the theories selected. This solver is integrated with the DPLL procedure, such that the DPLL procedure reasons about the Boolean logic, while the T solver checks feasibility on the theory predicates.

By using sufficiently expressive theories, it is possible to specify a configuration problem as an SMT problem directly, with no translation needed.

In this work we use the SMT solver Z3 [35], which is a well documented solver that accepts the full specification of the SMT-LIB standard [15]. It has proven to be very efficient, winning almost all awards in the SMT competition in recent years [16].

## 2.3 The stowage domain

Recall that the goal of a container stowage problem is to place a number of containers onto a container vessel, subject to a number of constraints. An

important part of those constraints arise from the properties of the different types of containers used, as well as from the design of container vessels.

To provide a better understanding of the issues arising from real-world container stowage, we describe the most common types of containers, followed by a design description of the cargo space in a container ship. We then present a short overview of research conducted in the field of non-interactive container stowage, before finally describing the specific stowage problem we cover in this thesis, along with its associated constraints.

### 2.3.1 ISO Containers

A standard ISO container (DC or Dry Cargo), is usually either 20 or 40 feet long, 8 feet wide and 8.6 feet high. 20' containers are a little shorter than 40' containers, making it possible to stow two 20' containers in a 40' bay. Longer containers with a length of 45, 48 and 53 feet exist, but are less common.

All containers are equipped with reinforced castings in each corner, designed to withstand great force. They carry the weight of containers stacked on top of each other, and are used as lift points when moving containers. It should be noted that 20' containers cannot be stacked on top of 40' containers due to their lack of castings in the middle.

#### Container types

Besides the standard ISO container described, several other container types exist. The major ones are:

- **High-cube containers**, usually refers to containers that are 9.6 feet tall, but containers of other non-standard heights exist. Note that there does not exist 20' high cube containers.
- **Reefer containers**, having a refrigeration module installed. They need a connection to the ship's power-supply.
- **IMO containers**, containing hazardous material. These containers have to be physically separated from other containers of the same type.
- **Pallet-wide containers**, being a little wider than normal to make a pallet fit inside. They use up the small gap between containers in stacks, and thus, two pallet-wide containers cannot be vertically aligned in adjacent stacks. Also note that stacks do get horizontally out of sync when high cube containers are placed in them. This can affect the possibility of placing pallet wide containers, as illustrated in figure 2.3.

### 2.3.2 Container vessel layout

Container vessels are large ships designed solely for container transportation. Their capacity is measured in Twenty-Foot Equivalent Units (TEU), where one

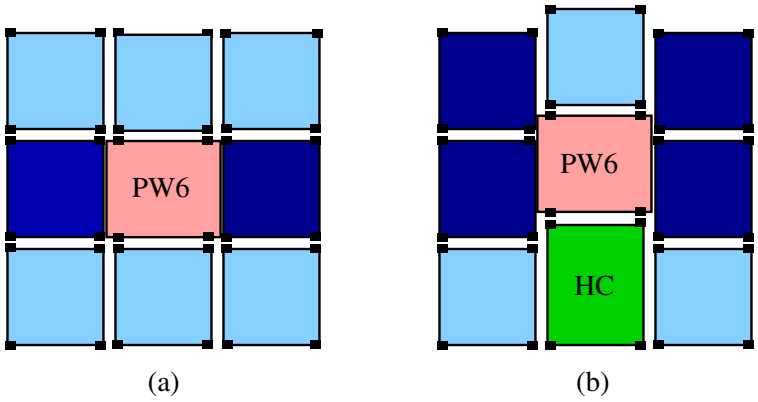


Figure 2.3: Three container stacks seen from the end (a) A pallet-wide container (red) aligned with normal ISO containers (blue) and affecting two of them (dark blue). (b) A pallet wide container (red) misaligned with normal ISO containers (blue) due to a high-cube container (green) causing the pallet wide container to affect four ISO containers (dark blue) rather than two.

40' container is equivalent to 2 TEUs. Today's vessels have a capacity in the range of 1.000 to 15.000 TEUs.

The cargo space of a vessel is divided into a number of bays, each consisting of a number of container stacks placed along the length of the ship. All bays are divided into an over deck and an under deck area, separated by flat, water-tight structures called *hatch covers*. A hatch cover, along with all the containers resting on it, has to be removed in order to gain access to the containers stored underneath.

Containers are stacked in vertically aligned cells, each matching the dimensions of an ISO 40' container. Each cell can either hold a single 40' container or two 20' containers. When placing a 40' container in a cell, it is said to be in a Forty-Foot Equivalent Unit (FEU) slot, and when placing two 20' containers in a cell, they are said to be in the *aft* and *fore* Twenty-Foot Equivalent (TEU) slots, referring to the two ends of the cell they are placed in.

If power plugs are available at a slot, it is said to be a *reefer slot*, accepting reefer containers.

Each cell on a vessel can be located using a (bay, stack, tier) tuple of numerical values. Even bay numbers point to FEU slots, while the associated aft and fore TEU slots of the same cell location are identified by odd bay numbers representing the FEU bay number $\pm$ 1. Stacks are numbered from the center of the ship and out, odd numbers in one direction, even numbers in the other. The container at a given height in a stack are identified by a tier number. Only even numbers are used. Under deck tier numbering starts from the bottom of the stack at 2, while the count of over deck stack tiers traditionally starts at 80. This numbering scheme is illustrated in figure 2.4.

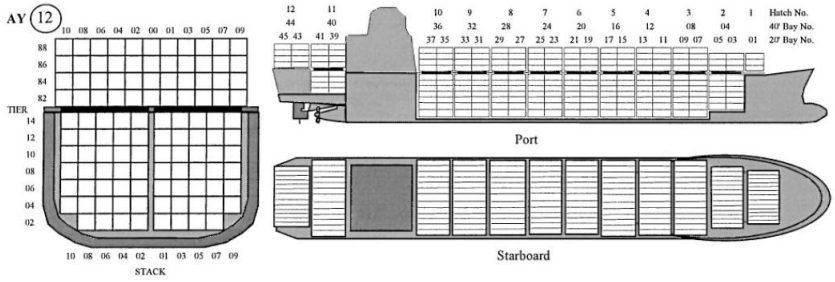


Figure 2.4: Stowage arrangement of a cellular container ship [36].

Containers of a non-standard length above 40 feet can normally be put in stacks of 40' containers, because they have extra castings aligned with those of a 40' container. This is illustrated in figure 2.5. However, this can usually only be done upper deck, since the slots under deck are normally limited in space to standard length containers.

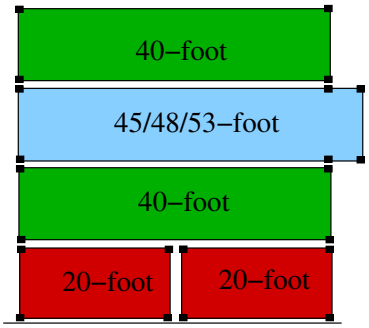


Figure 2.5: An example of a valid upper-deck container stack including a non-standard length container. The small black squares illustrate casting positions.

Each stack has a defined weight limit that cannot be exceeded. Thus, the sum of container weights in the stack must not be larger than its weight limit.

In addition to the container properties mentioned, all containers in a stowage problem have an associated discharge port. Since a container cannot be moved without removing the containers on top of it, it is important that the containers with the earliest discharge port are located at the top of stacks. If a container is stowed on top of another container that has an earlier discharge port, the top container needs to be moved to access the container underneath, which is then said to be *overstowed*. While overstowage is not prohibited in real-life container stowage, it is undesirable as time and money is wasted by unnecessary container movements.

### 2.3.3 Previous research on container stowage planning

To our knowledge, the only previous work done in the field of backtrack-free configuration for container stowage planning is [9]. This work focuses on providing interactive configuration for a single *location*. A location is a sub-part of a bay section located under deck, typically consisting of 2 to 65 cells for a large deep sea vessel. Being under-deck narrows the set of constraints down by avoiding difficult non-linear physical constraints of over-deck stowage. It also restricts the problem complexity by avoiding containers larger than 40', since they cannot normally fit into under-deck bays. This brings down the problem size to a manageable level for configurator implementation, while still preserving the NP-complete nature and thus the difficulty of of the problem.

[9] also includes the presentation of a simple graphical user interface where the user can place containers based on visual feedback about feasible placements, using a simple mouse-driven interface. Larger, commercial solutions of this type exist, such as [37, 38], but these systems lack the backtrack-free capability demonstrated in [9]. In those systems, when the user makes a choice that leads to an infeasible plan, they inform the user that the placement has rendered the plan infeasible, but the user is not prevented from making the choice in the first place. This could lead to large amounts of backtracking, as the source of the infeasibility might not be the most recently selected container.

All previous academic work in this field of study has focused on producing singular optimal stowage plans. Such a plan is required as the basis of the configurator approach just mentioned, since the configurator works by rearranging just a sub-part of a master plan. In the following, we will briefly sketch the different approaches that have been applied to the problem on the master level.

*Multi-phase approaches* work by decomposing the problem hierarchically into a number of smaller sub-problems. The most successful approaches demonstrated have been 2-phased [2, 36, 39, 40] or 3-phased [41, 42].

*Single-phase approaches* keep the representation of the problem or a chosen sub-problem in a single optimization model. Common approaches include:

- **MIP (Mixed Integer Programming)** encodes the problem using a number of variables, where some or all are restricted to the integer domain. See [43–46].
- **CP (Constraint Programming)** encodes the problem as constraints over a series of variables, onto which CP-solvers are applied. See [47, 48].
- **GA (Genetic Algorithms)** applies techniques inspired by natural evolution, selecting solutions evaluated to be of high quality on the basis of a fitness function, and combining them to produce new solutions of higher quality. See [49, 50].

Other approaches include *SA (Simulated Annealing)* [51], *Placement heuristics* [52], *3D-packing* [53], *Simulation* [54] and *Case-based methods* [55].

### 2.3.4 Constraints

In our configurator implementation we focus on the same sub-problem as [9], which is to rearrange containers in an under-deck location of a container vessel. This has the benefit of making our results based on a SAT-driven implementation directly comparable to this earlier work based on precompilation with BDDs, giving a good foundation for assessing the strengths and weaknesses of the two approaches.

The constraints included in our model are as follows:

- All containers must be placed.
- A cell can only hold one 40' container or up to two 20' containers
- 20' containers cannot be placed on top of 40' containers.
- Containers must have support from below, i.e. no empty slots can exist in the middle of stacks.
- 20' containers must be placed evenly, i.e. the difference in height between the aft and fore part of a stack must never exceed one.
- For each stack, the total weight of all containers in the stack must not exceed the stack's weight limit.
- Overstowage is completely disallowed, i.e. a container cannot be placed on top of a container having an earlier discharge port.
- Reefer containers can only be placed in reefer slots.

# Chapter 3

# Approach

In this chapter we present two different methods of encoding the stowage problem into a SAT representation for use in interactive configuration. The first approach focuses on a general solution for converting CSP problems, including the stowage problem, to SAT form, while the second represents a more direct encoding of the stowage problem, sacrificing the generality of the first approach in return for a more compact and less layered representation.

## 3.1 Extending CLab

A CSP model of the container stowage problem is presented in [9], where decision variables $X$ represent slots and each value in their domains $D$ represent placements of a container from the corresponding container group. These groups are constructed by dividing the containers into disjoint sets by type and weight similarity. The discretization done here is necessary to reduce the size of the BDD to an acceptable level. Since two different containers cannot be in the same cell at the same time, we have an alldifferent constraint on the slot variables. The complexity of representing an alldifferent constraint with a BDD is exponential, and this grouping of containers by weight is introduced to minimize the impact of this.

Each container in a group is assigned the weight of the group's heaviest container, thus guaranteeing the feasibility of all reachable stowage configurations, but risking the unreachability of certain feasible configurations of the real-world problem. A dynamic programming algorithm was introduced to minimize this "weight loss" when grouping containers. The authors used this model of the problem as input for the BDD-based configurator package CLab [17], specified in the associated CP language, and produced the results discussed earlier.

Using this problem model, we extend CLab to use SAT-based solving. Since our extension encompasses all linear constraints expressible in the CP language, it enables SAT-solver-based configuration support for *any* linear CSP supported by CLab, thus providing a basic tool for future research.

### 3.1.1 Configurator interface

Our extension includes the design of a shared interface, enabling programmers to seamlessly switch between the BDD and SAT-based implementations, without having to make any changes to the code interacting with CLab. This generic interface provides four basic commands:

- INIT($R : CSP$)
  Initializes the configurator with a chosen CSP problem, defined in the CLab CP language. In the C++ code, this task is performed in the constructor, taking a CP filename as parameter.
- SETVAR($x : variable, v : value$)
  Assigns a value to a variable, chosen from its domain of legal values. In our implementation, variables and values are represented as strings.
- UNSETVAR($x : variable$)[1]
  Cancels a value assignment, making the chosen variable unassigned again.
- VALIDASSIGNMENTS()
  Returns the valid domains for all variables in constant time.

This interface also provides a good framework for further extending CLab with new configurator implementations. This is done by building a new class inheriting the supplied configurator class.

### 3.1.2 CP language syntax

CLab accepts CSP descriptions in the CP language. The following is an excerpt from the CLab manual [56], which can be consulted for a more detailed description, including precedence and associativity of operators. An example CP file is given in figure 3.1, where a simple container vessel stowage instance is specified.

The CP language has two basic types: *range* and *enumeration*. A range is a consecutive and finite sequence of integers. An enumeration is a finite set of strings. The Boolean type is the range from 0 to 1. Range and enumeration types can be defined by the user. A CP description consists of a *type declaration*, a *variable declaration*, and a *rule declaration*. The type declaration is optional if no range or enumeration types are defined:

```
cp        ::= [ type {typedecl} ] variable {vardecl} rule {ruledecl}

typedecl  ::= id [ integer . . integer ] ;
          |   id { idlst } ;

vardecl   ::= vartype idlst ;
```

[1] This function is not implemented in the original CLab package or in our extension. It is however a trivial task to implement it using e.g. memorization of earlier valid domains, recomputation of those or a combination of both.

```
vartype     ::= bool
            |   id

idlst       ::= id {, idlst}

ruledecl    ::= exp ;
```

An identifier is a sequence of numbers, letters and the character "_" that does not begin with a number. An integer is a sequence of digits possibly preceded by a minus sign. The symbols `//` start a comment that extends until the end of the line. The syntax of expressions is given below:

```
exp         ::= integer
            |   id
            |   - exp
            |   ! exp
            |   ( exp )
            |   exp op exp

op          ::= * | / | % | + | - | == | !=
            | < | > | <= | >= | && | || | >>
```

The semantics, associativity, and precedence of arithmetic, logical, and relational operators are defined as in C/C++. Hence, `!`, `/`, `%`, `==`, `!=`, `&&`, and `||` denote logical negation, division, modulus, equality, inequality, conjunction, and disjunction, respectively. The only exception is the pipe operator `>>` that denotes implication. (...) Conversion between Booleans and integers is also defined as in C/C++. True and false is converted to 1 and 0, and any non-zero arithmetic expression is converted to true. Due to these conversion rules, it is natural to represent the Boolean constants true and false with the integers 1 and 0.

### 3.1.3 Conversion from CSP to CNF

Since SAT is a NP-complete problem, searching for a solution is exponential in the worst case. Therefore, SAT solvers use different techniques to help guide the search towards faster solutions, such as those of the DPLL algorithm already mentioned. When encoding a CSP as a SAT problem, the choice of encoding has a big influence on the solver performance, since different encodings might either help or obstruct the solver in doing effective propagation. Thus, the choice of conversion method has a great deal of impact on overall performance, and has consequently received a great deal of attention in the literature [57–61].

In the following we will give short descriptions of some of the methods devised, along with a presentation of the approaches we adopt.

```
type
cell40 [0..1];
cell20 [-2..0];

variable
cell20 C1_1_2_FTEU;
cell20 C1_1_2_ATEU;
cell40 C1_1_2_FEU;
cell20 C1_1_4_FTEU;
cell20 C1_1_4_ATEU;
cell40 C1_1_4_FEU;

rule
// Either the FEU or both TEU slots are empty
C1_1_2_FEU == 0 || (C1_1_2_FTEU == 0 && C1_1_2_ATEU == 0);
C1_1_4_FEU == 0 || (C1_1_4_FTEU == 0 && C1_1_4_ATEU == 0);

// Gravity constraint, containers must have something under them
(C1_1_2_FTEU == 0 || C1_1_2_ATEU == 0) >> (C1_1_4_FTEU == 0 &&
C1_1_4_ATEU == 0);
((C1_1_2_FTEU == 0 || C1_1_2_ATEU == 0) && C1_1_2_FEU == 0) >>
(C1_1_4_FEU == 0);

// A container can be placed only once
(C1_1_2_FTEU == -2) + (C1_1_2_ATEU == -2) + (C1_1_4_FTEU == -2) +
(C1_1_4_ATEU == -2) == 1;
(C1_1_2_FTEU == -1) + (C1_1_2_ATEU == -1) + (C1_1_4_FTEU == -1) +
(C1_1_4_ATEU == -1) == 1;
(C1_1_2_FEU == 1) + (C1_1_4_FEU == 1) == 1;

// Weight limits
((C1_1_2_FTEU == -2) + (C1_1_2_ATEU == -2) + (C1_1_4_FTEU == -2) +
(C1_1_4_ATEU == -2)) * 10 + ((C1_1_2_FTEU == -1) + (C1_1_2_ATEU ==
-1) + (C1_1_4_FTEU == -1) + (C1_1_4_ATEU == -1)) * 20 + ((C1_1_2_FEU
== 1) + (C1_1_4_FEU == 1)) * 20 <= 50;
```

Figure 3.1: An example of a simple stowage problem specification in the CP language. One 40' and two 20' containers are being stowed in a single stack with two cells, all to be discharged at the first port, for which reason there are no overstowage constraints. The variable naming convention are a 'C' followed by a bay_stack_tier_slot combination. Slots are either named FEU for a FEU slot, or ATEU/FTEU for the aft and fore TEU slots of a cell. The variable types cell20 and cell40 correspond to the two slot sizes. 20' containers are enumerated with negative integers, and 40' containers with positive integers.

### Direct Encoding

In the direct encoding [57], 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* and an *at-most-one* constraint over the Boolean variables representing the domain of a given CSP variable [58]. The number of Boolean variables produced with this encoding, just for representing the CSP variables, will be $O(vd)$, where $v$ is the number of variables and $d$ is the biggest domain of any variable.

We do not implement any direct encoding for CSP to CNF, but we do draw some inspiration from this line of thought for our model that translates the container vessel stowage problem directly to CNF form.

### Log Encoding

The log encoding [62] uses a logarithmic number of Boolean variables to represent the domain of a given CSP variable $v$. This is achieved by representing each domain value with a pattern over the Boolean variables. Since it is possible to make $2^n$ patterns with $n$ Boolean variables, the log encoding creates $O(v\lceil log(d)\rceil)$ variables to encode all the CSP variables. While the log encoding uses a much more compact encoding, it has been observed in the literature that the log encoding has weaker propagation than the direct encoding [58, 63]. This is important, as worse propagation leads to worse solving time, especially because the reduced problem size might not provide much improvement in itself, as hardness and problem size are at best weakly correlated. This has been seen with industrial SAT instances involving millions of variables being tractable, while much smaller random instances can easily be intractable [64–66].

The direct encoding and log encoding are the simplest encodings, and have been known for a long time. In recent years, a lot of newer encodings have been proposed, and been shown to be effective.

### Order Encoding

In the order encoding, as with the direct encoding, a Boolean variable is introduced for each domain value of each CSP variable. However, the meaning of the Boolean variables is 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 $(X, l, u, P, C)$ where $X$ is a set of integer variables, $l(v)$ and $u(v)$ are the lower and upper bounds of each variable $v \in X$, $P$ is a set of propositional variables and $C$ is a set of constraints. We then let $p(v \leq c)$ denote the Boolean variable representing whether the value of $v$ is less than some constant $c$.

As an example, consider some CSP variable $v$, with domain [0...2]. Three Boolean variables will be introduced for $v$, representing the truth values of the following statements: $p(v \leq 0)$, $p(v \leq 1)$, $p(v \leq 2)$. Note that $p(v \leq 2)$ is always

true, but we use it instead of the value *true* for ease of explanation. To ensure that only one value is selected, the following clauses are introduced in $C$ for each variable $v$:

$$\bigwedge_{l(v)<c\leq u(v)} \big(\neg p(v \leq c-1) \vee p(v \leq c)\big)$$

Any domain value $c$ can be specified by the unit clauses $\neg p(v \leq c-1) \wedge p(v \leq c)$. This encoding is well suited for reasoning about linear inequalities. We will try to give an intuition on why this approach works. For a thorough treatment we refer to [61].

Let $\alpha$ be an assignment of values to variables in a finite linear CSP, and let $e, f$ be linear expressions over the variables in $X$. Let $\alpha \models e$ denote the *satisfiability relation*, meaning that the assignment $\alpha$ makes expression $e$ true and that $l(x) \leq \alpha(x) \leq u(x)$ for all $x \in V$. Further, we extend the functions $l(e)$ and $u(e)$ such that they return the lowest and highest possible values for the linear expression $e$, given the domains of the variables used in $e$. As an example, $l(2x - 3y)$ = -9 and $u(2x - 3y) = 6$ when $l(x) = l(y) = 0$ and $u(x) = u(y) = 3$.

It can then be shown that the following holds for any integer $c \geq l(e) + l(f)$.

$$(\alpha \models e + f \leq c) \Leftrightarrow$$
$$(\alpha \models \bigwedge_{a+b=c-1} (e \leq a \vee f \leq b))$$

Parameters $a$ and $b$ are integers ranging over $l(e) - 1 \leq a \leq u(e)$ and $l(f) - 1 \leq b \leq u(f)$. The basic idea is that the inequality will only hold if all the conditions expressed in the clauses are fulfilled, and vice versa. A proof is given in [61]. Since $e$ and $f$ are linear expressions, they can themselves be split into components of linear expressions. In this way, the components in the expressions $e$ and $f$ can be inductively transformed. This is continued until the base case is hit, which is when an inequality has only one variable, $(v \leq a)$. At this point, we can describe the inequality with the Boolean variable $p(v \leq a)$. This is demonstrated for a simple case in example 3.

**Example 3.** Consider the inequality $x + 2y \leq 2$, where $l(x) = l(y) = 0$ and $u(x) = u(y) = 2$.

Transforming this to an order encoding can be done in two steps. First, if we let $e = x$ and $f = 2y$, then $l(f) = 0$ and $u(f) = 4$. We then get:

$$\alpha \models x + f \leq 2 \Leftrightarrow \alpha \models \bigwedge_{a+b=1} (x \leq a \vee f \leq b)$$

Which can be completely specified as:

$$(x \leq -1 \vee f \leq 2) \wedge (x \leq 0 \vee f \leq 1) \wedge (x \leq 1 \vee f \leq 0) \wedge (x \leq 2 \vee f \leq -1) \Leftrightarrow$$
$$(f \leq 2) \wedge (x \leq 0 \vee f \leq 1) \wedge (x \leq 1 \vee f \leq 0)$$

where the bi-implication is derived from the fact that $f \leq -1$ is always false and $x \leq 2$ is always true. Because $f = 2y = y + y$, we can then replace $f \leq 2$, $f \leq 1$ and $f \leq 0$ recursively in the sentence.

$f \leq 2$ is transformed to:

$$\alpha \models y + y \leq 2 \Leftrightarrow \alpha \models \bigwedge_{a+b=1} (y \leq a \vee y \leq b)$$
$$(y \leq -1 \vee y \leq 2) \wedge (y \leq 0 \vee y \leq 1) \wedge (y \leq 1 \vee y \leq 0) \wedge (y \leq 2 \vee y \leq -1) \Leftrightarrow$$
$$(y \leq 2) \wedge (y \leq 0 \vee y \leq 1) \Leftrightarrow y \leq 1$$

$f \leq 1$ is transformed to:

$$\alpha \models y + y \leq 1 \Leftrightarrow \alpha \models \bigwedge_{a+b=0} (y \leq a \vee y \leq b)$$
$$(y \leq -1 \vee y \leq 1) \wedge (y \leq 0 \vee y \leq 0) \wedge (y \leq 1 \vee y \leq -1) \Leftrightarrow$$
$$(y \leq 0)$$

$f \leq 0$ is transformed to:

$$\alpha \models y + y \leq 0 \Leftrightarrow \alpha \models \bigwedge_{a+b=-1} (y \leq a \vee y \leq b)$$
$$(y \leq -1 \vee y \leq 0) \wedge (y \leq 0 \vee y \leq -1) \Leftrightarrow (y \leq 0)$$

Replacing $f \leq 0$, $f \leq 1$ and $f \leq 2$ in the original sentence, we get:

$$\alpha \models (y \leq 1) \wedge (x \leq 0 \vee y \leq 0) \wedge (x \leq 1 \vee y \leq 1)$$

where all inequalities can be represented by Boolean variables in the order encoding. ◊

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 [61], and the CSP solver Sugar [21] used this encoding to win four categories at the 2008 CSP solving competition, and three categories at the 2009 competition [67, 68].

### Compact Order Encoding

In the compact order encoding [59], the CSP is transformed to what the authors call a *Compact 3ary CSP* before the order encoding is applied. The authors propose to do this through a three step conversion. First, the CSP is converted to a 3ary CSP.

**Definition 1.** A 3ary CSP is a tuple $(X, u, P, C)$ defined as follows.

$X$ = Finite set of integer variables with lower bound 0.
$u$ = Mapping from $X$ to $\mathbb{N}$ specifying the upper bounds for all $x \in X$.
$P$ = Finite set of propositional variables.
$C$ = Formula for the constraints:

$$C ::= p \mid \neg p \mid \sum_{i=1}^{n} \pm x_i \triangleright a \mid C \wedge C \mid C \vee C$$

Where $p \in P$, $n \in \{1, 2, 3\}$, $x_i \in X$, $a \in \mathbb{N}$, and $\triangleright \in \{\leq, \geq, =\}$.

Any finite linear CSP can be converted to a 3ary CSP in polynomial time using the following rules:

- When the lower bound $l$ of an integer variable $x$ is not 0, $x$ is replaced by a new integer variable $x' = x - l$.
- A constant multiplication $ax$ can be replaced by $\sum_{i=1}^{a} x$ when $a > 0$, or $\sum_{i=1}^{a} -x$ when $a < 0$.
- A linear expression containing more than three variables can be reduced to ternary expressions by replacing partial summations with new variables.

The 3ary CSP is then converted to a *Restricted 3ary CSP*.

**Definition 2.** A Restricted 3ary CSP is a 3ary CSP where the constraint formula $C$ is restricted to the following form, where $p \in P$, $x, y, z \in X$ and $a \in \mathbb{N}$.

$$C ::= p \mid \neg p \mid x \leq a \mid x \geq a \mid z = x + a \mid z = x + y \mid C \wedge C \mid C \vee C.$$

This amounts to reducing all combinations of signs and comparison operators in the expression $\sum_{i=1}^{n} \pm x_i \triangleright a$ to legal Restricted 3ary CSP constraints. Generally this is done by using algebraic manipulation such that no minus signs occur, at which point new variables representing subexpressions are introduced, such that a single variable is obtained for each side of the expression. As an example, consider the expression $x_1 - x_2 - x_3 \geq a$. We rewrite the expression as $x_1 \geq a + x_2 + x_3$, and then introduce the variables $z_1, z_2$, and clauses $(z_1 = a + x_2) \wedge (z_2 = z_1 + x_3)$, such that $z_2$ represents the right hand side of the expression. We can then write the expression as $(x_1 \geq z_2)$.

Finally, the Restricted 3ary CSP is converted to a *Compact 3ary CSP*, which is a 3ary CSP where we use a positional system with base $B \geq 2$ to encode the variables. The upper bounds on all integer variables are thus fixed to some integer constant $B - 1$.

**Definition 3.** Let $B \geq 2$ be the base. A Compact 3ary CSP $(B, X, P, C)$ is a 3ary CSP $(X, u, P, C)$ where $u(x) = B - 1, \forall x \in X$.

In this encoding, every original CSP variable $v$ 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 $v \in X$ using the selected base $B$. If we define the *maximum domain size* $d$ as follows:

$$d = 1 + max\big(\{u(x) \mid x \in X\} \cup \{a \mid a \text{ is an integer constant occurring in } C\}\big),$$

then the number of digits, or *order of magnitude*, denoted $m$, is equal to $\lceil log_B d \rceil$. Each digit is implemented as a Compact 3ary CSP variable, and the value for a variable $v$ is then selected through the assignment of values to the digits of $v$. We denote the digits of $v$ as $v_0, \ldots v_{m-1}$, with $v_0$ being the least significant digit. The digit $v_i$ for any value $a < d$ can then be determined by $v_i = \lfloor \frac{a}{B^i} \rfloor \bmod B$.

**Example 4.** Consider a 3ary CSP with some variable $v$, where $u(v) = 5$. In a Compact encoding with $B = 4$, $v$ would be represented by 2 digits, and the value 5 would be encoded with $v_1 = 1, v_2 = 1$, since $1 \times 4^0 + 1 \times 4^1 = 5$. $\Diamond$

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. We have included this possibility in our implementation.

### Tseitin transform

As mentioned in section 2.2.2, the conversion of a SAT problem to CNF form is exponential in the worst case. An example of this can be seen below.

**Example 5.** Consider a SAT problem consisting of a disjunction of 2-ary conjunctions, $\bigvee_{1 \leq i \leq n} (x_i \wedge y_i)$. Converting this problem to CNF form leads to the formula $(x_1 \vee \ldots \vee x_n) \wedge (x_1 \vee \ldots \vee x_{n-1} \vee y_n) \wedge \ldots \wedge (y_1 \vee \ldots \vee y_n)$, which uses $2^n$ clauses. $\Diamond$

To mitigate this problem, polynomially bounded methods that sacrifice logical equivalence but maintain equisatisfiability have been proposed. The most common of these is the *Tseitin transform* [22], which transforms any SAT problem $b$ to CNF form in linear time and space. It works by introducing equivalent variables for each subformula in a logical expression in $b$ and then using these new variables to build the CNF. The transformation rules are shown in figure 3.2. These rules are recursively applied, and the generated clauses are added to the CNF. Finally, a unit clause with the variable at the top of the recursive tree is added, as this must be satisfied. The number of variables generated by this procedure is linear in the number of literals.

The complexity of the Tseitin transform is linear, but there are many optimizations that can be done to the procedure, which do not change the worst case complexity of the transformation, but can make a huge difference in practice. The simplest optimization is the removal of the transform for negated expressions.

Since any subexpression is transformed to be represented by a single variable, we can just negate the new variable. This is shown in equation (3.1.1).

$$\neg a \rightarrow (\neg[a]) \tag{3.1.1}$$

$$
\begin{array}{lcl}
a \vee b & \rightarrow & (\neg[c] \vee [a] \vee [b]) \wedge ([c] \vee \neg[a]) \wedge ([c] \vee \neg[b]) \\
a \wedge b & \rightarrow & ([c] \vee \neg[a] \vee \neg[b]) \wedge (\neg[c] \vee [a]) \wedge (\neg[c] \vee [b]) \\
\neg a & \rightarrow & (\neg[a] \vee \neg[c]) \wedge ([a] \vee [c])
\end{array}
$$

Figure 3.2: The operations in the Tseitin transform. Square brackets denote the newly introduced variable for the corresponding subexpression and [c] denotes the newly introduced variable for the current expression.

A disjunction of more than two literals can be transformed with the introduction of only a single new variable, as can be seen in equation (3.1.2).

$$
\bigvee_{1 \leq i \leq n} a_i \rightarrow (\neg[c] \vee \bigvee_{1 \leq i \leq n} [a_i]) \wedge \bigwedge_{1 \leq i \leq n} ([c] \vee \neg[a_i]) \tag{3.1.2}
$$

Let $n$ denote the number of literals in the disjunction, this optimization uses $n + 1$ clauses to perform the equivalence procedure, which is $2n - 1$ less than the original transform and it also introduces $n - 1$ fewer variables. A similar optimization can be introduced for conjunctions of more than two literals, this optimization is shown in equation (3.1.3).

$$
\bigwedge_{1 \leq i \leq n} a_i \rightarrow ([c] \vee \bigvee_{1 \leq i \leq n} \neg[a_i]) \wedge \bigwedge_{1 \leq i \leq n} (\neg[c] \vee [a_i]) \tag{3.1.3}
$$

**Encodings implemented**

In this thesis we implement the log encoding, order encoding and compact order encoding, such that we can compare the performance of the various encodings and see which one is more efficient. For all of them we use the Tseitin transform to convert to CNF form.

### 3.1.4 SMT encoding

For the SMT-based configurator, we wish to select theories such that we are able to easily express the constraints of the CP language, without having to encode them in SAT form. To do this we use the theory of the integers, as this allows us to model linear expressions in the solver. By doing this we can map instances in the CP language almost directly to SMT format. Consider the example given in figure 3.1. Each variable is specified to be an integer in the range [0..1] or [-2..0], and all the rules are encoded as SMT statements. The rules can be mapped directly to SMT constraints using the theory of integers, except for arithmetic over constraints such as $x == v$, where $x$ is a variable and $v$ is a value in that variable's domain. For these expressions we introduce a new variable with range [0..1] to represent the truth value of $x == v$, which is used in the arithmetic expression. As mentioned earlier, we use the Z3 solver as our SMT engine.

## 3.2 Direct Encoding

As an alternative to the more general encoding that accepts any linear CSP problem specified in the CP format, we also try a direct encoding of the container vessel stowage problem as a SAT problem. The CP encoding has many steps between the problem description and the final SAT encoding, which might obscure the problem too much. This is a consequence of the general encoding we are using, which accepts any input in the CP format. A direct encoding could potentially by more efficient, as the model can be made more compact. The direct encoding we present here is inspired by [3], in which the authors present a binary Integer Programming model, where a Boolean variable is introduced for each cell and container combination, signifying whether the container is stored in this cell. We adopt the same approach, using one variable for each combination. Table 3.1 presents the constants, sets and Boolean variables used in the direct encoding.

| | **Constants and Sets** |
|---|---|
| $S$ | Stacks index set. |
| $T$ | Tiers index set, enumerated from bottom of stacks with increments of one, as opposed to the traditional numbering using $2, 4, 6$ and so forth. |
| $L$ | The set $\{F, AT, FT\}$, representing the 40' and two 20' slots in a cell. |
| $S_{ij}^{l}$ | The slot in stack $i$, tier $j$, where $l \in L$. |
| $C_F$ | 40' container index set. |
| $C_T$ | 20' container index set, also denoted as $C_{AT}$ and $C_{FT}$. |
| $C$ | Container index set. |
| $w_i$ | The weight of container $i$. |
| $p_i$ | Discharge port of container $i$. |
| | **Boolean Variables** |
| $b_{kji}^{F}$ | Container $i \in C_F$ being stored in slot $s_{kj}^{F}$. |
| $b_{kji}^{AT}$ | Container $i \in C_T$ being stored in slot $s_{kj}^{AT}$. |
| $b_{kji}^{FT}$ | Container $i \in C_T$ being stored in slot $s_{kj}^{FT}$. |
| $e_{kj}^{l}$ | Slot $S_{kj}^{l}$ being empty. |

Table 3.1: The constants, sets and variables we use.

### 3.2.1 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.

$$\bigvee_{s\in S, t\in T} b^F_{stc} \qquad \text{if } c \in C_F \tag{3.2.1}$$

$$\bigvee_{s\in S, t\in T} b^{FT}_{stc} \vee b^{AT}_{stc} \qquad \text{if } c \in C_T \tag{3.2.2}$$

**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.

$$\neg(b^m_{ikc} \wedge b^n_{jlc}) \tag{3.2.3}$$

**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.

$$\bigwedge_{l\in C_F,\ m\in C_T} (\neg b^F_{kjl} \vee \neg b^{FT}_{kjm}) \wedge (\neg b^F_{kjl} \vee \neg b^{AT}_{kjm}) \tag{3.2.4}$$

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.

$$\bigwedge_{l,k\in C_F} (\neg b^F_{ijk} \vee \neg b^F_{ijl}) \tag{3.2.5}$$

$$\bigwedge_{l,k\in C_T} (\neg b^{FT}_{ijk} \vee \neg b^{FT}_{ijl}) \wedge (\neg b^{AT}_{ijk} \vee \neg b^{AT}_{ijl}) \tag{3.2.6}$$

**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.

$$(\neg b^F_{kjl} \vee (\bigvee_{c\neq l\in C_F} b^F_{kj-1c}) \vee (\bigvee_{c\neq l\in C_T} b^{FT}_{kj-1c})) \tag{3.2.7}$$

$$(\neg b^F_{kjl} \vee (\bigvee_{c\neq l\in C_F} b^F_{kj-1c}) \vee (\bigvee_{c\neq l\in C_T} b^{AT}_{kj-1c})) \tag{3.2.8}$$

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.

$$(\neg b^x_{kjl} \vee (\bigvee_{c \neq l} b^{FT}_{kj-1c})) \tag{3.2.9}$$

$$(\neg b^x_{kjl} \vee (\bigvee_{c \neq l} b^{AT}_{kj-1c})) \tag{3.2.10}$$

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

This is implemented by the constraints (3.2.9) and (3.2.10), 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 (3.2.9) and (3.2.10), 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 (3.2.1) 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.

$$\bigwedge_{k<n} (\neg b^F_{mni} \vee \neg b^{AT}_{mkj}) \wedge (\neg b^F_{mni} \vee \neg b^{FT}_{mkj}) \tag{3.2.11}$$

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 $p_i > p_j$, we introduce:

$$\bigwedge_{k<n} (\neg b^{AT}_{mni} \vee \neg b^{AT}_{mkj}) \wedge (\neg b^{FT}_{mni} \vee \neg b^{FT}_{mkj}) \tag{3.2.12}$$

**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 (3.2.3).

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.

$$
\begin{aligned}
&t > \textsc{MaxTier}(s): \\
&\quad Y[s,t,W] \Leftrightarrow true \\
&t \leq \textsc{MaxTier}(s): \\
(1)\quad &Y[s,t,W] \Leftrightarrow (e^F_{st} && \vee \\
(2)\quad &\bigvee_{c \in C_{40} | w_c \leq W} b^F_{stc} \wedge Y[s, t+1, W - w_c] && \vee \\
(3)\quad &\bigvee_{c,k \in C_{20} | w_c + w_k \leq W} ((b^{AT}_{stc} \wedge b^{FT}_{stk}) \vee (b^{FT}_{stc} \wedge b^{AT}_{stk})) \wedge Y[s, t+1, W - w_c - w_k] && \vee \\
(4)\quad &\bigvee_{c \in C_{20} | w_c \leq W} (b^{AT}_{stc} \wedge e^{FT}_{st}) \vee (b^{FT}_{stc} \wedge e^{AT}_{st})) && (3.2.13)
\end{aligned}
$$

The $t >$ MaxTier(s) case of equation (3.2.13) is the base case. MaxTier is a function that returns the highest tier in the stack $s$. The $t \leq$ 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 $s$.

Algorithmically, for each stack $s$, we start out by enumerating all the possible container placements in the lowest tier of $s$. For each placement of a container $i$ in the current cell, we calculate the remaining weight limit on the stack $W - w_i$. Using dynamic programming, we then ask for a variable representing all the legal configurations for the next tier, given the newly calculated weight limit. By creating and remembering variables that correspond to the legal configurations, we allow reuse of the clauses representing these legal configurations, thereby using only a polynomial number of new clauses and variables to represent the legal configurations. The pseudocode for this is shown in Algorithm 4.

```
1  LP(s, t, W)
2  if t > MaxTier(s) then
3      return true
4  if (s, t, W) ∈ table then
5      return table[s, t, W]
6  Y = NewBooleanVar()
7  C = (¬Y)
8  for i ∈ 40FootContainersUnderWeightLimit(W) do
9      AddConstraint(Y ∨ ¬(b^F_sti ∧ LP(s, t + 1, W − w_i)))
10     C = (C ∨ (b^F_sti ∧ LP(s, t + 1, W − w_i)))
11 for i, j ∈ 20FootContainerPairsUnderWeightLimit(W) do
12     AddConstraint(Y ∨ ¬(b^AT_sti ∧ b^FT_stj ∧ LP(s, t + 1, W − w_i − w_j)))
13     C = (C ∨ (b^AT_sti ∧ b^FT_stj ∧ LP(s, t + 1, W − w_i − w_j)))
14     AddConstraint(Y ∨ ¬(b^FT_sti ∧ b^AT_stj ∧ LP(s, t + 1, W − w_i − w_j)))
15     C = (C ∨ (b^FT_sti ∧ b^AT_stj ∧ LP(s, t + 1, W − w_i − w_j)))
16 for i ∈ 20FootContainersUnderWeightLimit(W) do
17     AddConstraint(Y ∨ ¬(b^AT_sti ∧ e^FT_st))
18     C = (C ∨ (b^AT_sti ∧ e^FT_st))
19     AddConstraint(Y ∨ ¬(b^FT_sti ∧ e^AT_st))
20     C = (C ∨ (b^FT_sti ∧ e^AT_st))
21 C = C ∨ e^F_st
22 AddConstraint(C)
23 table[s, t, W]= Y
24 return Y
```

**Algorithm 4**: The LP (Legal Placement) procedure, a simplified version of the dynamic programming algorithm for specifying all legal container placements.

Lines 2-3 of the algorithm is the base case, stopping recursion when the top of the stack has been reached. Lines 4-5 does the memoization check, and returns the variable representing $Y[s, t, W]$ if it has already been created. Otherwise Line 6 creates a new Boolean variable and line 7 begins the construction of

a set of constraints that ensure the equivalence described in equation 3.2.13. Lines 8-9 iterate over all the 40' containers that are within the weight limit, and adds each of these as an option for $Y$. Each container option $i$ is specified by $b^F_{sti} \wedge \mathrm{LP}(s, t+1, W-w_i)$, where $b^F_{sti}$ denotes placing the container in the current tier in the stack, and $\mathrm{LP}(s, t+1, W-w_i)$ denotes the variable representing all the legal configurations for the remaining tiers in the stack. Lines 11-15 perform the corresponding operation for all 20' container pairs, where we add the two symmetric possibilities of placing container $i$ in the aft slot and container $j$ in the fore slot, or vice versa. Lines 16-20 add the two possibilities of placing a single 20' container in either the aft or fore slot of the current cell. Note that this has no recursive call, as no more containers can be added to the stack as a consequence of having one of the two TEU slots empty. Lines 21 add the possibility of having the whole cell empty, and line 22 adds the big clause $C$, which together with the smaller clauses added with AddConstraint ensures the equivalence of Y the legal choices of container placements for given (stack, tier)-combination. Finally lines 23-24 memoize the variable and return it.

Since the maximum weight limit is constant, the complexity of the number of clauses and variables generated by the LP procedure is $O(t(c_{40} + (c_{20})^2)$, where $t$ is the number of tiers, $c_{40}$ is the number of forty foot containers and $c_{20}$ is the number of twenty foot containers. Since this is applied to each stack we can describe the overall complexity as $O(st(c_{40} + (c_{20})^2)$, where $s$ is the number of stacks.

### 3.2.2 Configurator Optimizations

We conclude this chapter by giving a description of some of the optimizations we have used for increasing the solver performance.

Consider the Update procedure described in Algorithm 2. The essential step in this algorithm is the IsUnsatisfiable procedure. In general, a search based configurator needs to search for a satisfying solution every time this procedure is invoked. However, there are several optimizations that can be done for Update, such that the amount of searches can be minimized. The most simple optimization is to remember unsatisfiable assignments, such that if a given assignment was previously found to be unsatisfiable, this assignment will still lead to unsatisfiability if more constraints are added to the same problem.

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 *proof*. This can be exploited, as this proof 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 proof can be part of some satisfiable assignment. We utilize this fact by analyzing the whole returned proof, and remembering all the satisfiable assignments, such that we can immediately return $false$ in the IsUnsatisfiable procedure, when asked about these assignments in later invocations. It is important to note here, that the set of assignments known to be satisfiable 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.

Finally, we also try an optimization introduced in [14], which guides the solver towards solutions that provide proofs with as much new info as possible. This is done by modifying the branching of a solver, such that whenever it branches on some variable $v$, it tries assigning the unknown value to $v$ first, thereby preferring proofs that give new information. We implemented this optimization, but found that it had no impact on the solving speed for our solver. This could be related to the fact that modern SAT solvers perform aggressive restarting, making it hard to consistently guide the solver towards the unknown values.

# Chapter 4

# Results

## 4.1 Problem instances

Before presenting the results, we give an overview of the problem instances used in the experiments, followed by a description of the experimental setup.

### 4.1.1 Stowage

All stowage instances were built 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. This original instance consists of a bay with 20 stacks, with a total capacity of 198 cells, along with a total of 209 containers. 173 of these containers are 40', while 36 are 20'. Out of those, 9 are reefer containers, and all containers are distributed amongst 3 different discharge ports. From this instance, a range of down-scaled versions were created by removing one stack at a time, though retaining the proportions between the number of 20' and 40' containers found in the original problem. We believe that these instances provide a more accurate resemblance to a typical real-world problem than problems previously used for interactive container stowage configuration.

The experiments were performed grouping containers into between 5 and 10 classes, using the dynamic programming algorithm of [9], on a series of increasingly larger problems in terms of number of cells and containers.

### 4.1.2 CLib

To give a broader picture of the performance characteristics associated with the different interactive configurator implementations, we have included a suite of various configuration problems in our experiments. They all originate from a configuration problem package called CLib [10]. Here, we give a short description of the domains covered by the different instances.

- **Renault**: A large instance modelling the configuration of the Renault Megane family of cars [69].
- **PC, PC2, Big-PC**: Different instances of PC hardware configuration.
- **large-partial, large2, complex, 1-32.1**: Encoding a configuration problem involving parts of a power distribution network. It consists of local power sources supplying a number of electric lines, connected to transformer stations (sinks), consuming electricity and delivering it to the final customers. The instances were created in collaboration with the danish power distributor NESA [70].
- **Aralia**: A series of fault trees from various areas including avionic systems and the nuclear industry [71].
- **Bike2**: A bike configuration problem.

## 4.2 Experimental setup

### 4.2.1 Hardware

The experiments were performed on a PC running Ubuntu 11.04, having a 12 core 3.33 GHz Intel i7 processor. Each run was allowed to use one CPU core and a maximum of 8 GBs of RAM.

### 4.2.2 Limits

Each run was given a global timeout of 1 hour (CPU time), while a single run of the solver was given a maximum of 5 minutes. If this limit was exceeded, the whole experiment was considered timed out. Recall that a domain reduction consists of one solver call per value in the domain of each unset variable, so 5 minutes for a single solver call is highly unacceptable. As such, even one second is too much, if one wishes to have the domain reduction time perceived as being instant to the user, especially since the total waiting time accumulates with each solver call.

### 4.2.3 Measurements

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 pseudo-random variable assignments from the space of legal variable assignments, based on some random seed.

## 4.3 Stowage Results

On the following pages we present the results of the stowage experiments.

### BDD

In table 4.1 we present the results from the experiments with the BDD-based configurator. Each set of three columns describes the mean, median 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, "Timeout" refers to the BDD compilation process taking more than one hour, whereas "Out of 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. 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. The domain reductions also start to become cumbersome at the largest instances before compilation becomes infeasible. Especially for 6 container groups, where the instance with 77 cells and 70 containers has a maximum domain reduction time of almost 1 minute.

Building the BDD representation never took more than 10 minutes, even for the largest instances. It did, however, require significant amounts of memory. If the BDD construction process hits the memory barrier, garbage collection starts. This is a time consuming process, and experimentally it can be concluded that once this stage is reached, the chances of completing the BDD construction within the time-limit of 1 hour are minuscule. Thus, the critical resource when building BDDs is memory. Once built, the BDD configurator generally provided fast average domain reductions.

### SAT

In tables 4.2, 4.3, 4.4 and 4.5 we show the results using the general SAT configurator with the log encoding and compact order encoding with m set to 2, 3 and 4 digits respectively. These tables are organized the same way as the BDD table, except that for the SAT configurator, "Timeout" refers to a single solver call taking more than 5 minutes. For all encodings the solvers give strong performance for up to 34 slots and 15 containers. However, for all encodings we see severe scaling problems, as the instance with 34 slots and 20 containers gives some encodings trouble, and none of the encodings are able to satisfiably solve the instance with 34 slots and 25 containers. Surprisingly, the SAT configurator with compact order encoding is able to solve the instance with 77 slots and 60 containers for all choices of $m$, even when it has failed on most instances that are smaller than that.

Generally, compact SAT encoding with an order of magnitude equal to 3 or 4 provided the best results, with larger instances tending to work best with 4. This is not surprising, as a lower number of digits provides better propagation, but for very large instances the representation becomes too large for fast satisfiability checking. The latter phenomenon is clearly seen with order of magnitude set to 1 (equivalent to base = $\infty$), where the representation grows huge and fails on all instances. The log encoding, naturally, did not have these size issues, but generally performed weakly, most likely due to weak propagation in the solver.

Generalizing over all our experiments with the CSP to SAT encodings, we see that none of them prevented the solver from timing out on instances with more than 20 containers, with only a few of these instances being solved. Thus we see that, in general, the SAT-based configurator using the compact order encoding performs significantly worse than the BDD-based configurator, given that the BDD solution is provided with a significant amount of memory.

In table 4.6 we show the results from using the direct encoding from container vessel stowage problem to CNF form. This table is organized the same way as the tables for the general SAT configurator. While this approach is very fast for the very smallest instances, it quickly begins timing out, with instances involving 15 containers being infeasible. The reason for the earlier timeouts for the direct encoding could be because this encoding is not designed for taking advantage of the container weight class discretization. For the small instances the direct encoding performs better than the other SAT encodings, which supports this hypothesis, as the discretization has little effect on these instances, due to the low number of containers.

### SMT

Finally, we show the results from using the SMT-based configurator in table 4.7. The performance of the SMT-based configurator generally resembles that of the SAT encodings, with timeouts occurring as the problem size grows beyond the smallest instances. Generally, it performs worse than the compact encoding, with average runtime being about 8 times as much on the solved instances. Compared to the direct encoding, it manages to solve a larger quantity of the small instances, but on the instances with a large number of slots it times out on more instances.

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

Table 4.1: Online domain reductions for the BDD-based configurator.

| Size | | Number of Container Groups | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | # | 5 | | | 6 | | | 7 | | | 8 | | | 9 | | | 10 | | |
| Cel. | Cnt. | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | 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 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 14 | 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| 24 | 5 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 24 | 10 | 0.0 | 0.0 | 0.2 | 0.2 | 0.0 | 1.4 | 0.1 | 0.0 | 0.6 | 0.2 | 0.0 | 0.8 | 0.2 | 0.0 | 1.0 | 0.2 | 0.0 | 1.1 |
| 24 | 15 | 0.1 | 0.0 | 0.2 | 0.2 | 0.0 | 1.1 | 1.1 | 0.0 | 8.5 | 0.3 | 0.0 | 3.2 | 1.3 | 0.0 | 14.9 | 1.1 | 0.0 | 12.7 |
| 24 | 20 | 4.6 | 0.1 | 37.4 | 0.8 | 0.1 | 4.3 | 2.4 | 0.0 | 30.9 | 1.5 | 0.0 | 17.4 | Timeout | | | Timeout | | |
| 34 | 5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 34 | 10 | 0.0 | 0.0 | 0.0 | 0.2 | 0.0 | 1.1 | 0.1 | 0.0 | 0.6 | 0.1 | 0.0 | 0.8 | 0.1 | 0.0 | 0.9 | 0.1 | 0.0 | 0.7 |
| 34 | 15 | 0.2 | 0.0 | 0.9 | 2.2 | 0.0 | 7.6 | 1.5 | 0.0 | 7.6 | 0.9 | 0.0 | 8.3 | 1.3 | 0.1 | 5.4 | 1.6 | 0.1 | 7.9 |
| 34 | 20 | 11.9 | 0.2 | 124.2 | Timeout | | | 3.6 | 0.0 | 43.5 | 4.6 | 0.1 | 45.6 | 3.7 | 0.1 | 22.8 | Timeout | | |
| 34 | 25 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 34 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 10 | 0.1 | 0.1 | 0.4 | 0.6 | 0.1 | 4.8 | 0.3 | 0.1 | 2.1 | 0.8 | 0.2 | 5.6 | 0.6 | 0.1 | 4.1 | 1.7 | 0.1 | 11.2 |
| 44 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 10 | 0.2 | 0.1 | 0.6 | 0.9 | 0.3 | 5.6 | 0.8 | 0.3 | 4.8 | 1.2 | 0.2 | 6.7 | 1.0 | 0.1 | 5.9 | 2.1 | 0.2 | 12.6 |
| 55 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 40 | Timeout | | | Timeout | | | 16.6 | 6.8 | 71.9 | Timeout | | | Timeout | | | Timeout | | |
| 55 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 10 | 0.7 | 0.1 | 2.6 | 1.2 | 0.2 | 6.5 | 0.9 | 0.0 | 8.1 | 1.1 | 0.0 | 11.3 | 0.8 | 0.0 | 7.8 | 2.9 | 0.1 | 28.8 |
| 66 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 10 | 0.2 | 0.0 | 0.7 | 1.4 | 0.1 | 10.4 | 0.8 | 0.0 | 5.7 | 2.7 | 0.0 | 17.3 | 1.3 | 0.0 | 10.0 | 2.8 | 0.1 | 21.5 |
| 77 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 70 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |

Table 4.2: Online domain reductions for the SAT-based configurator with log encoding (base = 2).

| Size | | Number of Container Groups | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | # | 5 | | | 6 | | | 7 | | | 8 | | | 9 | | | 10 | | |
| Cel. | Cnt. | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | 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 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 14 | 10 | 0.0 | 0.0 | 0.1 | 0.1 | 0.1 | 0.3 | 0.1 | 0.1 | 0.2 | 0.2 | 0.1 | 0.7 | 0.1 | 0.1 | 0.5 | 0.2 | 0.1 | 0.6 |
| 24 | 5 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| 24 | 10 | 0.1 | 0.1 | 0.4 | 0.4 | 0.1 | 2.0 | 0.3 | 0.1 | 1.1 | 0.9 | 0.2 | 2.9 | 0.8 | 0.2 | 3.8 | 1.1 | 0.2 | 4.9 |
| 24 | 15 | 0.2 | 0.1 | 0.7 | 0.3 | 0.1 | 2.1 | 2.5 | 0.1 | 19.8 | 0.7 | 0.2 | 6.5 | 0.8 | 0.1 | 5.2 | 1.7 | 0.3 | 13.8 |
| 24 | 20 | 0.7 | 0.3 | 5.9 | 0.6 | 0.2 | 2.6 | 0.4 | 0.1 | 1.9 | 2.8 | 0.2 | 30.7 | Timeout | | | Timeout | | |
| 34 | 5 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| 34 | 10 | 0.1 | 0.0 | 0.2 | 0.2 | 0.1 | 1.1 | 0.2 | 0.1 | 0.8 | 0.4 | 0.2 | 1.8 | 0.3 | 0.2 | 1.5 | 0.5 | 0.3 | 2.3 |
| 34 | 15 | 0.4 | 0.2 | 1.6 | 1.9 | 0.2 | 6.7 | 1.4 | 0.2 | 8.5 | 2.2 | 0.4 | 11.5 | 2.6 | 0.5 | 14.6 | 5.1 | 0.5 | 33.8 |
| 34 | 20 | 2.3 | 0.5 | 29.5 | 4.8 | 0.3 | 39.6 | 1.2 | 0.3 | 14.3 | 21.6 | 0.8 | 239.0 | 9.3 | 0.9 | 103.8 | Timeout | | |
| 34 | 25 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 34 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 10 | 0.3 | 0.3 | 1.0 | 0.8 | 0.4 | 4.8 | 0.8 | 0.4 | 5.1 | 2.5 | 1.1 | 14.2 | 3.6 | 1.0 | 25.1 | 5.5 | 0.3 | 34.3 |
| 44 | 20 | 18.4 | 0.8 | 260.2 | 6.1 | 0.5 | 92.5 | 16.7 | 0.7 | 258.3 | Timeout | | | Timeout | | | Timeout | | |
| 44 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 10 | 0.4 | 0.4 | 1.2 | 1.5 | 0.8 | 6.8 | 2.3 | 1.5 | 9.8 | 4.1 | 1.1 | 22.1 | Timeout | | | Timeout | | |
| 55 | 20 | Timeout | | | 18.3 | 0.8 | 252.7 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 40 | 11.6 | 1.3 | 241.2 | Timeout | | | 6.6 | 4.5 | 23.3 | Timeout | | | Timeout | | | Timeout | | |
| 55 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 10 | 1.6 | 0.4 | 5.8 | 2.1 | 0.4 | 8.3 | 2.4 | 0.6 | 19.7 | Timeout | | | Timeout | | | Timeout | | |
| 66 | 20 | Timeout | | | 19.7 | 0.3 | 279.0 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 50 | 6.3 | 3.9 | 57.3 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 10 | 0.6 | 0.1 | 2.3 | 1.6 | 0.3 | 9.6 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 60 | 3.3 | 0.6 | 20.9 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 70 | Timeout | | | 9.8 | 6.6 | 52.2 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |

Table 4.3: Online domain reductions for the SAT-based configurator using compact encoding with order of magnitude = 2.

| Size | | Number of Container Groups | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | # | 5 | | | 6 | | | 7 | | | 8 | | | 9 | | | 10 | | |
| Cel. | Cnt. | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | 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 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 14 | 10 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.1 | 0.0 | 0.2 | 0.1 | 0.0 | 0.2 | 0.1 | 0.0 | 0.2 |
| 24 | 5 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| 24 | 10 | 0.1 | 0.0 | 0.2 | 0.2 | 0.0 | 1.2 | 0.1 | 0.0 | 0.5 | 0.5 | 0.1 | 2.2 | 0.4 | 0.1 | 2.2 | 0.6 | 0.1 | 3.3 |
| 24 | 15 | 0.1 | 0.0 | 0.2 | 0.1 | 0.0 | 1.1 | 0.7 | 0.0 | 5.6 | 0.3 | 0.1 | 3.1 | 0.4 | 0.0 | 2.8 | 0.8 | 0.1 | 7.2 |
| 24 | 20 | 0.3 | 0.1 | 2.4 | 0.4 | 0.1 | 1.8 | 0.2 | 0.0 | 1.5 | 0.6 | 0.0 | 7.8 | 0.6 | 0.1 | 5.2 | Timeout | | |
| 34 | 5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 34 | 10 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.5 | 0.1 | 0.0 | 0.3 | 0.1 | 0.0 | 0.5 | 0.1 | 0.1 | 1.0 | 0.2 | 0.1 | 1.2 |
| 34 | 15 | 0.1 | 0.1 | 0.6 | 0.7 | 0.1 | 2.8 | 0.5 | 0.0 | 2.6 | 0.6 | 0.1 | 3.1 | 0.9 | 0.1 | 5.1 | 1.6 | 0.1 | 8.7 |
| 34 | 20 | 1.2 | 0.2 | 10.2 | 0.8 | 0.1 | 4.0 | 0.5 | 0.1 | 6.2 | 11.9 | 0.2 | 135.7 | 2.8 | 0.2 | 28.2 | Timeout | | |
| 34 | 25 | Timeout | | | 25.6 | 0.2 | 282.3 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 34 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 10 | 0.2 | 0.1 | 0.5 | 0.7 | 0.2 | 4.7 | 0.3 | 0.1 | 2.0 | 0.9 | 0.3 | 5.8 | 1.5 | 0.3 | 11.3 | 2.1 | 0.1 | 14.1 |
| 44 | 20 | 7.0 | 0.4 | 100.9 | 7.8 | 0.2 | 128.5 | 4.5 | 0.2 | 61.5 | Timeout | | | Timeout | | | Timeout | | |
| 44 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 10 | 0.3 | 0.3 | 0.7 | 1.3 | 0.6 | 6.8 | 0.6 | 0.3 | 3.1 | 1.6 | 0.3 | 8.5 | 3.4 | 0.4 | 22.5 | 4.0 | 0.3 | 26.1 |
| 55 | 20 | 16.6 | 0.4 | 220.7 | 17.9 | 0.4 | 257.4 | 7.6 | 0.6 | 86.1 | Timeout | | | Timeout | | | Timeout | | |
| 55 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 10 | 0.7 | 0.2 | 2.6 | 1.7 | 0.3 | 6.4 | 0.7 | 0.1 | 6.4 | 1.3 | 0.1 | 11.1 | 2.5 | 0.1 | 22.4 | 2.5 | 0.1 | 22.8 |
| 66 | 20 | Timeout | | | 18.5 | 0.1 | 273.0 | 13.9 | 0.1 | 199.2 | Timeout | | | Timeout | | | Timeout | | |
| 66 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 50 | 4.5 | 6.2 | 43.3 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 10 | 0.2 | 0.0 | 0.9 | 1.2 | 0.1 | 7.6 | 0.8 | 0.1 | 5.7 | 2.5 | 0.1 | 16.3 | 5.7 | 0.1 | 43.1 | 4.7 | 0.1 | 36.0 |
| 77 | 20 | Timeout | | | 26.6 | 1.1 | 283.3 | 12.8 | 0.6 | 109.0 | Timeout | | | Timeout | | | Timeout | | |
| 77 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 60 | 1.5 | 0.2 | 8.2 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 70 | Timeout | | | 4.0 | 2.1 | 24.1 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |

Table 4.4: Online domain reductions for the SAT-based configurator using compact encoding with order of magnitude = 3.

| Size | | Number of Container Groups | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | # | 5 | | | 6 | | | 7 | | | 8 | | | 9 | | | 10 | | | | | |
| Cel. | Cnt. | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | 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 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | | |
| 14 | 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.2 | 0.1 | 0.0 | 0.2 | | | |
| 24 | 5 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | | | |
| 24 | 10 | 0.1 | 0.0 | 0.1 | 0.1 | 0.0 | 0.8 | 0.1 | 0.0 | 0.4 | 0.3 | 0.1 | 1.1 | 0.4 | 0.0 | 2.3 | 0.6 | 0.0 | 2.8 | | | |
| 24 | 15 | 0.1 | 0.0 | 0.2 | 0.1 | 0.0 | 0.4 | 0.5 | 0.0 | 4.2 | 0.2 | 0.0 | 1.9 | 0.4 | 0.0 | 2.4 | 1.9 | 0.1 | 21.9 | | | |
| 24 | 20 | 0.2 | 0.1 | 1.2 | 0.2 | 0.0 | 1.0 | 0.1 | 0.0 | 0.3 | 0.8 | 0.0 | 9.1 | Timeout | | | Timeout | | | | | |
| 34 | 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 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | | |
| 34 | 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.3 | 0.0 | 0.0 | 0.2 | 0.1 | 0.0 | 0.5 | 0.2 | 0.0 | 1.1 | 0.3 | 0.1 | 2.2 | | | |
| 34 | 15 | 0.1 | 0.0 | 0.5 | 0.5 | 0.0 | 2.9 | 0.4 | 0.0 | 1.9 | 0.5 | 0.1 | 2.3 | 0.9 | 0.1 | 5.2 | 2.2 | 0.1 | 10.2 | | | |
| 34 | 20 | 1.1 | 0.1 | 15.1 | 0.7 | 0.1 | 3.5 | 0.6 | 0.0 | 10.9 | 4.7 | 0.2 | 53.7 | 2.7 | 0.2 | 29.0 | 31.0 | 5.7 | 243.8 | | | |
| 34 | 25 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 34 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 44 | 10 | 0.1 | 0.1 | 0.4 | 0.6 | 0.2 | 4.6 | 0.3 | 0.1 | 2.1 | 1.1 | 0.3 | 7.2 | 1.2 | 0.2 | 9.3 | 3.3 | 0.1 | 22.9 | | | |
| 44 | 20 | 7.6 | 0.4 | 109.3 | 12.2 | 0.3 | 180.7 | 2.6 | 0.3 | 36.1 | Timeout | | | Timeout | | | Timeout | | | | | |
| 44 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 44 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 55 | 10 | 0.2 | 0.2 | 0.6 | 1.0 | 0.4 | 5.6 | 0.5 | 0.2 | 3.0 | 1.8 | 0.2 | 10.5 | 2.7 | 0.2 | 17.2 | 6.0 | 0.2 | 42.4 | | | |
| 55 | 20 | Timeout | | | 12.7 | 0.5 | 178.9 | 11.6 | 0.5 | 88.5 | Timeout | | | Timeout | | | Timeout | | | | | |
| 55 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 55 | 40 | 5.5 | 5.6 | 78.1 | Timeout | | | 1.9 | 0.6 | 8.2 | Timeout | | | Timeout | | | Timeout | | | | | |
| 55 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 66 | 10 | 0.4 | 0.1 | 1.6 | 1.6 | 0.3 | 8.1 | 0.6 | 0.1 | 5.6 | 2.0 | 0.1 | 20.1 | 4.1 | 0.1 | 40.9 | 4.3 | 0.1 | 42.1 | | | |
| 66 | 20 | Timeout | | | 17.5 | 0.1 | 263.1 | 12.0 | 0.1 | 178.7 | Timeout | | | Timeout | | | Timeout | | | | | |
| 66 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 66 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 66 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 66 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 77 | 10 | 0.2 | 0.0 | 0.9 | 1.2 | 0.1 | 8.6 | 0.7 | 0.1 | 4.7 | 2.7 | 0.1 | 16.9 | 4.3 | 0.1 | 33.3 | 5.6 | 0.1 | 44.0 | | | |
| 77 | 20 | 29.4 | 0.3 | 204.5 | Timeout | | | 13.8 | 0.4 | 112.2 | Timeout | | | Timeout | | | Timeout | | | | | |
| 77 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 77 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 77 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 77 | 60 | 1.4 | 0.2 | 8.2 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |
| 77 | 70 | Timeout | | | 7.8 | 3.4 | 46.0 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | | |

Table 4.5: Online domain reductions for the SAT-based configurator using compact encoding with order of magnitude = 4.

| Size | | Number of Container Groups | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | # | 5 | | | 6 | | | 7 | | | 8 | | | 9 | | | 10 | | | | |
| Cel. | Cnt. | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | 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 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | |
| 14 | 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | | |
| 24 | 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 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | |
| 24 | 10 | 0.1 | 0.0 | 0.2 | 0.1 | 0.0 | 0.6 | 0.1 | 0.0 | 0.8 | 0.2 | 0.0 | 0.8 | 0.2 | 0.0 | 1.0 | 0.2 | 0.0 | 1.2 | | |
| 24 | 15 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 24 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 34 | 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 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | |
| 34 | 10 | 0.4 | 0.0 | 1.2 | 0.3 | 0.0 | 2.2 | 0.4 | 0.0 | 3.0 | 0.4 | 0.0 | 2.6 | 0.4 | 0.0 | 3.2 | 0.3 | 0.0 | 2.1 | | |
| 34 | 15 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 34 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 34 | 25 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 34 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 44 | 10 | 1.1 | 0.0 | 6.3 | 2.1 | 0.0 | 19.4 | 1.3 | 0.0 | 12.0 | 2.3 | 0.0 | 22.3 | 2.2 | 0.0 | 21.4 | 3.1 | 0.0 | 23.9 | | |
| 44 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 44 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 44 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 55 | 10 | 1.7 | 0.0 | 10.7 | 4.3 | 0.1 | 37.3 | 3.2 | 0.1 | 27.9 | 6.8 | 0.0 | 52.1 | 6.4 | 0.0 | 49.3 | 4.7 | 0.0 | 35.9 | | |
| 55 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 55 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 55 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 55 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 66 | 10 | 10.4 | 1.0 | 39.8 | 9.6 | 0.4 | 94.3 | 13.5 | 0.0 | 143.1 | 22.5 | 0.0 | 238.8 | 20.1 | 0.0 | 213.7 | 27.5 | 0.0 | 293.8 | | |
| 66 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 66 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 66 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 66 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 66 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 77 | 10 | 6.2 | 0.0 | 30.5 | 4.5 | 0.0 | 43.7 | 8.3 | 0.0 | 81.8 | 13.4 | 0.0 | 93.2 | 6.9 | 0.0 | 55.0 | 8.7 | 0.0 | 69.5 | | |
| 77 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 77 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 77 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 77 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 77 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |
| 77 | 70 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | | |

Table 4.6: Online domain reductions for the SAT-based configurator with direct encoding.

| Size | | Number of Container Groups | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | # | 5 | | | 6 | | | 7 | | | 8 | | | 9 | | | 10 | | |
| Cel. | Cnt. | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max |
| 14 | 5 | 0.0 | 0.0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| 14 | 10 | 0.1 | 0.1 | 0.1 | 0.2 | 0.1 | 0.4 | 0.2 | 0.2 | 0.5 | 0.3 | 0.2 | 0.9 | 0.4 | 0.2 | 0.9 | 0.5 | 0.2 | 1.4 |
| 24 | 5 | 0.2 | 0.2 | 0.3 | 0.3 | 0.2 | 0.5 | 0.3 | 0.2 | 0.5 | 0.3 | 0.2 | 0.5 | 0.3 | 0.2 | 0.5 | 0.3 | 0.3 | 0.5 |
| 24 | 10 | 0.3 | 0.3 | 0.5 | 0.8 | 0.4 | 2.3 | 0.9 | 0.5 | 2.8 | 2.0 | 0.7 | 5.9 | 2.4 | 0.8 | 8.8 | 3.6 | 1.1 | 13.5 |
| 24 | 15 | 0.3 | 0.3 | 0.7 | 0.6 | 0.2 | 2.9 | 10.7 | 0.3 | 91.0 | 9.3 | 0.8 | 117.8 | Timeout | | | Timeout | | |
| 24 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 34 | 5 | 0.7 | 0.6 | 0.9 | 0.7 | 0.6 | 0.9 | 0.4 | 0.4 | 0.6 | 0.6 | 0.6 | 0.8 | 0.6 | 0.6 | 0.8 | 0.6 | 0.6 | 0.8 |
| 34 | 10 | 0.9 | 0.7 | 1.4 | 3.0 | 1.0 | 15.3 | 1.9 | 0.7 | 8.8 | 6.8 | 1.4 | 39.5 | 8.4 | 1.7 | 55.6 | 11.0 | 1.7 | 76.7 |
| 34 | 15 | 1.3 | 0.9 | 3.4 | 3.9 | 1.0 | 25.3 | 5.9 | 0.9 | 56.9 | 4.5 | 1.1 | 21.4 | 7.3 | 2.3 | 40.2 | 6.8 | 2.3 | 27.9 |
| 34 | 20 | 12.1 | 0.8 | 214.6 | 5.1 | 1.6 | 24.4 | 11.3 | 1.8 | 224.4 | Timeout | | | Timeout | | | Timeout | | |
| 34 | 25 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 34 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 10 | 2.1 | 1.9 | 3.5 | 4.0 | 2.5 | 17.1 | 4.7 | 3.1 | 19.5 | 8.1 | 4.6 | 41.0 | 10.3 | 4.6 | 59.8 | 16.4 | 5.3 | 87.4 |
| 44 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 44 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 10 | 4.1 | 4.2 | 6.2 | 8.2 | 5.9 | 26.3 | 10.4 | 7.9 | 34.0 | 18.2 | 9.2 | 70.3 | 20.0 | 5.0 | 105.0 | 18.0 | 6.1 | 87.9 |
| 55 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 10 | 8.1 | 6.6 | 14.0 | 13.2 | 10.0 | 29.7 | 17.4 | 8.9 | 70.4 | Timeout | | | Timeout | | | Timeout | | |
| 66 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 10 | 9.4 | 7.0 | 19.2 | 19.4 | 12.2 | 69.5 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 20 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 30 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 60 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 77 | 70 | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |

Table 4.7: Online domain reductions for the SMT-based configurator.

### 4.3.1 Encountering Exponential Runtime

Looking at the results for all our search-based configurators, a pattern is evident: Most solver calls are quite fast, but as soon as the problem size reaches a certain critical size, a number of extremely time consuming solver calls occur. While our time limit on a single solver call was set to 5 minutes, we did test a few instances without any time limit, these ran for over 24 hours without finding a solution. Our results suggest that search-based interactive configuration using SAT solving has some fundamental problems for solving container vessel stowage problems, as all our solver implementations, including the SMT solver, show the same type of behaviour with the medium and large sized instances.

It is interesting to note that most of the SAT instances solved are easy, even for large instances, while it is only for some specific assignments that the solver encounters the 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 underconstrained to overconstrained [72], or the distribution of solutions and structure in instances [73]. This is likely to be applicable here, as the configurator has to test all assignments, and some of these will have a large impact on the distribution of solutions, e.g. 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 hitting exponential complexity with one of these phase transitions increases, as many more assignments end up being tried.

#### Testing with other SAT solvers

Due to the complexity observed above, we wanted to make sure that our choice of solver was not the source of the bad performance observed. To test this, we implemented a method for saving particularly difficult instances to disk, such that these could be further experimented with. More specifically, we saved CNF files of all instances that took MiniSat more than one minute to solve, and then applied a variety of recently successful solvers to these instances. In addition to MiniSat, the solvers we applied were Lingeling [74], Glucose [75], Glueminisat [76] and Cryptominisat [77]. 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. The results of these experiments are shown in table 4.8.

Lingeling and CryptoMiniSat perform significantly better, both solving around 30 instances that MiniSat does not. But we can also see that 147 instances were unsolvable by all the solvers, and almost all instances take more than 10 seconds to solve for every solver, which means that none of the other solvers provide performance that can be used to avoid the exponential complexity experienced. It is also worth noting that these are single SAT instances, of which

| 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 |

Table 4.8: The number of instances solved by various solvers 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.

many typically have to be solved during a domain reduction, where the runtime of the whole reduction should be kept under one second.

### Finding the source of complexity

In an effort to get a clearer picture of which components of the container vessel stowage problem that make it difficult, we tried to remove various constraints from our problem instances, to see which ones are contributing to the exponential complexity. To do this, we picked a subset of instances that are representative of both of easy, intermediate and hard container vessel stowage instances. The subset of instances was chosen to include a very easy problem (14 slots / 10 containers), a series with 34 slots and an increasing amount of containers (5, 10, 15, 20, 25, 30), providing a snapshot of a situation where we see the picture shifting from easy to hard, and lastly, a few of the very hard problems.

For these instances we have tried removing three of the constraints: The weight constraint, because it accounts for a large part of the SAT instance size, especially for the direct encoding with its dynamic programming weight constraint algorithm, the no overstowage constraint and the gravity constraint, which requires that containers do not hang in free air. The last two constraints both have the property that placing a single container can have a huge impact on the problem structure. For example, placing a container with a late discharge port at a high tier will have a big effect on which containers can now be placed in that stack. Without these constraints, we still have something that closely resembles a container vessel stowage problem, while using much less complex constraints to represent the problem. All of these experiments were done using 10 container weight classes, as the number of weight classes did not have a big impact on the search-based results. The results can be seen in table 4.9. The columns labeled “All disabled” shows the results of running the experiments with all the constraints mentioned above disabled. The following columns show the results obtained from enabling each constraint, with the other two disabled.

As can be seen from the results, the removal of the complex constraints did make the solving of the instances that were already being solved faster, but for the intractable instances it did almost nothing to the complexity, even with all

| Size | | Enabled constraints | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | # | All disabled | | | Weight limits | | | No overstowage | | | Gravity | | |
| Cel. | Cnt. | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max |
| 14 | 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 |
| 34 | 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 |
| 34 | 10 | 0.0 | 0.0 | 0.0 | 0.1 | 0.1 | 0.4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 34 | 15 | 0.0 | 0.0 | 0.1 | 0.3 | 0.1 | 1.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| 34 | 20 | 0.0 | 0.0 | 0.1 | 0.4 | 0.1 | 1.5 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| 34 | 25 | 0.0 | 0.0 | 0.1 | 0.5 | 0.2 | 2.8 | Timeout | | | 0.0 | 0.0 | 0.1 |
| 34 | 30 | 0.1 | 0.0 | 2.1 | Timeout | | | Timeout | | | 0.1 | 0.0 | 2.1 |
| 44 | 40 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 55 | 50 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |
| 66 | 60 | Timeout | | | Timeout | | | 5.9 | 0.0 | 253.7 | Timeout | | |
| 77 | 70 | Timeout | | | Timeout | | | Timeout | | | Timeout | | |

Table 4.9: Online domain reductions for SAT-based configurator with direct encoding, 10 weight classes and various constraints disabled.

three constraints disabled, the solver was only able to scale to 10 more instances than earlier. Turning on the gravity constraints gives the same performance, while turning on either the weight limits or no overstowage constraints causes the solver to fail at 5 to 10 fewer containers.

We tried running several experiments with all the constraints disabled on the instance with 70 containers, with different random user choices for each experiment. The results from this can be seen in table 4.10. Each row in the table corresponds to a single experiment with the instance, with randomly selected user choices. For 5/10 instances we get very strong performance, with average latency well below one second. For the remaining five instances we get timeouts. For the five timeouts, we made two interesting observations. The first was that many user choices are made before the timeout occurs, with more than 50 assignments in each experiment. The second was that for all the timeouts, the previous assignment would be that some chosen cell must be empty.

| Experiment # | Mean | Median | Max |
|---|---|---|---|
| 1 | 0.590 | 0.190 | 2.980 |
| 2 | 0.492 | 0.150 | 3.100 |
| 3 | Timeout | | |
| 4 | Timeout | | |
| 5 | Timeout | | |
| 6 | 0.561 | 0.205 | 2.940 |
| 7 | 0.554 | 0.170 | 3.250 |
| 8 | 0.587 | 0.210 | 3.230 |
| 9 | Timeout | | |
| 10 | Timeout | | |

Table 4.10: Online domain reductions on the instance with 77 slots and 70 containers for the SAT-based configurator with direct encoding, using 10 weight classes and with the gravity, no overstowage and weight limit constraints disabled.

Inspired by these observations, we made a set of experiments where the user is not allowed to choose that cells must be empty, in order to see if this would impact the performance of the configurator. The results from this are shown in

table 4.11. When all three complex constraints are removed we see very strong performance, with all instances solved with low mean domain reduction times, and none of the single domain reductions taking more than one second. With weight limits enabled the configurator is still able to solve all instances, but with mean domain reduction times that are too high on instances with more than 40 containers. For cases where the no overstowage and gravity constraints are enabled, the performance is significantly worse than when the user can assign cells to be empty.

| Size | | Enabled constraints | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | # | All disabled | | | Weight limits | | | No overstowage | | | Gravity | | |
| Cel. | Cnt. | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max |
| 14 | 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 |
| 34 | 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 |
| 34 | 10 | 0.0 | 0.0 | 0.0 | 0.1 | 0.1 | 0.3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 |
| 34 | 15 | 0.0 | 0.0 | 0.1 | 0.2 | 0.2 | 0.9 | 0.0 | 0.0 | 0.1 | Timeout | | |
| 34 | 20 | 0.0 | 0.0 | 0.1 | 0.3 | 0.1 | 1.4 | Timeout | | | Timeout | | |
| 34 | 25 | 0.0 | 0.0 | 0.1 | 0.4 | 0.1 | 2.1 | Timeout | | | Timeout | | |
| 34 | 30 | 0.0 | 0.0 | 0.1 | 0.5 | 0.2 | 3.2 | Timeout | | | Timeout | | |
| 44 | 40 | 0.0 | 0.0 | 0.2 | 1.3 | 0.3 | 8.3 | Timeout | | | Timeout | | |
| 55 | 50 | 0.1 | 0.0 | 0.3 | 2.8 | 1.1 | 15.2 | Timeout | | | Timeout | | |
| 66 | 60 | 0.1 | 0.0 | 0.6 | 4.0 | 0.8 | 26.2 | Timeout | | | Timeout | | |
| 77 | 70 | 0.2 | 0.1 | 0.8 | 8.1 | 2.3 | 44.2 | Timeout | | | Timeout | | |

Table 4.11: The results from running the same experiments as in table 4.9, but disallowing assigning cells to be empty.

Based on all the results presented in this section, there are several conclusions that can be made. With the gravity, no overstowage and weight limit constraints disabled, the configurator is able to scale very well, solving all instances with very strong domain reduction speeds. We can also see that the solver with weight limits enabled is still able to scale decently, although domain reduction times get too high for the biggest instances. Finally, we see that there are at least three sources of complexity in this problem. Both the no overstowage and the gravity constraint is enough in itself to make the configurator encounter exponential complexity in some solver calls. These timeouts happen early in the initial domain reduction, where a few of the domain value assignments cause exponential runtime. This fits well with the previously mentioned work on hardness phase transitions, due to the significant changes in problem structure that these constraints can cause, even for single assignments.

Furthermore, we show that even without these constraints, the random user choices can be enough to cause exponential runtime, albeit only in cases where an unfortunate series of assignments are made.

## 4.4 CLib Results

The results from our experiments with the CLib benchmark suite can be seen in table 4.12. In these results we see performance that is more closely related to the performance reported in [14]. For the fault tree instances, the BDD-based

configurator times out on almost all of them, whereas conversely, the SAT-based configurator performs very strongly on almost all of them.

| x | Solver | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | BDD | | | SAT, b=2 | | | SAT, m=1 | | | SAT, m=2 | | | SAT, m=3 | | | SAT, m=4 | | |
| Problem | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max | Mean | Median | Max |
| FT baobab1 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT baobab2 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT baobab3 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT cea9601 | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT chinese | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9201 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9202 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9203 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9204 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9205 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9206 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9207 | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT das9208 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9209 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9601 | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT das9701 | Out of mem | | | 0.2 | 0.1 | 1.0 | 0.2 | 0.0 | 1.4 | 0.2 | 0.0 | 1.5 | 0.2 | 0.1 | 1.5 | 0.2 | 0.0 | 1.3 |
| FT edf9201 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT edf9202 | Out of mem | | | 0.0 | 0.0 | 0.3 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.3 | 0.0 | 0.0 | 0.3 |
| FT edf9203 | Out of mem | | | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.2 |
| FT edf9204 | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT edf9205 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT edf9206 | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT edfpa14b | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT edfpa14o | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT edfpa14p | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT edfpa14q | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT edfpa14r | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT edfpa15b | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT edfpa15o | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT edfpa15p | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT edfpa15q | Out of mem | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| FT edfpa15r | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT elf9601 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT ftr10 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT isp9601 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT isp9602 | 0.1 | 0.1 | 0.2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT isp9603 | Out of mem | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT isp9604 | 0.1 | 0.1 | 0.7 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT isp9605 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT isp9606 | 0.1 | 0.1 | 0.2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT isp9607 | 0.1 | 0.1 | 0.2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| FT jbd9601 | Out of mem | | | 0.1 | 0.1 | 0.3 | 0.1 | 0.1 | 0.3 | 0.1 | 0.1 | 0.3 | 0.1 | 0.1 | 0.3 | 0.1 | 0.1 | 0.4 |
| FT nus9601 | Out of mem | | | Timeout | | | 0.8 | 0.5 | 3.8 | Timeout | | | Timeout | | | Timeout | | |
| 1-32.1 | Infeasible | | | Infeasible | | | Timeout | | | Infeasible | | | Infeasible | | | Infeasible | | |
| Big-PC | 0.1 | 0.1 | 0.4 | 0.0 | 0.0 | 0.2 | Timeout | | | Timeout | | | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.1 |
| bike2 | 0.1 | 0.1 | 0.1 | Timeout | | | 0.0 | 0.0 | 0.0 | Timeout | | | 0.0 | 0.0 | 0.0 | Timeout | | |
| complex | Out of mem | | | 0.8 | 0.1 | 4.6 | Timeout | | | Timeout | | | 0.8 | 0.1 | 5.2 | 0.8 | 0.1 | 5.1 |
| pc | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | Timeout | | | Timeout | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| pc2 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | Timeout | | | Timeout | | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| renault | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | | Timeout | | |

Table 4.12: Online domain reductions for a range of configuration problems from the CLib package. Fault tree instances are prefixed with 'FT'.

On the PC and Bike configuration instances the BDD performs strongly, whereas the SAT-based approach struggles with the lowest orders of magnitude, suggesting that the CNF representation becomes too big for the SAT solvers to handle.

# Chapter 5

# Conclusion

In this thesis, we built a SAT and an SMT-based complete and backtrack-free interactive configurator, accepting problems with linear constraints, specified in the CSP style modeling language of the configuration package CLab.

As input to these configurator implementations we used instances of the container stowage problem, using an encoding originally devised by [9], along with a suite of configuration problems from different other domains. We also introduced a more direct encoding of the container stowage problem to the CNF format, bypassing the layers of the CSP to SAT conversion.

Compared to the available BDD-based solution, both encodings performed worse on the container stowage problem. The main reason for this was found to be certain difficult problem instances produced in the configuration process, where the task of solving reached exponential complexity. Our configurator implementations did, however, solve a significant amount of problems from other domains, where the construction of the BDD data structure failed.

Based on these results, it seems highly unlikely that SAT-based interactive configuration for the container vessel stowage problem is able to outscale BDD-based solutions. This is supported by the performance of the SMT-based configurator, for which the encoding is almost a direct translation of the CP problem to SMT format, and yet the same performance pattern is seen. We also showed that a host of state-of-the-art SAT solvers all have trouble with the hard instances encountered during our search.

By experimentally disabling various constraints, we have shown that the SAT-based configurator still encounters exponential runtime for simplified versions of the container vessel stowage problem. Most importantly, we have shown that, although it is by far the most complex constraint to express in Boolean form, the weight limit constraint is not the source of the exponential complexity. Furthermore, we have shown that both disallowing overstowage and requiring containers to have support from below are constraints that result in exponential runtimes.

In addition, we have shown that allowing the user to assign slots as empty can have a significant impact on the complexity, even when all three of the complex

constraints are disabled. Making many empty assignments can be enough to force the solver into exponential runtime.

For BDD-based configuration, we have shown that the compute valid domains procedure starts to become prohibitively expensive as the amount of containers gets near 100, even if the BDD can be compiled in reasonable time.

For non stowage-based configuration, we replicated the results from [14], showing that SAT-based configuration is able to significantly outperform BDD-based configuration on some domains.

Based on the results on BDD and CP-based interactive configuration presented in [4], the strong results using SAT in [14] and our results on BDD, SAT and SMT-based interactive configuration, we see that there is no single dominant strategy for solving these problems. The best solution technique for a given problem domain depends on the type of problem being solved.

# Chapter 6

# Future work

For future work there are several areas that would be interesting to explore, both for understanding what the current state-of-the-art is in solving of container vessel stowage problems, but also for improving these methods.

### Better understanding of hardness

Based on the results presented here on what makes the container vessel stowage configuration instances hard, it is possible to do a more thorough examination of this. For theory work, it could be examined how significant the changes in problem structure and solution space are, and if significant, show how and when the phase transition happens.

For experimental work, the reefer constraint and the no 20' containers on top of 40' containers constraint both remain to be thoroughly tested to see how they contribute to the complexity of the problem. These are both constraints that could have significant impact on the solving speed, although the latter constraint has already been tested somewhat, as part of the gravity constraint.

### Better SAT heuristics

Better domain-specific variable selection and variable polarity heuristics could be employed to improve the general speed of searches and possibly avoid some or all of the exponential time searches. The task of designing a good heuristic function requires a good understanding of the inner workings of the solver, when faced with the specific type of problems produced by the interactive container stowage problems and the associated encodings.

### Improved SAT solvers

As we have shown in our experiments, the current state-of-the-art in SAT solving is incapable of handling some of the problem instances that occur in the domain of interactive container vessel stowage. However, this has previously been the case for other domains, for which strong solving has now been achieved. One

example of this is cryptographic problems, where specific tools for handling such problems were introduced in [77]. Therefore, there is hope that future work on SAT solvers can alleviate some of the problems we see.

To help facilitate work on improved solvers, we intend to submit some of the hardest instances we produced to the SAT competition [13].

### Pseudo-Boolean constraints

One possible direction for future work is to encode our constraints as *pseudo-Boolean constraints*. Pseudo-Boolean constraints are linear inequalities where the variables take on the values 0 and 1, representing $false$ and $true$. This type of constraint could be used to express most of the constraints in the container vessel stowage problem in a manner that is simpler than using general linear arithmetic on integers. While the conversion of pseudo-Boolean constraints to proper CNF format is potentially expensive, several methods have been proposed for doing this efficiently [78, 79]. This has even been implemented in a specialized version of the MiniSat solver.

### Alternative SMT modeling

In this work we tested only a very limited subset of the modeling capabilities of SMT solvers. In future work we will experiment with alternative SMT encodings that could allow for stronger theory solvers, having more efficient reasoning capabilities for the structure of our problem. One modeling approach we wish to try is specifying variable domains as enumeration values, such that the solver can reason about the domains without considering arithmetic over the variables.

### Stronger BDD solutions

Another area where future research could be applied is in the strengthening of the BDD-based solution. As can be seen in our results, the domain reduction for the BDD goes well beyond one second for the biggest instances. This is problematic as big configuration problems, even if compilable, would still be infeasible, because the response time would be too slow. One method for mitigating this would be to optimize the look up procedure. Half the time spent in the look up procedure is spent applying the new user assignment to the current BDD. One possible direction would be to avoid applying this assignment, and only allowing usage of the edges corresponding to the assignment when traversing the BDD in future look-ups.

Another area that could be explored are other encodings for the BDD. As it is implemented now, only a single encoding has been tried for the BDD, whereas other encodings might be more scalable.

### Other implementations

While this thesis focused on comparing the performance of SAT, SMT and BDD-based interactive configuration, it would also be interesting to examine

other implementations to understand their performance characteristics. While CP-based configurators have previously been shown inferior to BDDs, recent work on CP models for the non-interactive container vessel stowage problem has shown solving speeds below one second for large instances with more than 150 containers [3], and methods combining BDDs and search have also been proposed to improve scalability [19]. Similarly, the authors show good performance for IP models in [3]. However, all these search methods could easily have the same scalability problems as we have seen with specific assignments, since our SAT-based approach also does well on the initial problem, even for large instances.

Another area that has largely been ignored in the literature is the compilation of configuration problems to representations as automata. This was introduced for interactive configuration in [80], and might be able to scale better than a BDD-based approach. However, this technology has not been studied much, and hence its performance characteristics are not well understood.

# Bibliography


[1] M. Avriel, M. Penn, and N. Shpirer, "Container ship stowage problem: Complexity and connection to the coloring of circle graphs," *Discrete Applied Mathematics*, vol. 103, pp. 271–279, 2000.

[2] D. Pacino, A. Delgado, R. M. Jensen, and T. Bebbington, "Fast generation of near-optimal plans for eco-efficient stowage of large container vessels," in *Proceedings of the 2nd International Conference on Computational Logistics (ICCL'11)*, LNCS, Springer, 2011.

[3] A. Delgado, R. M. Jensen, K. Janstrup, T. H. Rose, and K. H. Andersen, "A constraint programming model for fast optimal stowage of container vessel bays," *European Journal of Operational Research*, 2011.

[4] S. Subbarayan, R. M. Jensen, T. Hadzic, H. R. Andersen, J. Møller, and H. Hulgaard, "Comparing two implementations of a complete and backtrack-free interactive configurator," in *Proceedings of the CP-04 Workshop on CSP Techniques with Immediate Application*, pp. 97–111, Springer, 2004.

[5] H. R. Andersen, T. Hadzic, and D. Pisinger, "Interactive cost configuration over decision diagrams," *J. Artif. Intell. Res. (JAIR)*, vol. 37, pp. 99–139, 2010.

[6] T. Hadzic and H. R. Andersen, "A BDD-based polytime algorithm for cost-bounded interactive configuration," in *AAAI'06: Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence*, 2006.

[7] E. R. Hansen and H. R. Andersen, "Interactive configuration with regular string constraints," in *AAAI'07: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pp. 217–223, 2007.

[8] E. R. van der Meer, A. Wasowski, and H. R. Andersen, "Efficient interactive configuration of unbounded modular systems," in *SAC'06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pp. 409–414, 2006.

[9] R. M. Jensen, E. Leknes, and T. Bebbington, "Fast interactive decision support for modifying stowage plans using binary decision diagrams," in *International MultiConference of Engineers and Computer Scientists*, 2012.

[10] "Clib: Configuration benchmarks library, http://www.itu.dk/research/cla/externals/clib/."

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proceedings of the 38th annual Design Automation Conference*, DAC '01, (New York, NY, USA), pp. 530–535, ACM, 2001.

[12] G. Pan and M. Vardi, "Search vs. symbolic techniques in satisfiability solving," in *Theory and Applications of Satisfiability Testing*, vol. 3542 of *Lecture Notes in Computer Science*, pp. 898–899, Springer Berlin / Heidelberg, 2005.

[13] "SAT competition, http://www.satcompetition.org/."

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

[15] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.

[16] "SMT competition, http://www.smtcomp.org/."

[17] R. M. Jensen, "CLab: A C++ library for fast backtrack-free interactive product configuration.," in *CP* (M. Wallace, ed.), vol. 3258 of *Lecture Notes in Computer Science*, p. 816, Springer, 2004.

[18] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 8, pp. 677–691, 1986.

[19] A. H. Nørgaard, M. R. Boysen, R. M. Jensen, and P. Tiedemann, "Combining binary decision diagrams and backtracking search for scalable backtrack-free interactive product configuration," in *In Proceedings of the IJCAI09 Workshop on Configuration*, 2000.

[20] H. A. Kautz and B. Selman, "Planning as satisfiability," in *European Conference on Artificial Intelligence*, pp. 359–363, 1992.

[21] N. Tamura and M. Banbara, "Sugar: A CSP to SAT translator based on order encoding," in *Proceedings of the Third International CSP Solver Competition1*, pp. 815–822, 2008.

[22] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Studies in Constructive Mathematics and Logic*, vol. Part II, 1968.

[23] M. Davis, G. W. Logemann, and D. W. Loveland, "A machine program for theorem-proving," *Communications of The ACM*, vol. 5, pp. 394–397, 1962.

[24] H. Zhang and M. E. Stickel, "An efficient algorithm for unit propagation," in *In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH96), Fort Lauderdale (Florida USA*, pp. 166–169, 1996.

[25] J. P. M. Silva and K. A. Sakallah, "Grasp - a new search algorithm for satisfiability," in *International Conference on Computer Aided Design*, pp. 220–227, 1996.

[26] J. Huang, "The effect of restarts on the efficiency of clause learning," in *International Joint Conference on Artificial Intelligence*, pp. 2318–2323, 2007.

[27] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal speedup of las vegas algorithms," *Information Processing Letters*, vol. 47, pp. 128–133, 1993.

[28] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Theory and Applications of Satisfiability Testing*, pp. 294–299, 2007.

[29] N. En and N. Srensson, "Temporal induction by incremental SAT solving," *Electronic Notes in Theoretical Computer Science*, vol. 89, 2003.

[30] E. Koutsoupias and C. H. Papadimitriou, "On the greedy algorithm for satisfiability," *Information Processing Letters*, vol. 43, pp. 53–55, 1992.

[31] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artificial Intelligence*, vol. 126, pp. 43–62, 2001.

[32] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, 2008.

[33] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, "ISAC - instance-specific algorithm configuration," in *European Conference on Artificial Intelligence*, pp. 751–756, 2010.

[34] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Dpll( t): Fast decision procedures," in *Computer Aided Verification*, pp. 175–188, 2004.

[35] L. D. Moura and N. B. rner, "Z3: An efficient SMT solver," in *Tools and Algorithms for Construction and Analysis of Systems*, pp. 337–340, 2008.

[36] I. D. Wilson and P. A. Roach, "Principles of combinatorial optimization applied to container-ship stowage planning," *Journal of Heuristics*, vol. 5, pp. 403–418, 1999.

[37] Navis, "PowerStow." `www.navis.com`, 2011.

[38] Interschalt, "Seacos." `www.interschalt.de`, 2011.

[39] J. G. Kang and Y. D. Kim, "Stowage planning in maritime container transportation," *Journal of the Operational Society*, vol. 53, pp. 415–426, 2002.

[40] W.-Y. Zhang, Y. Lin, and Z.-S. Ji, "Model and algorithm for container ship stowage planning based on bin-packing problem," *Journal of Marine Science and Application*, vol. 4, no. 3, 2005.

[41] D. Ambrosino, D. Anghinolfi, M. Paolucci, and A. Sciomachen, "An experimental comparison of different heuristics for the master bay plan problem," in *Proceedings of the 9th Int. Symposium on Experimental Algorithms*, pp. 314–325, 2010.

[42] M. Yoke, H. Low, X. Xiao, F. Liu, S. Y. Huang, W. J. Hsu, and Z. Li, "An automated stowage planning system for large containerships," in *In Proceedings of the 4th Virtual Int. Conference on Intelligent Production Machines and Systems*, 2009.

[43] R. C. Botter and M. A. Brinati, "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*, pp. 217–229, North-Holland Publishing Co., 1992.

[44] D. Ambrosino and A. Sciomachen, "Impact of yard organization on the master bay planning problem," *Maritime Economics and Logistics*, no. 5, pp. 285–300, 2003.

[45] P. Giemesch and A. Jellinghaus, "Optimization models for the containership stowage problem." Proceedings of the International Conference of the German Operations Research Society, 2003.

[46] F. Li, C. Tian, R. Cao, and W. Ding, "An integer programming for container stowage problem," in *Proceedings of the Int. Conference on Computational Science, Part I*, pp. 853–862, Springer, 2008. LNCS 5101.

[47] D. Ambrosino and A. Sciomachen, "A constraint satisfaction approach for master bay plans," *Maritime Engineering and Ports*, vol. 36, pp. 175–184, 1998.

[48] A. Delgado, R. M. Jensen, and C. Schulte, "Generating optimal stowage plans for container vessel bays," in *Proceedings of the 15th Int. Conf. on Principles and Practice of Constraint Programming (CP-09)*, vol. 5732 of *LNCS Series*, pp. 6–20, 2009.

[49] Y. Davidor and M. Avihail, "A method for determining a vessel stowage plan, Patent Publication WO9735266," 1996.

[50] O. Dubrovsky, G. Levitin, and M. Penn, "A genetic algorithm with a compact solution encoding for the container ship stowage problem," *Journal of Heuristics*, vol. 8, pp. 585–599, 2002.

[51] M. Flor, "Heuristic algorithms for solving the container ship stowage problem," Master's thesis, Technion, Haifa, Isreal, 1998.

[52] M. Avriel, M. Penn, N. Shpirer, and S. Witteboon, "Stowage planning for container ships to reduce the number of shifts," *Annals of Operations Research*, vol. 76, pp. 55–71, 1998.

[53] A. Sciomachen and A. Tanfani, "The master bay plan problem: a solution method based on its connection to the three-dimensional bin packing problem," *IMA Journal of Management Mathematics*, vol. 14, pp. 251–269, 2003.

[54] W. C. Aye, M. Y. H. Low, H. S. Ying, H. W. Jing, and Z. Min, "Visualization and simulation tool for automated stowage plan generation system," in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 (IMECS 2010)*, vol. 2, (Hong Kong), pp. 1013–1019, 2010.

[55] S. Nugroho, "Case-based stowage planning for container ships," in *The Int. Logistics Congress*, 2004.

[56] R. M. Jensen, "Clab 1.0 user manual," 2004.

[57] J. D. Kleer, "A comparison of ATMS and CSP techniques," in *International Joint Conference on Artificial Intelligence*, pp. 290–296, 1989.

[58] T. Walsh, "SAT v CSP," in *Principles and Practice of Constraint Programming*, pp. 441–456, 2000.

[59] T. Tanjo, N. Tamura, and M. Banbara, "Towards a compact and efficient SAT-encoding of finite linear CSP," in *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation (ModRef 2010)*, 2010.

[60] T. Tanjo, N. Tamura, and M. Banbara, "Proposal of a compact and efficient SAT encoding using a numeral system of any base," in *Proceedings of the 1st International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT 2011)*, 2011.

[61] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara, "Compiling finite linear CSP into SAT," *Constraints*, vol. 14, no. 2, pp. 254–272, 2009.

[62] K. Iwama and S. Miyazaki, "SAT-variable complexity of hard combinatorial problems," in *IFIP Congress (1)*, pp. 253–258, 1994.

[63] M. Gavanelli, "The log-support encoding of CSP into SAT," in *Principles and Practice of Constraint Programming*, pp. 815–822, 2007.

[64] C. Anstegui, M. L. Bonet, J. Levy, and F. Many, "Measuring the hardness of SAT instances," in *National Conference on Artificial Intelligence*, pp. 222–228, 2008.

[65] H. Xu, R. A. Rutenbar, and K. A. Sakallah, "sub-SAT: a formulation for relaxed boolean satisfiability with applications in routing," in *International Symposium on Physical Design*, pp. 182–187, 2002.

[66] P. Kilby, J. K. Slaney, S. Thibaux, and T. Walsh, "Backbones and backdoors in satisfiability," in *National Conference on Artificial Intelligence*, pp. 1368–1373, 2005.

[67] "CSP competition 2008, http://www.cril.univ-artois.fr/cpai08/results/ranking.php?idev=15."

[68] "CSP competition 2009, http://www.cril.univ-artois.fr/cpai09/results/ranking.php?idev=30."

[69] J. Amilhastre, H. Fargier, and P. Marquis, "Consistency restoration and explanations in dynamic CSPs application to configuration," *Artificial Intelligence*, vol. 135, no. 12, pp. 199 – 234, 2002.

[70] R. Jensen and L. E. Sonne, "Power supply restoration," Master's thesis, Department of Innovation, IT University of Copenhagen, 2005.

[71] Y. Dutuit and A. Rauzy, "Approximate estimation of system reliability via fault trees," *Reliability Engineering & System Safety*, vol. 87, pp. 163–172, 2005.

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

[73] C. Coarfa, D. D. Demopoulos, A. S. M. Aguirre, D. Subramanian, and M. Y. Vardi, *Random 3SAT: The Plot Thickens*. 2000.

[74] A. Biere, "Lingeling and friends at the SAT competition 2011," in *SAT Competition*, 2011.

[75] G. Audemard and L. Simon, "glucose, solver description," in *SAT Competition*, 2011.

[76] H. Nabeshima, "glueminisat, solver description," in *SAT Competition*, 2011.

[77] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing*, pp. 244–257, 2009.

[78] O. Bailleux and Y. Boufkhad, "Efficient cnf encoding of boolean cardinality constraints," in *Principles and Practice of Constraint Programming*, pp. 108–122, 2003.

[79] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT," *JSAT*, vol. 2, no. 1-4, pp. 1–26, 2006.

[80] J. Amilhastre, H. Fargier, and P. Marquis, "Consistency restoration and explanations in dynamic CSPs application to configuration," *Artificial Intelligence*, vol. 135, pp. 199–234, 2002.