Constraint Optimization for Highly Constrained Logistic Problems

Maria Kinga Mochnacs
Meang Akira Tanaka
Anders Nyborg
Rune Møller Jensen

IT University Technical Report Series **TR-2007-2008-104**

ISSN 1600–6100 **September 2007**

ISSN 1600–6100

ISBN 978-87-7949-163-2

Copies may be obtained by contacting:

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

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

Abstract

This report investigates whether propagators combined with branch and bound algorithm are suitable for solving the storage area stowage problem within reasonable time. The approach has not been attempted before and experiments show that the implementation was not capable of solving the storage area stowage problem efficiently. Nevertheless, the report incorporates a detailed analysis of the problem, acts as a valuable basis for comparing the quality of alternative approaches and reveals the properties of the solution space.

Contents

Chapter 1

Introduction

Containerized transport in vessels traveling overseas is a field in rapid growth. As the trade increases, pressure is put on shipping companies to lower the cost of their transportation services. For that reason, there is an interest in the industry for developing algorithms, which can help placing containers efficiently aboard a vessel, respecting safety requirements and optimizing logistic criteria. Viewed from an academic perspective the problem has some interesting properties, as it contains subproblems, which have been proven to be NP-hard [1].

Placing containers on a vessel can be regarded as a combinatorial problem. The size of the combinatorial space can be roughly estimated as a permutation of placing a unique container for each slot available. Since a bay may accommodate up to 200 20-foot containers, the combinatorial space is immense, making it a very hard combinatorial problem.

One typical approach used within the field of operations research is to solve the problem by using integer programming. However nonlinear constraints cannot be modeled properly by the usage of integer programming. Consequently alternative approaches have to be considered.

In this report, an in depth study is given of the storage area stowage problem, which is a constraint optimization problem, consisting of arranging a set of containers below deck within a bay of a vessel. The safety requirements and logistic criteria are divided into hard and soft constraints respectively. A weight has been defined for each soft constraint to identify the importance of fulfilling each logistic criteria. Due to the ability of modeling nonlinear constraints in a simple fashion, functions known as propagators has been chosen to represent the constraints within the problem The chosen algorithm for solving the problem is branch and bound, described by [2] as: ”the most commonly known algorithm for solving constraint optimization problems”. The algorithm and choice of representation was selected based on the fact that no research within the field of containerized transport overseas exists, relying on this combination to solve the storage area stowage problem. The goal is that the research provided in this report will serve as a first step, in uncovering some

of the strengths and weaknesses by using a Constraint Satisfaction Problem(CSP)-model and branch and bound for solving the storage area stowage problem.

The issues which, this report would like to address is as follows:

Can backtrack combined with a CSP-Model find a solution within reasonable time for the storage area stowage problem.

Can branch and bound combined with a CSP-Model find an optimal solution within reasonable time for the storage area stowage problem.

Several task had to be formulated in order to answer the above issues. The first tasks, is to get a thorough understanding by formulating a mathematical model of the problem. The second task is to find a suitable CSP-Model by considering different candidate models, evaluate each of these, and select the most suitable candidate. The third tasks is to conduct experiments on the developed implementation and analyze the results.

Document outline

In chapter 2, the theoretical background of the report is established. Based on the theory and an informal problem description, the problem is formalized into a mathematical model in chapter 3. Three candidate CSP-Models are suggested based on an analysis of the formalized model in chapter 4. Chapter 5 formalize the chosen CSP-Modeland the formalization of the estimators are presented in chapter 6. Implementation of the CSP-Modeland search algorithms are represented in chapter 7. Based on the implementation, experiments are performed to cover different aspects of the search space and implementation in chapter 8. Chapter 9 concludes the report with some suggestion of what can be done in future.

Chapter 2

Background

This chapter provides a brief explanation of different constraint processing concepts used in the report.

2.1 Constraint satisfaction problems

Constraint satisfaction problems, or CSPs, are mathematical problems that typically involve finding out how to assign a discrete set of variables under certain constraints. Many solutions may actually satisfy the constraints of the problem.

For many hard constraint satisfaction problems, no algorithm has been discovered to solve the problem efficiently yet and some have been proved to be NP-hard. Solving these combinatorial problems is done by searching in the solution space, which is typically exponential in the size of the problem. A systematic search of the search space ensures that all candidate solutions are considered and the optimal one is found with certainty.

A *constraint network* is a model of a CSP that consists of a finite set of variables, a finite set of domains, and a finite set of constraints. A *variable* is a value holder for an entity of the problem. Each variable has its own *domain* that lists the possible values the variable can take. A *constraint* is a relation defined on a subset of variables that represents simultaneous legal assignments of the variables. A constraint can be specified explicitly by the list of satisfying tuples, or implicitly by a formula that characterizes the constraint.

An *instantiation* is the assignment of some subset of variables with some value from the domain of each variable. When all variables are assigned, the instantiation is said to be *complete*. Otherwise the instantiation is said to be *partial*. An instantiation is *consistent*, if it satisfies all of the constraints, whose scopes have no uninstantiated variables.

A *valid solution* of the constraint network is a consistent complete instantiation of all of its variables. An unsatisfiable problem does not have any solutions.

Constraint propagation

Searching for solutions can be viewed as traversing a search space, where the task is to reach a state, where all variables have been assigned with a legal value from their domain. Moving from one state to another state in the search space implies assigning or unassigning variables. The search space of the problem can be considerably larger than the solution space, potentially containing many inconsistent instantiations in respect to the given problem. Consequently, searching for solutions can be very inefficient. An approach is to tighten the search space by formulating an equivalent but more explicit model.

In general, the more explicit the model is, the more restricted the search space will be, making search more effective. When any consistent instantiation of a subset of variables can be extended to a consistent instantiation of all the variables, the model is said to be *globally consistent*. Having a model, which is globally consistent, makes it straightforward to find solutions, since any value chosen for any variable will lead to a solution. However computing a globally consistent model is intractable for sufficiently large problems. Instead, transforming a model into an approximation of a global consistent model may be preferable due to the lower computation cost.

Constraint propagation is the process of transforming a model into a tighter one. The tightening process can be done during the search itself, by inferring new knowledge using *local consistency enforcing algorithms* that perform a bounded amount of constraint inference during each iteration, such as arc or path consistency. A local consistency property is defined regardless of the domains or constraints of the CSP problem.

Another approach to do constraint propagation is *rules iteration*. Rules iteration tightens a model by iteratively applying reduction rules. A *reduction rule* or a *propagator* is a decreasing function that rules out domain values, which will not appear in a solution. A propagator depends on one or more *input* variables and changes the domain of one or more *output* variables. The assignment of an input variable *triggers* the propagator. Propagators can be seen as an actual implementation of the constraints themselves. When assigning one variable a specific value, the set of propagators remove values from the domain of the uninstantiated variables enforcing consistency with the newly instantiated variable.

A one-to-one relationship does not necessarily exists among the set of constraints and the set of propagators. The problems nature may imply that it is easier to construct several propagators that jointly implement a specific constraint. When a combination of several propagators together implement a given constraint, the propagators that the combination consists of is said to *contain the constraint*.

Example 2.1 *One of the constraints within the storage area stowage problem is that a 20-foot container cannot be stacked on top of a 40-foot container. One approach is to divide the constraints into two propagators: One, which ensures that no 40-foot containers can be placed below a 20-foot container and one, which ensures that no 20-foot container can*

be placed on top of a 40-foot.

For details of rules iteration and how a propagation engine works, refer to [3].

2.2 Search algorithms

The goal of a search algorithm is to find solutions to the CSP or conclude that the problem is unsatisfiable. Traversing the search space can be based on different strategies, each strategy resulting in a family of search algorithms. The search family this report considers is the backtrack search family.

Backtrack search algorithms belong to the family of systematic search and, as a consequence, are guaranteed to be complete. The completeness is attained by viewing the search space as a tree, where each node in the tree is an instantiation of a single variable and each branch is a possible assignment for that particular variable. The depth of the tree is determined by the number of variables and consequently the paths, from the root to a leaf node in the tree, are complete instantiations. The starting point, where all backtrack algorithms originate from, is the naive backtrack algorithm, which can be thought of as an algorithm which performs a depth first traversal of the tree, until a solution is found. If some variable along the path towards the leaf node results in an inconsistent partial instantiation, a backtrack occurs. *Backtracking* is the process, where the assignment of a previously assigned variable is reconsidered. Traversing the tree is clearly exponential in time in the worst case and is not practical for too large problems. Therefore, many variations of the naive backtrack algorithm, which tries to improve search time, have been suggested.

This report considers two members of the backtrack family: Forward checking and Dynamic Variable Forward Checking (DVFC). The motivation for choosing forward checking, is that the problem contains properties, which makes the algorithm suitable for finding optimal solutions efficiently along a static variable order. The DVFC has been chosen based on previous experience of being an algorithm, which could find a solution fast, since being based on the forward checking approach of pruning, ensures that the branching factor of the next variable to be instantiated is at a minimum [4].

Current variable is the variable, which the search algorithm currently is attempting to find an assignment for. The nodes of the search tree represent the current variable of a specific stage in the search.

Candidate value is any possible value an uninstantiated variable can be assigned to. In respect to the search tree the branches of a node are candidate values.

Current instantiation is the assignments that has been done until so far in the search. The path from the root of the search tree down to current variable is the current instantiation.

Future variables is the set of variables which still has to be assigned, excluding the current

variable. The set of nodes along any path from the current variable is the future variables.

Forward Checking

Forward checking is a simple improvement to naive backtracking. The principle is to prune domain values from future variables, that are inconsistent with the currently instantiation. Given a CSP-Model, where the constraints are represented with propagators, forward checking is achieved in a straightforward way: Whenever a variable is assigned, all propagators that specify the variable as input are applied. This guarantees that the current instantiation is consistent with any assignment of some future variable. Forward checking is superior compared to naive backtracking, in that it ensures that thrashing is avoided at an earlier stage of a given instantiation.

DVFC

DVFC is a heuristic based on the forward checking strategy that takes into account the benefits of variable orderings, which produces a small search space. DVFC determines the variable ordering dynamically, during search. It relies on the fail-first heuristic, by selecting the variable, which is most likely to restrict the search space as early as possible. By considering the variables that most likely restrict the search space as early as possible, DVFC strengthens the benefits of forward-checking look ahead by being able to detect dead ends as soon as possible considering the amount inference. All other factors being equal, the variable with the smallest number of viable values in its current domain, will have the fewest subtrees rooted at those values, and therefore the smallest search space below it.

For a thorough presentation of backtrack variations and their standard implementation refer to [2].

2.3 Constraint optimization problems

For some problems, candidate solutions must be ranked in terms of quality to some given criteria. In this case, constraint problems are *optimization problems* or COPs. The quality is described by an *objective* or *cost* function and the goal is to find a solution with as high quality as possible, in other words a solution with an optimal objective function value. In case the objective function has to be minimized, a minimization problem is considered, otherwise a maximization problem is considered.

A CSP-Model augmented with a cost function provides a framework to model a COP. Typically, the cost function is a weighted sum of several cost components. A *cost component*

is a problem-dependent real value function defined on a subset of variables. The cost components are also referred to as *soft constraints*, while the constraints of the problem are referred to as *hard constraints*.

An *optimal solution* for a minimization problem is any valid solution which has the lowest cost amongst all valid solutions.

For a detailed description of COP see [2].

2.3.1 Branch and Bound

Any backtracking algorithm can easily be modified in order to find an optimal solution of a COP: Rather than stopping with the first solution, the search is continued throughout the entire search space. Whenever a solution is found, evaluate its cost and maintain the current best cost solution. Given that the solution space is exponential this is intractable for sufficiently large problems. A straightforward improvement is to exploit the cost function. In case of a minimization problem, when the sum of cost components over the instantiated variables is already higher than the best solution found so far, the partial solution can be pruned away.

The above idea is the foundation of a popular search algorithm for constraint optimization, namely *branch and bound*. Branch and bound estimates the completion cost of a partial solution to prune potential solutions away. The algorithm maintains the cost of the best solution found so far. In case of a minimization problem this cost is an upper bound U for the cost of the optimal solution. Additionally, whenever a variable is instantiated, a *bounding evaluation function* f computes a lower bound L on the cost of any complete solution that extends the current partial instantiation. In case $L \geq U$, the partial solution cannot improve the current best cost and therefore the search along the current patch can be discontinued. In case $L < U$, the search continues along the current path, since there is a possibility to improve the current best cost. The algorithm terminates, when the first variable has no values left. The bounding evaluation function sums over two parts: the cost of the current partial instantiation and an estimated cost of the optimal completion of the current instantiation to a complete instantiation.

In order to ensure that all solutions, which improve the best cost are discovered, it is required that the estimated cost is an underestimate of optimal completion cost. On the other hand, in case the estimate is too weak, branch and bound will explore unnecessary solutions. The goal is to have an estimate as close as possible to the best completion cost.

For some problems, finding the optimal solution is not feasible and one may settle for less by computing an *approximation* of the optimal solution. The principle for computing an approximation of the optimal solution is to allow the estimation part to overestimate by a constant. Consequently, some solutions will be skipped and the search space is reduced.

The cost of the first found solution has an impact on the performance of branch and bound. The closer the cost is to the optimal cost, the more solutions are pruned away during search, and the sooner the search finishes. A *diving heuristic* is a heuristic to find a good initial solution.

For details of branch and bound refer to [2].

2.3.2 Model of a constraint optimization problem

This section formally presents the model this report uses for the given problem. It starts by defining the notions of a *domain mapper* and *propagator*, and concludes with the *CSP-Model* respectively *COP-Model* and a set of general notations.

Definition 2.1 (Domain mapper) *Let P be a CSP, let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of variables for P and let d_i be the initial domain for each $x_i \in X$.*
A domain mapper is a total function that specifies for each variable $x_i \in X$ a set of domain values $\mathcal{D}(x_i) \subseteq 2^{d_i}$.

A domain mapper $\mathcal{D}_1$ is *stronger* than a domain mapper $\mathcal{D}_2$, written $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$, if $\mathcal{D}_1(x_i) \subseteq \mathcal{D}_2(x_i) \forall x_i \in X$.

Definition 2.2 (Domain mapper consistent with an assignment) *Let $\mathcal{D}$ be a domain mapper and let $\langle x_j, v\rangle$ denote the assignment of an arbitrary value v to a variable x_j.*
A domain mapper consistent with the assignment $\langle x_j, v\rangle$ is a domain mapper $\mathcal{D}_{\langle x_j,v\rangle} \subseteq \mathcal{D}$ such that the assignment $\langle x_j, v\rangle$ can be extended consistently by any future assignment of any other variable x_i. In case x_i cannot extend consistently $\langle x_j, v\rangle$ then $\mathcal{D}_{\langle x_j,v\rangle}(x_i) = \emptyset$

$$\mathcal{D}_{\langle x_j,v\rangle}(x_i) = \begin{cases} D \subseteq \mathcal{D}(x_i) & \text{if } \langle x_j, v\rangle \text{ is consistent with } \langle x_i, u\rangle \text{ for any } u \in D \\ \emptyset & \text{otherwise} \end{cases}$$

The *strongest* domain mapper consistent with the assignment $\langle x_j, v\rangle$ is:

$$\mathcal{D}^*_{\langle x_j,v\rangle} \in \{\mathcal{D}_{\langle x_j,v\rangle} \,:\, \forall \mathcal{D}'_{\langle x_j,v\rangle} \sqsubseteq \mathcal{D} : (\mathcal{D}'_{\langle x_j,v\rangle} \neq \mathcal{D}_{\langle x_j,v\rangle} \Rightarrow \mathcal{D}'_{\langle x_j,v\rangle} \sqsubseteq \mathcal{D}_{\langle x_j,v\rangle})\}$$

Definition 2.3 (Propagator)
A propagator *is a triple $(\mathcal{P}, \mathcal{I}_\mathcal{P}, \mathcal{O}_\mathcal{P})$ consisting of:*

1. *a set of one input variables $\mathcal{I}_\mathcal{P} \subseteq X$. An assignment $\langle x_i, v\rangle$ of an arbitrary value v to an input variable $x_i \in \mathcal{I}_\mathcal{P}$* triggers *the domain decreasing function $\mathcal{P}$.*
2. *a set of output variables $\mathcal{O}_\mathcal{P} \subseteq X$. The domain of an output variable may be pruned when the $\mathcal{P}$ is triggered.*

3. *a domain decreasing function $\mathcal{P}$:*

$$\mathcal{P}(\mathcal{D})(x_i) = \begin{cases} \{v\} & \text{if } x_i \in \mathcal{I}_{\mathcal{P}} \text{ and } \langle x_i, v\rangle \text{ triggered } \mathcal{P} \\ \mathcal{D}^*_{\langle x_i,v\rangle}(x_k) & \text{if } x_k \in \mathcal{O}_{\mathcal{P}} \text{ and } \langle x_i, v\rangle \text{ triggered } \mathcal{P} \\ \mathcal{D}(x_i) & \text{otherwise} \end{cases}$$

Definition 2.4 (CSP-Model of a constraint satisfaction problem) *Let P be a CSP. A CSP-Model $\Re$ for P is a triple $(X, \mathcal{D}, \mathcal{C})$, consisting of:*

1. *a finite set of variables $X = \{x_1, x_2, \ldots, x_n\}$*
2. *the initial variable domains $\mathcal{D}(x_i) = d_i$*
3. *a finite set of constraints $\mathcal{C} = \{r_1, \ldots, r_m\}$ where each constraint r_j is explicitly specified as a set of propagators.*

The definition of the COP-Model extends the definition of the CSP-Model:

Definition 2.5 (COP-Model of a constraint optimization problem) *Let P^* be a COP. A COP-Model $\Re^*$ for P^* is defined as a pair $(\Re, F)$ where:*

1. *$\Re$ is the CSP-Model of the constraint satisfaction problem P^**
2. *F is the cost function that measures the quality of a solution $\vec{a}$ with regards to a finite set of cost components $\{F_1, F_2, \ldots, F_l\}$ and a finite set of weights $\{W_1, W_2, \ldots, W_l\}$.*

$$F(\vec{a}) = \sum_{j=1}^{l} W_j F_j(\vec{a})$$

Definition 2.6 (Bounding evaluation function) *Let P^* be a COP and let $(\Re, F)$ be the COP-Model of P^*.*
A bounding evaluation function f for a partial instantiation $\vec{a}_p$ is defined as:

$$f(\vec{a}_p) = \sum_{j=1}^{l} W_j \big(g_j(\vec{a}_p) + h_j(\vec{a}_p)\big)$$

where

$g_j(\vec{a}_p) = F_j(\vec{a}_p)$ is the true cost component F_j restricted to the partial instantiation $\vec{a}_p$ and

$h_j(\vec{a}_p)$ is the estimated completion cost of $\vec{a}_p$ into a complete, but not necessarily valid instantiation.

The following table summarizes the notations used throughout this report.

General notations		
X	:	variable set
$\mathcal{D}$	:	domain mapper
$\mathcal{P}^{name}$	:	propagator identified by a ***name***
$\mathcal{I}_{\mathcal{P}^{name}}$	:	input variable set of $\mathcal{P}^{name}$
$\mathcal{O}_{\mathcal{P}^{name}}$	:	output variable set of $\mathcal{P}^{name}$
$\vec{a}$	:	current instantiation
$\mathcal{S}$	:	scope of $\vec{a}$
$\pi_{\mathcal{S}_i}(\vec{a})$	:	projection of $\vec{a}$ on $\mathcal{S}_i \subseteq \mathcal{S}$
$\vec{a}_p$	:	partial instantiation of the first p variables
$\mathcal{S}_p$	:	scope of $\vec{a}_p$
$(\vec{a}_p, a_{p+1}, \ldots, a_n)$	:	complete instantiation extended from $\vec{a}_p$
F_{name}	:	cost component identified by a ***name***
W_{name}	:	unit weight for the cost component F_{name}
$g_{name}(\vec{a}_p)$	:	true cost of $\vec{a}_p$, restricted to the cost component F_{name}
$h_{name}(\vec{a}_p)$	:	estimated completion cost of $\vec{a}_p$, restricted to the cost component F_{name}
$h^*_{name}(\vec{a}_p)$	:	optimal completion cost of $\vec{a}_p$, restricted to the cost component F_{name}
$f(\vec{a}_p)$	:	bounding evaluation function for $\vec{a}_p$ i.e. estimated completion cost of $\vec{a}_p$
$f^*(\vec{a}_p)$	:	optimal completion cost of $\vec{a}_p$

A variable x ***belongs*** to an instantiation $\vec{a}$ if it is in the scope of the instantiation. This is denoted as $x \in \mathcal{S}$.

A domain value v ***belongs*** to an instantiation $\vec{a}$, if a variable within the scope of the instantiation is assigned to it. This is denoted as $v \in \pi_{\mathcal{S}}(\vec{a})$.

Chapter 3

The Storage Area Stowage Problem

The motivation for this chapter is to present a formalization of the storage area stowage problem. The formalization is based on an informal problem description, which can be found in the appendix B. The chapter begins with providing background information about various notions within the problem domain. After the background information the informal description is translated into a mathematical model.

3.1 Background

As goods are often manufactured far away from the consumer, the goods will have to be transported to the consumer. One way of doing this is by containerized transportation over sea, where vessels sail along preplanned routes. The preplanned routes makes it simple to decide, how containers can be transported from one destination to another. Each route forms a cycle and at each stop on the route, the vessel may unload containers or load additional containers destined for future ports. Therefore, vessels arriving at any port will usually have containers onboard. The containers arriving at a port may come inland e.g. by train or truck or by seaway e.g. other vessels. By connecting multiple routes together, it is possible to transport containers from one location to another without the establishment of a direct route. Each container will have a load port and a discharge port, which are the ports, where the container is loaded onto the vessel and where the container are destined to respectively.

In order to accommodate various goods, containers come in a range of sizes. The sizes are divided into standard measurements for containers, in order to alleviate planning of container placement aboard a vessel. For consistency however, this report only focuses on containers with the measurements denoted 20-foot and 40-foot, which are the most commonly used containers.

The stowage part of a vessel is divided into bays, dividing the ship in cross sections from

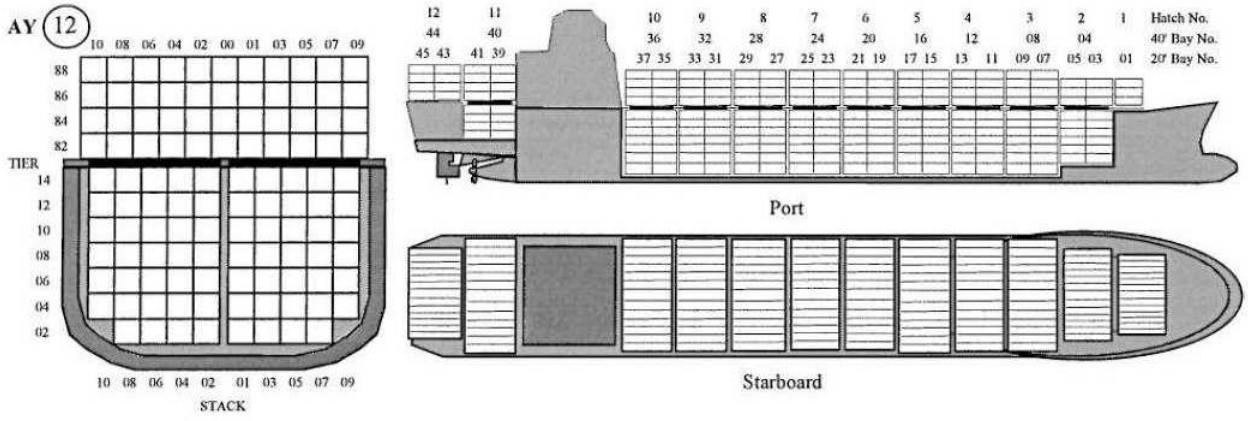


Figure 3.1: An overview of the layout of a ship. [7]

the stern to the bow. Containers are placed within each bay of a vessel according to a plan, referred to as the stowage plan. The containers are placed either below deck or above deck that is, inside the vessel or out in the open respectively. For later retrieval it is necessary to determine the exact location of a container. Several schemes exists in order to establish the position of each stowed container. The scheme, this report will use, is based on dividing the bays into slots fitting either one 40-foot container or two 20-foot containers. The slots within a bay is structured as a matrix, where each column is referred to as a stack and each row is referred to as a tier. Due to the shape of the vessel, some of the slots in the matrix are not allowed to hold any containers. Tiers are counted from the bottom of the matrix and up and stacks are counted from left to right. In order to identify two 20-foot container in a slot, a slot is further divided into two cells. As a consequence of stacking 20-foot container the term cell stack is introduced as cells in either one side of the stack or the other side of the stack.

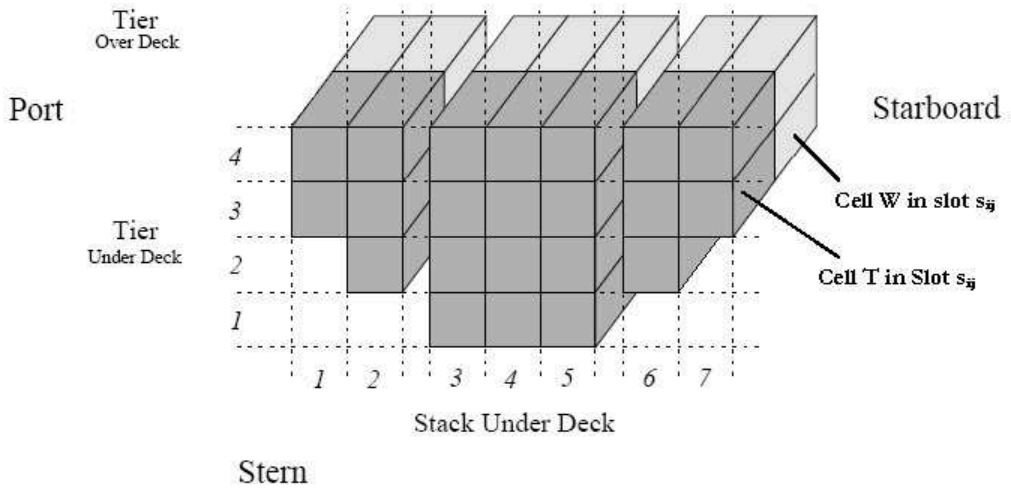


Figure 3.2: An overview of the cell layout of a bay below deck.

Different height of containers may cause the actual location of the container to not match the positioning system. However in order to ease identification of neighboring slots, this report will regard the neighboring slots as the slots, which are immediately adjacent to it according to the positioning system.

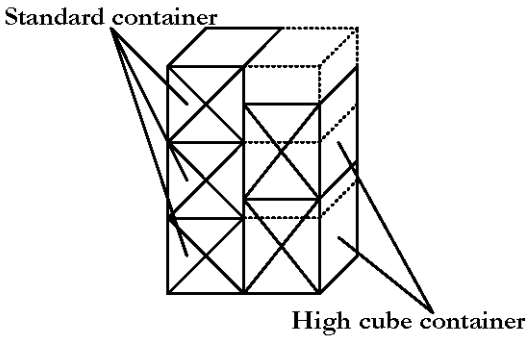


Figure 3.3: Models the positioning system for two neighbor stacks. The stack to the left has been filled with standard containers, while the stack to the right have been filled with high cube containers. The dotted lines represent the actual position of each container according to the positioning system.

Containers stowed above deck and below deck are physically separated by hatches on the deck of the ship. In order to sail safely, a number of safety precautions are given as rules for stowage of containers. As these rules vary from above and below deck, this report will focus only on containers stored below deck. The safety requirements are described in the following:

Due to the physical shape of the ship, a height and a weight restriction is put on the containers stowed in each stack. A maximum allowed height ensures that the containers stowed

below deck fits below the hatches on the deck. The maximum allowed weight of a stack ensures that the stress put on the hull of the ship by the stowed containers is within acceptable limits.

As the vessels travels overseas, the movement of the containers must be restricted. This is ensured by locking mechanisms attached to each corner of a container. Locking the containers in this manner restricts the placement of 20-foot containers such that they cannot be stowed on top of a 40-foot container.

Besides the physical properties of containers, goods have properties, which affect how containers can be arranged. Other properties, which will affect the arrangement of containers are the IMO level of a container and temperature requirements.

In order to be able to reduce damages from accidents, containers with hazardous goods such as fireworks, needs to be placed at a safe distance from other containers with hazardous goods. The IMO level is a description of how close container with certain goods can be placed next to each other. This mechanism simplifies the requirements for specialized knowledge of handling hazardous goods.

Perishable goods such as fruit or meat needs to remain at a consistent and low temperature in order to avoid decomposition. Therefore these types of goods will need to be placed in containers with temperature controlling devices. Containers of this kind are referred to as reefers. Power is necessary to make the temperature controlling device running, and therefore container can only be placed at designated areas with power supplying capabilities.

Given that containers may be placed according to the requirements above, many different stowage plans may still be possible. Selecting one of them is arbitrary if no preference has been defined. However, each valid stowage plan posses different qualities and may be preferred depending on defined objectives. One of the objectives usually defined by any company is profit maximization. For container transportation this can in principle be achieved either by increase the fee on transportation or reducing the cost of transporting containers. Due to the competition increasing the fee is not always a viable solution. Consequently companies are forced to look at the cost instead. The objectives for reducing the cost are defined as objectives for the storage area stowage problem and are mainly centered around arranging containers. The objectives are to minimize the following: Overstows, usage of stacks, wasted space and usage of reefer slots.

Cranes are necessary to unload or load containers and the cost of loading or unloading a container is calculated by a fee. An objective follows that containers, which are to be unloaded in the current port, is to be placed on top of each stack in order to avoid unnecessary container movement. Containers destined for future ports that are stacked on top of containers which are to be unloaded at the current port, is referred to as overstow containers. An overstow is inferred for each container stacked on top of another container with a smaller discharge port number.

As more containers are stowed within the same bay at future ports, it is of interest to keep as many stacks within a bay empty as possible. This will in turn provide freedom when stacking future containers to maximize optimization criteria.

The stowage of containers needs to be as compact as possible in order to transport as many containers as possible. If an arrangement of containers are placed, such that there is some space, which cannot be replaced by a container then that space is considered wasted.

Reefers can only be placed in designated areas, where power is being supplied. Placing non-reefer container in reefer slots may prevent a reefer container to be loaded onboard for some future port. Consequently as few reefer slots should be used to place non-reefer containers.

This section ends with a summary of the requirements for Storage Area Stowage Problem:

Physical requirements

Gravitation	Each container has to be supported either by the bottom of the deck or by containers.
Max Height	The total cellstack height cannot exceed the maximum cellstack height.
Max Weight	The total weight cannot exceed the maximum weight.
No 20-foot On Top	No 20-foot container can be on top of a 40-foot container.

Safety requirements

IMO	Each container are assigned an IMO level, and the rules is that two IMO-2 container have to be separated by a stack with no IMO-2. Each IMO-1 container cannot be adjacent to other container which is either IMO-1 or IMO-2.

Support requirements

Reefer	Each reefer has to be near a power supplying unit.

Objectives	
Overstow	Each overstow will be penalized.
Empty Stack	Each empty stack will be rewarded.
Wasted Space	The amount of wasted space is penalized.
Reefer slot	Each reefer slot which is occupied by a non-reefer container is penalized.

3.2 Formal definition of SASP

In the following, the requirements specified in the problem definition are translated into a formal definition of Storage Area Stowage Problem.

A *slot* is defined as the 40-foot stowage unit of a container vessel, uniquely identified by its bay, tier and stack position.

A *stack* denotes the slots of the same bay that have the same stack position. In each bay, stacks are counted from larboard to starboard, starting with 1.

Stack related notations $\alpha = (sc, tc_j, h_j, w_j)$		
sc	:	number of stacks
$tc_j \in \mathbb{N}$	:	number of tiers of stack j
$h_j \in \mathbb{R}^+$	:	height limit in foot of stack j
$w_j \in \mathbb{R}^+$	:	weight limit in kg of stack j
$J = \{1, \ldots, sc\}$	:	indexed set of stacks

A *storage area* denote lower-deck slots having belonging to the same bay. Consequently, each slot of a storage area is uniquely identified by its 2-dimensional position consisting of the tier position i counted bottom-up and its stack position j counted from left to right.

Since a slot may hold two 20-foot containers it is necessary to distinguish their positioning relative to the slot itself. A *cell* is defined as the part of a slot needed for a 20-foot container. The sides of a slot having place for two containers are referred to as the bow-side cell and the stern-side cell respectively. A slot, which can accommodate a single 20-foot container, is either placed on the bow-side or the stern-side depending on the ships physical structure. W denotes the bow-side of a cell. T denotes the stern-side of a cell. Let L denote set of cells possible for a slot.

Slot related notations $\beta = \{S, r_{i,j}, t^{20}_{i,j}, t^{40}_{i,j}, L_{i,j}\}$		
$s_{i,j}$	:	slot at stack j and tier i
$r_{i,j} \in \mathbb{B}$	:	$true$ if $s_{i,j}$ is a reefer
$t^{20}_{i,j} \in \mathbb{B}$	:	$true$ if $s_{i,j}$ can hold 20-foot containers
$t^{40}_{i,j} \in \mathbb{B}$	:	$true$ if $s_{i,j}$ can hold 40-foot containers
$L = \{W, T\}$	:	set of sides of a slot
$L_{i,j} \subseteq L$	:	set of cells that can be taken by 20-foot containers if $t^{20}_{i,j}$ is $true$ or by 40-foot containers if $t^{40}_{i,j}$ is $true$
$S = \{s_{i,j} : 1 \leq j \leq sc \wedge 1 \leq i \leq tc_j\}$	:	indexed set of slots

C^0 is the set of *containers already on board* before arriving to port 1, and remain on board after the vessel leaves port 1.

C^1 is the set of *containers to be loaded* into the storage area at port 1.

C denotes the entire set of containers, which will be onboard the ship when departing from port 1 i.e. $C = C^0 \cup C^1$.

P is the number of ports on the route the vessel sails.

Properties of container $c \in C$		
$dp_c \in \{1, \ldots, P\}$	:	discharge port number of container c
$w_c \in \mathbb{R}^+$	:	weight in kg of container c
$h_c \in \{8.5, 9.5\}$	:	height in feet of container c
$imo_c \in \{0, 1, 2\}$	:	IMO-level of container c
$l_c \in \{20, 40\}$	:	length in feet of container c
$r_c \in \mathbb{B}$	:	$true$ if container c is a reefer

Besides the properties defined above, on board containers specify their load port and their stowage position in the vessel.

Additional properties of a container $c \in C^0$		
$lp_c \in \{1, \ldots, P\}$	:	load port number of container c
$p_c \in S \times 2^L$	:	on board position of container c

For convenience, γ denotes the collection of container related properties:

$$\gamma = (dp_c, w_c, h_c, imo_c, l_c, r_c, lp_c, p_c)$$

An *assignment* or a *stowage plan* is an arrangement of containers within the vessel.

$A^0 : C^0 \rightarrow S \times 2^L$ defined by $A^0(c) = p_c$ is the stowage plan for containers on board.

$A^1 : C^1 \rightarrow S \times 2^L$ is the stowage plan for containers to be loaded at port 1.

Definition 3.1 (Assignment) *An assignment of containers in C is a total function $A : C \rightarrow S \times 2^L$*

$$A(c) = \begin{cases} A^0(c) & \textit{if } c \in C^0 \\ A^1(c) & \textit{if } c \in C^1 \end{cases}$$

The projections on slot and cell for a container c are:

$A_S : C \rightarrow S$ is the projection of A on S.

$A_L : C \rightarrow 2^L$ is the projection of A on 2^L.

Definition 3.2 (Storage Area Stowage Problem(SASP)) *The storage area stowage problem is a 5-tuple $(C^0, C^1, \alpha, \beta, \gamma)$.*

Example 3.1 Consider the stowage area shown in Figure 3.7, consisting of a single stack and a container on board.

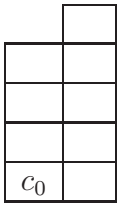


Figure 3.4: Stowage area with one stack.

Stack properties are $\alpha = (sc = 1, tc_1 = 5, h_1 = 43, w_1 = 14000)$
The set of slots is $S = \{s_{1,1}, s_{1,2}, s_{1,3}, s_{1,4}, s_{1,5}\}$
Slot properties are:

β	$t^{20}_{i,j}$	$t^{40}_{i,j}$	$L_{i,j}$	$r_{i,j}$
$s_{1,5}$	*true*	*false*	$\{W\}$	*false*
$s_{1,4}$	*true*	*true*	$\{W, T\}$	*false*
$s_{1,3}$	*true*	*true*	$\{W, T\}$	*true*
$s_{1,2}$	*true*	*false*	$\{W, T\}$	*true*
$s_{1,1}$	*true*	*false*	$\{W, T\}$	*true*

The set of containers on board is: $C^0 = \{c_0\}$
The set of containers to be loaded is: $C^1 = \{c_1, c_2, c_3, c_4\}$

Container properties are:

γ	dp_{c_i}	l_{c_i}	r_{c_i}	lp_{c_i}	p_{c_i}	w_{c_i}	h_{c_i}	imo_{c_i}
c_0	2	20	*true*	0	$(s_{1,1}, W)$	14000	8.5	0
c_1	2	20	*true*	-	-	1400	8.5	0
c_2	3	40	*false*	-	-	2800	8.5	0
c_3	4	20	*true*	-	-	1400	8.5	0
c_4	2	20	*false*	-	-	1400	8.5	0

The assignment for containers on board is $A^0(c_0) = p_{c_0}$

The SASP of this configuration is $(C^0, C^1, \alpha, \beta, \gamma, A^0)$

Before enumerating the constraints and objectives of the problem, some additional sets are constructed, that will ease the writing.

Container sets

$C_z = \{c \in C : l_c = z\}$	:	containers of length $z \in \{20, 40\}$
$C_{\text{IMO-z}} = \{c \in C : imo_c = z\}$	:	containers having IMO level $z \in \{0, 1, 2\}$
$C_j = \{c \in C : \exists i \,.\, 1 \leq i \leq tc_j \wedge A_S(c) = s_{i,j}\}$	:	containers assigned to stack j
$C_j^l = \{c \in C : c \in C_j \wedge l \in A_L(c)\}$	:	containers assigned to side l of stack j
$C_{i,j} = \{c \in C : A_S(c) = s_{i,j}\}$	:	containers assigned to $s_{i,j}$
$C_{nr} = \{c \in C : \neg r_c\}$	:	non reefer containers

Slot and Cell Sets

$S_{i,j}^{\text{IMO-1}} = \{s_{i-1,j}, s_{i+1,j}, s_{i,j-1}, s_{i,j+1}\}$	:	slots that cannot stow an IMO-1 container in case slot $s_{i,j}$ holds an IMO-1 container
$S_{i,j}^{\text{IMO-2}} = \{s_{k,j-1} : 1 \leq k \leq tc_{j-1}\} \cup \{s_{k,j+1} : 1 \leq k \leq tc_{j+1}\} \cup \{s_{k,j} : 1 \leq k \leq i \vee i < k \leq tc_j\}$	:	slots that cannot stow an IMO-2 container in case slot $s_{i,j}$ holds an IMO-2 container
$S^{nr} = \{A(c) : c \in C_{nr}\}$	:	cells storing non-reefer containers

Cell coverage

$T_{i,j} = \bigcup_{c \in C_{i,j}} A_L(c)$	:	cells covered by containers assigned to $s_{i,j}$
$o_{i,j} \Leftrightarrow L_{i,j} = T_{i,j}$	:	$o_{i,j}$ *true* if slot $s_{i,j}$ is fully occupied
$o_{i,j}^W \Leftrightarrow W \in T_{i,j}$	:	$o_{i,j}^W$ *true* if the bow side of slot $s_{i,j}$ is occupied
$o_{i,j}^T \Leftrightarrow T \in T_{i,j}$	:	$o_{i,j}^T$ *true* if the stern side of slot $s_{i,j}$ is occupied

Example 3.2 *if a 40-foot container has been placed at $s_{i,j}$ then $T_{i,j} = \{W, T\}$*

Constraints

CT1 All containers are assigned to a cell of a slot

$$\forall c \in C \,.\, A_S(c) = s_{i,j} \Rightarrow A_L(c) \subseteq L_{i,j}$$

CT2 A cell can hold at most 1 container

$$\forall c, c^{'} \in C \,.\, c \neq c^{'} \wedge A_S(c) = A_S(c^{'}) \Rightarrow A_L(c) \cap A_L(c^{'}) = \emptyset$$

CT3 A 40-foot container must cover both sides of a slot

$$\forall c \in C_{40} \,.\, |A_L(c)| = 2$$

CT4 A 20-foot container is allowed to cover one cell in a slot

$$\forall c \in C_{20} \,.\, |A_L(c)| = 1$$

CT5 Assigned slots above tier 1 must form stacks (gravity constraint)

$$\forall s_{i,j} \in S \,.\, o_{i,j} \wedge j > 1 \Rightarrow o_{i-1,j}$$
$$\forall s_{i,j} \in S \,.\, o^T_{i,j} \wedge j > 1 \Rightarrow o^T_{i-1,j}$$
$$\forall s_{i,j} \in S \,.\, o^W_{i,j} \wedge j > 1 \Rightarrow o^W_{i-1,j}$$

CT6 20-foot containers cannot be stacked on top of any 40-foot container

$$\forall c_{40} \in C_{40} \, \forall c_{20} \in C_{20} \,.\, A_S(c_{40}) = s_{i,j} \Rightarrow A_S(c_{20}) \neq s_{i+1,j}$$

CT7 The height of each cell stack is within its limits

$$\forall j \in J \forall l \in L \,.\, \textstyle\sum_{c \in C^l_j} h_c \leq h_j$$

CT8 The weight of each stack is within its limits

$$\forall j \in J \,.\, \textstyle\sum_{c \in C_j} w_c \leq w_j$$

CT9 Reefer containers must be placed in reefer slots

$$\forall c \in C \,.\, r_c \wedge A_S(c) = s_{i,j} \Rightarrow r_{i,j}$$

CT10 IMO rules are satsified for each container

$$\forall c \in C_{\text{IMO-1}} \cup C_{\text{IMO-2}} \, \forall c^{'} \in C_{\text{IMO-1}} \,.\, A_S(c) = s_{i,j} \Rightarrow A_S(c^{'}) \notin S^{\text{IMO-1}}_{i,j}$$
$$\forall c, c^{'} \in C_{\text{IMO-2}} \,.\, A_S(c) = s_{i,j} \Rightarrow A_S(c^{'}) \notin S^{\text{IMO-2}}_{i,j}$$

Objectives

OE1 Minimize overstows

There is a cost penalty of one unit for each container in a stack overstowing another container below it in the stack. The unit weight is W_{ov}.

There is an *overstow* between any two distinct containers in case they belong to the same cellstack and the discharge port of the container stowed at the lower tier is higher than the discharge port of the container stowed at the higher tier.

The binary relation $\prec$ on $S \times S$ defines whether two slots belong to the same stack and whether the first slot is located below the second slot:

$$s_{i,j} \prec s_{i',j'} \Leftrightarrow i < i' \wedge j = j'$$

$ov : C \times C \rightarrow \mathbb{B}$ defines if there is an overstow between two containers:

$$ov(c, c') \Leftrightarrow c \neq c' \wedge dp_c < dp_{c'} \wedge A_S(c) \prec A_S(c') \wedge A_L(c) \cap A_L(c') \neq \emptyset$$

Definition 3.3 (Overstow cost) $F_{ov}(A) = \left|\{(c, c') \in C \times C \,:\, ov(c, c')\}\right|$

OE2 Minimize the space wasted in a stack

The cost penalty is the length of wasted space. The unit weight is W_{ws}.

A stack consists of two cellstacks that do not necessarily have the same number of containers stacked into them. Therefore the two cellstacks can grow to different heights. The *wasted space* of the stack is defined as the sum of the wasted space of its two cellstacks. There is no wasted space in a cellstack, if there is enough space space to fit a standard container, otherwise the wasted space is the space left in the cellstack.

$fs : J \times L \rightarrow \mathbb{R}$ defines the available space on side l of stack j:

$$fs(j, l) = h_j - \sum_{c \in C_j^l} h_c$$

$ws : J \times L \rightarrow \mathbb{R}$ defines the wasted space on side l of stack j:

$$ws(j, l) = \begin{cases} 0 & \text{if } fs(j, l) \geq h_{st} \\ fs(j, l) & \text{if } fs(j, l) < h_{st} \end{cases}$$

Definition 3.4 (Wasted space cost) $F_{ws}(A) = \sum_{j \in J}(ws(j, T) + ws(j, W))$

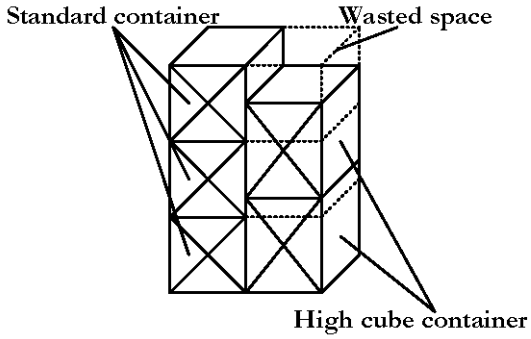


Figure 3.5: Skewed positioning system
Due to a skewed positioning of containers in the right stack, wasted space is introduced in the top in which no containers can be placed.

OE3 Avoid loading non-reefers into reefer slots

The cost penalty is one unit for each non-reefer container in a reefer slot. The unit weight is W_r.

Definition 3.5 (Reefer cost) $F_r(A) = \left|\{(s_{i,j}, l) \in S^{nr} \,:\, r_{i,j}\}\right|$

OE4 Avoid starting new stacks

The cost penalty is one unit per new stack used. The unit weight is W_{es}.

Definition 3.6 (Empty stack cost) $F_{es}(A) = \left|\{j \in J \,:\, |C_j| > 0\}\right|$

Definition 3.7 (Cost of an assignment) *The cost of a solution is the weighted sum of the costs defined for objectives (***OE1***) - (***OE4***):*

$$F(A) = W_{ov}F_{ov}(A) + W_{ws}F_{ws}(A) + W_rF_r(A) + W_{es}F_{es}(A)$$

Definition 3.8 (Valid Solution of the SASP) *A valid solution of SASP is an assignment of containers to be loaded in port 1 $A^1 : C^1 \rightarrow S \times 2^L$, such that the total assignment of containers $A : C \rightarrow S \times 2^L$ satisfies constraints (***CT1***)-(***CT10***).*

Definition 3.9 (Solution space) *The solution space for $SASP$ is the set $\mathcal{A}^1$ of all valid solutions.*

$$\mathcal{A}^1 = \{A^{1'} \,:\, A^{1'} is\ a\ valid\ solution\ of\ SASP\}$$

Definition 3.10 (Optimal Solution of the SASP) *An optimal solution A^{1*} of SASP is a valid solution $A^{1'}$ that minimizes the cost function $F(A)$.*

$$A^{1*} = \underset{A^{1'} \in \mathcal{A}^1}{\operatorname{argmin}} F(A)$$

Example 3.3 Consider SASP defined in Example 3.1:

$$\begin{aligned} A^1(c_1) &= \big(s_{1,1}, \{W\}\big) \\ A^1(c_2) &= \big(s_{1,3}, \{W, T\}\big) \\ A^1(c_3) &= \big(s_{1,2}, \{T\}\big) \\ A^1(c_4) &= \big(s_{1,2}, \{W\}\big) \end{aligned}$$

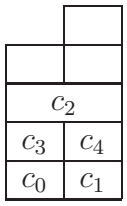


Figure 3.6: A valid solution with cost 1150.

$$\begin{aligned} A^1(c_1) &= \big(s_{1,2}, \{W\}\big) \\ A^1(c_2) &= \big(s_{1,3}, \{W, T\}\big) \\ A^1(c_3) &= \big(s_{1,1}, \{T\}\big) \\ A^1(c_4) &= \big(s_{1,2}, \{T\}\big) \end{aligned}$$

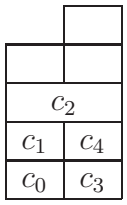


Figure 3.7: An optimal solution with cost 950.

Chapter 4

Evaluating CSP representations of SASP

Many variations exist on how to represent the Storage Area Stowage Problem as a constraint satisfaction problem. Three possibilities have been considered on how to model SASP as a CSP - namely *"Container as variables and slot as domain values"*, *"Slot as variables and container as domain values"* and *"Cell as variables and container halves as domain values"*. These suggestions will be referred to as *container-model*, *slot-model* and *cell-model* respectively. A presentation for each model is given with a brief description, followed by how domain values are pruned then pros and cons are outlined. After the presentation, a scoreboard follows with a conclusion of which model was chosen. In this chapter the initial step of how to translate the constraints into propagators is shown by identifying the pruning operations.

4.1 Pruning operations

Before outlining each model, the pruning operations, which are required to satisfy the constraints in SASP, are presented. The necessary pruning operations have been identified by analyzing each constraint and extracting the operations required. Table 4.1 shows the pruning operations identified. Table 4.2 illustrates, how each constraint is covered by some pruning operations. As the table shows, no pruning could be inferred from **CT1**, so this constraint must be implemented by other means.

PG1 - Uniqueness
Each container to be loaded are used from the same pool, which implies placed containers cannot be considered for another slot. Each cell can only be used once.

PG2 - Gravity
When placing a container sufficiently high in a stack, it is required that some containers are placed underneath it, in order to avoid it from falling to the bottom of the ship. The term support is introduced to state that a container is required in order to ensure that a placed

ID	Name
PG1	Uniqueness
PG2	Gravity
PG3	Reefer
PG4	Pick IMO-1 container
PG5	Pick IMO-2 container
PG6	Pick 20-foot container
PG7	Pick 40-foot container
PG8	Cover 40-foot container
PG9	Height
PG10	Weight

Table 4.1: Pruning operations

Constraint	Pruning
CT1	N/A
CT2	**PG1**
CT3	**PG8**
CT4	**PG1**
CT5	**PG2**
CT6	**PG6**, **PG7**
CT7	**PG9**
CT8	**PG10**
CT9	**PG3**
CT10	**PG4**, **PG5**

Table 4.2: Pruning coverage

container stays at its position.

PG3 - Reefer
A reefer container cannot be in a non-reefer slot. However, since it is given by the problem which slots have reefer capability this pruning can occur prior to the search.

PG4 - Pick IMO-1 container
According to the IMO constraint, neighboring slots is not allowed to accommodate IMO-1 container once an IMO-1 have been placed in a given slot.

PG5 - Pick IMO-2 container
According to the IMO constraint, slots in the current and neighboring stacks is not allowed to accommodate IMO-2 containers, once a given slot in a current stack is assigned with an IMO-2 container. Furthermore, neighboring slots to the given slot may not contain an IMO-1 container as well.

PG6- Pick 20-foot container
It should not be possible to place any 20-foot container on top of a 40-foot container. Therefore in the case when a 20-foot container has been placed no 40-foot container can be considered for any slots below it.

PG7 - Pick 40-foot container
A constraint states that it should not be possible to place any 20-foot container on top of a 40-foot container. Therefore, in the case where a 40-foot container has been placed, no 20-foot container can be considered for any slots above it.

PG8 - Cover 40-foot container
A 40-foot container must cover an entire slot.

PG9 - Height
In the problem definition, a height limitation constraint has been given. Since all stacks are divided into slots and only two different container heights available, pruning will not

occur until one empty slot in a given stack remains. For this reason, no pruning will occur based on the height constraint. An alternative mechanism needs to ensure that the height limitation is respected.

PG10 - Weight
The weight constraint states that each stack cannot exceed its weight limit w_j. The remaining weight, is the weight that can be added to stack j, before exceeding w_j no matter if some containers have been placed or not. The remaining number of slots, will be the number of slots, where nothing has been placed yet and the remaining available containers are the containers, which still needs to be placed within the bay. In the general case it will be that either all remaining available containers can be placed within stack j or some n lightest containers can be placed before exceeding the remaining weight. If n is less than the number remaining empty slots, then it can be inferred that only n slots can be filled up with the n lightest containers before exceeding the weight limitation. Therefore the rest of the slots cannot be assigned to any container. Since gravity rule requires that containers are supported, the bottom available slots have to be filled and the upper available slots can be left with nothing.

4.2 Container-model

Variables:	Container
Domain values:	Slot

Approach: The idea behind this model is to have containers represented as variables and then consider, which slot each container should be assigned to. Since this is the task of SASP this representation seems to be a natural choice for representing the CSP-model.

Pruning:

PG1 Slot $s_{i,j}$ can be pruned away as a candidate value, when it has been fully covered by containers. When the assigned container covering the slot is a 40-foot container, $s_{i,j}$ can be removed immediately. Assigning a 20-foot container, requires that $s_{i,j}$ is checked for whether it has been fully covered before being pruned away.

PG2 When placing containers it has to be ensured that there are enough containers to support it. When the number of containers needed to support some other container is the same as the number of available containers to be placed, then slots, which do not have any placed containers above the picked slot, can be pruned away as candidate values from all the available containers to be placed.

PG3 Each reefer container can only be placed in a reefer slot, while a non-reefer container can be placed in either a reefer or a non-reefer slot.

PG4 Picking some slot $s_{i,j}$ for an IMO-1 container prunes any slot according to the IMO constraint for any IMO-1 container.

PG5 Picking some slot $s_{i,j}$ for an IMO-2 container prunes any slot according to the IMO constraint for any IMO-1 and IMO-2 container.

PG6 Picking some slot $s_{i,j}$ for a 20-foot container prunes any slot in stack j below tier i as candidate values for any 40-foot containers.

PG7 Picking some slot $s_{i,j}$ for a 40-foot container prunes any slot in stack j above tier i as candidate values for any 20-foot containers.

PG8 40-foot container has the same dimension as a slot, it is therefore ensured by the model that a 40-foot container fully covers a slot.

PG9 As described previously this pruning operation will not be considered.

PG10 When only the n lightest containers can fit in a stack j. All available containers, which are not among the n lightest containers needs to get slots pruned away. The slots, which are required to be pruned away, are those described in **PG10** in section 4.1.

Advantages:

- Since the search goes through all containers, it is ensured by the model that every container will be assigned.
- Reefers can be pruned prior to search
- Do not need to prune anything for **PG8**.

Disadvantages:

- The gravity constraint is difficult to ensure, since this model relies on forcing some containers to pick specific cells.
- Placed container may potentially affect where all other containers can be placed. For instance placing an IMO-1 container will affect where all other IMO-1 containers can be placed.
- The number of variables, which will be affected by **PG10** are all unassigned variables.

The following model takes the reverse of the previous approach by looking at the stowage area and examines what can be fitted into each slot. Since there cannot be more containers than slots available, some slots are assigned but left empty. An *air* value has been introduced to denote that a slot remains empty.

4.3 Slot-model

Variables: Slot
Domain values: Containers

Approach: This model uses the slots as variables and containers as domain values. In example, for each slot, one can chose which container it should accommodate. Since a slot can accommodate a 40-foot container and some containers may be 20-foot long, placing two 20-foot containers within a slot poses an issue. One approach is to construct pairs of 20-foot containers, which will result in $|C_{20}|^2$ of such combinations. In addition, one 20-foot containers may be placed in a slot alone, leaving half of the slot empty. Furthermore, special slots exists, which can only hold a single 20-foot container.

Pruning:

PG1 When a container c has been used, it needs to be pruned away as a possibility from all other unassigned slots. If c is a 20-foot container, then all domain values, in which c appears, has to be pruned away as a candidate value as well.

PG2 When a container is placed in a slot, all slots underneath it cannot select the introduced air value for assignment. That is, when a 40-foot container is placed in slot $s_{i,j}$, the air value is pruned away from the domain of any variable positioned in the same stack beneath $s_{i,j}$.

PG3 Each reefer container can only be placed in a reefer slot, while any non-reefer container can be placed in either a reefer or a non-reefer cell.

PG4 Picking some IMO-1 container for slot $s_{i,j}$ prunes any IMO-1 container as candidate value for slots according to the IMO constraint.

PG5 Picking some IMO-2 container for slot $s_{i,j}$ prunes any IMO-1 and IMO-2 containers as possible candidate values according to the IMO constraint.

PG6 Picking some 20-foot container for slot $s_{i,j}$ prunes any 40-foot container as candidate value for slots in stack j below tier i.

PG7 Picking some 40-foot container for slot $s_{i,j}$ prunes any 20-foot containers as candidate value for slots in stack j above tier i.

PG8 40-foot container has the same dimension as a slot, it is therefore ensured by the model that a 40-foot container fully covers a slot.

PG9 As described previously this pruning will not be considered.

PG10 Each stack j will only be able to accommodate the n lightest containers before exceeding the weight limitation. All other containers to be placed can be pruned away as candidate values from any slots in j. Furthermore if the number of slots available in the stack exceeds n it can be inferred that all but the lowest n slots will have to accommodate air, since any containers above tier 1 needs to be supported.

Advantages:

- The search can be done such that the slots are filled in a bottom up approach, thereby respecting the gravity constraint.
- Reefers can be pruned prior to search.
- Simple to reason about containers placed in a stack.
- Do not need to prune anything for **PG8**.
- The number of variables, which will be affected by **PG10**, are limited to only one stack when using this model, as opposed to the container-model, where all variables are affected.

Disadvantages:

- Since any slot initially can pick air as a candidate value, not all containers may be placed within the stowage area. This has to be ensured by introducing additional propagator.
- Implacable IMO-1/IMO-2 containers are potentially discovered late.

4.4 Cell-model

Variables: Cells
Domain values: Containers

Approach: The drawback of using the slot-model, is the number of domain values. To address this issue, the following model is introduced, which avoids the combination of 20-foot containers by using cells as variables. By having cells as variables one can fit exactly a 20-foot container. However, 40-foot containers will not fit within a cell. This issue can be handled by splitting 40-foot containers into matching halves. For convenience, when a container half is mentioned in this section, it refers to both a 20-foot container or a 40-foot half container.

Pruning:

PG1 Picking container c for slot $s_{i,j}$ prunes c from any other slot.

PG2 When a container is placed in a cell, all cells underneath it can not select the introduced air value for assignment. That is, when a 40-foot container is placed in slot $s_{i,j}$, the air value is pruned away from the domain of any variable positioned in the same stack beneath $s_{i,j}$.

PG3 Each reefer container can only be placed in a reefer cell, while a non-reefer container can be placed in either a reefer or a non-reefer cell.

PG4 Picking some IMO-1 container cell l in slot $s_{i,j}$ prunes any IMO-1 container in slots according to the IMO constraint.

PG5 Picking some IMO-2 container cell l in slot $s_{i,j}$ prunes any IMO-1 and IMO-2 container in slots according to the IMO constraint.

PG6 Picking some 20-foot container for cell l in slot $s_{i,j}$ prunes away any 40-foot containers in cells below tier i in stack j and on the same side as cell l.

PG7 Picking some 40-foot container for cell l in slot $s_{i,j}$ prunes away any 20-foot containers in cells below tier i in stack j and on the same side as cell l.

PG8 40-foot container needs to be cut in half to fit a cell, therefore ensuring that the two halves are placed next to each other is required. This can be achieved by pruning all domain values except the other half from the domain of the neighbor cell.

PG9 As described previously this pruning will not be considered.

PG10 Each stack j will only be able to accommodate the n lightest containers before exceeding the weight limitation. The containers to be placed, can be pruned away as candidate values from any cells in j, which only can accommodate air.

Advantages:

- The search can be done such that each stack is filled bottom up, thereby respecting the gravity constraint in a natural way.
- Maintaining 20-foot container pairs is not needed, which results in a narrower search tree than the slot-model, due to smaller domains.
- Non-reefer cells can have their initial domains pruned to only select the non-reefer container halves.
- The number of variables, which will be affected by **PG10**, are limited to only one stack when using this model, as opposed to the container-model, where all variables are affected.

- Simple to reason about containers placed in a stack.
- Do not need to prune anything for **PG8**.

Disadvantages:

- Number of variable is doubled, compared to the slot-model, which results in a deeper search tree.
- Additional constraints needs to be added:
  - 40-foot container half has to be placed next to its other half.
  - 20-foot containers cannot be placed next to 40-foot container halves.

4.5 Conclusion

The variable and domain sizes are summarized in table 4.3. Assuming that there are sufficient slots for the containers, the table shows that the cell-model has the most variables and thus results in the deepest search tree. The container-model will have the lowest number of variables and therefore have the shallowest search tree. The domain sizes shows that slot-model has the largest domain size and therefore also provides the widest search tree. The cell-model result in the narrowest search tree due to the domain size.

	Container-model	Slot-model	Cell-model
Variables	$\|C\|$	$\|S\|$	$\|S\|\|L\|$
Domains	$\|S\|$	$\|C_{40}\| + \|C_{20}\|^2 + 2\|C_{20}\|$	$\|C\|$

Table 4.3: space complexities for the given elements in the future application

Table 4.4 shows how many variables are affected when pruning based on a constraint is carried out. Let $tc = \max_{j \in \{1,\ldots,sc\}}\{tc_j\}$ denote the number of tiers for the stack in the bay, which has the most tiers. As the table shows, the container-model will depend on the number of containers when pruning. For the other two models the amount of pruning is mainly dependent on the stack size. Since it is expected that the number of slots which appears in a stack is significantly less than the number of containers, it is expected that either the slot-model or the cell-model is affecting less variables than the container-model. The number of domain values in the slot-model is significantly higher than in the cell-model. Based on the above observation, the model chosen is the cell-model.

	Container-model	Slot-model	Cell-model
PG1 - Uniqueness	$\|C\|$	$\|S\|$	$\|S\|\|L\|$
PG2 - Gravity	$\|C\|$	$\|tc\|$	$\|tc\|$
PG3 - Reefer	-	-	-
PG4 - Pick IMO-1 container	$\|C\|$	4	9
PG5 - Pick IMO-2 container	$\|C\|$	$3\|tc\|$	$3\|L\|\|tc\|$
PG6 - Pick 20-foot container	$\|C\|$	$\|tc\|$	$\|L\|\|tc\|$
PG7 - Pick 40-foot container	$\|C\|$	$\|tc\|$	$\|L\|\|tc\|$
PG8 - Cover 40-foot container	-	-	1
PG9 - Height	-	-	-
PG10 - Weight	$\|C\|$	$\|tc\|$	$\|tc\|$

Table 4.4: Shows the maximum number of affected variables when performing different pruning operations in the three different models.

Chapter 5

CSP representation of SASP

Based on the analysis for selecting a proper representation of SASP, the CSP-Model needs to be detailed further. This chapter presents the CSP-Model in terms of variables, domains and propagators. It is shown how pruning operations can be transformed into propagators.

5.1 Variables

Each variable in the CSP model corresponds to a cell as defined in the SASP. The set of variables is:

$$X = \{x^l_{i,j} \,:\, s_{i,j} \in S \wedge l \in L_{i,j}\}$$

Table 5.1: Model specific variable sets

$X^{\text{IMO-z}}_{i,j} = \{x^l_{i,j} \in X \,:\, s_{i,j} \in S^{\text{IMO-z}}_{i,j}\}$	:	variables that cannot stow an IMO level z, when an IMO level z has been placed in $s_{i,j}$
$X^R = \{x^l_{i,j} \in X \setminus \mathcal{S} \,:\, \perp \notin \mathcal{D}(x^l_{i,j})\}$	:	all unassigned variables, which cannot accommodate air

For convenience, a neighboring operation is defined on the set of cells L, to denote the other side of a cell within a slot:

$$\overline{W} = T,\ \overline{T} = W.$$

5.2 Domains

Since variables are cells, a variable cannot be assigned to a 40-foot container. Consequently, 40-foot containers are divided into two halves, each half maintaining the properties of the

original 40-foot container. To identify the two halves that make up an original 40-foot container, the halves are given the same unique identifier. In this model, the *container* term is used both for 20-foot containers and 40-foot container halves. A 40-foot container half is marked similarly to a cell, as being bow or stern.

Model specific container sets

$C^H_{40} = \{c^T, c^W \,:\, c \in C_{40}\}$	:	40-foot container halves
$C^H_{nr} = \{c^\lambda \in C^H_{40}, c \in C_{20} \,:\, \neg r_c\}$	:	non-reefer 40-foot halves and 20-foot containers
$C^H_{r} = \{c^\lambda \in C^H_{40}, c \in C_{20} \,:\, r_c\}$	:	reefer 40-foot halves and 20-foot containers
$C^H_{\text{IMO-z}} = \{c^\lambda \in C^H_{40}, c \in C_{20} \,:\, imo_c = z\}$	:	40-foot halves and 20-foot containers with IMO-z

The neighboring operation on the set of cells still holds. That is, the corresponding half of a 40-foot half c^λ is $c^{\overline{\lambda}}$.

In case the bay has more cells than containers, some of the cells will remain empty. Let $\perp$ denote the domain value that indicates that a cell is left empty. The "air" term is used as a synonym for $\perp$. The properties for $\perp$ are:

$$h_\perp = 0,\ w_\perp = 0,\ l_\perp = 0,\ imo_\perp = 0,\ r_\perp = \mathit{false},\ lp_\perp = 0 \text{ and } dp_\perp = 0.$$

The domain of a variable consists of the containers the cell can accommodate. Some slots can stow a single 20-foot container. Therefore 40-foot containers are excluded from the domain of the cells belonging to these slots. Reefer containers can only be placed into reefer slots and therefore reefer containers are excluded from the domain of non-reefer cells.

Let $\mathcal{D}(x^l_{i,j})$ be the initial domain for each variable $x^l_{i,j}$.

$$\mathcal{D}(x^l_{i,j}) = \begin{cases} C^H_{40} \cup \{\perp\} & : \ \neg t^{20}_{i,j} \wedge t^{40}_{i,j} \wedge r_{i,j} \\ (C^H_{40} \cap C^H_{nr}) \cup \{\perp\} & : \ \neg t^{20}_{i,j} \wedge t^{40}_{i,j} \wedge \neg r_{i,j} \\ C_{20} \cup \{\perp\} & : \ t^{20}_{i,j} \wedge \neg t^{40}_{i,j} \wedge r_{i,j} \\ (C_{20} \cap C^H_{nr}) \cup \{\perp\} & : \ t^{20}_{i,j} \wedge \neg t^{40}_{i,j} \wedge \neg r_{i,j} \\ C^H_{40} \cup C_{20} \cup \{\perp\} & : \ t^{20}_{i,j} \wedge t^{40}_{i,j} \wedge r_{i,j} \\ ((C^H_{40} \cup C_{20}) \cap C^H_{nr}) \cup \{\perp\} & : \ t^{20}_{i,j} \wedge t^{40}_{i,j} \wedge \neg r_{i,j} \\ \{c\} & : \ \exists c \in C_{20} \,.\, A^0_S(c) = s_{i,j} \wedge l \in A^0_L(c) \\ \{c^\lambda\} & : \ l = \lambda \wedge \exists c^\lambda \in C^H_{40} \,.\, A^0_S(c) = s_{i,j} \wedge l \in A^0_L(c) \\ \emptyset & : \ \neg t^{20}_{i,j} \wedge \neg t^{40}_{i,j} \end{cases}$$

Example 5.1 Consider the SASP defined in Example 3.1.
The set of variables is:

$$X = \{x^W_{1,1}, x^T_{1,1}, x^W_{1,2}, x^T_{1,2}, x^W_{1,3}, x^T_{1,3}, x^W_{1,4}, x^T_{1,4}, x^W_{1,5}\}$$

The set of 20-foot containers is:

$$C_{20} = \{c_0, c_1, c_3, c_4\}$$

The set of 40-foot container halves is:

$$C_{40}^H = \{c_2^W, c_2^T\}$$

The initial domains are :

$$\begin{array}{l}
\mathcal{D}(x_{1,1}^W) = \{c_0\} \\
\mathcal{D}(x_{1,1}^T) = \{c_1, c_3, c_4, \bot\} \\
\mathcal{D}(x_{1,2}^W) = \{c_1, c_3, c_4, \bot\} \\
\mathcal{D}(x_{1,2}^T) = \{c_1, c_3, c_4, \bot\} \\
\mathcal{D}(x_{1,3}^W) = \{c_1, c_2^W, c_2^T, c_3, c_4, \bot\} \\
\mathcal{D}(x_{1,3}^T) = \{c_1, c_2^W, c_2^T, c_3, c_4, \bot\} \\
\mathcal{D}(x_{1,4}^W) = \{c_2^W, c_2^T, c_4, \bot\} \\
\mathcal{D}(x_{1,5}^T) = \{c_4, \bot\}
\end{array}$$

5.3 Additional constraints and pruning operations

Besides the constraints given in SASP, this model introduces three additional constraints:

CT11 The two halves of a 40-foot container must be placed in the same slot

$$\forall c \in C_{40} \,.\, x_{i,j}^l = c^\lambda \Rightarrow x_{i,j}^{\overline{l}} = c^{\overline{\lambda}}$$

CT12 A cell that accommodates a 20-foot container excludes the possibility of its neighbor cell to accommodate a 40-foot half

$$\forall x_{i,j}^l, x_{i,j}^{\overline{l}} \in X \,.\, x_{i,j}^l = c \wedge c \in C_{20} \wedge x_{i,j}^{\overline{l}} = c' \Rightarrow c' \notin C_{40}^H$$

CT13 Allowing each container only to appear once

$$\forall x_{i,j}^l, x_{i,j}^{\overline{l}} \in X \,.\, x_{i,j}^l \neq x_{i,j}^{\overline{l}} \wedge x_{i,j}^l = c \wedge x_{i,j}^{\overline{l}} = c' \Rightarrow c \neq c'$$

In addition the following pruning operation is defined:

Constraint	Pruning
CT11	**PG8**
CT12	**PG11**
CT13	**PG1**

Table 5.2: Pruning coverage for additional constraints

PG11 - Placing a 20-foot container excludes any 40-foot container
By definition a slot can typically accommodate either two 20-foot containers or a single 40-foot container. Placing a 20-foot container excludes any 40-foot container to be placed in that slot.

5.4 Propagators

As defined in section 2.1 propagators remove values in conflict with constraints. Table 5.3 presents the propagators derived by analyzing the pruning operations described in section 4.1.

ID	Name
PR1	Uniqueness
PR2	Gravity
PR3	Air
PR4	IMO-1
PR5	IMO-2
PR6	No 20-foot container on top
PR7	No 40-foot container below
PR8	Correct halves
PR9	Overfitting
PR10	Forced air due to space
PR11	Forced air due to weight

Table 5.3: Propagators

Pruning	Propagators
PG1	**PR1**
PG2	**PR2**, **PR3**
PG3	N/A
PG4	**PR4**
PG5	**PR5**
PG6	**PR7**
PG7	**PR6**
PG8	**PR8**
PG9	N/A
PG10	**PR11**
PG11	**PR9**

Table 5.4: Coverage

PR1 *Uniqueness*: a container cannot be assigned to more than one variable at a time

$$\mathcal{P}^u(\mathcal{D})(x) = \begin{cases} \{n \in \mathcal{D}(x) \,:\, n \neq x^l_{i,j}\} & x \in X \setminus \{x^l_{i,j}\} \\ \mathcal{D}(x^l_{k,j}) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^u} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^u} = X \setminus \{x^l_{i,j}\}$$

PR2 *Gravity*: the cells below a cell that stows a container cannot be left empty

$$\mathcal{P}^g(\mathcal{D})(x^l_{k,j}) = \begin{cases} \{n \in \mathcal{D}(x^l_{k,j}) \,:\, x^l_{i,j} \neq \perp \Rightarrow n \neq \perp\} & \text{if } k < i \\ \mathcal{D}(x^l_{k,j}) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^g} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^g} = \{x^l_{k,j} \,:\, k < i\}$$

PR3 *Air*: the cells above a cell that is empty cannot stow a container

$$\mathcal{P}^a(\mathcal{D})(x^l_{k,j}) = \begin{cases} \{n \in \mathcal{D}(x^l_{k,j}) \,:\, x^l_{i,j} = \perp \Rightarrow n = \perp\} & \text{if } k > i \\ \mathcal{D}(x^l_{k,j}) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^a} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^a} = \{x^l_{k,j} \,:\, k > i\}$$

PR4 *IMO-1*: a cell stowing an IMO-1 container restricts the stowage of IMO-1 containers according to the IMO constraint

$$\mathcal{P}^{\text{IMO-1}}(\mathcal{D})(x) = \begin{cases} \{n \in \mathcal{D}(x) \,:\, imo_{x^l_{i,j}} = 1 \Rightarrow n \notin C^H_{\text{IMO-1}}\} & \text{for all } x \in X^{\text{IMO-1}}_{i,j} \\ \mathcal{D}(x) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^{\text{IMO-1}}} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^{\text{IMO-1}}} = X^{\text{IMO-1}}_{i,j}$$

PR5 *IMO-2*: a cell stowing an IMO-2 container restricts the stowage of IMO-1 and IMO-2 containers according to the IMO constraint

$$\mathcal{P}^{\text{IMO-2}}(\mathcal{D})(x) = \begin{cases} \{n \in \mathcal{D}(x) \,:\, imo_{x^l_{i,j}} = 2 \Rightarrow n \notin C^H_{\text{IMO-1}}\} & \text{for all } x \in X^{\text{IMO-1}}_{i,j} \\ \{n \in \mathcal{D}(x) \,:\, imo_{x^l_{i,j}} = 2 \Rightarrow n \notin C^H_{\text{IMO-2}}\} & \text{for all } x \in X^{\text{IMO-2}}_{i,j} \\ \mathcal{D}(x) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^{\text{IMO-2}}} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^{\text{IMO-2}}} = X^{\text{IMO-1}}_{i,j} \cup X^{\text{IMO-2}}_{i,j}$$

PR6 *No 20-foot container on top* : the cells above a cell stowing a 40-foot half cannot stow 20-foot containers.

$$\mathcal{P}^{\text{40-20}}(\mathcal{D})(x^y_{k,j}) = \begin{cases} \{n \in \mathcal{D}(x^y_{k,j}) \,:\, l_{x^l_{i,j}} = 40 \Rightarrow n \notin C_{20}\} & \text{for all } k > i \\ \mathcal{D}(x^y_{k,j}) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^{\text{40-20}}} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^{\text{40-20}}} = \{x^y_{k,j} \,:\, k > i\}$$

PR7 *No 40-foot container below*: the cells below a cell stowing a 20-foot container cannot stow 40-foot halves.

$$\mathcal{P}^{\text{20-40}}(\mathcal{D})(x^y_{k,j}) = \begin{cases} \{n \in \mathcal{D}(x^y_{k,j}) \,:\, l_{x^l_{i,j}} = 20 \;\Rightarrow n \notin C^H_{40}\} & \text{for all } k < i \\ \mathcal{D}(x^y_{k,j}) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^{\text{20-40}}} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^{\text{20-40}}} = \{x^y_{k,j} \,:\, k > i\}$$

PR8 *Correct Halves*: the halves of a 40-foot container must be placed next to each other.

$$P^{\text{40-40}}(\mathcal{D})(x) = \begin{cases} \{n \in \mathcal{D}(x) \,:\, l_{x^l_{i,j}} = 40 \wedge x^l_{i,j} = c^z \Rightarrow n = c^{\overline{z}}\} & x \in \{x^{\overline{l}}_{i,j}\} \\ \{n \in \mathcal{D}(x) \,:\, l_{x^l_{i,j}} = 40 \wedge x^l_{i,j} = c^z \Rightarrow n \neq c^{\overline{z}}\} & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^{\text{40-40}}} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^{\text{40-40}}} = X$$

PR9 *Overfitting*: a 20-foot container placed in a cell excludes the possibility of its neighbor cell to accommodate a 40-foot half.

$$\mathcal{P}^{\text{20-20}}(\mathcal{D})(x) = \begin{cases} \{n \in \mathcal{D}(x) \,:\, l_{x^l_{i,j}} = 20 \Rightarrow n \in C_{20} \cup \{\bot\}\} & x \in \{x^{\overline{l}}_{i,j}\} \\ \mathcal{D}(x) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^{\text{20-20}}} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^{\text{20-20}}} = \{x^{\overline{l}}_{i,j}\}$$

The above defined propagators are sufficient for representing the constraints of the problem. However, additional propagators have been introduced in order to enhance pruning and discover deadends and solutions earlier.

PR10 *Forced air due to space*: if the number of containers, not assigned to a cell, is equal to the number of cells that must hold a container, the rest of the cells are left empty.

$X^A = \{x^l_{i,j} \in X \setminus \mathcal{S} \,:\, \bot \in \mathcal{D}(x^l_{i,j})\}$ is the set of all unassigned variables, which can accommodate air.

$$\mathcal{P}^{\text{a-s}}(\mathcal{D})(x) = \begin{cases} \{n \in \mathcal{D}(x) \,:\, |C^H_{40} \cup C_{20} \setminus \pi_{\mathcal{S}}(\vec{a})| = |X^R| \Rightarrow n = \bot\} & x \in X^A \\ \mathcal{D}(x) & \text{otherwise} \end{cases}$$

$$\mathcal{I}_{\mathcal{P}^{\text{a-s}}} = \{x^l_{i,j}\}$$

$$\mathcal{O}_{\mathcal{P}^{\text{a-s}}} = X^A$$

PR11 *Forced air due to weight*: As described in **PG10** in section 4.1, cells that cannot accommodate any container should have its domain pruned to only contain air. In order to resolve the cells, which cannot accommodate any container, following notation is introduced:

I_j^{20} is the index set of variables of stack j assigned to a 20-foot container:

$$I_j^{20} = \{(i,l) \,:\, x_{i,j}^l = c \wedge c \in C_{20}\}$$

I_j^{40} is the index set of variables of stack j assigned to a 40-foot half:

$$I_j^{40} = \{(i,l) \,:\, x_{i,j}^l = c^\lambda \wedge c^\lambda \in C_{40}^H\}$$

W_j is the current weight of stack j:

$$W_j = \sum_{(i,l)\in I_j^{20}} w_{x_{i,j}^l} + \frac{1}{2} \sum_{(i,l)\in I_j^{40}} w_{x_{i,j}^l}$$

n is the number of available containers:

$$n = |C_{40}^H \cup C_{20} \setminus \pi_{\mathcal{S}}(\vec{a})|$$

Φ is the sequence of available containers ordered by weight:

$$\Phi = \langle c_1, c_2, \ldots, c_n \rangle, c_i \in C_{40}^H \cup C_{20} \setminus \pi_{\mathcal{S}}(\vec{a}) \text{ and } w_{c_1} \leq w_{c_2} \leq \ldots \leq w_{c_n}$$

m_j is the number of cells in stack j that can be filled without weight excess:

$$m_j = \operatorname*{argmax}_{k\in 1\ldots n} \sum_{z=1}^{k} w_{\Phi_z} \leq (w_j - W_j)$$

X_j^l is the set of unassigned cells in stacks j side l above tier m_j:

$$X_j^l = \{x_{i,j}^l \,:\, x_{i,j}^l \in X \setminus \mathcal{S} \wedge i \geq m_j\}, l \in L$$

I_j^l is the index set of unassigned variables of stack j side l:

$$I_j^l = \{i \,:\, x_{i,j}^l \in X \setminus \mathcal{S}\}, l \in L$$

$$\mathcal{P}^{a\text{-}w}(\mathcal{D})(x) = \begin{cases} \{n \in \mathcal{D}(x) \,:\, m_j < |I_j^l| \Rightarrow n = \perp\} & x \in X_j^l \\ \mathcal{D}(x) & \text{otherwise} \end{cases}$$

$\mathcal{I}_{\mathcal{P}^{a\text{-}w}} = \{x_{i,j}^l\}$

$O_{\mathcal{P}^{a\text{-}w}} = X_j^l$

5.5 Early termination criteria

Some requirements cannot be discovered by the propagators e.g. if the number of containers to be loaded exceeds the available space. Other criteria do not prune away enough values or happen so late that it does not pay off to prune e.g. the height limit. Early termination criteria are introduced to represent constraints that are not modeled by propagators. A termination criterion is a boolean function that indicates whether a specific constraint is satisfied. In case the early termination criterion is true a dead end has been discovered, and the search cannot continue along current search path. An early termination criterion is denoted as $\mathcal{E}^{\alpha}$ where α stands for the name of the termination criterion.

ETC1 The height of a stack has exceeded its limit.

$$\forall j \,.\, \mathcal{E}^{H}(j) \Leftrightarrow \sum_{z=1}^{tc_j} h_{x^l_{z,j}} > h_j$$

ETC2 The weight of a stack has exceeded its limit.

$$\forall j \,.\, \mathcal{E}^{W}(j) \Leftrightarrow \sum_{z=1}^{tc_j} \sum_{l \in L} w_{x^l_{z,j}} > w_j$$

ETC3 Too many containers: all containers cannot be placed within the space available.

$$\mathcal{E}^{XC} \Leftrightarrow |C^H_{40} \cup C_{20} \setminus \pi_{\mathcal{S}}(\vec{a})| > |X \setminus \mathcal{S}|$$

ETC4 Too many reefers: all reefers cannot be placed within the available reefer cells.

$$\mathcal{E}^{XR} \Leftrightarrow |C^H_r \setminus \pi_{\mathcal{S}}(\vec{a})| > |\{x^l_{i,j} \in X \,:\, r_{i,j}\} \setminus \mathcal{S}|$$

ETC5 Too few containers: not enough containers to support the containers already placed.

$$\mathcal{E}^{FA} \Leftrightarrow |C^H_r \setminus \pi_{\mathcal{S}}(\vec{a})| < |X^R|$$

5.6 Correctness of propagators

Propagators and early termination criteria may not have a one to one relationship with constraints in the formal model. In some cases, several propagators or a combination among propagators and early termination criteria implement a single constraint. This section establish correspondence between propagators/early termination criteria and constraints from the formal model. The correctness is shown through series of argumentation.

CT1 All containers are assigned to a cell of a slot

The constraint is represented by $\mathcal{E}^{XC}$.

$\mathcal{E}^{XC}$ guarantees that a complete instantiation has all containers assigned to a cell. The proof is by contradiction: assume that the search can return a complete instantiation where not all containers are assigned. This contradicts the definition of $\mathcal{E}^{XC}$, which stops search if the number of available cells is less than the number of containers to be assigned.

CT2 A cell can hold at most 1 container

Since variables are cells and each variable can be assigned to one value at a time, the constraint is implicitly represented by the model itself.

CT3 A 40-foot container must cover both sides of a slot

The constraint is represented by $\mathcal{P}^{\text{40-40}}$.

When a 40-foot container is assigned to a cell, the domain of the neighbor cell is pruned to contain only the other half and all other variables will have the other half pruned away from their domain.

CT4 A 20-foot container is allowed to cover one cell in a slot

The constraint is represented by $\mathcal{P}^{u}$.

Whenever a container c is assigned to a cell $x^{l}_{i,j}$, $\mathcal{P}^{u}$ prunes away c as a candidate value from all other cells than $x^{l}_{i,j}$, which ensures that c can never be chosen again for any other cell.

CT5 Assigned slots must form stacks

The constraint is represented by $\mathcal{P}^{g}$ and $\mathcal{P}^{a}$.

Whenever a container c is assigned to a cell $x^{l}_{i,j}$, $\mathcal{P}^{g}$ removes air as candidate value from all cells beneath $x^{l}_{i,j}$. Therefore, air cannot be chosen beneath a cell that accommodates a container.

Whenever air is assigned to a cell $x^{l}_{i,j}$, $\mathcal{P}^{a}$ prunes away all other values than air from any cells above $x^{l}_{i,j}$. Therefore, a container can never be placed above some air.

The combination of $\mathcal{P}^{g}$ and $\mathcal{P}^{a}$ ensures that there will never be air between two containers in a stack.

CT6 20-foot containers cannot be stacked on top of any 40-foot container

The constraint is represented by $\mathcal{P}^{\text{40-20}}$ and $\mathcal{P}^{\text{20-40}}$.

For this constraint it will only matter if a cell is assigned a 40-foot container or a 20-foot container.

In case a 40-foot container is assigned to a cell $x^{l}_{i,j}$, $\mathcal{P}^{\text{40-20}}$ prunes away 20-foot containers from the domain of all cells above $x^{l}_{i,j}$, ensuring that 20-foot containers can never be placed on top of a 40-foot container.

In case a 20-foot container is assigned to a cell $x^l_{i,j}$, $\mathcal{P}^{\text{20-40}}$ prunes away 40-foot containers from the domain of all cells underneath $x^l_{i,j}$, ensuring that 40-foot containers can never be placed below a 20-foot container.

The combination of $\mathcal{P}^{\text{40-20}}$ and $\mathcal{P}^{\text{20-40}}$ ensures that it can never be the case that a 40-foot container can be underneath a 20-foot container or vice versa.

CT7 The height of each stack is within its limits

The constraint is exactly represented by the early termination criterion $\mathcal{E}^H$, which is the negation of the actual constraint.

CT8 The weight of each stack is within its limits

The constraint is represented by the early termination criterion $\mathcal{E}^W$, which is the negation of the actual constraint.

CT9 Reefer containers must be placed in reefer slots

The constraint is ensured by the domain mapper $\mathcal{D}$ and constraint $\mathcal{E}^{XR}$.

$\mathcal{D}$ ensures that reefer containers cannot be considered as candidate values for non-reefer cells. $\mathcal{E}^{XR}$ guarantees that a complete instantiation has all reefer containers assigned to a reefer cell. The proof is by contradiction, similar to the one given for **CT1**.

CT10 IMO rules are satisfied for each container

The IMO-1 constraint is represented by $\mathcal{P}^{\text{IMO-1}}$.

For a placed IMO-1 containers the constraint states that any cells adjacent to $x^l_{i,j}$ cannot accommodate an IMO-1 container. $\mathcal{P}^{\text{IMO-1}}$ ensures this by pruning all IMO-1 container from cells adjacent to $x^l_{i,j}$.

The IMO-2 constraint is represented by $\mathcal{P}^{\text{IMO-1}}$ and $\mathcal{P}^{\text{IMO-2}}$.

For a placed IMO-2 container, the constraint states that any cells adjacent to $x^l_{i,j}$ cannot accommodate an IMO-1 container, which is ensured by $\mathcal{P}^{\text{IMO-1}}$. Additionally no IMO-2 container can be assigned to the rest of the cells of stack j and to all cells in stacks $j+1$ and $j-1$. $\mathcal{P}^{\text{IMO-2}}$ ensures this, by pruning away IMO-2 containers from any cells in stacks $j+1$ and $j-1$ and any cell in the stack j except cell $x^l_{i,j}$.

CT11 Two halves of a 40-foot container must be placed in the same slot

This is constraint is similar to **CT3** and is ensured by the same pruning operation.

CT12 A cell that accommodates a 20-foot container excludes the possibility of its neighbor cell to accommodate a 40-foot half

The constraint is represented by $\mathcal{P}^{\text{20-20}}$.

When a 20-foot container is assigned to a cell, the domain of the neighbor cell is pruned to contain only the 20-foot container.

CT13 Allowing each container only to appear once

The constraint is represented by $\mathcal{P}^u$.

Whenever a container c is assigned to a cell $x^l_{i,j}$, $\mathcal{P}^u$ prunes away c as a candidate value from all other cells, which ensures that c can never be chosen again.

Chapter 6

Estimation

The purpose of this chapter is to give the reader an understanding of the estimation the branch and bound algorithm uses. The chapter is divided into four sections, where each section explains one objective of the storage area stowage problem. Each section defines the estimated cost, proves that the estimate is an underestimate of the real completion cost and argues for the efficiency of the estimator.

6.1 Overstowage Bounding

Given a partial solution $\vec{a}_p$, we wish to calculate a lower bound on the number of overstows of any solution extending it. Due to the exponential number of combinations, the "brute-force" way of building all complete solutions that extend $\vec{a}_p$ and computing the number of overstows for each stowage plan is infeasible. The goal is therefore to efficiently calculate a good, but not necessarily tight bound.

The main idea is to relax the problem by not taking into consideration the overstows among containers belonging to the extension of the partial solution.

The air container must not be counted in for the number of overstows. The following notation defines the domain values different than air that belong to the current instantiation:

$$\pi^{\perp}_{\mathcal{S}}(\vec{a}) = \pi_{\mathcal{S}}(\vec{a}) \setminus \{\perp\}$$

Formally, the costs are defined as follows:

$F^{0\times0}_{ov}(\vec{a}_p)$ is the number of overstows of $\vec{a}_p$.

$$F^{0\times0}_{ov}(\vec{a}_p) = \sum_{j=1}^{s_c} \sum_{l\in L} \left|\{(x^l_{i,j}, x^l_{k,j}) \in \pi^{\perp}_{\mathcal{S}_p}(\vec{a}_p)^2 \ : \ dp_{x^l_{i,j}} > dp_{x^l_{k,j}} \wedge i > k\}\right|$$

$F^{0\times1}_{ov}(\vec{a}_p, a_{p+1:n})$ is the number of overstows among containers belonging to $\vec{a}_p$ and containers belonging to an extension $(a_{p+1}, \ldots, a_n)$ of $\vec{a}_p$ into a complete instantiation.

$F_{ov}^{0\times1}(\vec{a}_p, a_{p+1:n}) =$
$\sum_{j=1}^{s_c} \sum_{l \in L} \left| \{(x_{i,j}^l, x_{k,j}^l) \in \pi_{\mathcal{S}_p}^{\perp}(\vec{a}_p, a_{p+1:n}) \times \pi_{X-\mathcal{S}_p}^{\perp}(\vec{a}_p, a_{p+1:n}) \ : \ dp_{x_{i,j}^l} > dp_{x_{k,j}^l} \wedge i > k\} \right|$

$F_{ov}^{1\times1}(\vec{a}_p, a_{p+1:n})$ is the number of overstows among containers belonging to an extension $(a_{p+1}, \ldots, a_n)$ of $\vec{a}_p$ into a complete instantiation.

$$F_{ov}^{1\times1}(\vec{a}_p, a_{p+1:n}) = \sum_{j=1}^{s_c} \sum_{l \in L} \left| \{(x_{i,j}^l, x_{k,j}^l) \in \pi_{X-\mathcal{S}_p}^{\perp}(\vec{a}_p, a_{p+1:n})^2 \ : \ dp_{x_{i,j}^l} > dp_{x_{k,j}^l} \wedge i > k\} \right|$$

The overstow cost of the partial instantiation $\vec{a}_p$ is:

$$g_{ov}(\vec{a}_p) = F_{ov}^{0\times0}(\vec{a}_p)$$

The cost of the optimal completion of $\vec{a}_p$ is

$$h_{ov}^*(\vec{a}_p) = \min_{a_{p+1:n}} \left(F_{ov}^{0\times1}(\vec{a}_p, a_{p+1:n}) + F_{ov}^{1\times1}(\vec{a}_p, a_{p+1:n}) \right)$$

where $(\vec{a}_p, a_{p+1:n})$ is a complete and valid instantiation.

The estimated completion cost is

$$h_{ov}(\vec{a}_p) = \min_{a'_{p+1:n}} \left(F_{ov}^{0\times1}(\vec{a}_p, a'_{p+1:n}) \right)$$

where $(\vec{a}_p, a'_{p+1:n})$ is a complete, but not necessarily valid instantiation.

Proposition 6.1 *The bounding cost h_{ov} is always an underestimate of the optimal completion cost h_{ov}^* i.e. $h_{ov}^* \geq h_{ov}$*

Proof.

$$\begin{aligned}
h_{ov}^*(\vec{a}_p) &= \min_{a_{p+1:n}} \left(F_{ov}^{0\times1}(\vec{a}_p, a_{p+1:n}) + F_{ov}^{1\times1}(\vec{a}_p, a_{p+1:n}) \right) \\
&= \min_{a_{p+1:n}} \left(F_{ov}^{0\times1}(\vec{a}_p, a_{p+1:n}) \right) + \min_{a_{p+1:n}} \left(F_{ov}^{1\times1}(\vec{a}_p, a_{p+1:n}) \right) \\
&\geq \min_{a_{p+1:n}} \left(F_{ov}^{0\times1}(\vec{a}_p, a_{p+1:n}) \right) \\
&\geq \min_{a'_{p+1:n}} \left(F_{ov}^{0\times1}(\vec{a}_p, a'_{p+1:n}) \right) \\
&= h_{ov}(\vec{a}_p)
\end{aligned}$$

which completes the proof. □

How to calculate h_{ov}?

Having proved the correctness of the estimator, the goal is to calculate its value efficiently. For a particular $\vec{a}_p$, create a weighted graph $G = (V, E, w)$ in the following manner: all unassigned containers and all cells not part of the instantiation, become nodes in the graph, and an edge is added between a container and a cell, if the container belongs to the variables domain. The weight of the edge is the number of overstows that result from stowing the container into the cell.

When $T = \{c \in C_{20}, c^l \in C_{40}^H \,:\, c \notin \pi_{\mathcal{S}_p}(\vec{a}_p)\}$ denotes the set of unassigned containers and $R = \{x_{i,j}^l \in X \setminus \mathcal{S}_p\}$ denotes the set of not yet instantiated cells, the set of nodes is

$$V = T \cup R$$

the set of edges is

$$E = \{(c, x_{i,j}^l) \in T \times R \,:\, c \in D(x_{i,j}^l)\}$$

and

$$\begin{aligned} w(c, x_{i,j}^l) \;\; = \;\; & \left|\{x_{k,j}^l \in \pi_{\mathcal{S}_p}^{\perp}(\vec{a}_p) \,:\, dp_c > dp_{x_{k,j}^l} \wedge k < i\}\right| + \\ & \left|\{x_{k,j}^l \in \pi_{\mathcal{S}_p}^{\perp}(\vec{a}_p) \,:\, dp_c < dp_{x_{k,j}^l} \wedge k > i\}\right| \end{aligned}$$

for each edge $(c, x_{i,j}^l) \in E$.

G is bipartite and, due to the way it is constructed, h_{ov} is equal to the cost of a complete matching of minimum cost. Therefore, efficiently calculating h_{ov} is reduced to efficiently find a maximum matching of minimum cost of the associated bipartite graph and verifying if the maximum matching is complete. In case the maximum matching is not complete, it can be concluded that the current instantiation cannot be extended to a valid solution.

Example 6.1 Consider the CSP-Model defined in Example 5.1 and a partial instantiation $\vec{a}_p = \{(x_{1,1}^W, c_0), (x_{1,2}^T, c_1)\}$.
The bipartite graph used for overstow estimation is shown in Figure 6.1.

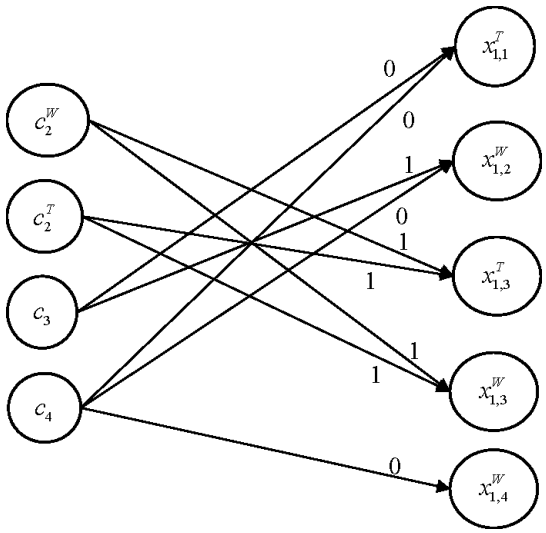


Figure 6.1: Bipartite Graph.

Definition 6.1 (Minimum Cost Matching Problem(MCMP)) *Given a weighted bipartite graph $G = (T \cup R, E, w)$ where $w : E \to \mathbb{N}$, the MCMP is to find a maximum matching of minimum cost: $M \subseteq E$, such that no edges of M have common endpoints, $|M|$ is maximized and $\sum_{e \in M} w(e)$ is minimized.*

Flow algorithms are state of the art algorithms that can efficiently solve a MCMP. These algorithms build a flow network and find a maximum flow of minimum cost [6].

Example 6.2 A solution of the matching problem built of Figure 6.1 is:

$$M = \{(c_2^W, x_{1,3}^W), (c_2^T, x_{1,3}^T), (c_3, x_{1,1}^T), (c_4, x_{1,2}^W)\}$$

The matching is complete and therefore $h_{ov}^* = 3$.

6.2 Emptystack Bounding

In this case, the goal is to calculate a lower bound on the number of used cellstacks of any solution extending a partial instantiation $\vec{a}_p$. A cellstack is *used* if at least one container has been stowed into it. Otherwise a cellstack is *empty*.

To accurately bound the number of used cellstacks of a complete solution, an estimation algorithm should fill already used cellstacks before starting empty cellstacks. Identifying a cellstack independently of its parent stack makes it easy to build an ordering among the cellstacks. A straightforward way to identify a cellstack is by its relative position to the left side of the stowage area. Cellstacks are counted from larboard to starboard, starting with 1.

$K = \{2j - 1, 2j \,:\, j \in J\}$ is the set of cellstacks.

Accordingly, one can identify the parent stack of a cellstack and the side the cellstack represents. W is considered to come before T and it is assumed that each side can be identified by its relative position to the parent slot.

Cellstack properties

$j_k = \lfloor \frac{1}{2}(k+1) \rfloor$	:	stack to whom cellstack k belongs
$l_k = \begin{cases} W & \text{if } \lfloor (k+1) \mod 2 \rfloor = 0 \\ T & \text{otherwise} \end{cases}$	:	the side cellstack k represents
$fs(k) = h_{j_k} - \sum_{i \in \{i \,:\, x_{i,j_k}^{l_k} \in \mathcal{S}_p\}} h_{x_i,j_k}^{l_k}$	:	free space of cellstack k
$ws(k) = \begin{cases} fs(k) & \text{if } fs(k) < h_{st} \\ 0 & \text{otherwise} \end{cases}$	:	the wasted space of cellstack k

$U = \{k \in K \,:\, \exists\, i \,.\, x_{i,j_k}^{l_k} \in \pi_{\mathcal{S}_p}(\vec{a}_p)\}$ is the set of used cellstacks.
$E = K \setminus U$ is the set of empty cellstacks.

Having defined the set of used cellstacks and the set of empty cellstacks, we are ready to introduce an ordering relation on the set of cellstacks K.

For any permutation ρ of U, $\prec^{\rho}$ is a pre-order on K that orders the used cellstacks accordingly to ρ while the free cellstacks are ordered in decreasing order of their available cells and follow used cellstacks.

$$\begin{aligned} k \prec^{\rho} m \quad \Leftrightarrow \quad & \big(k, m \in U \ \wedge \ \rho_k < \rho_m\big) \vee \\ & \big(k \in U \ \wedge \ m \in E\big) \vee \\ & \big(k, m \in E \ \wedge \ \big|\{i \ : \ x_{i,j_k}^{l_k} \notin \pi_{\mathcal{S}_p}(\vec{a}_p)\}\big| \leq \big|\{i \ : \ x_{i,j_m}^{l_m} \notin \pi_{\mathcal{S}_p}(\vec{a}_p)\}\big|\big) \end{aligned}$$

$\prec_k^{\rho}$ denotes the kth cellstack in the ordering.

For a particular ordering $\prec^{\rho}$, the minimum number of cellstacks needed for S standard containers and H highcube containers, when the k-th cellstack in the ordering has σ available space is defined recursively in terms of the optimal solutions to subproblems:

$$u_{\rho}(S, H, k, \sigma) = \begin{cases} k, \text{ if } S = 0 \ \text{ and } \ H = 0 \\ min(u_{\rho}[S-1, H, k, \sigma - h_{st}), u_{\rho}(S, H-1, k, \sigma - h_{hc})) \\ \qquad \text{if } \ S \geq 1 \ \text{ and } \ H \geq 1 \ \text{ and } \ \sigma \geq h_{hc} \\ u_{\rho}(S, H-1, k, \sigma - h_{hc}) \text{ if } S = 0 \ \text{ and } \ H \geq 1 \ \text{ and } \ \sigma \geq h_{hc} \\ u_{\rho}(S-1, H, k, \sigma - h_{st}) \text{ if } S \geq 1 \ \text{ and } \ H = 0 \ \text{ and } \ \sigma \geq h_{st} \\ u_{\rho}(S-1, H, k, \sigma - h_{st}) \text{ if } S \geq 1 \ \text{ and } \ H \geq 1 \ \text{ and } \ h_{st} \leq \sigma < h_{hc} \\ u_{\rho}(S, H, \prec_{k+1}^{\rho}, fs(\prec_{k+1}^{\rho})) \text{ if } S + H \geq 1 \text{ and } \ \sigma < h_{st} \\ u_{\rho}(S, H, \prec_{k+1}^{\rho}, fs(\prec_{k+1}^{\rho})) \text{ if } H \geq 1 \text{ and } \ h_{st} \leq \sigma < h_{st} \end{cases}$$

The structure of an optimal solution of the problem can be characterized in the following way. Let $A_{1:j}$ denote the optimal arrangement of j containers. Let i be the first container of the last cellstack k, needed to accommodate the j containers. The goal is to prove that $A_{1:i-1}$ is an optimal arrangement. The proof is by contradiction: let's assume that $A_{i:1-1}$ is not optimal. In this case there is another arrangement $A'_{1:i-1}$ that is optimal and needs $k' < k-1$ cellstacks. In this case, we can construct arrangement $A'_{1:j}$ consisting of $A'_{1:i-1}$ followed by containers i to j stowed in cellstack $k'+1$. Since cellstack $k'+1$ comes before cellstack k in the cellstack ordering, it is certain that it can accommodate containers i to j. Observing that $k' + 1 < k$ we have just constructed an arrangement for containers 1 to j that needs fewer cellstacks than the optimal arrangement $A_{1:j}$, contradicting the optimality of $A_{1:j}$.

We are now ready to formally define the cost functions. Let S be the number of unassigned 20-foot containers and H be the number of unassigned 20-foot containers:

$$\begin{aligned} S &= \big|\{c \in C_{20} \ : \ c \notin \pi_{\mathcal{S}_p}(\vec{a}_p)\}\big| \\ H &= \big|\{c^l \in C_{40}^H \ : \ c^l \notin \pi_{\mathcal{S}_p}(\vec{a}_p)\}\big| \end{aligned}$$

The empty stack cost of the partial instantiation $\vec{a}_p$ is:

$$g_{es}(\vec{a}_p) = |U|$$

The cost of the optimal completion of $\vec{a}_p$ is:

$$h^*_{es}(\vec{a}_p) = \max\left(0, \min_{\rho \in \mathcal{P}(U)} u_\rho(S, H, \prec^\rho_1, fs(\prec^\rho_1)) - g_{es}(\vec{a}_p)\right)$$

The estimated completion cost is:

$$h_{es}(\vec{a}_p) = \max\left(0, u_{\rho_0}(S, H, \prec^{\rho_0}_1, fs(\prec^{\rho_0}_1)) - g_{es}(\vec{a}_p)\right),$$

where ρ_0 is an arbitrary permutation of the used cellstacks set U.

$g_{es}(\vec{a}_p)$ has to be subtracted from the above costs since it is already included in u.

Proposition 6.2 *The bounding cost h_{es} is always an underestimate of the optimal completion cost h^*_{es} i.e. $h^*_{es} \geq h_{es}$*

Proof.

Since, used stacks are filled before empty stacks, and empty stacks are considered in the same order, the proof that h_{es} is a correct bounding function for h^*_{es} is reduced to showing that the number of newly started empty stacks does not depend on the order in which the used cellstacks are considered. Given two permutations ρ_1 and ρ_2 of the used cellstacks, u_{ρ_1} and u_{ρ_2} will examine in different order, but nevertheless the same stowage combinations, and therefore choose the same optimal arrangement, guaranteeing that the same number of additional empty cellstacks are necessary. Thus the order in which the used stacks are considered is irrelevant, proving that $h_{es} = h^*_{es}$.

□

How to calculate h_{es}?

We have already argued for the optimal structure of the problem. Additionally, if the space of subproblems is small, in the sense that the recursive algorithm solves the same subproblems repeatedly, dynamic programming is applicable and efficiently solves the problem.

To analyze how subproblems overlap, a problem tree is built for a problem with a single cellstack with infinite space. The root of the tree is the initial problem and each node identifies a subproblem for a number of standard containers and a number of highcube containers. Due to the construction procedure the number of standard containers of each node is at most 1 less than the number of standard containers of its parent node. The same holds for highcube containers. Picture 6.2 shows part of the subproblem tree.

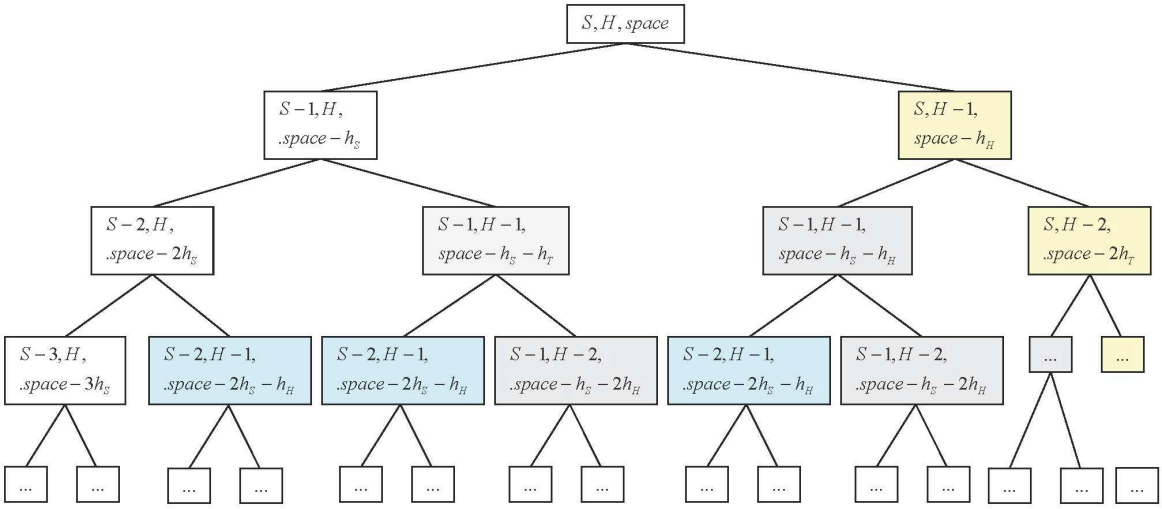


Figure 6.2: Overlapping subproblems for a single cellstack.

Considering the root of the tree having level 0, observe that each sequence to a subproblem found at level k is identified by a sequence of choices of length k, where each choice is either a standard container or a highcube container. To count the number of distinct subproblems of level k is reduced to counting the different possible ways of building a sequence of length k where the accumulative number of standard containers, respective highcube containers matters, but the order in which they appear in the sequence is irrelevant. There are $\begin{pmatrix} k \\ S \end{pmatrix}$ ways to build a sequence having S standard containers and $k - S$ highcube containers. However, since order is irrelevant, these $\begin{pmatrix} k \\ S \end{pmatrix}$ subproblems are identical. Since we can build sequences having $0 \leq S \leq k$ standard containers (and accordingly $k - S$ highcube containers), the number of distinct subproblems found at level k is $k + 1$.

Consequently, the total number of distinct subproblems for a single cellstack problem with infinite space is:

$$\sum_{k=0}^{H+S}(k+1) = \frac{(H+S+1)(H+S+2)}{2}$$

A similar proof holds for the general problem, where a cellstack has a limited amount of space and there are several stacks available by viewing the accumulated available space of the set of cellstacks as the available space of a single cellstack. The main difference is that whenever the space associated with a particular cellstack is filled, the wasted space should be substracted from the accumulated available space.

Dynamic programming optimizes the inefficient recursive algorithm by maintaining and reusing solutions to subproblems. A table maintains solutions to subproblems. Initially all table entries contain the nil value to indicate that the entry has not been calculated yet. When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and stored. Each subsequent time the subproblem is encountered,

the value stored in the table is simply returned.

6.3 Wastedspace Bounding

In this case, the goal is to calculate a lower bound on the wasted space of any solution extending a partial instantiation $\vec{a}_p$. The idea behind this estimator is the same with the one presented for Emptystack bounding: an ordering is chosen for the set of cellstacks, and all stowage combinations that fill cellstack after cellstack as defined in the ordering are built and evaluated.

The main difference lies, when the recursive algorithm reaches a boundary condition. This happens, when a cellstack cannot accommodate further containers, or when the stowage plan is complete. In the first case, the remaining space is wasted, since it is less than the space needed for a standard container. In the later case, the wasted space is calculated as being the wasted space of the the current cellstack, added together with the wasted space of all cellstacks that follow in the ordering.

For a particular ordering $\prec^\rho$, the minimum wasted space for S standard containers and H highcube containers, when the k-th cellstack in the ordering has σ available space is defined recursively in terms of the optimal solutions to subproblems:

$$w_\rho(S,H,k,\sigma) = \begin{cases} \sigma^* + \sum_{j=k+1}^{|K|} ws(\prec_j^\rho), \text{ if } S=0 \text{ and } H=0 \\ min(w_\rho[S-1,H,k,\sigma-h_{st}), w_\rho(S,H-1,k,\sigma-h_{hc})) \\ \qquad \text{if } S\geq 1 \text{ and } H\geq 1 \text{ and } \sigma \geq h_{hc} \\ w_\rho(S,H-1,k,\sigma-h_{hc}) \text{ if } S=0 \text{ and } H\geq 1 \text{ and } \sigma \geq h_{hc} \\ w_\rho(S-1,H,k,\sigma-h_{st}) \text{ if } S\geq 1 \text{ and } H=0 \text{ and } \sigma \geq h_{st} \\ w_\rho(S-1,H,k,\sigma-h_{st}) \text{ if } S\geq 1 \text{ and } H\geq 1 \text{ and } h_{st} \leq \sigma < h_{hc} \\ \sigma + w_\rho(S,H,\prec_{k+1}^\rho, fs(\prec_{k+1}^\rho)) \text{ if } S+H\geq 1 \text{ and } \sigma < h_{st} \\ \sigma + w_\rho(S,H,\prec_{k+1}^\rho, fs(\prec_{k+1}^\rho)) \text{ if } H\geq 1 \text{ and } h_{st} \leq \sigma < h_{st} \end{cases}$$

where

$$\sigma^* = \begin{cases} \sigma & \text{if } \sigma < h_{st} \\ 0 & \text{otherwise} \end{cases}$$

is the wasted space of the current cellstack. The cost functions are defined below.

The wasted space cost of the partial instantiation $\vec{a}_p$ is:

$$g_{ws}(\vec{a}_p) = \frac{\sum_{k\in K} ws(k)}{h_{st}}$$

The cost of the optimal completion of $\vec{a}_p$ is:

$$h^*_{ws}(\vec{a}_p) = \frac{\min_{\rho \in \mathcal{P}(U)} w_\rho(S, H, \prec_1^\rho, fs(\prec_1^\rho))}{h_{st}} - g_{ws}(\vec{a}_p)$$

The estimated completion cost is:

$$h_{ws}(\vec{a}_p) = \frac{u_{\rho_0}(S, H, \prec_1^{\rho_0}, fs(\prec_1^{\rho_0}))}{h_{st}} - g_{ws}(\vec{a}_p)$$

where ρ_0 is an arbitrary permutation of the used cellstacks set U.

$g_{ws}(\vec{a}_p)$ has to be subtracted from the above costs since it is already included in w.

Proposition 6.3 *The bounding cost h_{ws} is always an underestimate of the optimal completion cost h^*_{ws} i.e. $h^*_{ws} \geq h_{ws}$*

Proof.

The correctness proof for h_{ws} is the same as for h_{es} in proposition 6.2

□

An efficient way to calculate it, is by using dynamic programming.

6.4 Reefer Bounding

For any partial instantiation $\vec{a}_p$, the goal is to estimate the number of reefer cells that are occupied with non-reefer containers for any complete solution that completes $\vec{a}_p$. The estimate is achieved by counting the number of reefer cells that will certainly be instantiated with a not-air container and subtracting it from the count of reefer container halfs. Formally, the costs are defined as follows:

X_r^{nr} is the set of reefer cells of $\vec{a}_p$ instantiated with non reefer containers.

$$X_r^{nr} = \{x^l_{i,j} \in \mathcal{S}_p \,:\, r_{i,j} \wedge x^l_{i,j} \in C^H_{nr}\}$$

X_r^r is the set of reefer cells of $\vec{a}_p$ instantiated with reefer containers.

$$X_r^r = \{x^l_{i,j} \in \mathcal{S}_p \,:\, r_{i,j} \wedge x^l_{i,j} \in C^H_r\}$$

X_r is the set of uninstantiated reefer cells whose domain does not contain air.

$$X_r = \{x^l_{i,j} \in X^R \,:\, r_{i,j}\}$$

The reefer cost of the partial instantiation $\vec{a}_p$ is the number of reefer cells of $\vec{a}_p$ instantiated with non reefer containers:

$$g_r(\vec{a}_p) = |X_r^{nr}|$$

The cost of the optimal completion of $\vec{a}_p$ is the number of reefer cells that with certainty will be instantiated with non reefer containers plus an ε number of reefer cells that may be instantiated with non reefer containers:

$$h_r^*(\vec{a}_p) = \max\big(0, |X_r| - (|C_r^H| - |X_r^r|)\big) + \varepsilon$$

The estimated completion cost is:

$$h_r(\vec{a}_p) = \max\big(0, |X_r| - (|C_r^H| - |X_r^r|)\big)$$

Proposition 6.4 *The bounding cost h_r is always an underestimate of the optimal completion cost h_r^* i.e. $h_r^* \geq h_r$*

Proof.

Trivially, $h_r^*(\vec{a}_p) \geq h_r(\vec{a}_p)$.

□

The bounding evaluation function used by the branch and bound search algorithm is:

Definition 6.2 (Bounding evaluation function)

$$f(\vec{a}_p) = W_{ov}\big(g_{ov}(\vec{a}_p)+h_{ov}(\vec{a}_p)\big)+W_{es}\big(g_{es}(\vec{a}_p)+h_{es}(\vec{a}_p)\big)+W_{ws}\big(g_{ws}(\vec{a}_p)+h_{ws}(\vec{a}_p)\big)+W_r\big(g_r(\vec{a}_p)+h_r(\vec{a}_p)\big)$$

Proposition 6.5 *The bounding cost $f(\vec{a}_p)$ is always an underestimate of the optimal completion cost $f^*(\vec{a}_p)$ i.e. $f^*(\vec{a}_p) \geq f(\vec{a}_p)$*

Proof.

The proof follows trivially from the individual proofs for each cost component:

$$\begin{aligned} f^*(\vec{a}_p) &= W_{ov}\big(g_{ov}(\vec{a}_p) + h^*_{ov}(\vec{a}_p)\big) + W_{es}\big(g_{es}(\vec{a}_p) + h^*_{es}(\vec{a}_p)\big) + \\ &\quad W_{ws}\big(g_{ws}(\vec{a}_p) + h^*_{ws}(\vec{a}_p)\big) + W_r\big(g_r(\vec{a}_p) + h^*_r(\vec{a}_p)\big) \\ &\geq W_{ov}\big(g_{ov}(\vec{a}_p) + h_{ov}(\vec{a}_p)\big) + W_{es}\big(g_{es}(\vec{a}_p) + h_{es}(\vec{a}_p)\big) + \\ &\quad W_{ws}\big(g_{ws}(\vec{a}_p) + h_{ws}(\vec{a}_p)\big) + W_r\big(g_r(\vec{a}_p) + h_r(\vec{a}_p)\big) \\ &= f(\vec{a}_p) \end{aligned}$$

□

Chapter 7

Implementation

The implementation chapter gives a description on how the main parts have been implemented. We consider the main parts to be representation of data, search algorithms, propagation engine, estimator and evaluation calculator.

The chapter begins with explaining fundamental concepts, which are necessary in order to understand how the running time is improved. Then an explanation of how data are represented is given. After the data representation, the pseudocode for the search algorithms, propagation engine, estimators and evaluation calculator is given.

This chapter outline an approach of how effective pruning can be achieved and a data structure for represent a collection of characteristics for some of the objects within SASP.

7.1 Fundamental concepts

One of the main focuses in the implementation has been to be able to prune domains and backtrack efficiently. In order to do that, the concepts of shared domains and labels have been conceived.

Shared Domains

Each variable maintains some characteristic information about properties that possible domain values should posses, in order to be eligible for being assigned to a particular variable. One possible naive implementation is to let each variable maintain a domain with all the candidate values, it can select. A domain value is then removed, when it is no longer eligible for that particular variable. Domain changes are recorded for each variable, such that a domain value reappears, when it is eligible for selection again. The basic operations, which the domain is required to support, are adding and removing domain values. Using a hashtable to represent a domain - adding and removing a value is expected to be O(1). Pruning away several domain values from a domain will therefore be $O(|D|)$. The domain values in SASP can be viewed as being shared amongst the variables. That is, containers

to be placed are taken from the same pool. Based on this observation, it is required that any used domain value has to be removed as a candidate value from the domains of the remaining unassigned variables. Using the time complexities above for a set of $|X|$ variables - adding or removing an already selected value from the domains of the remaining unassigned variables is expected to be $O(|X|)$. Table 7.1 summarize the time complexities for a naive implementation of domains.

Table 7.1: Time complexities for naive implementation

Adding a domain value to a domain	$O(1)$
Removal of a domain value from a domain	$O(1)$
Pruning several values from a domain	$O(\|D\|)$
Adding a domain value to the domains of all unassigned variables	$O(\|X\|)$
Removal of a domain value from the domains of all unassigned variables	$O(\|X\|)$

Since each variable in the worst case has all domain values represented in its domain, the space requirement is expected to be $O(|X||D|)$. Table 7.2 summarize the space complexities for a naive implementation of domains.

Table 7.2: Space complexities for naive implementation

Representing domain values in a single domain	$O(\|D\|)$
Representing domain values in each domain	$O(\|X\|\|D\|)$

Due to the nature of how pruning occurs in SASP, it has been discovered through analysis that the pruning mainly occurs on the basis of the container type and not so much on the container itself e.g. if a cell cannot accommodate a container due to a property such as IMO-1, then that cell will not be able to accommodate any container with that property. In addition, the variation of container types that is, the number of unique combinations of container properties, are quite limited. Further more, several cells are able to accommodate the same kind of containers, which implies that several variables can potentially share the same domain. If the cardinality of the set of unique domains U is considerable less than the total number of variables, then the time and space requirements can be lessened considerably. This holds in the case of SASP and the concept is presented as *shared domain*. The idea behind shared domains is that exactly one copy of each unique domain is maintained, which each variable is able to refer to by using some characteristic description.

This characteristic description is referred to as a *label*, and will be explained in further detail in the following section. Based on a particular label $\mathcal{L}$, a reference to a domain with label $\mathcal{L}$ can be obtained. The concept of pruning several values from the domain of a single variable will simply be a matter of changing the label for that variable and obtaining a new reference to a domain with the newly constructed label. Obtaining a reference can easily be implemented by using a hashtable, which uses the label as the key and the reference to a shared domain as the value, retrieval can therefore be expected to be $O(1)$. If changing the label and obtaining the reference can be expected to be carried out in constant time,

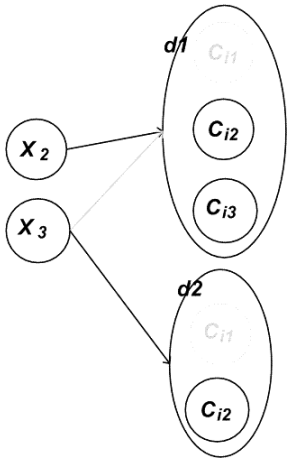


Figure 7.1: Variable x_2 and x_3 are initially pointing to domain $d1$. Container C_{i1} is then assigned to some variable and has to be pruned away from all domains due to uniqueness propagation. Some properties when assigning the domain value C_{i1} causes variable x_3 to change its domain to $d2$.

then pruning based on the label can be achieved in constant time. Table 7.3 shows the time complexities for implementing shared domains.

Table 7.3: Time complexities for shared domain implementation

Adding a domain value to a domain	$O(1)$
Removal of a domain value from a domain	$O(1)$
Pruning several values from a domain	$O(1)$
Adding a domain value to domains of all unassigned variables	$O(\|U\|)$
Removal of a domain value from domains of all unassigned variables	$O(\|U\|)$

Since variables can share the same domain, the space requirement for all domains can be reduced from $O(|X||D|)$ to $O(|U||D|)$. The table 7.4 summarizes the space complexities for implementing shared domain.

Constructing unique domains

The number of unique domains in U, will depend on the structure of the problem. Through some analysis of SASP, the following properties, were taken into consideration when constructing the unique domains: IMO, Container length, reefer and air properties.

The possible IMO combinations is based on the fact that a cell, which can accommodate an IMO-2 container can also accommodate an IMO-1 container, which in turn can accommodate an IMO-0. This is induced from the propagators $\mathcal{P}^{\text{IMO-1}}$ and $\mathcal{P}^{\text{IMO-2}}$, which describes the IMO constraint for SASP.

Table 7.4: Space complexities for shared domain implementation

Representing domain values in a single domain	$O(\|D\|)$
Representing domain values in all domains	$O(\|U\|\|D\|)$

Three combinations based on the length property of containers are constructed. They are selected on the basis that cells typically can accommodate either a 40-foot half, or a 20-foot container before any containers have been placed within the bay. Since a slot cannot accommodate both a 20-foot container and a 40-foot container, placing a 20-foot container in a cell excludes any 40-foot container half in the neighbor slot, which is ensured by propagator $\mathcal{P}^{\text{20-20}}$. The same argumentation holds for 40-foot containers and is ensured by propagator $\mathcal{P}^{\text{40-40}}$. This yields two additional combinations: 20-foot containers and 40-foot containers.

The possible reefer combinations is constructed on the fact that any non-reefer cell may only accommodate non-reefer containers, while reefer cells may accommodate either reefer containers or non-reefer containers.

The air property is finally taken into consideration, for which two combinations exists. Initially a cell may either hold a container or air. Cells, which cannot accommodate air, are constructed as a result of propagator $\mathcal{P}^{g}$. The combinations for each chosen category are summarized in table 7.5.

IMO	Length	Reefer	Air
IMO-0 ∪ IMO-1 ∪ IMO-2	40-foot ∪ 20-foot	non-Reefer ∪ Reefer	Air
IMO-0 ∪ IMO-1	40-foot	non-Reefer	non-Air
IMO-0	20-foot		

Table 7.5: Combinations in the selected categories

The properties selected for constructing the unique domains have been chosen based on the low number of combinations within each criteria. The upper bound on the number of unique domains is calculated by multiplying the cardinality of the combinations created for each selected criterion. Criterions with many possible combinations are therefore unsuitable to be distinguished upon. The weight property is an example of this.

Having three possible IMO combinations, three container length combinations, two reefer combinations and two air combinations, the total number of domains results in 36 unique domains.

Any domain d for SASP has a corresponding domain $d^{\perp}$, which has an additional air domain value. $d^{\perp}$ is referred to as the air domain. Whenever a domain value v is removed from d then v has to be removed from $d^{\perp}$ as well. Besides the air domain value the domains have the same behavior. This fact can be exploited by having a data structure, which keeps a reference to domain d, such that when reasoning about the number of elements in $d^{\perp}$, then it would be all elements in d with the additional air domain value. Consequently air

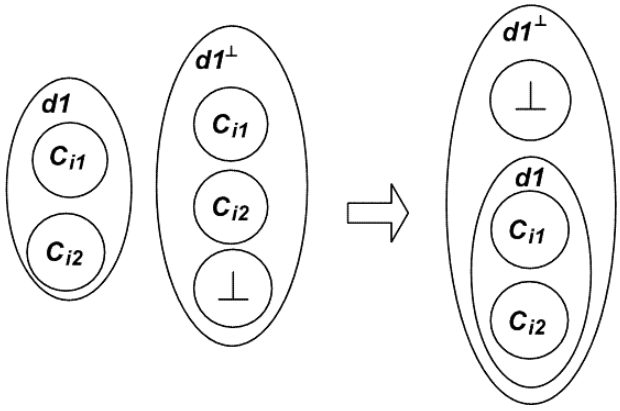


Figure 7.2: The domain $d1$ and $d1^{\perp}$ have the same containers except for the air container. When some container appearing in $d1$ has to be removed the same container has to be removed from $d1^{\perp}$ as well. $d1^{\perp}$ can therefore include the containers, which $d1$ has

domain values do not need to get pruned away since pruning can be achieved by changing to the corresponding domain, which does not accommodate air. This technique halves the number of unique domains.

Single container domain

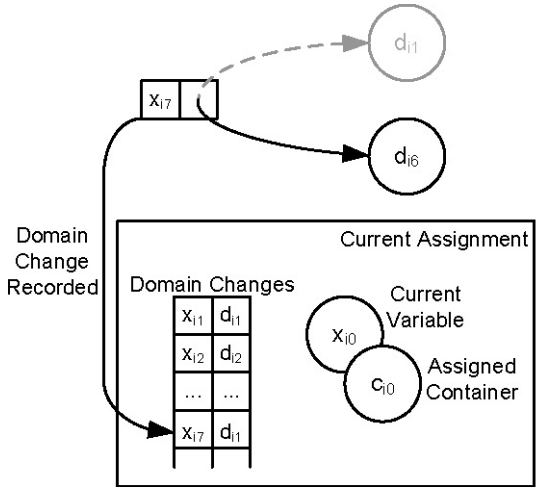
The propagator $\mathcal{P}^{40\text{-}40}$ requires that a 40-foot container half needs to be placed next to its other half. Consequently the other container half cannot be considered as an eligible value for any other variable than the neighbor cell of the placed 40-foot container half. When using shared domain this has an impact on the number of unique domains and pruning. The idea of *Single container domain* and *Collapsing 40-foot container halves* has been conceived to remedy this issue. Having the requirement of pruning all containers, such that only a 40-foot container half is left, makes the number of domains grow linearly in the number of 40-foot containers. These domains are named single container domains. They have the property that only a single 40-foot container is present. $\mathcal{P}^u$ forces the 40-foot container half to be pruned away once it has been assigned to a cell, and consequently the time complexity for removal of a single domain value is degraded to $O(|U|+|C_{40}|)$, because it is unknown, which one of the single container domain that needs to be pruned. The running time can be improved by collapsing containers. Instead of having two container halves c^T and c^W for container c, a single container half c' is used to represent c. Since the two container parts c^T and c^W are the same container all properties for one of the container halves are the same as the other half and vice versa. Further more, there is nothing in SASP, which requires to distinguish the container halves of the same container from each other, thus no issue will appear in that relation. When placing a 40-foot container c in a cell, the container is removed from any shared domain by $\mathcal{P}^u$ and the domain for the neighbor cell needs to be the single container domain for container c, since that is the only container, which should be considered. Since $\mathcal{P}^u$ removes c from any shared domain, c cannot be considered as candidate value for any other cell. The single container domain ensures that the other half is the only value to be chosen. If backtracking occurs, c should

not be inserted into any shared domains before all assignments, where c appears have been removed. The single container domain is only used after the first half is chosen. From this it can be inferred, that a 40-foot container should only be reinserted into domains, once the half which was assigned first has been unassigned.

Further pruning will not be necessary and consequently no additional cost is added to the running time, when removing a domain value from domains and the running time of $O(|U|)$ is therefore retained.

Domain pruning

Changing a domain reference for a variable in itself does not remove any domain values. This is however necessary for domain values, which have already been assigned to a specific variable. All other factors being equal, when not using any particular data structure, the removal of a domain value can only be attained by iterating over all unique domains, which results in a running time of $\Theta(|U|)$. In order to reduce running time, the domains are divided into lists, where each list holds a reference to all domains, which share a given combination of container properties. These lists are then stored in a hashtable, where the given property is used as key. Using this data structure, running time can be improved to $O(|U|)$. Removing a domain value with a given set of properties is then simply a matter of retrieving the list of domains, with the corresponding properties and then iteratively remove it from the domains in the list. The proposed datastructure is referred to as *Label-Domain table*. The pseudocode for the described approach can be found in the appendix C.3. Example 7.1 below shows how pruning can be carried out.

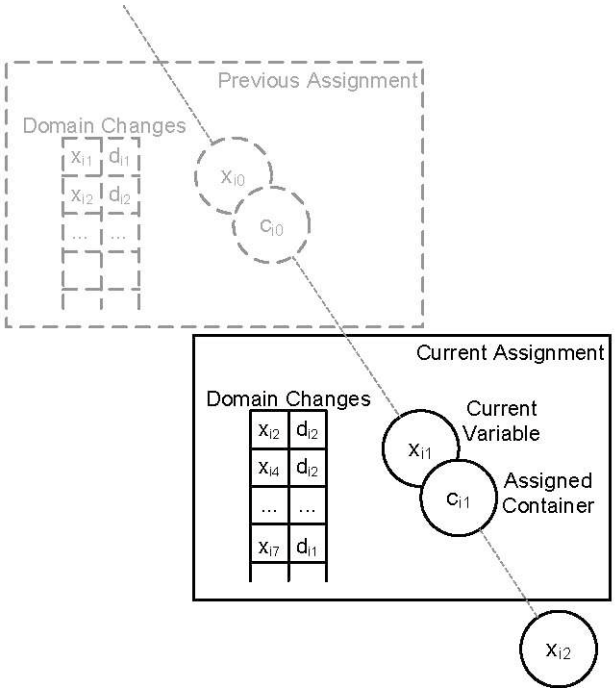


Example 7.1
Cell x_{i0} is stowed with container c_{i0} causing domain pruning for all affected cells. The domain reference for cell x_{i7} changes from domain d_1 to d_6

Domain restoration

Restoring shared domains when backtracking is a bit more complicated than when variables maintains a domain on its own. In order to be able to restore domains when a backtrack

occurs, domain reference changes, that were made during an assignment of variable x, has to be recorded. Example 7.2 shows how domain restoration is carried out.



Example 7.2
Cell x_{i0} has been stowed with container c_{i0} previously causing domain reference changes for any affected variables to be recorded in a list. An additional domain change for cell x_{i2} is recorded when the current instantiation of cell x_{i1} is being instantiated with container c_{i1}

The list of domain reference changes is a list of pairs, such that the first component is a future variable x' and the second component is a reference to the shared domain x' was referring to before x was assigned. When backtracking to x, any reference changes are restored by traversing the list and updating each variable with the domain it previously referred to. A restriction induced by this data structure, is that the domains can only be restored properly, if the variables are unassigned in the same order as they were assigned.

Example 7.3 *Suppose a given unassigned variable have multiple neighbors assigned with IMO-1 containers. The domain of the unassigned variable must not be changed to a domain allowing IMO-1 containers until all neighboring assignments with IMO-1 have been unassigned. As this information is only kept within the first neighbor, which was assigned with an IMO-1 container, its domain would be restored incorrectly if the assignment order for assigned variables is altered.*

Since it is expected that changing reference for a variable can be done in constant time, restoring all domains to a previously domain is expected to be $O(|X|)$.

Making a domain change does not add the domain value v back to any domains, and consequently does not make it eligible for future variables. The domain value needs to be

reinserted to any domains, which contained v prior to v was used. Having in mind the label-domain table described previously, the label $\mathcal{L}$ for v can identify the domains v needs to be added to, thus the running can still be kept to $O(|U|)$. The pseudecode for the described operation can be found in appendix C.3.

Maintenance of search state

A variable may be attempted to be assigned with several different domain values from its domain to see if that value will lead to a consistent instantiation. Once a domain value has proven not to lead to any consistent instantiation, it should not be considered a candidate value for that variable again. However using shared domains makes it not possible just to remove domain values as the removal of a domain value will be visible for any other variable which shares the same domain. This issue can be addressed, if the domains can guarantee a fixed order of the domain values. By using a combination of an order and a pointer all domain values from the beginning of the order to the pointer can be regarded as domain values, which has been considered as candidate value, while the rest are eligible candidate value for instantiation. Initially the pointer for a variable x is pointing to the first domain value within the ordered list of a domain. For each considered candidate value the index is incremented. Whenever a backtrack occurs and a previously instantiated variable x' needs to consider a new candidate value, the pointer for x is reset to the beginning of the domain, such that all domain values can be reconsidered for x under the new assignment of x'.

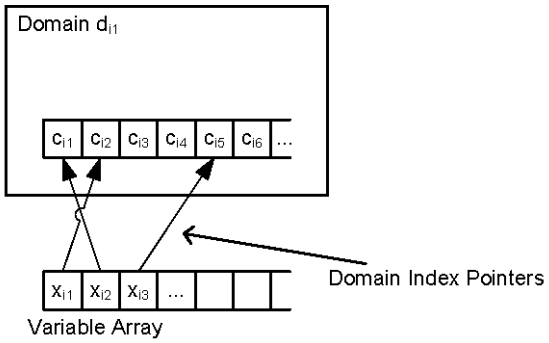


Figure 7.3: Index pointer to domain d_1 of the variables $x_{i1}, x_{i2}, x_{i3}, \ldots$

Label

Each container, cell, domain, and propagator have a set of characteristics, which collectively are referred to as *label*. A label is implemented as an array of bits in order to support following operations efficiently:

- Constructing a label
- Elementhood test

- Testing a label for a specific property

Each property in the label is represented as a sequence of boolean flags and indicates the presence of following properties: IMO-0, IMO-1, IMO-2, 20-foot container, 40-foot container, reefer, non-reefer and air. An additional flag, which marks whether a container is standard height or highcube, is used as well.

The labels are interpreted differently depending on the type of object, which possesses the label. For a cell, the label is interpreted as the type of container the cell can accommodate. For a domain, the label is interpreted as the set of containers the domain consists of. This gives an easy association between cells and their corresponding domains, as a cell and its corresponding domain have identical labels.

For a container, the label describes the properties of the given container. To determine whether a container belongs to a given domain, it must be verified that the properties described within the label of the container are also present within the label of the domain.

For the propagators, the label works as an association between propagators and containers, in a similar way as the label works between domains and containers. A propagator is scheduled to be run if the label of the propagator has properties, which exist in the label of the container. As an example, $\mathcal{P}^{\text{IMO-1}}$ is scheduled to run propagation for an input variable, if the IMO-1 property is present within the label of the container, that the input variable is assigned with.

A propagator prunes the domain of a cell, by altering its label such that a reference to a new domain is obtained. As an example, the $\mathcal{P}^{\text{IMO-1}}$ prunes the domain of the output variables, by ensuring that the flag for IMO-1 within the label for a given output variable is reset. Pseudocode for $\mathcal{P}^{\text{IMO-1}}$ can be found in appendix C.4.2

Due to the fact that labels are shared amongst different label holders some combinations of flags are not possible for a specific label holder. For instance, a container is not able to both be a 20-foot and 40-foot container at the same time, thus only one of these two bits can be set. However, a cell may accommodate both a 20-foot and 40-foot container and therefore allows both flags to be set at the same time. Still, this does not pose any issue as long as the label is interpreted correctly.

The space needed for a label is 9 flags or bits and can therefore be placed within a single machine word on a 32-bit machine. This makes it possible to support the desired operations with few instructions.

The bit layout of the label is showed in the left table in example 7.4, while the right table depicts the legend of each flag.

Example 7.4 *Layout for label*

Bit:	8	7	6	5	4	3	2	1	0
	h	a	0	1	2	r	n	t	f

Flags	
h	highcube
a	air
0	IMO-0
1	IMO-1
2	IMO-2
r	reefer
n	non-reefer
t	20-foot container
f	40-foot container

In the following, the operations supported by a label are detailed. In order to explain how the different operations are computed, some notation are introduced: the bitwise AND operation is denoted as $\&$, the bitwise OR operation is denoted as $\mid$ and the bitwise NOT operation is denoted as $\sim$.

Constructing a label

Since the label is a bit array, bits can easily be set or reset to construct labels. For instance, this is used when a variable needs to retrieve a new shared domain.

Pruning a label
Let p be a label and q be a label identifying properties which should not be part of p. Pruning of label p can be achieved by the following operation:

$$p \ \& \ \sim q$$

Including properties in a label
Let p be a label and q be a label identifying properties desired to be part of p. The following operation includes q in p:

$$p \mid q$$

Elementhood test

The elementhood test checks whether a label p is contained within another label q, for instance if a container is a member of a domain. This is achieved by checking whether all flags in p are in q. If some of the flags in p are not set in q it means that some property was not supported by q.

This corresponds to performing a logical AND for each flag and then check if the flags still are intact in respect to q. By using this fact, one can support the elementhood operation by using two instructions (one bitwise AND and one equal instruction).

Let p and q be labels. The following operation returns *true* if p is contained within q:

$$(p \ \& \ q) = q$$

Example 7.5 *Elementhood test Let $u = 001001010$ be the label for a 20-foot IMO-0 reefer container, $v = 001000111$ be the label of the domain which can contain 20-foot or 40-foot, IMO-0 and only non-reefer containers. Elementhood test returns false.*

Testing a label for a specific property

Testing a label for a specific property is used for example to determine if a container is a 40-foot container. This is achieved by doing following operation:

Let p be a label and q be a label with the properties being tested. The following operation returns *true* if the properties identified by q are present in p:

$$(p \ \& \ q) <> 0$$

7.2 Representing data

The main data structure used in the pseudocode is a set-like data structure, in the following referred to as a *setarray*. Setarrays mimics the behavior of sets known from mathematic i.e. all elements are distinct, a new set can be constructed by the union of two sets or by applying the set difference and so forth. The difference is that the elements of the sets are ordered along the insertion order and the nth element can be accessed by using the postfix operator $[n]$. This structure can be implemented by combining a hashtable to maintain the elements and a linked list to maintain the element order.

In the pseudocode, setarrays are used to represent both the instantiation and the domains:

1. The instantiation $\vec{a}$ is a setarray of pairs $m = (x, v)$, where $x \in X$ and $v \in \mathcal{D}(x)$.
2. The instantiation's scope, $\pi_{\mathcal{S}}(\vec{a})$, is a setarray of variables.
3. Each domain $\mathcal{D}(x)$ is a setarray of containers.

The algorithm for picking a container as a candidate value has been implemented by choosing elements according to some order. The search space will differ depending on the chosen order among containers. Since complete solutions are desired, the goal is to place domain values in an order, which maximizes the probability of reaching a complete solution. An example is to always attempt to place IMO containers first, in order to ensure that IMO containers are stowed as close together as possible. Containers in a domain are sorted based on the label of the container. The idea is that, since the label is an array of bits and thereby represents a value, the organization of the flags can be ordered such that the most important criteria will appear first in the setarray. Given that elements in the setarray are ordered in an increasing order of their label, the most important criteria within a label must be placed at the least significant bit position while the second most important criteria is

placed at the second least significant bit position and so forth. Using the organization of the bits within a label gives the advantage that sorting based on the label flag can be done without an additional cost of constructing some value representing the order, as the order is directly represented. In addition to the label used as a sorting criteria, the discharge port is added as a second criteria. This can be used, to ensure that containers with an upcoming discharge port, are stowed in the top of stacks.

7.3 Algorithm

The pseudocode is written with the intention of explaining how the optimal solution can be found. It is kept at an abstract level, such that language specific details does not clutter the important details. Details about the actual implementation can be found in the appendix A

Domain management function

The usage of shared domains requires that a domain can be retrieved by the label of the domain. A domain function $\mathcal{M}$ is introduced to retrieve a domain based on its label. However, to avoid introducing a function that extracts the label from a variable, the domain function maps a variable x, to a domain by the label associated with variable x.

Two additional procedures, REINSERTDOMAINVALUE and REMOVEDOMAINVALUE, have been introduced in order to maintain the shared domains, by inserting or removing a domain value for some shared domain respectively. These procedure take a value v as input and based on the label associated with v, all shared domains, which can accommodate that value will be affected. The pseudocode for REINSERTDOMAINVALUE and REMOVEDOMAINVALUE can be found in appendix C.3.

7.4 Search

The search section is divided into two subsections. The first part describes the algorithms to find a single solution within the search space. The second part presents the algorithm to find the optimal solution within the search space.

Maintenance of search state

A global array $searchstate$ maintains information on how much of the search space has been traversed. Each element $searchstate[x]$ contains the pointer to the domain value it currently considers as the candidate value, within the domain associated to x for variable x. The details of maintaining the search state can be found in section 7.1.

7.4.1 Single solution search

Three backtrack algorithms have been implemented with variation on how the next variable for instantiation is being selected. The similarity of these three algorithms has been extracted into the procedure SINGLESOLUTIONSEARCH and each variation is outlined in its own procedure. The SINGLESOLUTIONSEARCH is a modification of a standard backtrack algorithm, which can be found in [2].

Given a partial instantiation $\vec{a}$, a cost F and a SELECTNEXTVARIABLE function, the algorithm searches in a problem instance, until a solution is found which has better cost than F, or concludes that no more solutions could be found. Once a complete solution is found, the algorithm returns the solution found. SELECTNEXTVARIABLE is a variable ordering function, which sets the strategy on how the next variable should be chosen.

SINGLESOLUTIONSEARCH relies on the following procedures, described in details later:

- SELECTANDASSIGNVALUE
  Extends $\vec{a}$ with the assignment of a given variable x and carries out propagation to enforce consistency. A boolean is returned to indicate wether $\vec{a}$ could be consistently extended.

- ESTIMATIONCALC
  The estimation calculator maintains the estimators and based on the given partial instantiation, it returns an estimated cost of any solution extending it.

```
SINGLESOLUTIONSEARCH(a⃗, F, SELECTNEXTVARIABLE)
1   x ← SELECTNEXTVARIABLE(a⃗)
2   while x ≠ null ∧ S ⊂ X
3      consistent ← SELECTANDASSIGNVALUE(x, a⃗)
4      if consistent ∧ ESTIMATIONCALC(a⃗) < F
5         x ← SelectNextVariable(a⃗)
6      else
7         if S = {}
8            x ← null
9         else
10           REINSERTDOMAINVALUE(v)
11           searchstate[x] ← 0
12           x ← S[ |S| ]
13           a⃗ ← a⃗ \ {(x, π_x(a⃗))}
14  return a⃗
```

Figure 7.4: SINGLESOLUTIONSEARCH

The SingleSolutionSearch algorithm, shown in figure 7.4, traverses through the search space of a problem, using a while loop, which terminates if the selected variable is **null**, or if the instantiation is no longer partial.

The algorithm uses the boolean returned by SELECTANDASSIGNVALUE in line 3, to determine whether the selected value was consistent. If the value is consistent and the estimated cost is below the cost F, the search proceeds to the next variable on line 5. Otherwise, the algorithm attempts to perform a backtrack. Line 7 checks whether the instantiation is empty. If this is the case, it means that the no solution has been found, and line 8 sets x to *null* in order to end the while loop. Otherwise a backtrack is performed in lines 10-13. The algorithm returns either a complete or an empty instantiation on line 14.

Select and Assign Value

Given a variable x and a partial instantiation $\vec{a}$, the procedure SELECTANDASSIGNVALUE attempts to assign x in $\vec{a}$. The *exhausted* variable signals if $\vec{a}$ cannot lead to a solution when extended with an assignment of x. Initially *exhausted* is set to $true$. The while loop in lines 2-10 attempts to find a consistent candidate value for x. It runs until either the domain for x is exhausted or a valid candidate value for x is found. In line 3, the domain m is retrieved from the domain manager $\mathcal{M}$, based on the label associated with variable x and a candidate value v is chosen from m as the $searchstate[x]$th element in line 4. The assignment is added to $\vec{a}$ in line 5 and the pointer for $searchstate[x]$ is incremented in line 6. Line 7 runs the propagators, to check if the assignment causes the domain for any future variable to get exhausted. Lines 9-10 remove the assignment for x in case a future domain is exhausted. Lines 11-14 return whether the assignment of x can lead to a solution. It should be noted that the value v does not need to be removed from the domains since it will be removed when $\mathcal{P}^u$ is applied.

The pseudocode for SELECTANDASSIGNVALUE is shown in figure 7.5.

```
SELECTANDASSIGNVALUE(x, a⃗)
1   exhausted ← true
2   while |M(x)| ≤ searchstate[x] ∧ exhausted
3       m ← M(x)
4       v ← m[searchstate[x]]
5       a⃗ ← a⃗ ∪ {(x, v)}
6       searchstate[x] ← searchstate[x] + 1
7       exhausted ← RUNPROPAGATORS(a⃗)
8       if exhausted
9           REINSERTDOMAINVALUE(v)
10          a⃗ ← a⃗ \ {(x, v)}
11  if exhausted
12      return false
13  else
14      return true
```

Figure 7.5: The SELECTANDASSIGNVALUE algorithm

Variable ordering heuristics

Different parts of the backtrack algorithms can be changed in order to improve the search. This report focuses on heuristics, which select a good variable ordering. The variable ordering strategies, which are chosen are: Smallest domain first, static order bottom-up and minimum overstow. Each of these ordering functions can be used as the SELECT-NEXTVARIABLE function used by the procedure SINGLESOLUTIONSEARCH.

DVFC

This variable ordering selects the variable with the smallest domain, as the next variable to be instantiated. It is motivated by the fact that all other things being equal, the variable with the smallest domain will have the smallest number of subtrees. Combined with the SINGLESOLUTIONSEARCH this variable ordering heuristic is the DVFC described in [2].

Figure 7.6 shows the pseudocode for the selecting the next variable by using the DVFC heuristic.

```
SELECTVARIABLEDVFC(a⃗)
1  if|S| = |X|
2     return null
3  U ← X \ S
4  j ← 1
5  for i ← 2 to |U|
6     if |M(U[i])| < |M(U[j])|
7        j ← i
8  return U[j]
```

Figure 7.6: The SELECTVARIABLEDVFC algorithm

Static order bottom-up

This approach relies on the Forward Checking approach described by [2]. The algorithm is motivated by the fact that overstows should be avoided. Knowing that the containers in a domain are ordered by label and discharge port, the idea is to assign variables in a bottom up approach on each stack, in an attempt to naturally minimize overstows. SELECTVARIABLEBOTTOMUP implements the variable order such that it looks for the first available variable in a bottom up fashion for each stack. This variable ordering heuristic is also referred to as static order bottom-up. The pseudocode for SELECTVARIABLEBOTTOMUP is shown in figure 7.7.

Minimum overstow

This variable ordering heuristic focuses on minimizing overstowage. The idea behind this technique is to find the variable, which is estimated to give the least amount of overstow, if assigned with the first domain value from the set of all available domain values, maintained in a sorted order on label and discharge port. Using the minimum cost matching algorithm, the optimal stowage can be obtained for a partial instantiation and the variable matched

```
SELECTVARIABLEBOTTOMUP(a⃗)
1   for j ← 1 to sc            // iterating through stacks
2     for l ← 1 to |L|         // iterating through cells
3       for i ← 1 to tc_j      // iterating through tiers
4         if x^l_{i,j} ∉ a⃗
5           return x^l_{i,j}
5   return null
```

Figure 7.7: The SELECTVARIABLEBOTTOMUP algorithm

with the first domain value can be easily retrieved.

The following procedures are introduced to ease the description of the SELECTVARIABLEOVERSTOW procedure.

- REPRESENTATIONCOST computes the weights of the edges as described under "How to calculate h_{ov}?" in section 6. The pseudocode is listed in appendix C.
- MINCOSTMATCH solves a minimum cost matching problem given as parameter. It returns the matching as a set of edges, where an edge (c, x) represents the matching between a container c and a cell x. A description on how to solve the minimum cost matching problem is in Chapter 5 of [6].
- FIRST returns the first endpoint of an edge.
- SECOND returns the second endpoint of an edge.

An example of a graph returned by MINCOSTMATCHGRAPH is shown in figure 7.8.

Figure 7.9 lists the pseudocode of the SELECTVARIABLEOVERSTOW procedure. Lines 1-3 initialize the set of unassigned variables, the set of containers yet to be placed and the weights for the minimum cost matching graph. The minimum cost matching graph is computed in line 4. Lines 5-7 search for the variable matched with the first container to be placed. In case the first container is not matched to a variable, *null* is returned. This situation occurs, when the overstowage estimation discovers that the current instantiation cannot be extended to a solution.

7.4.2 Depth First Branch and Bound

The implementation of depth first branch and bound, BRANCH&BOUND, relies on two backtrack algorithms, INITIALSEARCH and MAINSEARCH, given as input besides an empty instantiation $\vec{a}$. This technique opens the possibility of giving an optimized *diving heuristic* as the initial search algorithm that finds the first solution. This algorithm may be specialized in finding a very good solution, but may be too slow to traverse the entire problem in

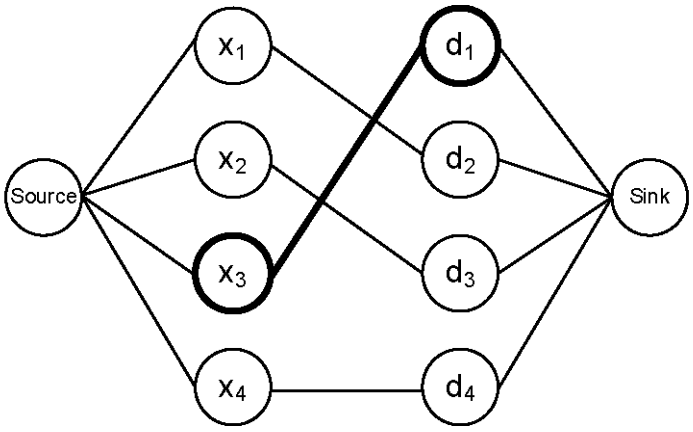


Figure 7.8: DivingOverstowSearch graph returned by MINCOSTMATCHGRAPH. The graph shows a sample query, imagining that d_1 is the first domain value in the domain. The graph shows that the variable to be instantiated with d_1, in order to create the least amount of overstow is x_3.

```
SELECTVARIABLEOVERSTOW(a⃗)
1   U ← X \ S
2   C^U ← C \ π_S(a⃗)
3   w ← REPRESENTATIONCOST(C^U, U, a⃗)
4   M ← MINCOSTMATCH(C^U, U, w)
5   foreach e ∈ M
6      if FIRST(e) = C^U[1]
7         return SECOND(e)
8   return null
```

Figure 7.9: SELECTVARIABLEOVERSTOW

general. For this reason, the second algorithm would typically be specialized in traversing the search space fast to discover the remaining solutions.

The pseudo code is shown in figure 7.10. BRANCH&BOUND searches for the initial solution by using the procedure given as INITIALSEARCH. If a solutionis found, a better solution is searched by using the procedure given as the MAINSEARCH. The algorithm maintains the variables F^* and *Best* for respectively the best cost and the best solution found so far. Line 3 invokes the initial search. In case an initial solution is found, lines 5 - 13 iteratively search for a new solution. The solution is evaluated in line 6 and if the solution is an improvement, lines 7-9 update the best solution found so far. Lines 10-12 retrieve the last assigned variable and remove its assignment, to allow MAINSEARCH to continue traversing the search space. BRANCH&BOUND returns the best found solution in line 14.

```
BRANCH&BOUND(INITIALSEARCH, MAINSEARCH, a⃗)
1   F* ← ∞
2   Best ← null
3   Sol ← INITIALSEARCH(a⃗, F*)
4   if Sol ≠ null      // A solution was found
5      while Sol ≠ null
6         F ← EVALUATIONCALC(Sol)
7         if F < F*
8            Best ← Sol
9            F* ← F
10        x ← S[ |S| ]     // Unassign the last instantiated variable
11        REINSERTDOMAINVALUE(π_{x}(a⃗))
12        a⃗ ← a⃗ \ {(x, π_{x}(a⃗))}
13        Sol ← MAINSEARCH(Sol, F*)
14  return Best
```

Figure 7.10: The Branch & Bound algorithm

Propagation engine

When a variable is assigned, propagation is performed by the procedure RUNPROPAGATORS, which is the propagation engine in the application. The main part of the propagation engine is to iterate through a list of propagators and apply each propagator in turn. A naive approach has one list, which the engine iterates through causing all propagators to be scheduled both in the best and the worst case. An improved approach over the naive implementation is to use a hashtable, which uses the label $\mathcal{L}$ of a container as key and the associated value is a list of propagators, which should be executed for $\mathcal{L}$, ensuring that only the propagators necessary for $\mathcal{L}$ are executed.

Evaluation calculator

The cost of a solution is computed by the procedure EVALUATIONCALC, which takes a solution as an argument. The procedure sums over a predefined set of evaluation functions. Each result from the evaluation function is multiplied with some weight.

OVERSTOW$(\vec{a})$ and WASTEDSPACE$(\vec{a})$ are detailed in appendix $C.1$.

Estimation calculator

The estimation of a partial solution is carried out by the procedure ESTIMATIONCALC. The procedure takes a partial instantiation as input and sums over a predefined set of estimator functions. Each result from the estimation function is multiplied with a weight, predefined according to relevance.

procedure EVALUATIONCALC($\vec{a}$)
1 **return** $W_{ov} * \text{OVERSTOW}(\vec{a})$
$+ \quad W_{ws} * \text{WASTEDSPACE}(\vec{a})$
$+ \quad W_{es} * \text{EMPTYSTACK}(\vec{a})$
$+ \quad W_r * \text{REEFER}(\vec{a})$

Figure 7.11: Cost evaluation for $\vec{a}$.

procedure ESTIMATIONCALC($\vec{a}$)
1 **return** $W_{ov} * \text{ESTIMATEOVERSTOW}(\vec{a})$
$+ \quad W_{ws} * \text{ESTIMATEWASTEDSPACE}(\vec{a})$
$+ \quad W_{es} * \text{ESTIMATEEMPTYSTACK}(\vec{a})$
$+ \quad W_r * \text{ESTIMATEREEFER}(\vec{a})$

Figure 7.12: Cost estimation for $\vec{a}$.

ESTIMATEOVERSTOW($\vec{a}$) and ESTIMATEWASTEDSPACE($\vec{a}$) are detailed in appendix $C.2$.

Chapter 8

Experiments

This chapter describes the experiments, which were conducted to investigate the performance of the implementation and being able to identify characteristic, which could be decisive for the performance. The section begins by describing the test components available for experimentation and an explanation of what is measured. Then experiments are performed to investigate various aspect of the implementation. Each experimental section is build up with a motivation for the experiment. The test setup follows with a conclusion of the result. This chapter shows a method to compute the upper bound the search space. In addition a technique for avoid excessive estimation is being investigated. In addition a method to measure the quality of alternative heuristics is also being provided.

8.1 Test components

This section presents the individual parts, which are used to conduct the experiments. The presentation begins with the test data, then propagators and early termination criteria are described. The various search algorithms are then presented and the section concludes with a description of what is to be measured.

8.1.1 Test data

For testing purposes the structural layout of a vessel has been generated in order to simulate a realistic size vessel. The vessel will be referred to as **VESSEL-1**. The vessel is divided into 18 bays numbered from 0 to 17. The widest bay is eighteen stacks wide and the thirteen tiers deep below deck at the deepest level. In addition, a stowage plan for **VESSEL-1** has been generated. However it soon proved that we were not able to solve the SASP within reasonable time and therefore an alternative had to be considered.

A smaller vessel, **VESSEL-2**, has therefore been constructed such that the properties of the bays varies from one another in order to ensure different aspects can be examined. Furthermore, it is also required that the bays could be solved within a reasonable amount

of time. Table 8.1 summarize the most important properties of the bays in **VESSEL-2**. The two first columns show the dimensions of each bay. The third column shows how many cells are available for placing a container. The fourth and fifth column shows the number of 20-foot and 40-foot container, and the last column shows how many different discharge ports are there for the containers to be loaded.

	Tiers	Stacks	Available cells	20-foot	40-foot	Number of discharge ports
Bay V	2	2	7	3	1	2
Bay S	5	3	22	5	5	2
Bay B	5	3	25	8	7	6
Bay A	5	4	33	4	10	6

Table 8.1: Vessel data for **VESSEL-2**

Propagators and early termination criteria

The constraints of SASP are represented by propagators and early termination criterions, which are described in details in section 5.4 and 5.5 respectively. A standard set of propagators are defined to ensure that the constraints of SASP is respected. The standard set is defined in table 8.2.

Table 8.2: standard propagator sets

PRS1	$\{\mathcal{P}^{u}, \mathcal{P}^{g}, \mathcal{P}^{a}, \mathcal{P}^{\text{IMO-1}}, \mathcal{P}^{\text{IMO-2}}, \mathcal{P}^{\text{40-20}}, \mathcal{P}^{\text{20-40}}, \mathcal{P}^{\text{40-40}}, \mathcal{P}^{\text{20-20}}, \mathcal{P}^{a\text{-}s}\}$

No aspects in relation to early termination criteria was considered and therefore all early termination criterions are used for all experiments.

8.1.2 Search

The purpose of the experiments is to reason about the search space and see how efficient the implemented search algorithms are. The efficiency of the search algorithm can be examined on two aspects: ”How fast can a single solution be found?” and ”How fast can the entire search space be traversed?” This section begins with an explanation on how to quantify the search space followed by a presentation of the different algorithms to find a single solution. The search section is concluded with a presentation of the branch and bound variant used in the experimentation.

Bounding search space

The upper bound of the search space gives an idea of how vast the search space is. Having an upper bound, makes it possible to reason about how much of the search space gets

pruned away when using different heuristics. Throughout the experiments the common measure is the number of iterations. This corresponds to the number of nodes visited in the search space and can be compared to the theoretical upper bound.

As described in section 2.2, the search space is shaped as a tree where each node represents an assignment of some variable x and branches represents the candidate values for x. One possible way to bound the number of complete instantiations of the tree for SASP is to let the air value be unique, such that the number of possible candidate values always will be the number of available cells. The number of solutions can then be calculated by the number of unique orderings of the candidate values including the unique air values. The number of complete assignments for the cell-model is therefore upper bounded by $O(|X|!)$. However the bound is quite weak, since air is not unique and therefore branches, where air has been considered for a variable should be considered as being the same subtree. A tighter bound can be derived by following consideration. Having i cells and j containers we assume three cases:

1. having exactly enough slots to place all containers
2. more slots available but no more containers to be placed
3. more slots available than containers to be placed.

For convenience the case where there are more containers than slots is not being considered.

In the case where there are enough slots to place all containers the number of possibilities for placing all containers will be a simple permutation. When there are no more containers to be placed, only one possibility is left, which is to fill the rest of the slots with air. In the case, where slots exceeds the number of containers, one has the choice to either choose to place a container or air. For each choice a subtree can be constructed with the remainding possibilities. If a container has been chosen a subtree with $j-1$ container has to be placed within $i-1$ slots, while chosen air to be placed creates a subtree with j containers to be placed $i-1$ slots. This leads to following recursive function:

Let i be the number of cells, let j be the number of containers. $\psi(i,j)$ gives the number of complete assignments.

$$\psi^P(i,j) = \begin{cases} j! & j = i \\ 1 & j = 0 \\ j(\psi^P(i-1,j-1)) + \psi^P(i-1,j) & \text{otherwise} \end{cases}$$

The number of nodes in the search tree can be counted by using similar argumentation and results in the following function:

$$\psi'(i,j) = \begin{cases} 1 + \sum_{k=1}^{j} \prod_{l=k}^{j} l & j = i \\ i & j = 0 \\ j + 1 + j(\psi'(i-1,j-1)) + \psi'(i-1,j) & otherwise \end{cases}$$

The function ψ' computes the number of nodes a subtree excluding the root. The base case $i = j$ represents the number of nodes where j containers have to placed within i cells. The tree is $i + 1$ high and have $j!$ paths. From the root the number of containers available for selection is j, which results in j new nodes. At the next level all nodes will have the possibility to choose between $j-1$ containers, which result in $j(j-1)$ new nodes. Extending this argument the number of nodes at level k is the product from j down to k i.e. $\prod_{l=k}^{j}$. The sum of the nodes in the entire tree will therefore be the root node in addition to the sum of nodes at each level in the tree. In the case where there are no more containers left to be placed the remaining cells will be placed with air thus there will be i nodes left. In the recursive case either j containers or air can be placed a cell, which results in $j + 1$ possibilities for each cell. Placing a container in a cell results in j subtrees with one less container to be placed in one less slot i.e. $\psi'(i-1, j-1)$. Placing an air container results in a subtree where one less slot can be used i.e. $\psi'(i-1, j)$.

$$\psi^N(i, j) = 1 + \psi'(i, j)$$

The function ψ^N computes the number of nodes in a subtree including the root.

For some of the bays it was possible to traverse the entire search space and thereby count the number of solutions. Table 8.3 show the number of solutions and the number of how consistent solutions and the percentage.

Table 8.3: Solutions

Bay	Candidate solutions	Complete instantiations	Percentage
Bay S	336	3,160,080	1.06E-04
Bay B	2880	870,072,320	3.31E-06
Bay A	23040	150,994,944	1.53E-04

Searching for a single solution

Different heuristics have been considered in order to improve the efficiency of the backtrack algorithms. The variants used in this report are all based on variable ordering as described in section 7.4.1. The variants of the backtrack algorithms used in this report can be found in table 8.4.

Table 8.4: Single solution varieties

Name		Select variable procedure
DVFC	:	SELECTVARIABLEDVFC
Forward checking	:	SELECTVARIABLEBOTTOMUP
Overstow	:	SELECTVARIABLEOVERSTOW

Traversal of the search space

The implemented branch and bound applies the diving heuristic by take two algorithms as input: One which is efficient for finding a good solution the other for efficiently traversing the search space. When performing experiments related to traversal of the search space the DVFC is used as the initial search algorithm and as the main search.

8.1.3 Measurements criteria

After an experiment has been executed data is being gathered gathered and analyzed. Depending on what is required to be examined, the measurement is based on one or more of the following criterions:

- Number of iterations
  In a search an iteration is a step for extending a consistent partial instantiation to another consistent partial instantiation by one variable. The unit is defined to be defined as one variable. Some propagators have been enhance to automatically assigned a variable when there is only a single domain value left to consider, however these are not considered as an iteration step.

- Number of backtracks
  In search the number of backtrack is defined as the number of times some variable is being reconsidered for another value after it has been assigned to a domain value, which were consistent. One unit is defined for each time a backtrack is enforced by an inconsistent assignment.

- Duration
  The duration describes the time used to perform the experiment. The unit is miliseconds or seconds.

- Estimation backtracks
  An estimation backtrack refers to a partial solution, for which the estimated cost is higher than the current best cost. This results in a backtrack, which is not inferred from a domain exhaustion, but due to the estimators. The unit is defined each time an estimator forces a backtrack.

8.2 Propagator improvements

Although the initial developed propagators modeled the constraints in SASP, the amount of pruning was not sufficient. Several measures have been done to strengthen the pruning power of propagators. One heuristic is to assign containers to cell immediately, when there is only one value left to be selected. The propagators, that could be improved by this heuristic, are $\mathcal{P}^{a}$ and $\mathcal{P}^{40\text{-}40}$, which are denoted as $\mathcal{P}'^{a}$ and $\mathcal{P}'^{40\text{-}40}$ respectively. Furthermore

a weight propagator $\mathcal{P}^{a\text{-}w}$ was implement as well, which is described in more details in section 5.4.

8.2.1 Searching for a single solution

The motivation for doing a single solution test is to examine if the improvement of the new propagators has any effect, when searching for a single solution in the search space. The setup can be seen in table 8.5. The bays chosen has been based whether a solution could be found within reasonable time.

Table 8.5: Setup for finding a single solution using different propagators

Search algorithm	:	DVFC
Problem instance	:	Bay V, Bay S, Bay B, Bay A, Bay 0, Bay 1, Bay 2, Bay 6, Bay 16
Propagators	:	$\mathcal{P}^{u}, \mathcal{P}^{g}, \mathcal{P}^{a}, \mathcal{P}^{\text{IMO-1}}, \mathcal{P}^{\text{IMO-2}}, \mathcal{P}^{\text{40-20}}, \mathcal{P}^{\text{20-40}}, \mathcal{P}^{\text{40-40}}, \mathcal{P}^{\text{20-20}}, \mathcal{P}^{a\text{-}s}, \mathcal{P}^{a\text{-}w}, \mathcal{P}'^{a}, \mathcal{P}'^{\text{40-40}}$
Measurement	:	Iteration, Backtrack

Three new sets of propagators has been constructed, which are shown in the table 8.6. **PRS2** is the set, which replaces the propagators $\mathcal{P}^{a}$ and $\mathcal{P}^{\text{40-40}}$ with the improved propagators $\mathcal{P}'^{a}$ and $\mathcal{P}'^{\text{40-40}}$. **PRS3** and **PRS4** are the sets **PRS1** and **PRS2** respectively with an additional propagator $\mathcal{P}^{a\text{-}w}$.

Table 8.6: propagator sets

PRS2	$\{\mathcal{P}^{u}, \mathcal{P}^{g}, \mathcal{P}'^{a}, \mathcal{P}^{\text{IMO-1}}, \mathcal{P}^{\text{IMO-2}}, \mathcal{P}^{\text{40-20}}, \mathcal{P}^{\text{20-40}}, \mathcal{P}'^{\text{40-40}}, \mathcal{P}^{\text{20-20}}, \mathcal{P}^{a\text{-}s}\}$
PRS3	$\mathbf{PRS1} \cup \{\mathcal{P}^{a\text{-}w}\}$
PRS4	$\mathbf{PRS2} \cup \{\mathcal{P}^{a\text{-}w}\}$

Conclusion on finding a single solution

The results are shown in table 8.7. As the table shows set **PRS2** yields a lower amount of iterations, but has a higher amount of backtracks compared to **PRS1**. This is to be expected, as immediately assigning variable avoids instantiating through select value calls, but does not avoid backtracks if the assignment triggering instant assignments is invalid.

Set **PRS3** shows no improvement over **PRS1**, when searching for a single solution. This could be explained by the low amount of iterations.

	PRS1	PRS2	PRS3	PRS4
Bay A				
Iterations	9659	**7219**	9659	**7217**
Backtracks	4813	**4517**	4813	**4517**
Bay B				
Iterations	213	**159**	213	**159**
Backtracks	94	**92**	94	**92**
Bay S				
Iterations	28	**15**	28	**15**
Backtracks	0	0	0	0
Bay V				
Iterations	7	7	7	7
Backtracks	0	0	0	0
Bay 0				
Iterations	22	**13**	22	**13**
Backtracks	0	0	0	0
Bay 1				
Iterations	53	**30**	53	**30**
Backtracks	1	**0**	1	**0**
Bay 2				
Iterations	80	**46**	80	**46**
Backtracks	0	0	0	0
Bay 6				
Iterations	172	**46**	172	**46**
Backtracks	0	0	0	0
Bay 16				
Iterations	89	**53**	89	**53**
Backtracks	9	9	9	9

Table 8.7: Result of Single solution search with propagator sets. Bolded entries are the best for entry for each bay.

8.2.2 Traversal of the search space

Since searching for a single solution did not show any improvements, it was considered, whether it was due to the fact that the number of iterations were too small. As an alternative an additional test has been setup to investigate if that had been the case. In order not to prune any solutions away, the branch and bound was carried out without using estimations. The bays on **VESSEL-1** were too large to to be able to complete and consequently it was only bays from **VESSEL-2**, which was subjected to experiments. The test setup is shown in table 8.8. A description of the sets of propagators used for the experiment can be found in table 8.6.

Table 8.8: Test setup

Search algorithm	:	Branch & Bound
Estimators	:	-
Problem instance	:	Bay V, Bay S, Bay B, Bay A
Propagators	:	$\mathcal{P}^{u}$, $\mathcal{P}^{g}$, $\mathcal{P}^{a}$, $\mathcal{P}^{\text{IMO-1}}$, $\mathcal{P}^{\text{IMO-2}}$, $\mathcal{P}^{\text{40-20}}$, $\mathcal{P}^{\text{20-40}}$, $\mathcal{P}^{\text{40-40}}$, $\mathcal{P}^{\text{20-20}}$, $\mathcal{P}^{a\text{-}s}$, $\mathcal{P}^{a\text{-}w}$, $\mathcal{P}'^{a}$, $\mathcal{P}'^{\text{40-40}}$
Measurement	:	Iteration, Backtrack

Conclusion of the traversal of the search space

Shown in table 8.9 is the result of iterations and backtracks made, when using the propagator set on each problem. The table shows that set **PRS2** does not always perform better than **PRS1**, which can be interpreted as the improvements is weak compared to simply just evaluating whether the current weight of the stack exceeds the maximum allowed weight. This is to be expected, as the improvement cannot trigger backtracks, unless the weight for the containers placed in the stack is close to the maximum weight of the stack. **PRS4** has the most bolded entries, showing that the best propagator set is the set containing all improved propagators.

	PRS1	**PRS2**	**PRS3**	**PRS4**
Bay V				
Iterations	351	**339**	351	**339**
Backtracks	**146**	**146**	**146**	**146**
Bay S				
Iterations	96,111	76,503	85,983	**71,751**
Backtracks	47,816	46,744	**42,824**	43,624
Bay B				
Iterations	105,403	82,917	86,779	**69,525**
Backtracks	50,590	50,046	41,950	**41,246**
Bay A				
Iterations	10,023,317	**7,932,603**	10,023,317	**7,932,603**
Backtracks	5,000,139	**4,881,829**	5,000,139	**4,881,829**

Table 8.9: Result of branch and bound with propagator sets. Bolded entries are the best for entry for each bay.

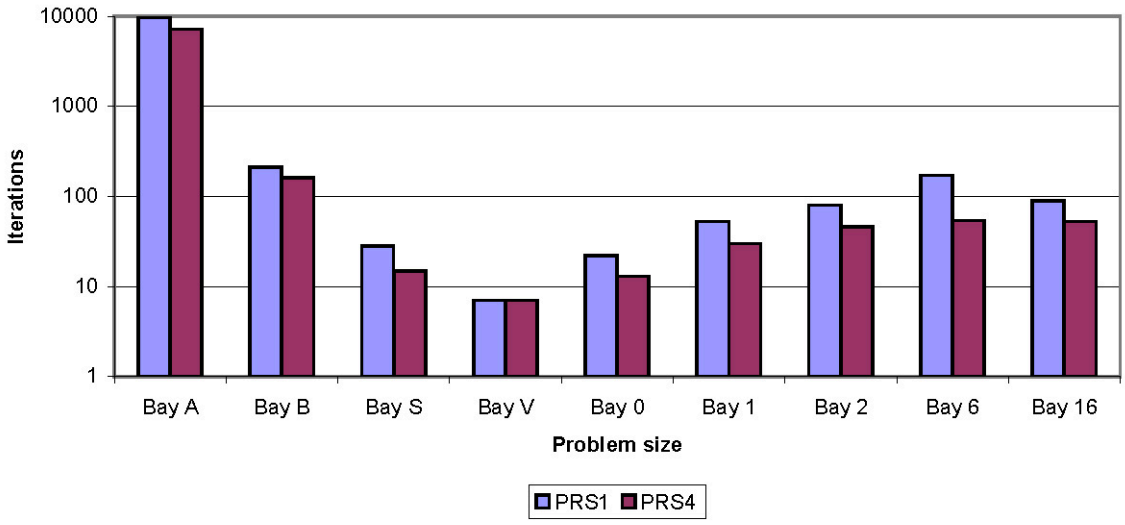


Figure 8.1: A histogram of the two propagator sets **PRS1** and **PRS4** for a range of different bays. Each column in the diagram shows the number of iterations for each propagator set on a given problem.

Figure 8.1 shows that the sets **PRS1** and **PRS4** follow each other very closely, showing that the improvement is merely by a constant factor. This is to be expected, as the pruning performed by **PRS4**, mainly concerns skipping iterations, by immediately assigning variables when possible. Although the problem size of some problems are indicated to be smaller, the amount of iterations spent before finding a solution varies. This shows that although the problem is smaller, the initial solution is not always present early in the search space.

8.3 Estimators

8.3.1 Traversal of the search space

Experimentation with the estimators focuses on how different variations of estimators would affect the search. Another interesting observation is to investigate how early the estimators are able to backtrack in order to reason how much the underestimation is. The estimators is based on the four objectives **OE1-OE4** defined in SASP, which are detailed in section 3.2.

For each objective, a weight is provided to express the impact of violating one of the objectives. The weight is expressed in terms of dollars and are meant to show the economical impact, when not satisfying a given objective. The provided weights are shown in table 8.10.

The test setup is shown in table 8.11. The sets of estimator combinations are shown in table 8.12. It should be noted that the implementation of h_{es} and h_{ws} has been merged into one for efficiency and therefore only eight instead of 16 sets have been constructed.

Table 8.10: Weights

Overstow	W_{ov}	=	200
Wasted space	W_{ws}	=	100
Reefer	W_r	=	50
Empty stack	W_{es}	=	200

Table 8.11: Setup for estimation

Search algorithm	:	DVFC
Estimators	:	$h_{ov}, h_r, h_{es}, h_{ws}$
Problem instance	:	Bay S, Bay B, Bay A
Propagators	:	**PRS1**
Measurement	:	Iteration, Duration, Estimation backtrack

Conclusion for traversal of the search space for estimation improvements

The result of the test execution is shown in table 8.13 with the different estimator setups given in each column. An initial test run are given in the first column, where the algorithm are run without using estimators. As shown in the table, the implemented estimators does not always yield a better cost, compared to not using any estimation at all. Especially the overstow estimator, when running alone, which uses very little time on the first two bays, but suddenly jumps to use the most time among all the estimators when run on Bay S, surpassing even the approach of not running estimators.

The reason for this is that the containers to be loaded in Bay S have almost no variation on their discharge port, which makes the overstow estimator merely as extra workload. This affects all the estimator setups that the overstow estimator is part of for Bay S, which shows that the overstow estimator furthermore is dominant on the time it uses on each iteration. From this it can be concluded that although the estimation techniques allows to skip solutions, the extra work they infer does not necessary pay off.

8.4 Lazy Estimation

8.4.1 Traversal of the search space

A partial solution might be too small for the estimators to calculate a cost, which is close to any solution that can be extended from it. For that reason, if estimation is started too early, it might introduce a substantial overhead. To avoid this, the idea of *lazy estimation* is introduced, which refers to avoiding of estimation, until a certain percentage of the variables in a given problem instance are assigned. The motivation is to examine whether it lazy estimation is beneficial. Setup for lazy estimation can be found in table 8.14. The chosen bays are those that are able to find a solution within reasonable time for this experiment.

Table 8.12: Estimator sets

ERS1	:	-
ERS2	:	h_{es}, h_{ws}
ERS3	:	h_{ov}
ERS4	:	h_r
ERS5	:	h_r, h_{ov}
ERS6	:	h_{ov}, h_{es}, h_{ws}
ERS7	:	h_r, h_{es}, h_{ws}
ERS8	:	$h_r, h_{ov}, h_{es}, c_{ws}$

	ERS1	ERS2	ERS3	ERS4	ERS5	ERS6	ERS7	ERS8
Bay B								
Iterations	69,525	714	**512**	35,168	8,038	**512**	35,168	8,062
Duration	2,750.018	**109.376**	203.126	1,546.885	2,843.768	125.001	2,171.889	3,046.895
Estimation Backtracks	0	165	35	**5,056**	1,266	35	**5,056**	1,266
Bay A								
Duration	7,932,603	**98,769**	274,214	3,167,664	401,881	259,022	2,823,876	476,231
Time spent	305,970.708	**7,484.423**	112,828.847	153,672.859	158,766.641	116,188.244	199,829.404	201,729.166
Estimation Backtracks	0	22,389	43,397	**375,678**	57,586	40,191	359,922	62,735
Bay S								
Iterations	7,1751	42,074	**27,782**	56,808	42,504	32,196	61,308	42,281
Duration	2,687.517	**2,156.264**	7,078.170	2,593.767	9,937.564	8,578.180	3,671.899	10,747.179
Estimation Backtracks	0	2,661	1,600	**5,399**	2,493	2,018	1,266	2,357

Table 8.13: Estimation setup results

The experiments for lazy estimation is being performed by performing lazy estimation ranging from starting estimation when 100% of the variables has been assigned to 0% when no variable has been assigned. It should be noted that having set the threshold to 100% implies that estimations is never carried out, 45% implies that estimation is only performed for partial instantiations were more than 45% of the variables have been assigned and 0% indicates the estimation is performed for any partial instantiation.

When comparing two setups, where one is using a lazy estimation percent greater than zero, and the other is performing estimation all the time, the setup using lazy estimation will always use at least as many or more iterations as the other setup, since it will at most be able to perform estimation backtracks as often as when performing estimation for all partial instantiations. For this reason, the only criteria for comparison that remains is the time spent, for each of the two setups.

Line 4 in SINGLESOLUTIONSEARCH needs to be modified to incorporate lazy estimation for a threshold of lz which are shown in figure 8.2

Table 8.14: Test setup

Search algorithm	:	Branch & Bound
Estimators	:	$h_{ov}, h_r, h_{es}, h_{ws}$
Problem instance	:	Bay S, Bay B, Bay A
Propagators	:	**PRS1**
Range(%)	:	0 - 100
Measurement	:	Duration

4 **if** $consistent \ \wedge (\frac{|\mathcal{S}|}{|X|} \geq lz \ \vee \ \text{ESTIMATIONCALC}(\vec{a}) < C)$

Figure 8.2: INCORPORATION LAZY ESTIMATION IN SINGLESOLUTIONSEARCH

Conclusion of traversing the search space for lazy estimation

Shown in figure 8.3 is test results for lazy estimation for Bay A. The initial bump in the graph is the time taken for the estimator setups involving the estimation for overstow. This shows that the estimators are actually started too late and that the time spent on calculating the estimated cost is slower than simply searching without estimators in the remaining subtree of that partial solution. As estimation is allowed with less assigned variables, the time taken decreases until a lazy percent 45%, after which increasing the lazy percent does not have any effect on the time taken. As the time for traversing the problem with zero percent for lazy estimation is equal to the lowest time, it can be concluded that lazy estimation does not pay off for this problem instance.

As figure 8.4 shows **ERS2** and **ERS3** performs the best for Bay B, once the lazy estimation percent is below 55%. Combined with the results from the previous graph, this suggest that these estimators actually perform the best when started early with a decreasing performance as the lazy estimation percentage increases.

ERS8 performs almost as bad as not using estimators (100% lazy percent), regardless of the lazy estimation percentage. This shows that although some of the estimators, when tested alone performs well, their performance is penalized if combined with an estimator performing poorly for a given problem instance.

In figure 8.5 for the final problem instance, **ERS8** performs worse than not running estimators at all at 100% lazy estimation.

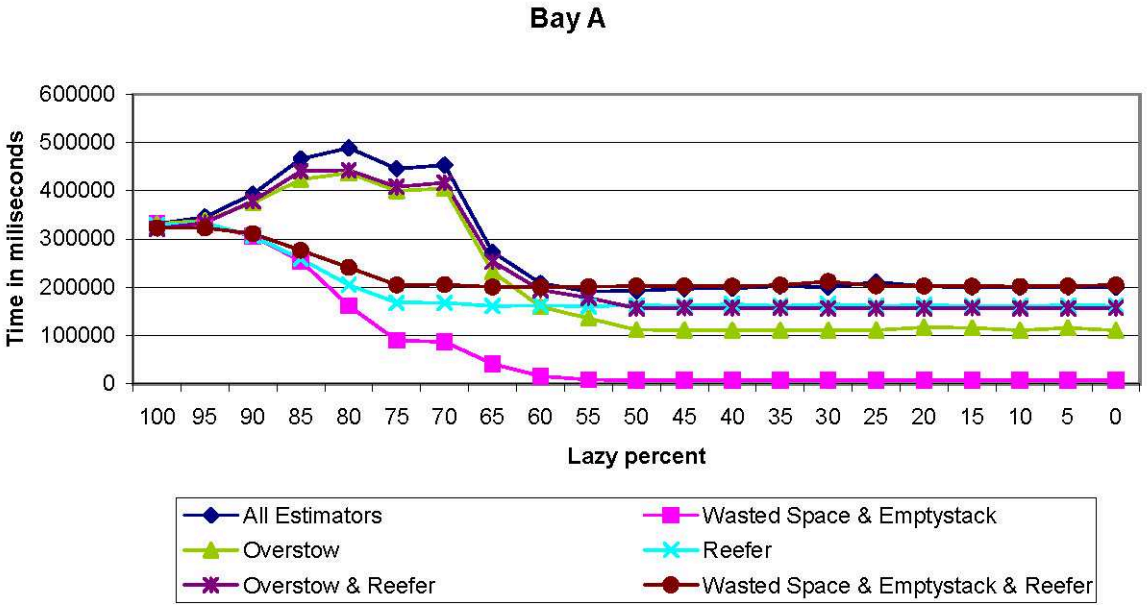


Figure 8.3: Result

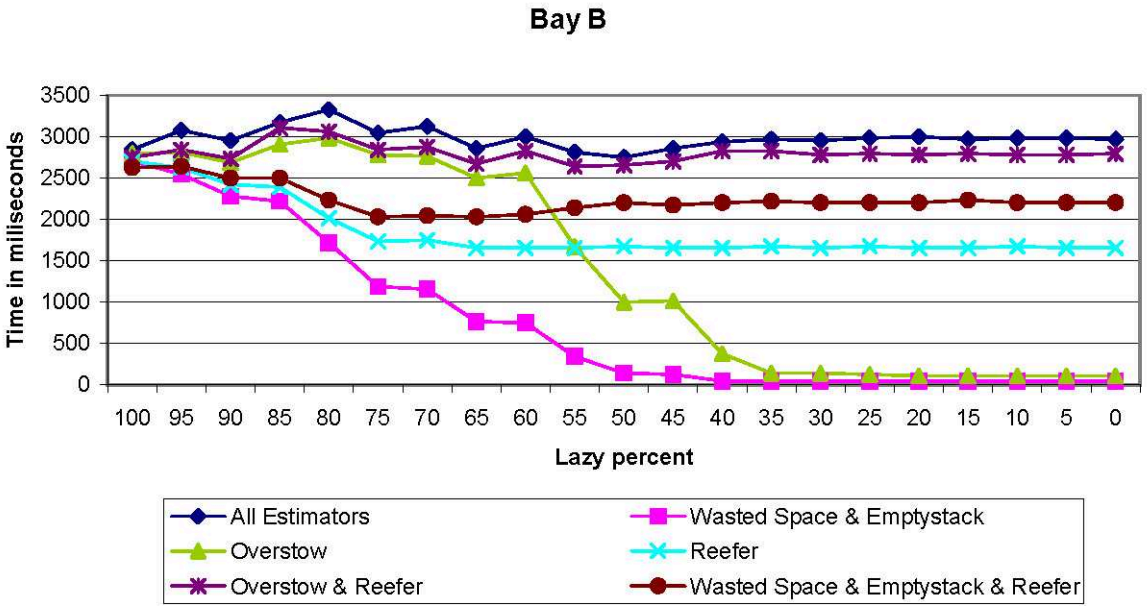


Figure 8.4: Result

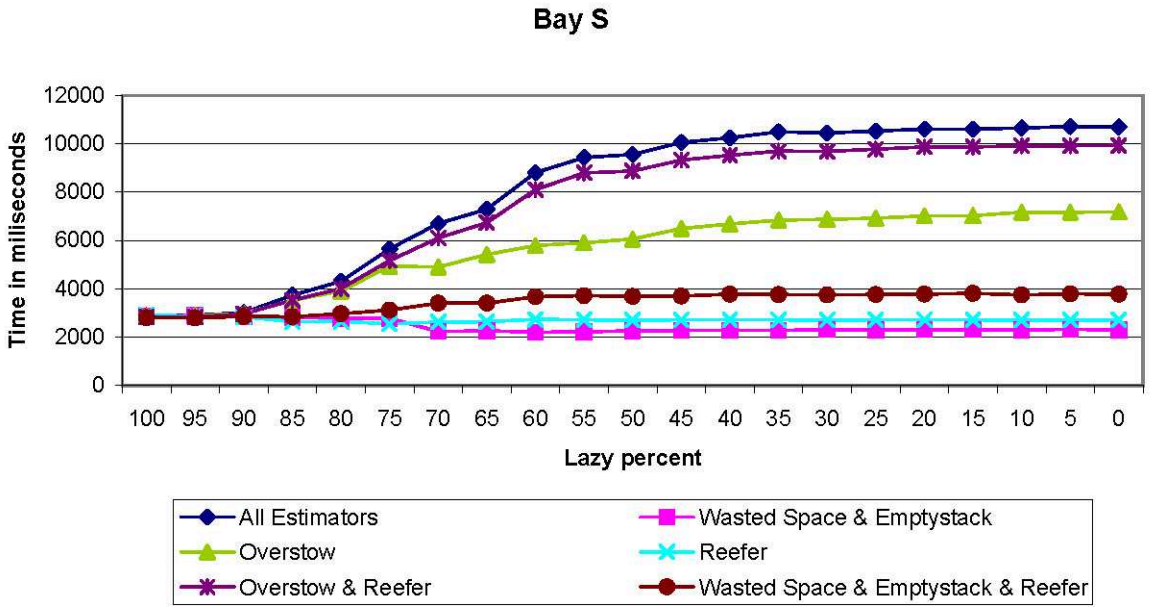


Figure 8.5: Result

This could explain the reason for why the overstow estimator works poorly as the estimator may estimate a very low cost until a container with high discharge port is assigned to a cell late in the search space.

8.5 Approximation

8.5.1 Traversal of the search space

The branch and bound algorithm can easily be made into an approximation algorithm, by adding a percentage to the cost calculated by the estimators. The motivation for this experiment is to investigate how approximating the estimated cost of a partial instantiation would affect the performance of the search at the expense of optimality.

Line 4 in SINGLESOLUTIONSEARCH needs to be modified to incorporate approximation δ which are shown in figure 8.6

$$4 \qquad \textbf{if } consistent \ \wedge \ (1+\delta) * \text{ESTIMATIONCALC}(\vec{a}) < C$$

Figure 8.6: Incorporation of approximation in SINGLESOLUTIONSEARCH

Using this modification, the goal is to find solution close to the optimal in significantly less time, compared to when searching for the optimal one. The setup for experimentation is shown in table 8.15. The chosen bays are the ones, which a solution could be found in reasonable time.

Table 8.15: Test setup for branch and bound approximation

Search algorithm	:	Branch & Bound
Estimators	:	$h_{ov}, h_r, h_{es}, h_{ws}$
Problem instance	:	Bay S, Bay B, Bay A
Propagators	:	**PRS1**
Range(%)	:	0 - 50
Measurement	:	Iteration, Estimation backtrack

Figure 8.7 shows the results from approximating with an approximation percent ranging from 50-0% on the x-axis and the number of estimation backtracks on the y-axis. Red dots on each line marks when the best found solution is improved.

The number of estimation backtracks used grows steadily in respect to the approximation percent, unless the best found solution is improved. From less estimation backtracks it can be inferred that larger branches of the search space is cut away, which, as expected, shows that approximating the estimated cost enables the algorithm to backtrack sooner in the search space. Once the best found solution is updated, the graph shows a drastic drop

in the number of estimation backtracks performed. This is to be expected, as the estimators will have better terms for triggering a backtrack if the cost of the best found solution is lower. This in turn affects approximating the estimated cost as well.

When approximating by a certain percentage, it is indirectly given that the best solution will be improved in leaps by at least a percentage equal to the approximation percent. Although the chosen problem instances hold many candidate solutions, the graph appears to show very few improvements to the best found solution. While this partially can be blamed on the approximation percentage, it can also be argued that the amount of improvements to the best found solution is low, if the solutions initially found are good.

It can be observed for Bay B, the solution initially found never improves regardless of the approximation percent, which indicate that the solution initially found is the optimal one.

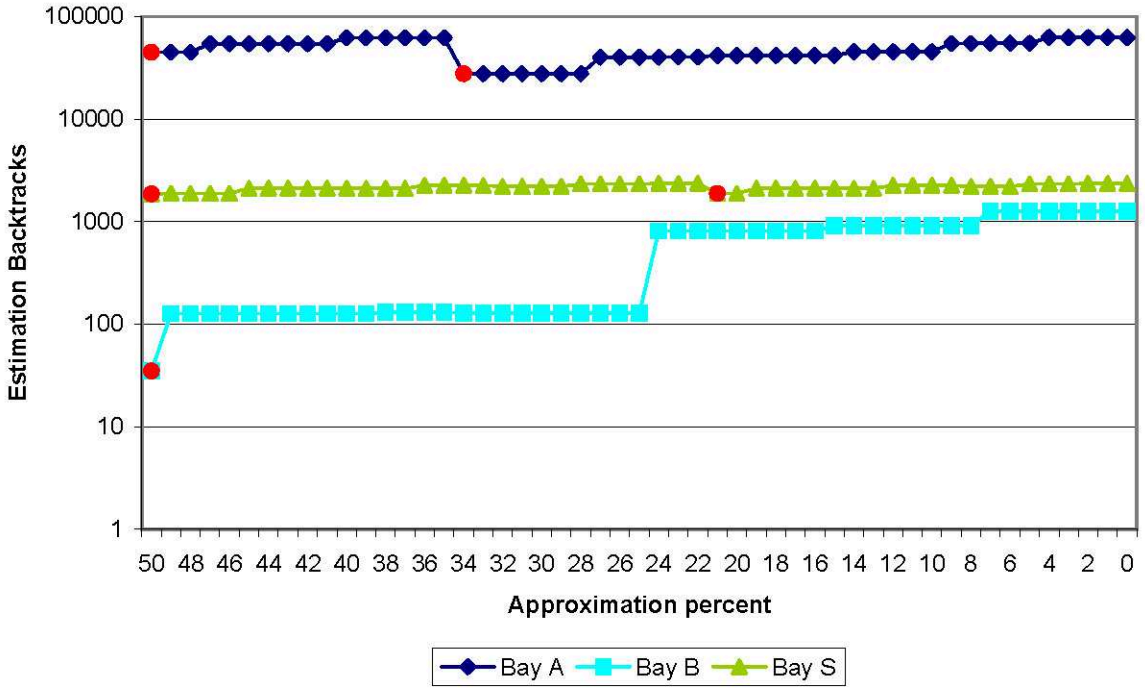


Figure 8.7: Approximating the estimated results from 50-0%, showing the number of estimation backtracks used for three different bays.

Extracted from the approximation tests, Figure 8.8 show the number of iteration for the same problem and the same range as the previous graph. Comparing the two graphs, shows a close correspondence between the measured estimation backtracks and the number of iterations used. From this, it can be concluded that the amount of time used has a close correspondence to the approximation percentage and the best found solution in the problem.

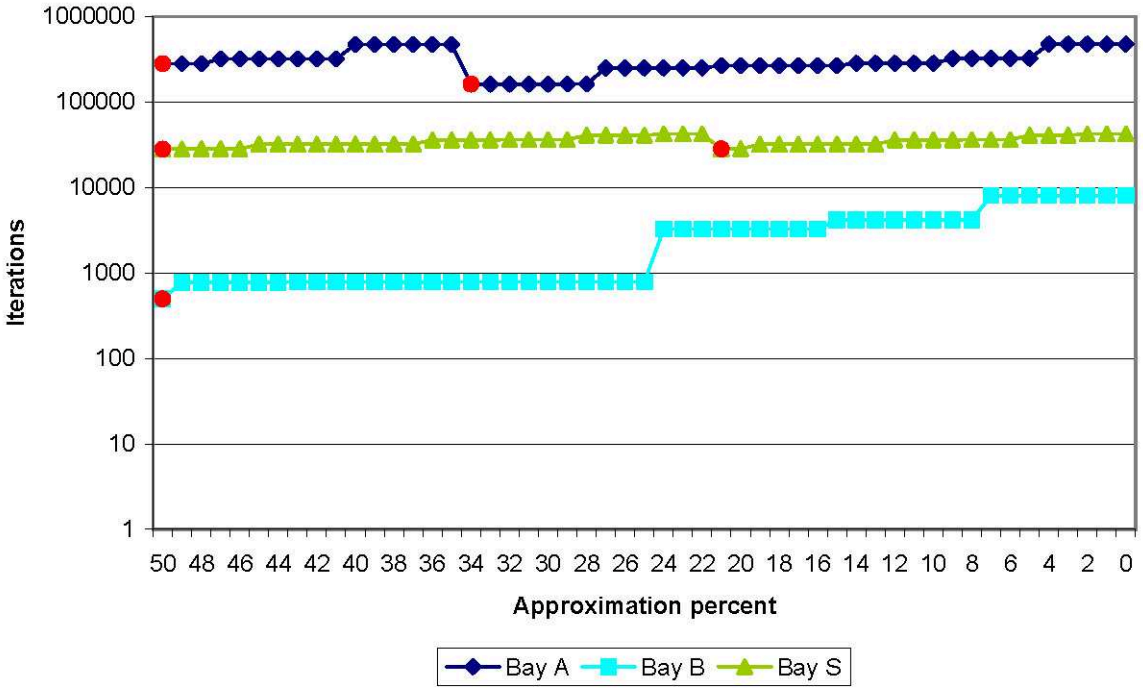


Figure 8.8: Approximating the estimated results from 50-0%, showing the number of iterations used for three different bays.

Conclusion on traversal of search space when using approximation

Based on the test data on which approximation is performed, it can be concluded that approximation has a strong potential, in that the best solution is found even with a high approximation percentage. The amount of test bays on which approximation is performed is however very few and to properly verify the results, a more extensive number of bays is required to be tested. Due to time restrictions, this has however been left as future work.

8.6 Variable ordering

8.6.1 Searching for a single solution

Different variable ordering heuristics has been considered in order to improve the efficiency of finding the initial solution. Three different algorithms has been created for that purpose and a further description can be found in section 8.4. The motivation for this experiment is to measure, whether any of the variable ordering heuristics are beneficial, as the intention with finding an initially better solution is to decrease the total search space.

The other components in the experiment can be found in table 8.16.

Table 8.16: Setup for variable ordering

Search algorithm	:	DVFC, Forward Checking, Overstow
Estimators	:	$h_{ov}, h_r, h_{es}, h_{ws}$
Problem instance	:	**VESSEL-2, Bay0**
Propagators	:	**PRS1**
Measurement	:	Cost, Iteration

Conclusion

The diagram in figure 8.9 shows the cost of the initially found solution for four different bays, when using Forward Checking, DVFC or Overstow as variable ordering. Giving the best cost for all bays except Bay A is forward checking. DVFC and Overstow gives an almost equally good cost, which indicates that the extra work made by the overstow heuristic does not pay off.

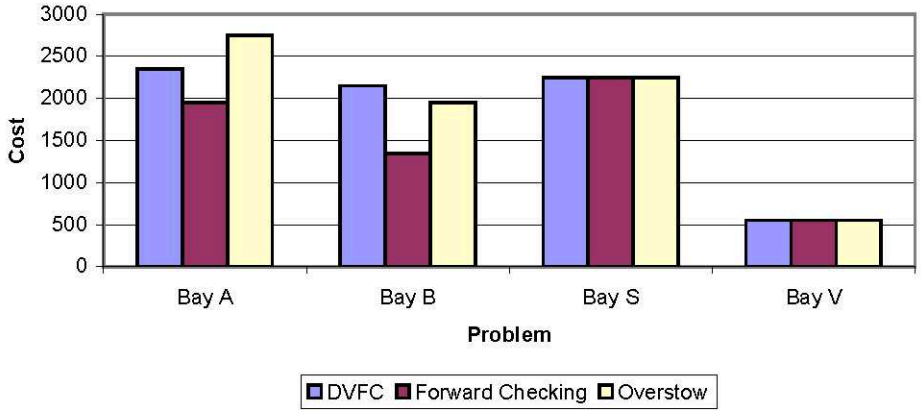


Figure 8.9: Cost of the initial solution found on four different bays

Figure 8.10 shows the number of iterations taken for each of the algorithms for traversing the entire search space in the three different bays. Holding a slight advantage in all bays except for Bay A is the Forward Checking. A reason why the forward checking performs badly on Bay A could involve the fact that the initially found solution has a high cost, as can be seen in the diagram above. Overstow diving does not yield less iterations for each problem compared to using DVFC, from which it together with the results from figure 8.9, can be concluded that this heuristic does not work.

Table 8.17 shows the cost of the initial solution found, when using different variable ordering heuristics and the cost is compared to the optimal solution for each bay in **VESSEL-2**. The intention is to show the diversion between the optimal solution and the solution found by using a particular variable ordering. A small difference shows that the given variable ordering found a solution with a cost close to the optimal.

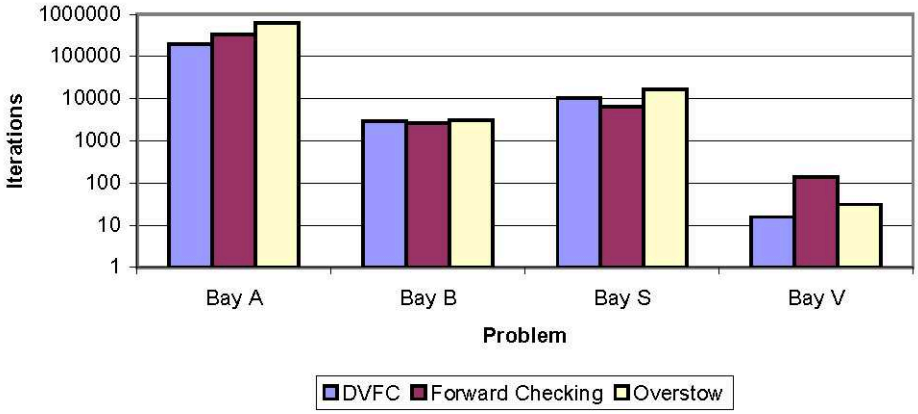


Figure 8.10: Measuring the iterations used before finding the initial solution on three different bays, using forward checking, DVFC and Overstow as diving heuristic

Shown below is the number of iterations used for the four different bays for the different diving heuristic setups in the branch and bound algorithm. As can be seen in the table, none of the diving heuristics finds a solution with a cost closer than 41% to the optimal cost, showing that none of the diving heuristics is capable of finding a solution reasonable close to the optimal one.

	Diving cost				Distance in % to the optimal solution		
	DVFC	Forward Checking	Overstow	Optimal	DVFC	Forward Checking	Overstow
Bay A	2350	1950	2750	1150	51	41	58
Bay B	2150	1350	1950	750	65	44	62
Bay S	2250	2250	2250	2250	0	0	0
Bay V	550	550	550	550	0	0	0

Table 8.17: Diving heuristic evaluation table

8.7 Profiling

A performance analysis has been made of the implementation, in an attempt to identify which parts of the program that is the best candidates for optimization. A profiling tool has been used to gather the data and shows a percentage of processing time used by the different components which the program consist of. The results of the analysis is intended as a basis for future improvement of the processing time. The setup can be found in table 8.18.

Conclusion on profiling

Figure 8.11 shows an overview of the time used by various parts of the program. Taking the most time by far is the estimators with over 80% of the total processing time this makes

Table 8.18: Test setup

Search algorithm	:	Branch & Bound
Estimators	:	$h_{ov}, h_r, h_{es}, h_{ws}$
Problem instance	:	Bay B
Propagators	:	**PRS1**
Measurement	:	Processing time

the component the best candidate for further optimization.

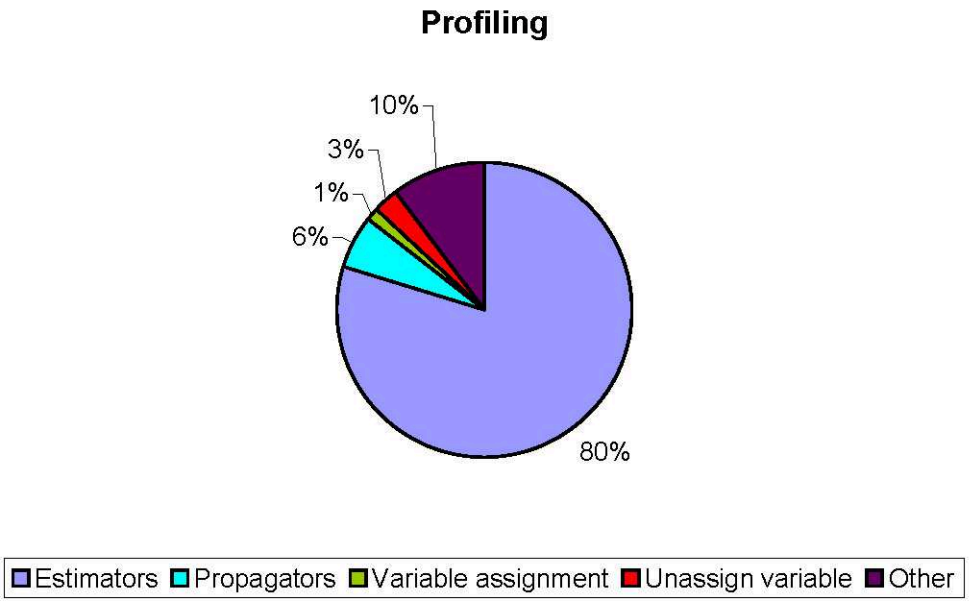


Figure 8.11: Profiling run on all program components

The figure 8.12 diagram focuses on the estimator component, with the aim to identify which estimators that takes the most processing time. Taking more than 80% of all processing time in the estimator component, the overstow estimator is shown as the best candidate for optimization.

Although the time spent by the overstow estimator is huge, the estimator is also one of the strongest, capable of inferring estimation backtracks at an early stage of a given instantiation. As the algorithm for implementing overstow, is most likely the one causing the overhead, serious considerations should be made before replacing it an alternative implementation.

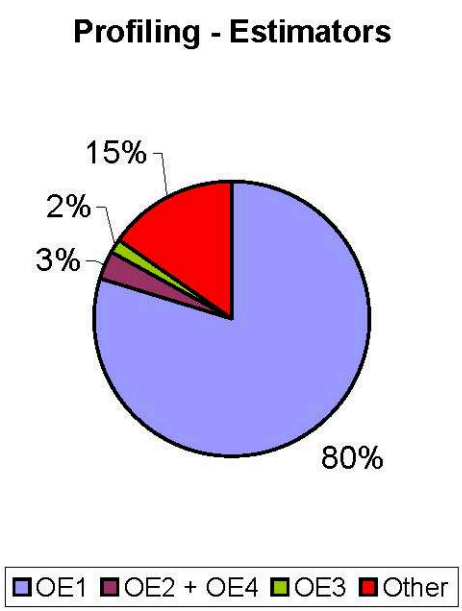


Figure 8.12: Processing time on estimators

Figure 8.13 shows the processing time distributed on the propagators. Improvements on the propagators have mainly focused on being able to prune away containers from the domains more extensively and not much on optimization. In the diagram below, some of the propagators have been omitted, as the processing time spent in these propagators was so low, that they were irrelevant. The three propagators taking the most time is:

- **ETC3**
- **PR8**
- **PR3**

Although the propagators might appear to be using a lot of processing time, some of them might simply be using the processing time because they are scheduled many times. An example of this is the implementation of $\mathcal{P}^{40\text{-}z}$, which simply ensures that the correct half of a 40-foot container is placed next to its other half. This is a propagator, which essentially does very little work is constant asymptotically. However the problem, on which the profiling tool was executed, had mainly 40-foot containers in its stowage plan, which in turn affects the propagator.

The most interesting discovery in the profiling test was the amount of time taken by the overstow estimator. The overstow estimator is however very important and future optimizations would most likely consist of experimentation with alternative algorithms for calculating the estimate or, lazy estimation. Propagators takes very little time, which shows that the implementation effectively supports fast pruning of domain values. This also shows that making the propagators prune more efficiently shows promise, as additional propagators would not affect the total processing time considerably.

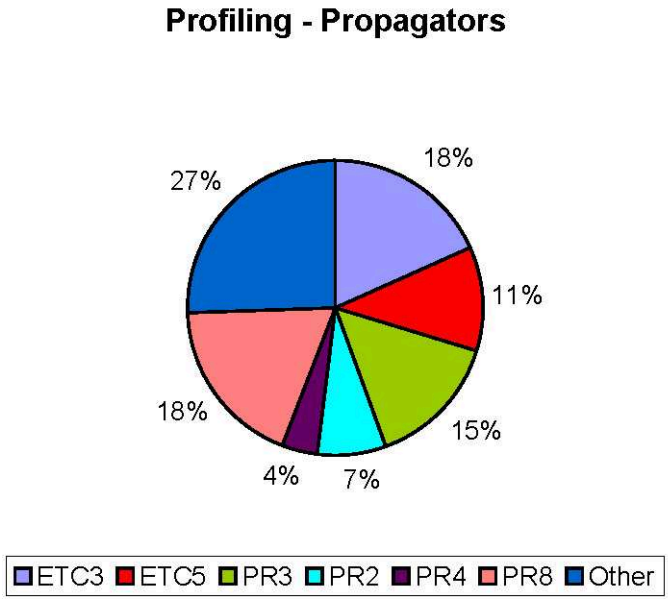


Figure 8.13: Profiling run on the estimator component

8.8 Solution discoveries

8.8.1 Traversal of the search space

The experiments revealed that the capability of solving the storage area stowage problem for our implementation had not been satisfactory. This issue had to be further investigated to determine whether the implementation or the choice of method was the issue. The first aspect, which had to be investigated, was the solutions found by our implementation. The branch and bound algorithm was modified to show the cost for each solution found. In addition the estimators were omitted to ensure, that no solutions in the search tree were disregarded. The test setup is shown in table 8.19.

Table 8.19: Test setup

Search algorithm	:	Branch & Bound
Estimators	:	-
Problem instance	:	Bay A
Propagators	:	$\mathcal{P}^{u}$, $\mathcal{P}^{g}$, $\mathcal{P}^{a}$, $\mathcal{P}^{\text{IMO-1}}$, $\mathcal{P}^{\text{IMO-2}}$, $\mathcal{P}^{\text{40-20}}$, $\mathcal{P}^{\text{20-40}}$, $\mathcal{P}^{\text{40-}y}$, $\mathcal{P}^{\text{20-20}}$, $\mathcal{P}^{C}$, $\mathcal{P}^{w}$, $\mathcal{P}'^{a}$, $\mathcal{P}'^{\text{40-}y}$
Measurement	:	Cost of solution

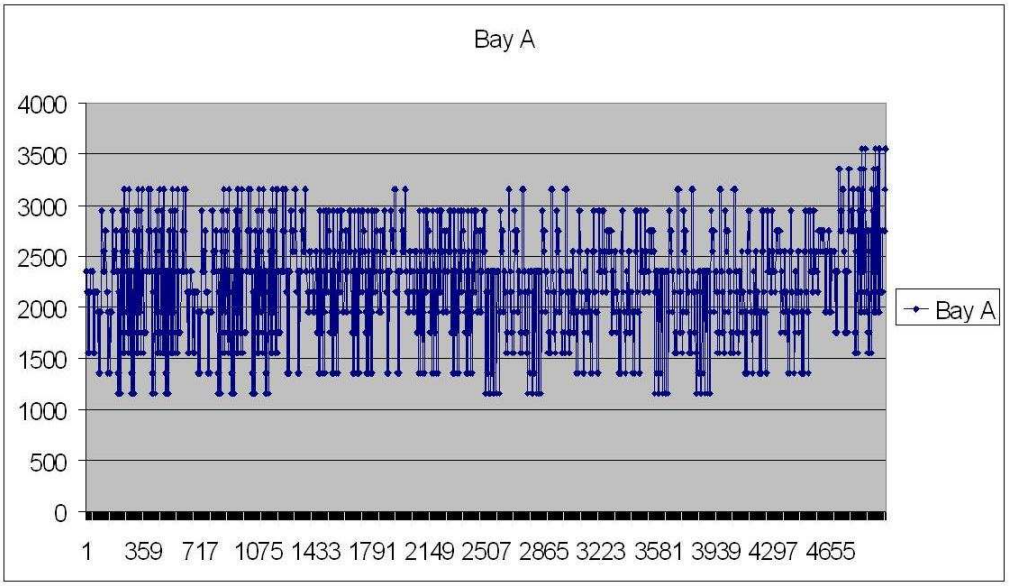


Figure 8.14: Result

Conclusion of the traversal of the search space

Figure 8.14 shows the cost for the first 5000 solutions found in the search space for Bay A, where the y-axis denotes the cost and the x-axis denotes the order of the solutions. For Bay B and Bay S it has been possible to traverse the entire search space, which is depicted in figures 8.15 and 8.16 respectively. Through inspection of the graphs following has been observed: Solution with same cost seems to be clustered. This could be explained by the fact that containers with the same set of properties can be permutated without the cost being affected. Containers with the same properties will be referred to as an equivalence set. The other interesting observation is that even though solutions are clustered they seemed to be reappearing continuously during the search. The explanation for this could be that if several equivalence sets exists then any permutation between elements in any particular set would result in a new solution without affecting the cost. These two observations serve as the lower bound of how many solutions is to be found within a problem instance i.e. Let E be a set of equivalence sets of containers in problem P then the lower bound of solutions for a problem instance will be $\prod_{e \in E} |e|!$.

The containers to be loaded on board **VESSEL-2** was further examined in order to see if this observation was indeed correct. Equivalence sets was formed based on height, weight, IMO level, size and discharge port. However the graphs showed considerable more solutions than expected and therefore further analysis was carried out. For Bay B the solutions for a particular cost was examined in order to explained why more solutions than expected was found. It became quite clear that our definition on equivalence sets was too strict.

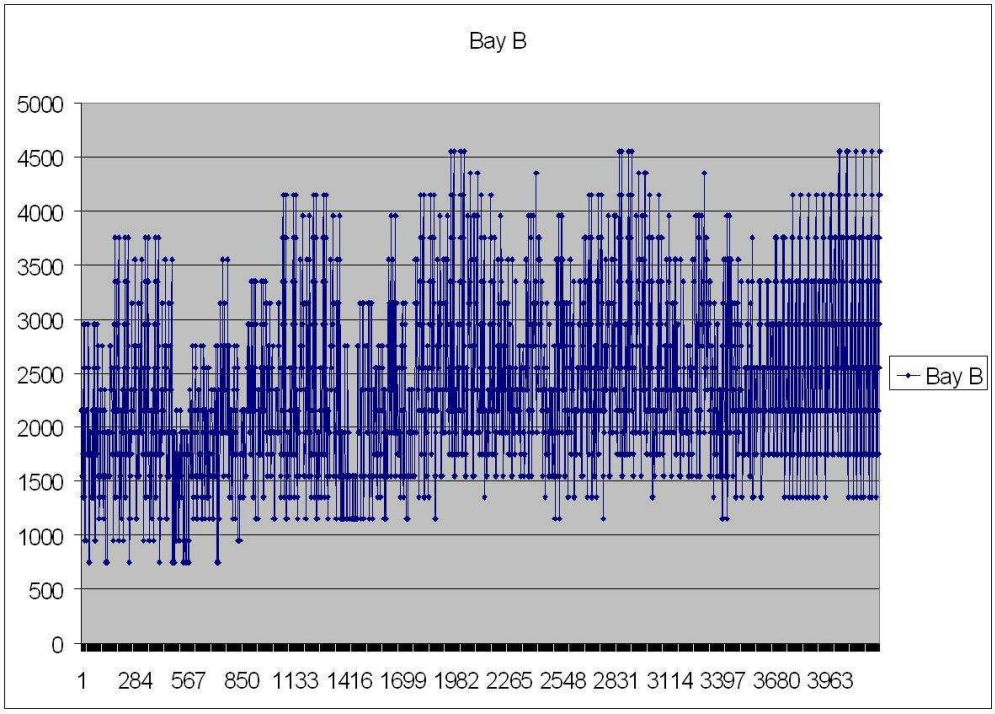


Figure 8.15: Result

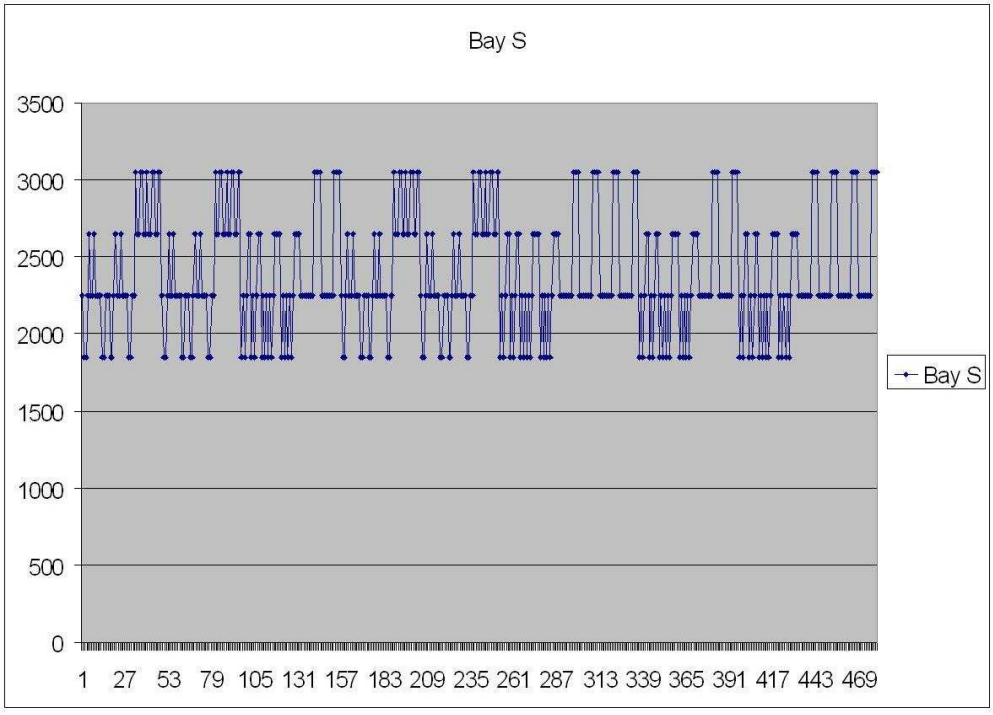


Figure 8.16: Result

Through observation it could be observed that swapping two 20-foot container on the same tier produced two new solution without changing the cost. This would be the case where all properties on the container remain the same except for the weight or IMO level. Furthermore it was also observed that as long as the number of overstows did not change for a permutation of containers it would be possible to swap containers without affecting the cost. This is particular true when considering containers in a single stack since the total height and the weight will remain unchanged. E.g. having two 20-foot containers on the different tier, with different discharge port can be swapped if these are to be discharged later than any containers above them. Due to these observation no simple rule on how these equivalence sets should be form could be given.

8.9 Conclusion on experiment

The performed experiments gave a good insight to the strengths and weaknesses of the chosen implementation, showing the advantages of using one combination of components over another combination, and giving some indication how different characteristics for approximately equal-sized problem instances made the search space vary significantly. It also gave some explanation why our implementation did not perform as well as we could have hoped for.

The set of problem instances used was quite small and it would be necessary to perform more experiments to strengthen the conclusion given in the chapter. Additionally it would be interesting to do further experimentation on how characteristics of containers would affect the search. However due to the time constraint this has not been done.

Chapter 9

Conclusion

The goal for this report has been to investigate whether an optimal solution could be found for a combinatorial problem by using branch and bound combined with propagators. This question has been inspired by the industry, however it has as well an interest in the academic world since branch and bound algorithm combined with propagators has according to our knowledge not been attempted before to solve the storage area stowage problem. Relating back to the issues this report would consider:

Can backtrack combined with a CSP-Model find a solution within reasonable time for the storage area stowage problem.

Can branch and bound combined with a CSP-Model find an optimal solution within reasonable time for the storage area stowage problem.

Experiments show that the algorithm is capable of solving the problem, but does not scale well to realistic problem sizes. Nevertheless because this approach has not been used previously, no means to measure the quality of any algorithm, which could solve SASP existed. Guaranteing optimality, we are able to provide a tool to measure the quality of alternative algorithms.

Furthermore the experiments showed that the storage area stowage problem contained several properties. The first interesting discovery was, that the constraints were too weak to restrict the search space considerably. Secondly, many solution with the same cost appear during a search, which indicates that the objectives did not restrict the solution space sufficiently, consequently requiring the estimators to be rather precise in order to cut of branches.

The third discovery was that the implemented estimators behave differently depending on the properties of the containers to be loaded. What is interesting in that respect would be to investigate which decisive properties could improve the estimators. Lazy estimation was an interesting approach, which showed promising results, provided that the estimators were beneficial for that given problem instance.

Future work

Even though a lot of ground have been covered in this report much work still needs to be done in order to solve the SASP. Based on the result from the experiments and the conclusion the future work should focus on how to solve the issue with solutions reappearing with the same cost. We suggest the following:

- Considering equivalent container as a single solution

  Having n containers with the same characteristics will yield $n!$ different solutions without the cost changing. Having this in mind the search space could be reduced if it is possible to identify any of the $n! - 1$ solutions and skip these. In connection to this it would be interesting to see if forcing the equivalent containers to be placed early in the search rather than late will improve the search. The motivation for placing equivalent containers early in the search is that containers with the same characteristics are only rearranged once rather than for each subtree.

- Using estimation value to cut of branches in the search tree

  The estimators computes an estimation of what the actual cost will be when extending the partial solution to a complete solution. However if there is a tight correspondence between the estimation value and the actual cost, then it would be possible to avoid unnecessary exploration of the search tree by comparing the estimated value for the current partial instantiation with some estimated value for a previous partial instantiation.

- Alternative search methods

  The branch and bound algorithm relies on estimators to cut of branches within the search tree. Having many solution with the same cost will cause the estimator to not being able to cut of branches and thus many solutions with the same cost are discovered.

Due to time constraints several other aspects were not examined. Following aspects could have been interesting to examine in depth:

- Scalability

  An experiment to see how the performance is affected by the size of the problem instance. Since it is already known that the size grows exponentially, it would still be interesting to see how much time is used when as search space grows. The sheer size of problem instances would not generate sufficient results to be concluded upon on the scalable test, and has therefore been disregarded.

- Domain value ordering

  Section 7.4.1 describes how the order of the domain value is composed by the label and discharge port and used for selecting domain values which restricts the search

space the most. What could be interesting is to examine the effect of altering the sorting order, however this would require almost all experimentations to be redone and was therefore disregarded due to time restriction.

- Variable orderings

  Experimenting with variable ordering has a lot of potential for improving on the iteration count as the possibilities for variable ordering heuristic approaches are many. One interesting approach could be to combine the DVFC with the Overstow heuristic, which would allow to select the optimal value, according to the overstow, for variables with the smallest domain. However, this necessitates extensive changes in the current implementation and has for this reason been considered as future work.

- Weights on the objectives

  More experiments could be performed with different sets of weights, in order to see if it would affect the execution of estimators. However several tests has to be redone for each set of weights and it was disregarded due to time constraints.

- Identifying decisive characteristics for the estimators

  The conclusion for the estimators was somewhat unclear and the performance of them seems to be dependent on some characteristics of the problem instance. For future work it could be interesting to identify, which characteristics are decisive for the performance of the implemented estimators.

- Lazy estimation

  In general, lazy estimation shows many interesting properties: for some problem instances it is the most interesting to do estimation with only few assigned variables, while for other problem instances it is not beneficial to do estimation at all. It would be interesting to be able to identify the properties that make estimation beneficial, as the reward appears to be great. This has however been left as future work.

Bibliography

[1] Mordecai Avriel, Michal Penn, Naomi Shpirer: *Container ship stowage problem: Complexity and connection to the coloring of circle graphs*, Elsevier (1999)

[2] Dechter, Rina: *Constraint Processing*, Elsevier Science (2003), ISBN 1-5860-890-7

[3] F. Rossi, P. Van Beek, T. Walsh: *Handbook of Constraint Programming*, Elsevier (2006), ISBN 978-0-444-52726-4

[4] M. Mochnacs, M. Tanaka, A. Nyborg: *An experimental analysis of constraint processing algorithms*, (2006)

[5] T. Cormen, C. Leiserson, R. Rivest, C. Stein: *Introduction to Algorithms*, McGraw-Hill (2003), ISBN 0-262-03293-7

[6] Kenneth P. Bogart: *Introductory Combinatorics*, Academic Press (1983), ISBN 0-12-110830-9

[7] I. D. Wilson and P. A. Roach: *Principles of combinatorial optimization applied to container- ship stowage planning*, Journal of Heuristics (1999)

Appendix A

Program organization

Shown below is a component diagram of the implementation developed. It presents the program as divided up into four basic components, of which the most central is the Model component, which contains the problem description and the instantiation build upon it, and the Branch & Bound component, containing the actual search algorithm. Each of the components are described in further detail later in this section.

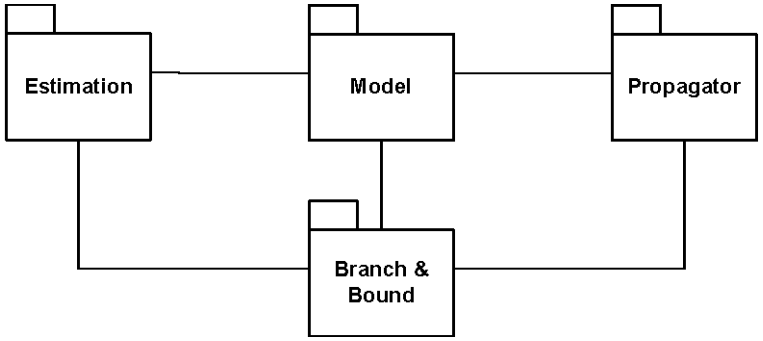


Figure A.1: The implementation presented as a component diagram

Model Component

Containing the constraint network representation of the problem is the model component. The variables are kept within the problem class in a three dimensional array, with the rows, tiers, stacks and cells as indexes on the different dimensions. Additional information regarding maximum height and weight allowed for each stack, is kept in the stack class.
Besides the problem, the model component also contains the current instantiation being built upon the problem assignments that the instantiation consists of.

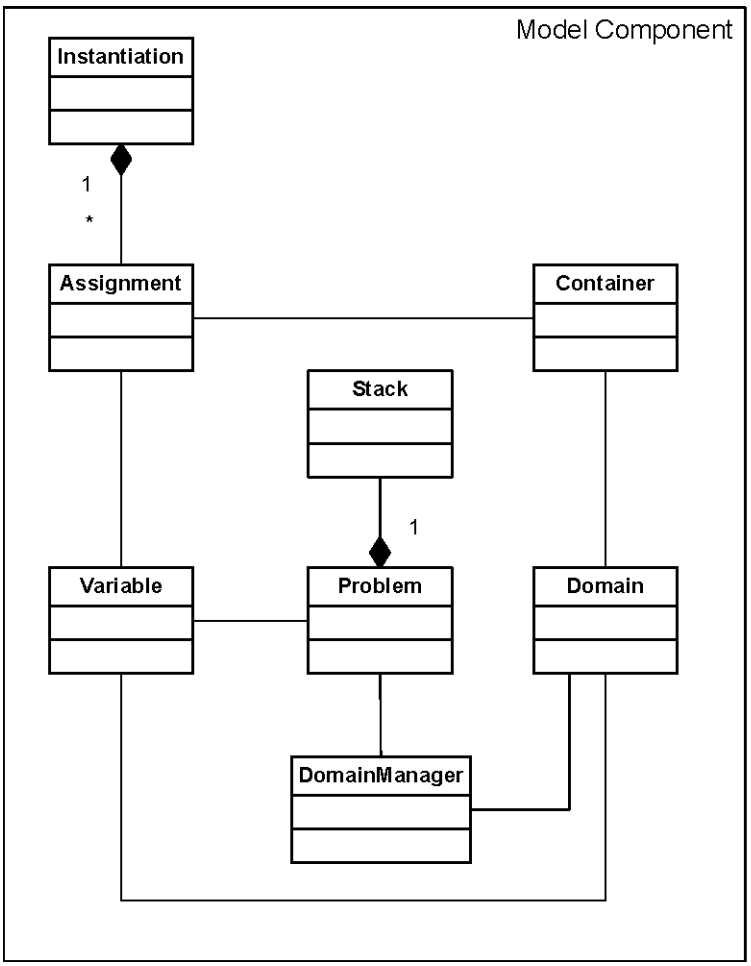


Figure A.2: The Branch & Bound component presented as a class diagram

Propagator Component

The main class within the propagator component is the scheduler, which maintains the underlying propagators. Based on the assignment of a given input variable, it determines which propagators should be run using the label concept described in the report. The propagators are kept within the scheduler in a simple list structure. All communication with the propagators go through the scheduler, making the scheduler serve as an abstract layer between the propagators and the remaining application.

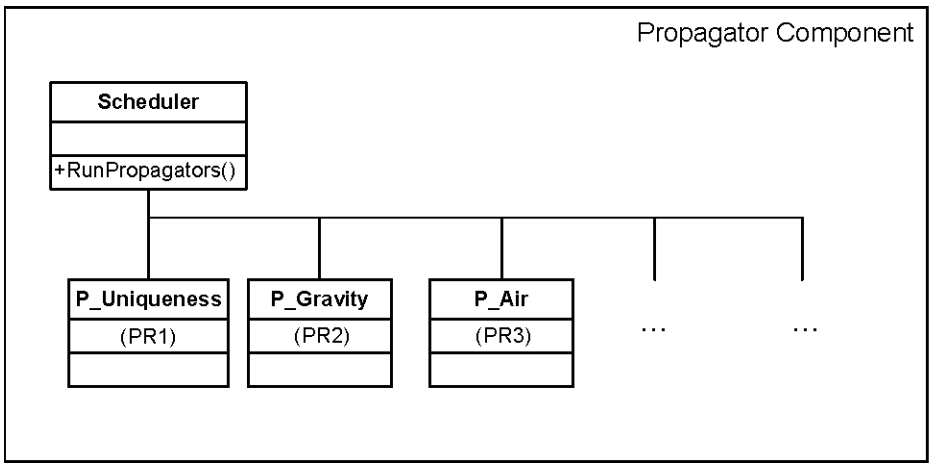


Figure A.3: The propagator component presented as a class diagram

Estimator Component

The class responsible for running the estimators, is the EstimationCalculator which schedules the estimators one by one and sums up the returned cost from each. A reference to each estimator is kept in a list structure within the EstimationCalculator.

Also kept within the estimator component is evaluators, which works in a similar fashion as the estimators. Evaluators are used to calculate the cost of complete solutions only and calculates the cost faster than the estimators. The main reason for using the estimators was for debugging purposes, in an attempt to verify that the estimators always underestimated the cost of any solution which could be extended from a given partial solution. It was kept as a part of the final implementation, as they give a slight search time reduction, due to the more simply and efficient way of calculating the cost for complete solutions.

All communication with the estimators and evaluators go through the EstimationCalculator and EvaluationCalculator, making these classes serve as an abstract layer between the estimators/evaluators and the remaining application

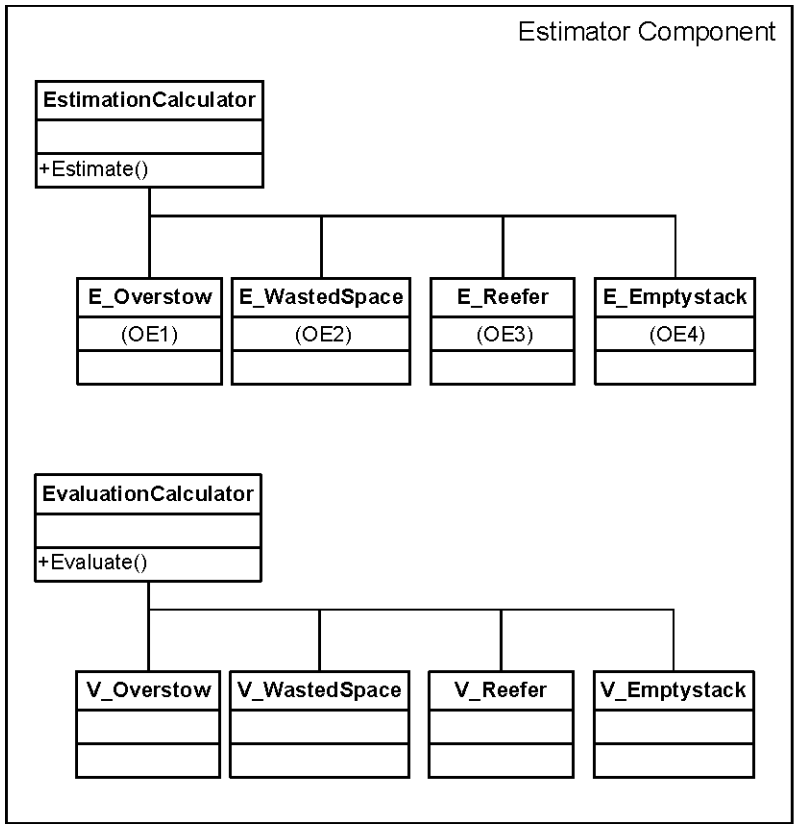


Figure A.4: The estimator component presented as a class diagram

Branch & Bound Component

This section gives a detailed description of the main classes handling the actual search within a problem instance. Shown below is an UML diagram of the classes within the Branch & Bound component.

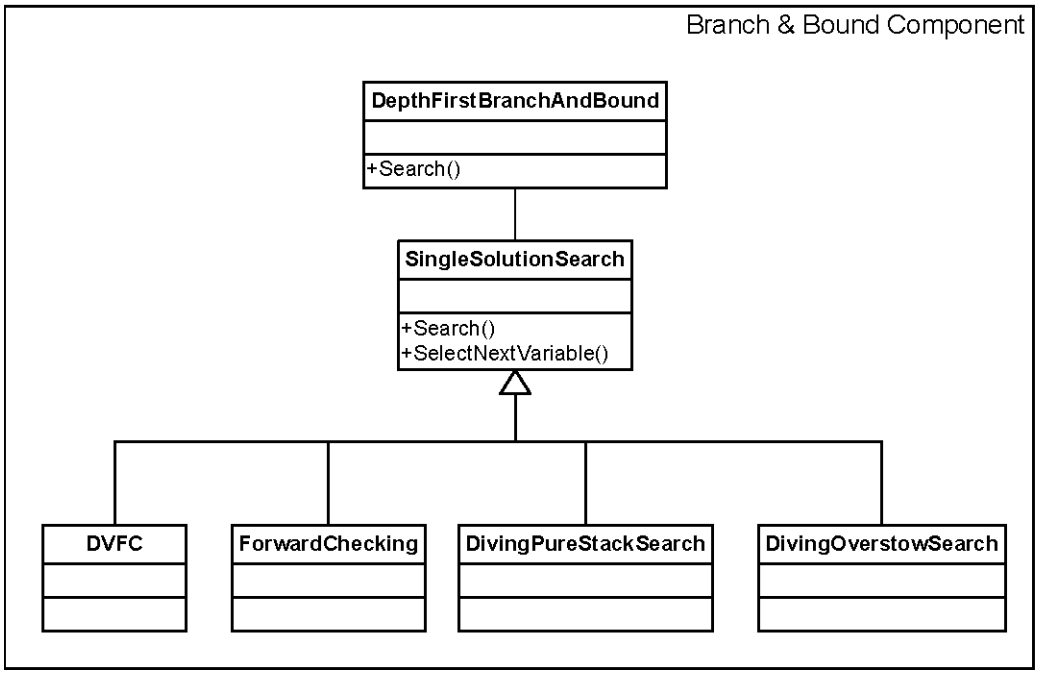


Figure A.5: The Branch & Bound component presented as a class diagram

The different implementations of the backtracking algorithm is shown in the class diagram as inheritance from SINGLESOLUTIONSEARCH with the option to override the procedure SELECTNEXTVARIABLE. DEPTHFIRSTBRANCHANDBOUND holds the actual branch and bound algorithm, described with pseudocode in the report.

Appendix B

Informal description

Stowage Problem for Under Deck Storage Area

Input

1. Current port number p $\{0,\ldots,N\}$
2. A physical layout of a container vessel under deck storage area defining:
   a) A number of standard container cells organized in stacks
   b) Max height for each stack
   c) Max weight for each stack
   d) Attributes for each cell

      Reefer cell (Y/N).
      Max number of 20’ containers cell can hold (0,1,2)
      Max number of 40’ containers cell can hold (0,1)
      (we assume no 45-foot bays)
3. A list of containers already stored in the storage area containing for each container:
   a) The cell the container is assigned to
   b) The weight of the container
   c) The height of the container (high-cube/not high-cube)
   d) The length of the container (20’ or 40’)
   e) Load port (0,1,. . . ,N)
   f) Discharge port (0,1,. . . ,N)
   g) IMO level (0,1,2)
   h) Reefer (Y/N)

4. A list of containers to load into the storage area containing for each container:

   a) The weight of the container
   b) The height of the container (high-cube/not high-cube)
   c) The length of the container (20' or 40')
   d) Discharge port (0,1,. . . ,N)
   e) IMO level (0,1,2)
   f) Reefer (Y/N)

Output
An assignment of containers in the load-list under bullet 4 above to cells in the storage area such that the constraints and objectives below are achieved:

1. Constraints (a valid assignment must satisfy all of them)

   a) Assigned slots must form stacks (containers stand on top of each other in the stacks. They cannot hang in the air)
   b) Reefer containers must be placed in reefer slots (obs. if a reefer slot can hold more than one 20' container, both of these can be reefer.)
   c) A slot can at most hold two 20' containers or one 40 container.
   d) A slot can only hold the max number of containers of different length as described by its attributes under bullet 2.d.ii-iii above.
   e) 20' containers can not be stacked on top of 40 containers (this is physically impossible)
   f) The height of each stack is within its limits
   g) The weight of each stack is within its limits
   h) IMO rules are satisfied for each container:
      i. level 0: no rule
      ii. level 1: level 1 containers must be separated from level 1 and 2 containers by at least one slot vertically and horizontally (obs. as described in the note below, we map containers to slots successively from the bottom of stacks, independent of whether some of these containers are high-cube). Thus, if we have an IMO level 1 at slot i in stack j, then the container at slot i+1 and i-1 in stack j cannot be a level 1 or level 2, and the container at level i in stack j+1 and j-1 can not be a level 1 or level 2
      iii. Level 2: level 2 containers must be separated from level 1 containers according to the rules of level 1 containers. In addition, level 2 containers must be separated from any other level 2 containers by a stack without level 2 containers. Thus, if a stack i contains a level 2 container, then stack i+1 and i-1 cannot contain a level 2 container.

2. Objectives

   (a) Minimize overstows. Cost penalty: one unit for each container in a stack overstowing another container below it in the stack. Unit weight Woverstow

   (b) Minimize the space wasted in a stack. Cost penalty: length of wasted stack space. Unit weight WspaceWaste

   (c) Avoid loading non-reefers into reefer slots. Cost penalty: one unit for each non-reefer container in a reefer slot. Unit weight Wreefer

   (d) Keep stacks empty if possible. Cost penalty one unit per new stack used. Unit weight Wemptystack

- Note on interpretation of cell restrictions

  High-cube containers may get containers in a stack out of sync with the cell positions. For this reason, it may be unclear how cell restrictions are to be interpreted for each container in a stack. Given the current state of affairs, however, it is safe to apply the cell restrictions to the containers in the order they appear in the stack rather than their actual position. Thus, the restrictions on the i'th cell counted from the bottom of a stack apply to the i'th container in the stack independent of its actual position. For 20-foot containers, this is the case, because 20-foot containers never get out of sync with the cell level. No high-cube 20-foot containers exist and any legal stack either consists fully of 20-foot containers or has a single shift from 20-foot containers in the bottom to 40 and 45-foot containers in the top. 40 and 45-foot containers, on the other hand, can get out of sync with the cell level since 40 and 45-foot containers can overstow a high-cube container. We need to argue for each cell restriction in turn that it can be applied to the relative rather than absolute position of a container in a stack. For reefer restrictions this is the case because power connections only exist at the bottom tiers over and under deck. So even if high-cube containers are stored at the bottom tiers, the stack levels never get so much out of sync that the power lines cannot reach the connectors. For length restrictions this is the case since we only have one rule where misalignment of containers may be important. Over deck 45-foot containers must be placed over the lashing bridge. This rule, however, is only relaxed by placing high-cube containers in bottom tiers.

Appendix C

Pseudo code

This appendix presents the pseudocode for helper functions, which was chosen to omit from the report itself. For each pseudocode procedure, a detailed description of the key elements is provided for clarification.

C.1 Evaluation

C.1.1 Overstowage Evaluation

OVERSTOW returns the real overstow cost g_{ov} of the instantiation $\vec{a}$ given as parameter. The algorithm iterates over the entire stowage location and updates the overstow count whenever a container with a lower discharge port is stowed below a container with a higher discharge port.

The time complexity for OVERSTOW is $O(\mathrm{sc\,T})$, where T is the maximum number of tiers of all stacks.

```
procedure OVERSTOW(a⃗)
1   g_ov ← 0
2   for j ← 1 to sc do
4      for i1 ← tc_j downto 1 do
5         for i2 ← i1 − 1 downto 1 do
3            foreach l ∈ L do
6               if dp π_{x^l_{i1,j}}(a⃗) > dp π_{x^l_{i2,j}}(a⃗) then
7                  g_ov ← g_ov + 1
8               endif
9            endfor
10        endfor
11     endfor
12  endfor
13  return g_ov
```

Figure C.1: Overstow cost of the instantiation $\vec{a}$.

C.1.2 Wastedspace Evaluation

WASTEDSPACE returns the real wastedspace cost g_{ws} of the instantiation $\vec{a}$ given as parameter. The algorithm considers each cellstack in turn, and calculates accumulatively the wasted space of the entire stowage configuration.

```
procedure WASTEDSPACE(a⃗)
1   g_ws ← 0
2   foreach k ∈ K do
4      g_ws ← g_ws + WASTEDSPACEOFSINGLESTACK(fs(k))
5   endfor
6   g_ws ← g_ws / h_st
6   return g_ws
```

Figure C.2: Wastedspace cost of the instantiation $\vec{a}$.

The time complexity for WASTEDSPACE is $O(|K|)$, since WASTEDSPACEOFSINGLESTACK is $O(1)$. fs takes constant time, by being maintained along with the assignments of the search.

procedure WASTEDSPACEOFSINGLESTACK(σ)
1 $w \leftarrow 0$
2 **if** $\sigma < h_{st}$ **then**
3 $w \leftarrow \sigma$
4 **endif**
5 **return** w

Figure C.3: Wasted space from σ free space.

C.2 Estimation

C.2.1 Overstowage Estimation

ESTIMATEOVERSTOW returns the estimated overstow cost $(g_{ov} + h_{ov})$ of the instantiation $\vec{a}$ given as parameter.

procedure ESTIMATEOVERSTOW($\vec{a}$)
1 $U \leftarrow X \setminus \mathcal{S}$
2 $C^U \leftarrow C \setminus \pi_{\mathcal{S}}(\vec{a})$
3 $g_{ov} \leftarrow$ OVERSTOW($\vec{a}$)
4 $M \leftarrow$ MINCOSTMATCH$\left(C^U, U,$ REPRESENTATIONCOST$(C^U, U, \vec{a})\right)$
5 **if** $|M| = |C^U|$ **then**
6 $h_{ov} = \sum_{e \in M} w(e)$
7 **return** $g_{ov} + h_{ov}$
8 **else**
9 **return** ∞

Figure C.4: Estimated overstow cost of the instantiation $\vec{a}$.

MINCOSTMATCH is a routine that takes as arguments a Minimum Cost Matching Problem and returns the maximum matching of minimum cost as a set of weighted edges. The algorithm can be found in Chapter 5 of [6].

OVERSTOW calculates the real overstow cost of the instantiation given as parameter, as described in section C.1.1.

The REPRESENTATIONCOST procedure details the computation of the representation costs. As parameters, it receives the collection C^U of containers yet to be loaded, the collection U

of available stowage cells, and the current instantiation $\vec{a}$. The trivial algorithm to calculate the representation costs is optimized based on the following observations: two containers with the same discharge port stowed in the same location result in the same amount of overstows, and a container stowed in a particular cell gives the same amount of overstows as if it was stowed in a neighboring below or above cell of the same cellstack, if such a cell exists. Based on these ideas, the containers are considered in increasing order of their discharge ports and the free cells of each stack in a bottom up order.

Initially, the algorithm calls the SORT subroutine to sort the container list according to the partial order defined by the CONTAINERCOMPARATOR comparison function, that is in increasing order of their discharge ports. The algorithm maintains two variables, the container considered at the previous iteration $c^{'}$ and the cell considered at the previous iteration $l^{'}$. In case the current container c and previous container $c^{'}$ are similar or current cell l and previous $l^{'}$ cell can accommodate the same type of container then the previously calculated value can be reused. In case where the container are distinct and cell cannot accommodate the same container the number of overstows is counted explicitly by calling OVERSTOWOF. The RC hash-table maintains the resulting costs.

```
procedure REPRESENTATIONCOST(C^U, U, a⃗)
1   SORT(CONTAINERCOMPARATOR, C^U)
2   c' ← nil
3   foreach c ∈ C^U
4     if CONTAINERCOMPARATOR(c', c) = 0 then
5       RC[c] ← RC[c']
6     else
7       c' ← c
8       l' ← nil
9       foreach l ∈ U *** this considers cells per stack, bottom up ***
10        if ISIN(c, l) then
11          if CELLCOMPARATOR(l', l) = 0 then
12            RC[c][l] ← RC[c][l']
13          else
14            RC[c][l] ← OVERSTOWOF(a⃗, c, l)
15            l' ← l
16          endif
17        endif
18      endfor
19    endif
20  endfor
21  return RC
```

Figure C.5: Representation costs computation.

```
procedure OVERSTOWOF(ā, c, x^l_{i,j})
1   ov ← 0
2   for t ← 1 to i − 1 do
3      if dp_{π_{{x^l_{t,j}}}(ā)} < dp_c then
4         ov ← ov + 1
5      endif
6   endfor
7   for t ← i + 1 to tc_j do
8      if dp_{π_{{x^l_{t,j}}}(ā)} > dp_c then
9         ov ← ov + 1
10     endif
11  endfor
12  return ov
```

Figure C.6: Overstow cost of stowing container c in cell $x^l_{i,j}$.

The CONTAINERCOMPARATOR defines the partial ordering of the container set according to discharge port then label. Sorting according to this $O(1)$ comparison function results in the container list sorted in the increasing order of discharge ports followed by their label. The CELLCOMPARATOR is a $O(1)$ comparison routine that receives as parameters two cells l_1 and l_2 and returns 0 if the cells belong to the same cellstack, are situated one on top of the other and have the same label.

The worst case running time of REPRESENTATIONCOST is $O(|C^U|log|C^U| + |C^U||U|\,T)$, since sorting takes $O(|C^U|log|C^U|)$ and there are at most $|C^U||U|$ calls to OVERSTOWCOSTOF, each call using $O(T)$ time. A tighter bound of $O(|C^U|log|C^U| + DI\,T)$ can be derived observing that the number of calls to OVERSTOWCOSTOF depends on the number of different discharge ports, written D, and the number of neighboring cellstack regions, written I.

C.2.2 Wastedspace Estimation

ESTIMATEWASTEDSPACE returns the estimated wasted spaced cost $(g_{ov} + h_{ov})$ of the instantiation $\vec{a}$ given as parameter. On line 4 it calls the procedure WASTEDSPACEOF which is the dynamic programming implementation of the recursive definition given in section 6.3.

procedure ESTIMATEWASTEDSPACE$(\vec{a})$
1 $S \leftarrow \left|\{c \in \bigcup_{x \in X \setminus \mathcal{S}} \mathcal{M}(x) \ : \ h_c = 8.5\}\right|$
2 $H \leftarrow \left|\{c \in \bigcup_{x \in X \setminus \mathcal{S}} \mathcal{M}(x) \ : \ h_c = 9.5\}\right|$
3 $\rho_0 \leftarrow$ decreasing order of available cells, used stacks first
4 **return** $\frac{\text{WASTEDSPACEOF}(S,H,\prec_1^{\rho_0},fs(\prec_1^{\rho_0}))}{h_{st}}$

Figure C.7: Estimated wastedspace cost of the instantiation $\vec{a}$.

WASTEDSPACEOFSINGLESTACK is described in section C.1.2.

```
procedure WastedSpaceOf(S, H, k, σ)
1   if W[S, H, k, σ] = nil
2     if σ < h_st then
3       if S ≥ 1 or H ≥ 1 then
4         w ← NoSpaceInStack(S, H, k, σ)
5       else
6         w ← NoContainers(k, σ)
7       endif
8     elseif σ ≥ h_hc then
9       if S ≥ 1 and H ≥ 1 then
10        w ← min(WastedSpaceOf(S − 1, H, k, σ − h_st),
                  WastedSpaceOf(S, H − 1, k, σ − h_hc)
11      elseif H ≥ 1 then
12        w ← WastedSpaceOf(S, H − 1, k, σ − h_hc)
13      elseif S ≥ 1 then
14        w ← WastedSpaceOf(S − 1, H, k, σ − h_st)
15      else
16        w ← NoContainers(k, σ)
17      endif
18    else
19      if S ≥ 1 then
20        w ← WastedSpaceOf(S − 1, H, k, σ − h_st)
21      elseif H ≥ 1
22        w ← NoSpaceInStack(S, H, k, σ)
23      else
24        w ← NoContainers(k, σ)
25      endif
26    endif
27    W[S, H, k, σ] ← w
28  endif
29  return W[S, H, k, σ]
```

Figure C.8: Wasted space estimation algorithm.

procedure NOCONTAINERS(k, σ)
1 $w \leftarrow$ WASTEDSPACEOFSINGLESTACK(σ)
2 **for** $j \leftarrow k+1$ **to** $|K|$ **do**
3 $w \leftarrow w +$ WASTEDSPACEOFSINGLESTACK$(fs(\prec_j^{\rho_0}))$
4 **endfor**
5 **return** w

Figure C.9: Update of wasted space due to no more containers.

procedure NOSPACEINSTACK(S, H, k, σ)
1 $w \leftarrow 0$
2 **if** $k+1 \geq |K|$ **then**
3 $w \leftarrow$ WASTEDSPACEOFSINGLESTACK(σ)
4 **else**
5 $w \leftarrow \sigma +$ WASTEDSPACEOF$(S, H, \prec_{k+1}^{\rho_0}, fs(\prec_{k+1}^{\rho_0}))$
6 **endif**
7 **return** w

Figure C.10: Update of wasted space due to no more space in cellstack k.

C.3 Domain management function

A global Label-Domain table τ as described in section 7.1 is maintained by the two procedures REINSERTDOMAINVALUE and REMOVEDOMAINVALUE.

Reinsertion of domain value

The procedure REINSERTDOMAINVALUE inserts the value v given as input into all domains, which can accommodate v. From τ a list of all domains is retrieved by the label of v. Each domain in the list gets value v reinserted.

```
REINSERTDOMAINVALUE(v)
1   m ← τ[DomainValueLabel(v)]
2   foreach s ∈ m
3       s ← s ∪ {v}
4       τ[DomainValueLabel(v)] ← m
```

Figure C.11: Pseudocode for REINSERTDOMAINVALUE

Removal of domain value

The procedure REMOVEDOMAINVALUE removes the value v given as input from all domains, which can accommodate v. From τ a list of all domains is retrieved by the label of v. Each domain in the list gets value v removed.

```
REMOVEDOMAINVALUE(v)
1   m ← τ[DomainValueLabel(v)]
2   foreach s ∈ m
3       s ← s \ {v}
4       τ[DomainValueLabel(v)] ← m
```

Figure C.12: Pseudocode for REMOVEDOMAINVALUE

C.4 Propagators examples

C.4.1 Uniqueness

The uniqueness propagator ensures that a domain value v does not remain an eligible candidate value, when it has been assigned. It relies on the procedure REMOVEDOMAINVALUE described in the appendix C.3. Given as input is the current partial instantiation $\vec{a}$, the current variable being assigned $x^l_{i,j}$ and the candidate value v, which are removed from any shared domain. Since no domain is not checked for exhaustion *false* is returned. The omission of the check of exhaustion, causes the algorithm to potentially backtrack much later. However this only occurs if a cell, which has some container placed above it, has a reference to an empty domain. This is caught by $\mathcal{E}^{FA}$.

PROPAGATORUNIQUENESS$(\vec{a}, x^l_{i,j}, v)$
1 REMOVEDOMAINVALUE(v)
2 **return** *false*

Figure C.13: Pseudocode for uniquness propagator

C.4.2 IMO-1

The IMO-1 propagator ensures that the cells according to the IMO rule are not able to consider an IMO-1 and IMO-2 containers. It relies on the procedure REMOVEDOMAINVALUE described in the appendix C.3. A current partial instantiation $\vec{a}$, the current variable $x^l_{i,j}$ being instantiated and the candidate value v is given as input. All assignable cells in set $X^{\text{IMO-1}}_{i,j}$, which is defined in table 5.1, are getting the IMO-1 and IMO-2 property removed from the celllabel in line 2. A check whether the pruning caused exhaustion of their domain is carried out in line 4, in the case that exhaustion occurred the procedure returns *true* otherwise *false* is returned if no of the assignable variables caused exhaustion.

PROPAGATORIMO-1$(\vec{a}, x^l_{i,j}, v)$
1 **foreach** $x \in X^{\text{IMO-1}}_{i,j}$
2 **if** $x \in \mathcal{S}$
3 $Label[x] \leftarrow Label[x]$ & $\sim(Label\text{IMO-1} \mid Label\text{IMO-2})$
4 **if** $\mathcal{D}(x) = \emptyset$
5 **return** *true*
6 **return** *false*

Figure C.14: Pseudocode for IMO-1 propagator