Applications

JDMS

Journal of Defense Modeling and Simulation: Applications, Methodology, Technology
1–22

DOI: 10.1177/1548512918812804
journals.sagepub.com/home/dms
SAGE

Constraint programming heuristics and software tools for amphibious embarkation planning

Paul A Chircop and Timothy J Surendonk

Abstract

We outline the development and performance of heuristic approaches to obtain prioritized load planning solutions for the embarkation of cargo onto the deck of an amphibious ship. The heuristic techniques are underpinned by a constraint programming paradigm and have been implemented in a Java-based software package called COmPacT (Constraint Optimization Packing Tool). COmPacT utilizes the modeling and solver libraries of the IBM ILOG CPLEX Optimization Studio. For the purposes of mathematical modeling, the embarkation planning problem is akin to packing a set of rectangular items onto a larger rectangular space (the deck), which could contain obstacles and may be subject to mass balance constraints. The modeling and algorithmic approaches are outlined in connection to the software development of COmPacT. Finally, we demonstrate how COmPacT may be used in conjunction with a planner's knowledge and expertise to enable iterative packing techniques, thereby combining the strengths of both automated and manual methods.

Keywords

Constraint programming, embarkation planning, amphibious operations, bin packing problem, software development, heuristics, unloading constraints

1. Introduction

1.1. Contextual background

An important problem when planning an amphibious operation, that is, the disembarkation of army units from a ship at sea and their subsequent transportation to and landing upon the shore, is how to optimally pack such units onto the ship's deck (loading area). In this context, the units in question are typically dominated by various vehicles, tanks, and other cargo items that must be carefully positioned within the loading area. These items/units may be subject to unloading constraints (priority ordering) in accordance with mission-specific disembarkation requirements. Furthermore, a ship's stability and trim properties may mean that certain mass balance constraints are imposed on item positions, and these may be further constrained by the presence of natural barriers and/or obstacles on the deck. In this study, we consider a single deck only.

The problem of positioning items onto a ship deck in the manner described above is referred to as the *amphibious embarkation planning problem*. In this paper, we develop an algorithmic and software-based framework to obtain prioritized load planning solutions for this embarkation planning problem. Our intent is to design a mathematical modeling and algorithmic solution approach that can be developed and tested in an interactive software system. The ultimate goal is to produce a system that can be easily deployed and used by military operators or planners to rapidly experiment with various embarkation options for amphibious operations planning. We will demonstrate how such a software system could be used to leverage an operator's expertise in order to facilitate both automated and manual packing methods in a complementary fashion.

Defence Science and Technology Group, Australia

Corresponding author:
Paul A Chircop, DST Eveleigh, Department of Defence, Locked Bag 7005, Liverpool, NSW 1871, Australia.
Email: paul.chircop@dst.defence.gov.au

1.2. Description of the problem

An amphibious embarkation planning problem can be represented as a *(bin) packing problem*. A packing problem is concerned with placing a set of rectangles (also referred to as rectangular items or just simply items) of varying length, width, and mass onto a larger rectangular space (also called a *bin*). An item can be placed in the bin if it does not overlap with the bin's boundary or any other packed item.

When modeled as a packing problem, the amphibious embarkation planning problem is concerned with positioning a set of army units (represented as rectangular items) of varying length, width, and mass onto a ship's deck (the bin). A feasibly positioned army unit must be entirely contained within the deck and not overlap with any other unit that is also entirely contained within the deck. If any item cannot be positioned on the deck in such a manner, or if it violates any other side constraints, it is labeled as unpacked. Each item must also be positioned according to its given orientation (no rotation allowed). An illustration of the problem concept can be found in Figure 1.

There may be circumstances in which the number of candidate rectangular items is so large that it is physically impossible for all of them to be packed onto the deck in the manner described above. In such cases, an optimal solution will entail that a number of items remain unpacked, for example, the situation depicted in Figure 2.

If there is a requirement to allow for some additional spacing around individual items (for example, to accommodate tie-downs or to create personnel thoroughfares), then this can be easily accommodated by augmenting the size of each item with an artificial border. The border width can be set for each rectangle individually.

In addition to position feasibility, there may also be a requirement to balance the mass distribution of all packed items. Such a requirement may be necessary to ensure the ship's stability with a candidate packing solution. A deck may also impose constraints on the individual mass of each item, that is, there may be loading constraints that preclude unusually heavy items from being packed on the deck. Such items would be immediately labeled as unpacked, along with any others having a width (length) exceeding the deck's width (length). This would be carried out in a pre-processing phase.

Another consideration that must be accounted for in any packing problem is the presence of inherent natural barriers and/or obstacles on the deck, around which rectangular items can be potentially placed without overlapping. Obstacles may represent the presence of ramps, elevators, or walls, or may be included to model decks that are not wholly rectangular (i.e., piecewise rectangular decks).

In some specialized mission contexts, there may be a requirement to pack the rectangular items (representing the members of an ORBAT - ORder of BATtle - a set of units, vehicles, equipment or cargo to be transported) onto the deck according to a priority ordering. A prioritized packing problem can be accounted for by designing either hard or soft constraints that track the relative positions of the packed items according to their priority values. In this context, a soft constraint might allow for the violation of the priority ordering conditions to within pre-defined tolerance limits.

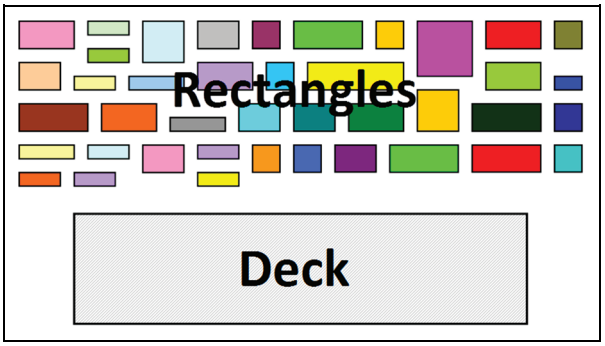

Figure 1. Concept illustration of a packing problem.

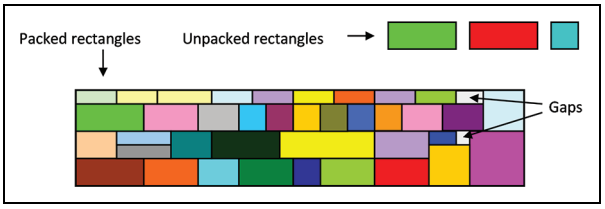

Figure 2. An optimal solution to the problem in Figure 1.

The overall objective of the problem is to minimize the total unused space on the deck or, equivalently, to maximize the total area of all packed rectangles, subject to the aforementioned packing constraints. For example, an optimal solution to the packing problem of Figure 1 can be found in Figure 2, which also highlights the packing gaps (unused space on the deck).

1.3. Review of the literature

The amphibious embarkation planning problem described in the previous section bears a number of similarities to the well-known and comprehensively studied problems in the rectangular bin packing literature. In this section, we will briefly outline a few of these related problems and discuss their mathematical modeling and solution approaches.

The classical two-dimensional bin packing problem (2DBPP), which is known to be $\mathcal{NP}$-hard (see Johnson et al.[1] and Jansen and Prädel[2]), is concerned with assigning

a list of rectangular items to a minimum number of identical rectangular bins. No overlapping of the rectangular items is permitted in any of the bins. (The fundamental difference between this classical problem and the amphibious packing problem concerns the objective and the number of bins.) An extension to the classical 2DBPP incorporating variable bin sizes of varying cost can be found in Pisinger and Sigurd[18] and Bettinelli et al.[19] Furthermore, see the recent work by Polyakovskiy and M'Hallah[20] on an extension to the 2DBPP with due dates for the packed items.

A number of optional constraints may be imposed on the standard problem statement of the 2DBPP. If the rectangular items must packed with their sides parallel to that of the bin edges, then the problem is referred to as *orthogonal*. If the rectangular items are allowed to be rotated by 90 degrees, then the problem has the additional descriptor of *orientable*. When applied to wood and metal cutting problems, *guillotine* constraints are usually imposed on the placements of items in each bin so that feasible patterns may be obtained. The reader should consult the articles by Lodi et al.[3,4] for other applications of the 2DBPP.

Solution approaches to the classical 2DBPP are many and varied, including the application of meta-heuristics, such as *tabu search* (see Lodi et al.[5]), approximation algorithms (see Lodi et al.[6]), and exact methods utilizing Dantzig–Wolfe-based branch-and-price-and-cut (see Pisinger and Sigurd[7]). Furthermore, the article by Pisinger and Sigurd,[7] motivated by the work of Martello et al.,[8] proposed a *Constraint Satisfaction Problem* (CSP) approach to their *column generation* pricing problem. Martello et al.[8] adopted a CSP approach to their problem due to performance considerations, whereas Pisinger and Sigurd[7] selected a CSP to enable greater modeling flexibility.

Côté et al.[9] introduced a branch-and-cut algorithm for solving a 2DBPP with *unloading constraints* (relevant to delivery-based vehicle routing problems, see Gendreau et al.[10]). The problem is concerned with feasibly packing a set of rectangular items onto the loading area of a vehicle (a single finite rectangular bin). A feasible packing is described as one in which the items belonging to a customer are directly retrievable at the moment of delivery, that is, the required items may be accessed without moving other items. An exact branch-and-cut solution approach for the problem was proposed and outlined by Côté et al.[9] A slightly different approach was taken by da Silveira et al.,[11] who studied the *strip packing* variant of the unloading problem with approximation algorithms. The strip packing problem assumes a single bin of a given width and undetermined length, that is, the bin is open at one end. The problem is to pack a list of items such that the resulting bin can be truncated at minimum length.

At this point, we wish to draw attention to a few key modeling and solution features that are relevant to the scope of our study. Firstly, consider the constraint programming (CP) paradigm adopted by Martello et al.[8] and Pisinger and Sigurd.[7] Given that our amphibious embarkation planning problem contains a number of complex constraints, the expressive power of a CP framework provides us with a number of exploratory modeling options. These options are facilitated by the specialized logic-based constraints and filtering algorithms common to most CP solvers. A CP approach to the embarkation planning problem would ensure that constraints for mass balance, priority ordering, and the inclusion of obstacles can be succinctly stated and processed coherently by the solver. Secondly, we note that heuristic and approximation approaches to the strip packing problem with unloading constraints have recently been considered by da Silveira et al.[11] in preference to exact algorithms. Because our intent is to explore iterative solution approaches via automated and manual packing techniques, a CP modeling framework will permit the opportunity to experiment with various heuristics on a range of initial candidate packing configurations.

A CP-based modeling approach to the amphibious embarkation planning problem, combined with the ability to explore iterative heuristic and manual packing techniques, appears to be an unexamined area of research in the bin packing domain. This provides ample justification for adopting such an approach in this study. Before introducing our model and the proposed heuristic approaches, the next section will cover the essential theory and features of CP.

2. Primer on constraint programming

Before we present our modeling approach to the amphibious embarkation planning problem, it is important to introduce the vocabulary and features of CP. This *primer* section, which is included for the purposes of completeness, may be skipped if one is already familiar with the basics of CP. The content of this section has been inspired and motivated from the work of Dechter,[12] Milano and Trick,[13] and Smith.[14] These texts, which are referenced throughout, are recommended as resources for further didactic reading.

2.1. Relations

Given a set of *variables* $X = \{x_1, \ldots, x_n\}$ with associated *domains* $D = \{D_1, \ldots, D_n\}$, a *relation* R on the set of variables is defined as a subset of the *cartesian product* of the domains:

$$R \subseteq \bigotimes_{i=1}^{n} D_i := \{(x_1, \ldots, x_n) | x_1 \in D_1, \ldots, x_n \in D_n\}. \quad (1)$$

The *scope* of a relation is denoted by S, and is the set of variables on which a relation is defined. The cardinality of the scope of a relation is called the *arity* of the relation.

That is, if the scope of a relation is $\{x_1, \ldots, x_n\}$, the relation is said to have arity n.

For example, say we have two variables, x_1 and x_2, with the following domains:

$$x_1 \in D_1 = \{\text{blue, pink, yellow}\}, \tag{2}$$

$$x_2 \in D_2 = \{\text{moon, panther, submarine}\}. \tag{3}$$

Then the following set forms an arity-2 relation on the scope $S = \{x_1, x_2\}$:

$$R = \{(\text{blue, moon}), (\text{pink, panther}), (\text{yellow, submarine})\}. \tag{4}$$

2.2. Constraints

Let $X = \{x_1, \ldots, x_n\}$ be a set of variables with associated domains $D = \{D_1, \ldots, D_n\}$. A set $C = \{c_1, \ldots, c_m\}$ on (X, D) is called a *constraint set* if the elements of C are scope–relation pairs: $c_i = (S_i, R_i)$ for all $i \in \{1, \ldots, m\}$, where $S_i = \{x_{i_1}, \ldots, x_{i_{|S_i|}}\}$. Since a constraint c_i can be stated solely in terms of a scope and an associated relation, it need not have a closed mathematical form. (See Lustig and Puget[21] for more information.)

A constraint can also be defined as a mapping:

$$c_i : \bigotimes_{j=1}^{|S_i|} D_{i_j} \to \{0, 1\}, \quad \forall i \in \{1, \ldots, m\}. \tag{5}$$

An *instantiation* of values to the variables of the scope S_i of a constraint c_i is an assignment, $x_{i_j} \mapsto v_{i_j} \in D_{i_j}$, for all $j \in \{1, \ldots, |S_i|\}$. An instantiation is said to *satisfy* a constraint if it is an element of the associated relation:

$$c_i(v_{i_1}, \ldots, v_{i_{|S_i|}}) = \begin{cases} 1 & \text{if } (v_{i_1}, \ldots, v_{i_{|S_i|}}) \in R_i, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

To give a concrete example of a constraint, consider the variables x_1 and x_2 with domains given by the following:

$$x_1 \in D_1 = \{\text{family, full, growing}\}, \tag{7}$$

$$x_2 \in D_2 = \{\text{guy, house, matters, pains, ties}\}. \tag{8}$$

We can define a constraint c on the scope $S = \{x_1, x_2\}$ as follows: (x_1, x_2) forms the name of a television situation comedy. In this case $c = (S, R)$, where the relation R is given by the following:

$$R = \{(\text{family, guy}), (\text{family, matters}), (\text{family, ties}), (\text{full, house}), (\text{growing, pains})\}. \tag{9}$$

As demonstrated above, one of the benefits of adopting a constraint-based approach to a combinatorial problem is the ability to declaratively state complex constraints through the use of relations. These may include constraints with non-arithmetic and logic-based operators. The CP paradigm also permits the use of specialized symbolic or *global constraints*. A global constraint encapsulates a specialized relation between variables and serves as a succinct representation of a group of more fundamental constraints. Global constraints are also equipped with tailored constraint propagation techniques known as *domain filtering* algorithms, which will be discussed after the following section on modeling.

2.3. Modeling

Let X be a set of variables with respective domains in the set D, and let C be a set of constraints on (X, D). A CSP seeks an assignment of values to the variables such that each constraint is satisfied. Such an assignment is called a *feasible solution*. The goal of a CSP may be to find one, some, or all feasible solutions.

As there may be more than one way to represent a CSP, the triple (X, D, C) can be referred to as a *model* of a CSP. We note that the correctness of a model is only a necessary condition for its selection – it may not be a sufficient one if we are concerned with solving a problem in the most efficient way.

On model selection, Smith[14] suggests that a ''good'' model should contain as few constraints as possible, with all constraints expressed in a concise manner. Constraints that are known to possess efficient and low complexity propagation algorithms should be selected in preference to ones that do not possess comparable performance. However, there may be situations in which adding constraints can safely ignore unnecessary parts of the search space and hence shrink the combinatorial landscape. This is generally true of problems that contain a high degree of symmetry, where adding symmetry breaking constraints can greatly increase the efficiency of propagation and search.[9]

2.4. Propagation

In CP, the solving process is composed of two primary techniques; these are *constraint propagation* and *constructive search*. Constraint propagation is performed on the CSP's initial formulation (this is referred to as initial propagation) and then subsequently interleaved with the constructive search. There are three fundamental steps to constructing and solving a CSP (see Milano and Trick[13]):

1. defining the variables and their domains;
2. stating the constraints between the variables;
3. selecting a search strategy.

Constraint propagation involves domain filtering, that is, removing values from the domains of decision variables

that are inconsistent with the constraint set. The primary purpose of constraint propagation is to reduce the size of the search space. (The search space is the set of all possible combinations of assignments of values to decision variables.) Propagation generally relies on *consistency techniques*. Consistency techniques are enforced locally through partial assignments and attempt to remove values from the domains of variables that are logically inconsistent with a feasible solution. Such values are therefore said to be locally inconsistent. A local inconsistency is a partial assignment to the variables that may satisfy some of the constraints but cannot be extended to additional variables and, hence, cannot be part of a solution to the CSP. In general, CP solvers will utilize two types of consistency, namely, *arc consistency* and *bounds consistency*.

2.4.1. Arc consistency. A binary constraint on variables x_1 (domain D_1) and x_2 (domain D_2) is arc consistent from x_1 to x_2 if and only if for each $v_{x_1} \in D_1$ there exists $v_{x_2} \in D_2$ satisfying c. This is written symbolically as follows:

$$(\forall v_{x_1} \in D_1)(\exists v_{x_2} \in D_2) \\ [(x_1 \mapsto v_{x_1}) \wedge (x_2 \mapsto v_{x_2}) \Rightarrow c(x_1, x_2) = 1]. \quad (10)$$

The constraint c is arc consistent if it is arc consistent from x_1 to x_2 and arc consistent from x_2 to x_1. An instance of a CSP (that is, an instantiation of values to variables) is arc consistent if all of its binary constraints are arc consistent. The concept of arc consistency can be extended to constraints of larger arity through generalized arc consistency (also known as hyper-arc consistency).

2.4.2. Bounds consistency. Given a constraint c with scope S, a variable $x \in S$ is bounds consistent relative to c if $x \mapsto \min\{D_x\}$ (or $x \mapsto \max\{D_x\}$) and can be extended to assignments of values to variables in $S \backslash \{x\}$ which satisfy c. Constraint c is bounds consistent if each of its variables is bounds consistent relative to c (see Dechter[12]). According to Smith,[14] CP solvers will try to enforce arc consistency on some, but not necessarily all, binary constraints and enforce bounds consistency on arithmetic constraints (this is the default setting for the ILOG CP Optimizer). Smith[14] also notes that CP solvers will not usually try to maintain generalized arc consistency on non-binary constraints, except for global constraints that possess efficient filtering algorithms.

2.5. Constructive search

Initial constraint propagation will terminate in one of the following states (see Milano and Trick[13]):

1. at least one variable has an empty domain, in which case, the problem is unsatisfiable;
2. a solution is found, that is, each variable is left with exactly one value in its domain;
3. at least one domain of a variable contains more than a single value.

So while constraint propagation may be sufficient to solve some CSPs without any additional techniques, it does not necessarily guarantee that solutions will be found. In other words, after initial constraint propagation has been completed, the remaining values in each variable domain may not guarantee the satisfaction of all constraints. Therefore, propagation has to be interleaved with a branching strategy that can explore the remainder of the search space.

The constructive search is performed after initial constraint propagation and involves a guided traversal of the branches of a search tree (representing the search space). The leaf nodes of the search tree contain the possible combinations of assignments of values to decision variables. The most commonly used search strategy is *depth-first search* with *chronological backtracking* (see Milano and Trick[13]), with the amount of propagation during the search determined by the type of problem under consideration.

A general backtracking search takes a partial assignment and tries to extend it by assigning values to un-instantiated variables. If an assignment produces a constraint violation and cannot be extended to a solution, then it is discarded (the tree node is pruned) and a different assignment is tried (backtracking). A search tree is generated as different assignments are tried, and the method of extending a node in the search tree is called a *branching strategy*. A branching strategy coupled with a set of constraint propagation rules forms the basis of a backtracking algorithm.

2.6. Constraint optimization

If a CSP also requires that an objective function, for instance $f : X' \subseteq X \rightarrow \mathbb{R}$, be maximized or minimized, then it is called a *Constraint Optimization Problem* (COP). When considering a COP, we want to find the maximum or minimum value of some objective function. In such cases, CP solvers will implement a pseudo branch-and-bound algorithm to find the optimal solution. This is carried out by solving a sequence of feasibility problems that produce successively better objective values. When a better objective value is found for a new feasible solution, it is used to bound the objective function (from below or above, depending on whether the max or min is under consideration) with respect to all sub-problems in the search tree.

3. Problem statement and solver

At this point, we have provided a high-level description of the amphibious embarkation planning problem and detailed the salient features of CP approaches to hard combinatorial problems. In this section, we will use the vocabulary introduced heretofore to provide a detailed CP modeling approach to the amphibious embarkation planning problem.

Our modeling approach to the amphibious packing problem broadly follows the conventions defined by Onodera et al.[15] and Chen et al.[16] Even though these studies utilized a binary integer programming modeling approach, we can easily adapt the same paradigm to a CP framework. The details of our CP modeling approach will be outlined after we have introduced the input data, coordinate system for the deck, and the problem's decision variables.

3.1. Input data

The input data for a packing problem consists of a ship deck and a set of non-homogeneous, non-orientable (non-rotatable) rectangular items. The deck and each rectangular item can be defined by a collection of parameters. A list is provided below in the form of "*symbol* (*units*) – *description*":

- W (mm) – width of the deck (along the x-axis);
- L (mm) – length of the deck (along the y-axis);
- A (mm^2) – area of the deck ($= W \times L$);
- M (kg) – mass holding capacity of the deck;
- μ (kg) – mass threshold for rectangular items;
- r_x (mm) – center of mass tolerance along the x-axis;
- r_y (mm) – center of mass tolerance along the y-axis;
- $\mathcal{R}$ (N/A) – set of rectangular items;
- $\mathcal{R}''$ (N/A) – set of obstacles ($\mathcal{R}'' \subset \mathcal{R}$);
- X_i (mm) – x-ordinate of obstacle $i \in \mathcal{R}''$;
- Y_i (mm) – y-ordinate of obstacle $i \in \mathcal{R}''$;
- w_i (mm) – width of rectangle $i \in \mathcal{R}$;
- l_i (mm) – length of rectangle $i \in \mathcal{R}$;
- a_i (mm^2) – area of rectangle $i \in \mathcal{R}$ ($= w_i \times l_i$);
- m_i (kg) – mass of rectangle $i \in \mathcal{R}$.

3.2. Coordinate system

The origin of the coordinate system $O(0, 0)$ is defined to coincide with the bottom left-hand corner of the deck. The positive x-axis runs along the bottom side of the deck, while the positive y-axis runs along the left-hand side of the deck (see Figures 3 and 4). For each rectangular item $i \in \mathcal{R}$, the position at which it is packed on the deck (x_i, y_i) is also with respect the rectangle's bottom left-hand corner.

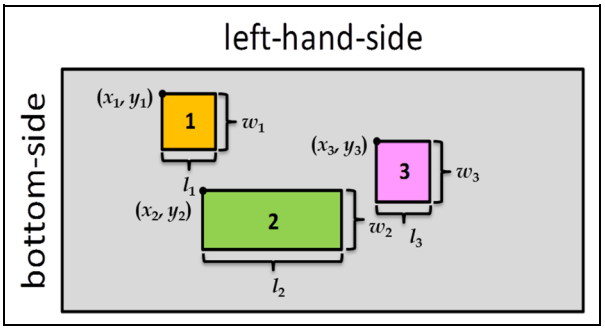

Figure 3. Illustration of the relative positions of three items packed on a deck. The bottom side and left-hand side of the deck are clearly labeled. Note that the deck is visually depicted as having been rotated clockwise by 90 degrees.

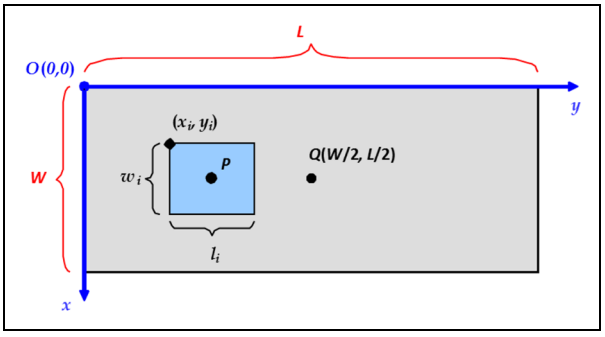

Figure 4. Coordinate system and deck for a packing problem.

We will use the terms *left* and *below* relative to the deck's coordinate system, where the bottom side of the deck is the x-axis and the left-hand side of the deck is the y-axis (Figure 4). Throughout this paper, however, all pictorial depictions of packing configurations on decks will be displayed as having been rotated clockwise by 90 degrees. (This choice has been made to aid formatting and visual presentation.) An illustrative example of a deck with three packed items, with the bottom side and left-hand side clearly labeled, can be found in Figure 3.

The geometric center (the centroid) of the deck is located at $Q(W/2, L/2)$, where W is the width of the deck (with respect to the x-axis) and L is the length of the deck (with respect to the y-axis). A diagrammatic representation can be found in Figure 4. It is important to note that center of mass calculations are performed by finding the relative position of a rectangular item's centroid from the deck's centroid:

$$\overrightarrow{OP} = \left(x_i + \frac{w_i}{2}\right)\hat{i} + \left(y_i + \frac{l_i}{2}\right)\hat{j}, \tag{11}$$

where $\hat{i}$ and $\hat{j}$ are the unit basis vectors for a two-dimensional cartesian coordinate system.

This assumes that the center of mass coincides with the centroid. We note that this is not always the case in practice. We expect that the model can be easily adapted to accommodate alternative geometries. The centroid P of a rectangular item $i \in \mathcal{R}$ is given through the position vector. If the centroid of the deck is given through the position Q, then the relative position of the centroid of rectangular item $i \in \mathcal{R}$ from the deck's centroid is given by the following:

$$\overrightarrow{QP} = \overrightarrow{OP} - \overrightarrow{OQ}, \tag{12}$$

$$= \left(x_i + \frac{w_i - W}{2}\right)\hat{i} + \left(y_i + \frac{l_i - L}{2}\right)\hat{j}. \tag{13}$$

3.3. Decision variables

The modeling approach chosen for the packing problem is a CP paradigm. CP was chosen due to its powerful modeling flexibility, the broad scope for user interaction with the model, and the ability to combine customized heuristic techniques with a commercial or open-source CP branch-and-bound solver.

The decision variables for the problem are discrete. Each decision variable can take a value from its domain, that is, the set of possible assignments of values to variables. A list of the decision variables, their respective domains, and qualitative descriptions can be found below.

- $u_i \in \{0, 1\}$: a boolean, equal to 1 (TRUE) if rectangle $i \in \mathcal{R}$ is packed, and 0 (FALSE) otherwise.
- $x_i \in \{0, 1, \ldots, W - w_i\}$: an integer-valued variable defining the x-ordinate of a rectangle's position on the deck.
- $y_i \in \{0, 1, \ldots, L - l_i\}$: an integer-valued variable defining the y-ordinate of a rectangle's position on the deck.
- $\ell_{ij} \in \{0, 1\}$: a boolean, equal to 1 if rectangle $i \in \mathcal{R}$ is placed to the *left* of rectangle $j \in \mathcal{R}$ ($i \neq j$), and 0 otherwise. (i is to the left of j if $x_i + w_i \leq x_j$.)
- $b_{ij} \in \{0, 1\}$: a boolean, equal to 1 if rectangle $i \in \mathcal{R}$ is placed *below* rectangle $j \in \mathcal{R}$ ($i \neq j$), and 0 otherwise. (i is below j if $y_i + l_i \leq y_j$.)

To see how these variables relate to each other in a given packing configuration, we can refer to the set-up given in Figure 3. Under this arrangement, it can be seen that $x_1 + w_1 \leq x_2$, and therefore rectangle 1 is to the left of rectangle 2, yielding $l_{12} = 1$. Since $y_1 + l_1 \leq y_3$ and $y_2 + l_2 \leq y_3$, rectangles 1 and 2 are below rectangle 3, and hence $b_{13} = b_{23} = 1$.

Note that we need to know the relative positions of the rectangles on the deck, not just their packed positions. This is to ensure that items do not overlap with each other, nor with the boundary of the deck space. Thus, for any two packed rectangles i and j, at least one of ℓ_{ij}, ℓ_{ji}, b_{ij}, b_{ji} must be equal to 1. This is expressed mathematically in constraint (25).

3.4. Constraint Optimization Problem

3.4.1. Objective. The objective of the packing problem is to maximize the total area of packed rectangles:

$$\max\left(z = \sum_{i \in \mathcal{R}} u_i a_i\right). \tag{14}$$

With this objective, the problem is similar to that of the two-dimensional cutting knapsack problem (2DCKP, see Gilmore and Gomory[17]).

3.4.2. Derived bounds. If a rectangle $i \in \mathcal{R}$ has width (length) exceeding the width (length) of the deck or mass exceeding the mass threshold, then the rectangle cannot be packed:

$$\begin{aligned} &[(w_i > W) \vee (l_i > L) \vee (m_i > \mu)] \\ &\Rightarrow \quad (u_i = 0), \quad \forall i \in \mathcal{R}. \end{aligned} \tag{15}$$

Any rectangle that cannot be packed because of a constraint violation (15) is placed in the set of un-packable rectangles, which we denote as $\mathcal{R}' \subset \mathcal{R}$. Equipped with this set, a bound can be derived for the objective function and the total mass of packed rectangles:

$$\sum_{i \in \mathcal{R}} u_i a_i \leq \min\left\{A, \sum_{i \in \mathcal{R} \setminus \mathcal{R}'} a_i\right\}, \tag{16}$$

$$\sum_{i \in \mathcal{R}} u_i m_i \leq \min\left\{M, \sum_{i \in \mathcal{R} \setminus \mathcal{R}'} m_i\right\}. \tag{17}$$

In addition, the variable domains for the rectangle positions imply the following bounds constraints:

$$0 \leq x_i \leq W - w_i, \quad \forall i \in \mathcal{R}, \tag{18}$$

$$0 \leq y_i \leq L - l_i, \quad \forall i \in \mathcal{R}. \tag{19}$$

3.4.3. Obstacles. The set of obstacles is a subset of the set of rectangular items. We write this set as $\mathcal{R}'' \subset \mathcal{R}$, with $\mathcal{R}''$ being mutually exclusive with the set of un-packable rectangles, that is, $\mathcal{R}' \cap \mathcal{R}'' = \varnothing$. An obstacle $i \in \mathcal{R}''$ must be placed at a pre-determined location (X_i, Y_i) on the deck.

Hence, $u_i = 1$, $x_i = X_i$ and $y_i = Y_i$ for all $i \in \mathcal{R}''$. This can be expressed as follows:

$$(u_i = 1) \wedge (x_i = X_i) \wedge (y_i = Y_i), \quad \forall i \in \mathcal{R}''. \tag{20}$$

From the pre-determined positions of the obstacles, the decision variables for the relative positions can also be determined as follows:

$$(X_i + w_i \leq X_j) \Rightarrow (\ell_{ij} = 1), \quad \forall i, j \in \mathcal{R}''(i \neq j), \tag{21}$$

$$(Y_i + l_i \leq Y_j) \Rightarrow (b_{ij} = 1), \quad \forall i, j \in \mathcal{R}''(i \neq j). \tag{22}$$

3.4.4. Packing feasibility. The constraints pertaining to the relative positions between all pairs of rectangles are defined as follows:

$$(\ell_{ij} = 1) \Leftrightarrow (x_i + w_i \leq x_j), \quad \forall i, j \in \mathcal{R}(i \neq j), \tag{23}$$

$$(b_{ij} = 1) \Leftrightarrow (y_i + l_i \leq y_j), \quad \forall i, j \in \mathcal{R}(i \neq j). \tag{24}$$

For any two rectangles $i, j \in \mathcal{R}$ (where $i < j$), if $u_i = u_j = 1$, then i must be placed to the left, right, below, or above j. This is expressed mathematically as follows:

$$\ell_{ij} + \ell_{ji} + b_{ij} + b_{ji} + (1 - u_i) + (1 - u_j) \geqslant 1, \quad \forall i, j \in \mathcal{R}(i < j). \tag{25}$$

For a pair of rectangles $i, j \in \mathcal{R}$ (where $i < j$), if at least one of u_i and u_j is zero, then constraint (25) becomes redundant (see Pisinger and Sigurd[7]). Finally, to ensure that any two distinct rectangles $i, j \in \mathcal{R}$ (where $i \neq j$) do not overlap within the confines of the deck, we require the following constraints:

$$x_i - x_j + W\ell_{ij} \leq W - w_i, \quad \forall i, j \in \mathcal{R}(i \neq j), \tag{26}$$

$$y_i - y_j + Lb_{ij} \leq L - l_i, \quad \forall i, j \in \mathcal{R}(i \neq j). \tag{27}$$

3.4.5. Mass balance. The center of mass must be within the prescribed threshold values. This can be expressed mathematically through the following constraints:

$$\left| \sum_{i \in \mathcal{R}} u_i m_i \left(x_i + \frac{w_i - W}{2} \right) \right| \leq r_x \sum_{i \in \mathcal{R}} u_i m_i, \tag{28}$$

$$\left| \sum_{i \in \mathcal{R}} u_i m_i \left(y_i + \frac{l_i - L}{2} \right) \right| \leq r_y \sum_{i \in \mathcal{R}} u_i m_i. \tag{29}$$

3.4.6. Priority ordering and symmetry breaking. If the set of rectangles $\mathcal{R}$ is an indexed set defining a priority ordering, then the rectangles should be packed in a way that closely reflects the prioritization. If $i < j$, then j should not be packed below i. If $j = i + 1$, then j must be to the left or to the right of i. These conditions exclude the obstacles. Mathematically, we write this as follows:

$$b_{ji} \neq 1, \quad \forall i, j \in \mathcal{R} \backslash \mathcal{R}''(i < j), \tag{30}$$

$$\ell_{ij} + \ell_{ji} \geqslant 1, \quad \forall i, j \in \mathcal{R} \backslash \mathcal{R}''(j = i + 1). \tag{31}$$

If priority ordering is not considered, then it is advantageous to add symmetry breaking constraints to eliminate swapping between equivalent arrangements of rectangles on the deck. If any two rectangles are identical, we require that the rectangle with the lower index be packed to the left of or below the other. That is, for any two rectangles, $i, j \in \mathcal{R} \backslash \mathcal{R}''$ with $i < j$, we have the following:

$$(w_i = w_j) \wedge (l_i = l_j) \wedge (m_i = m_j) \Rightarrow (\ell_{ij} + b_{ij} \geqslant 1) \wedge [(x_i \leq x_j) \vee (y_i \leq y_j)], \quad \forall i, j \in \mathcal{R} \backslash \mathcal{R}''(i < j). \tag{32}$$

3.5. Solver

Once the CP model has been constructed from the input data, it can be passed to a CP solver to find an optimal solution that satisfies the feasibility constraints. We have chosen to use the ILOG CP Optimizer – a state-of-the-art and commercial optimization engine – as the solver for our CP model. As referenced in Section 2, the solver uses two techniques for solving a COP, namely, *constructive search* and *constraint propagation*.

3.6. Example

In this section, we present an example of a packing scenario that is typical of problems that can be efficiently solved with the direct application of our modeling framework and the ILOG CP Optimizer (no additional heuristics required). Consider Example Deck *A*, of dimensions 15.8 meters × 93.5 meters and containing three obstacles, as shown in Figure 5. We consider the problem of attempting to pack a set of 36 candidate items onto the deck. A comparison of the relative sizes of the candidate items can be

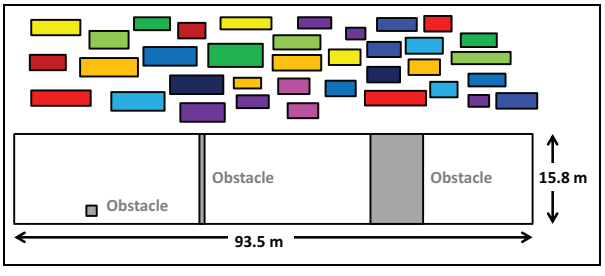

Figure 5. Example Deck *A* with three rectangular obstacles.

Table 1. The candidate pack list for Example Deck *A*. There are three obstacles (O) and 36 rectangular items (R) to be packed (of which there are 18 types).

Item	Quantity	Mass (tonne)	Dimensions (m)
R1	9	18.00	5.79×2.60
R2	4	63.10	9.83×3.66
R3	3	12.00	6.54×2.50
R4	2	13.60	10.67×3.05
R5	2	19.04	9.02×2.50
R6	2	8.10	11.01×2.50
R7	2	10.80	5.40×2.69
R8	2	3.69	3.50×1.98
R9	1	14.50	8.99×2.59
R10	1	21.11	7.00×3.30
R11	1	6.50	6.50×1.84
R12	1	11.76	5.00×2.70
R13	1	6.45	8.27×3.43
R14	1	19.26	9.48×2.50
R15	1	4.60	4.78×2.17
R16	1	15.00	7.18×2.48
R17	1	12.04	7.30×2.60
R18	1	5.60	6.10×2.07
O1	1	0.00	9.50×15.80
O2	1	0.00	1.75×2.00
O3	1	0.00	0.50×15.80

seen in Figure 5, with information pertaining to quantity, mass, and dimensions of each item type located in Table 1.

The CP formulation of this problem contains roughly 3,000 variables and 15, 800 constraints. If we relax the mass balance constraints, that is, we allow for a large tolerance on the location of the center of mass, then it takes only 0.68 seconds to pack all the items listed in Table 1 onto the deck of Figure 5. In this case, the number of branching decisions taken by the solver was 864 with eight fails (pruned nodes). If we impose rather stringent mass balance constraints, namely, we stipulate that the center of mass of the packed items must coincide with the geometric center of the deck, then it takes the solver 2.90 seconds to solve (approximately four times longer than the relaxed case). In this case, the solver had to work significantly harder than before to find a solution, with 5,014 branching decisions and 605 fails. These examples were run on a 3.16 GHz dual-core desktop with 4.00 GB of RAM running Windows XP.

4. Constraint programming-based heuristics

As demonstrated in the previous example, the CP solver underpinning the packing model may need to explore a large space of possibilities to find a feasible or optimal solution. When there are many options open to the solver it slows down as it tries to determine which options are likely to be the most fruitful. In the event that there are too many options, the exploration space may become so large that the solver will not show anything more than an initial/partial solution in a reasonable amount of time.

It may be advantageous, therefore, to help the solver find an optimal solution or high-quality partial solution in a reasonable time frame by limiting the choices that it can make. If we do this in a way that helps the solver find a good quality partial solution quickly, but cannot guarantee the optimal solution, we call this approach a *heuristic*.

We have chosen to employ two main heuristics. The first aggregates and packs rectangular items (primarily to avoid symmetries), and the second sequentially packs rectangular items while concomitantly constraining positions. The sequential packing routine is designed to limit the degree to which packed items can move around on the deck as other items are subsequently packed.

4.1. Aggregation

If two rectangular items are identical in every way (width, length, and mass), then a feasible packing in which both items are present will be identical to a feasible packing in which the positions of both items are swapped. Accordingly, the solver would be wasting effort if it tries to find both of these feasible packing solutions. In common with many other optimization problems, we can improve solution outcomes by trying to ''break'' symmetries. One way to do this is to add constraints to the model that reduce the number of equivalent feasible solutions the solver can explore.

Alternatively, we can make some decisions that try to remove these symmetries from the problem statement to begin with. The approach here is to use an aggregation heuristic. This heuristic attempts to remove the presence of identical items by aggregating them together into larger items.

The aggregation heuristic loops through the priority list of items to be packed, making comparisons between the characteristics of adjacent items. When two adjacent items have the same dimensions and mass, they are aggregated into a single item. We restrict ourselves to items adjacent in the list in order to produce packing solutions that should reflect the priority ordering (when it is invoked).

We allow the user to execute this heuristic by specifying the number of times to sweep through the list and aggregate adjacent pairs of items. In the first iteration, we arbitrarily aggregate identical pairs by aggregating along one side (the vertical side or the horizontal side). The next iteration sweeps through the updated list by aggregating items along the other side. This then continues for the number of iterations specified by the user. An example of this aggregation routine is illustrated in Figure 6.

In addition to removing symmetry, the aggregation heuristic can improve the solver's runtime by lowering the

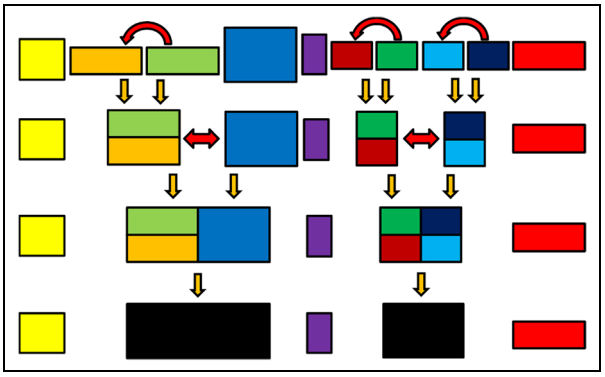

Figure 6. Example of a two-step aggregation process. The first step is a top–bottom aggregation and the second is a side-by-side aggregation.

number of candidate items to be packed. This approach may produce sub-optimal solutions, as it is possible that some constituent elements of an aggregated item could be more profitably used elsewhere on the deck, such as, for example, fitting into small gaps of the packing solution.

4.2. Constraining positions

Initially, the model assumes that an item can be packed anywhere on the deck. It is, however, possible that an operator may have a good idea of where the item should be packed. At the simplest level, the operator should be able to indicate this as a constrained range for the item. (for example, see Figure 7).

Drawing on this concept, the ''constraining positions'' heuristic uses a rectangle's current location (if it has one) as an indication of approximately where it should go and allows it to be offset by either a fixed distance or a distance proportional to its size.

This heuristic can be executed from within the model as an additional constraint that reads pre-set range restrictions for each item and requires that all items, if packed, are within the range settings. The heuristic can also be implemented outside of the model, where it simply allows the user to make a choice of how much allowance to give each rectangle from its current position.

4.3. Sequential packing

It is possible to use the constraining positions heuristic (described in the above section) in an iterative manner to sequentially pack items onto the deck. This is performed according to the following sequential packing heuristic:

1. pack the first item (this is usually a trivial task);
2. constrain all the currently packed items to be near their current position;
3. add a new item to be packed;
4. execute the solver on this restricted problem; and
5. if infeasible, or another type of error is found, break, otherwise continue by going back to 2.

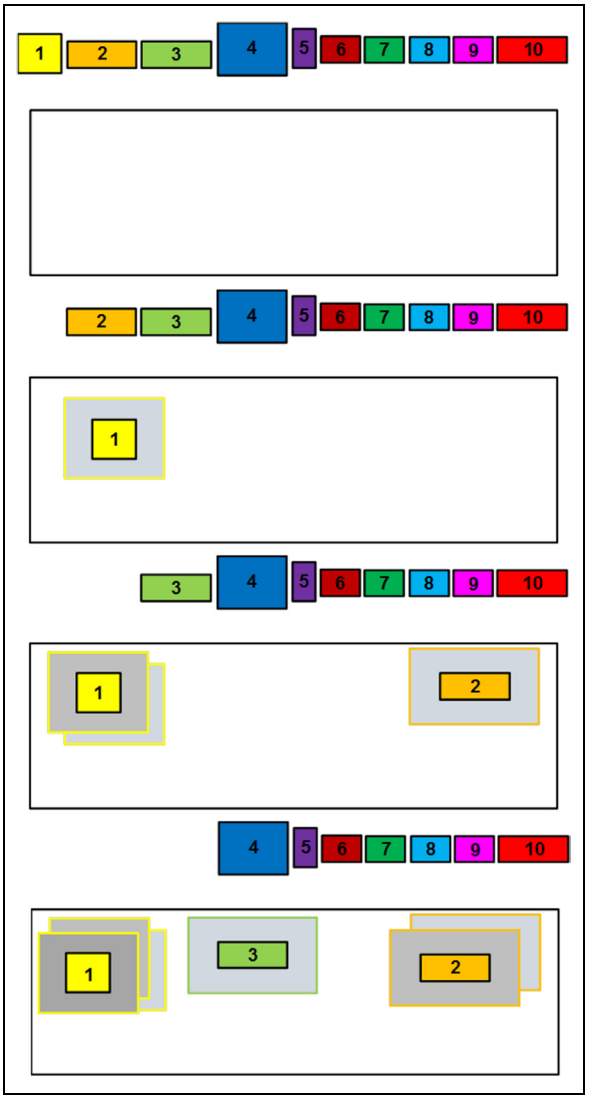

Figure 7. A pictorial representation of the iterations in a sequential packing routine. As each item is packed, its range of movement in the next iteration is restricted to a local neighborhood defined by its current position.

A pictorial depiction of the sequential packing routine can be found in Figure 7.

In each execution of the solver, if all of the packed items are tightly constrained then the resulting problem should solve fairly quickly. When there are only a few items to be packed they can be feasibly placed in almost

random locations. It is important, therefore, to ensure that they can move around to more optimal locations as additional items are packed on the deck. We can accomplish this by giving generous movement allowances early on in the sequence and then slowly limiting the allowances as the sequence progresses and the density of packed items increases. This is an example of a local or neighborhood search algorithm, but expressed in the parlance of CP. For a comprehensive introduction to this class of techniques, see Van Hentenryck and Michel.[22]

The sequential packing heuristic assumes that the packing process takes place in two phases. The first phase limits the allowable movement of each packed item by an absolute distance; the second phase limits the movement of each packed item by a fixed fraction of the item's size.

More formally, the user provides the following four inputs to the sequential packing heuristic:

1. ρ_0, the distance that all items can move once packed at the start of phase 1;
2. ρ_1, the distance that all items can move once packed, at the end of phase 1;
3. k, the iteration of the sequential packing routine where the heuristic first changes from phase 1 to phase 2; and
4. λ, the ratio (expressed as a percentage) of an item's size, which limits its movement during phase 2.

Mathematically, the constraints on the position (x_i, y_i) of an item i that has width w_i and length l_i and was packed at position $(X_i^{(n)}, Y_i^{(n)})$ in the preceding iteration n will be according to (33) and (34):

$$|x_i - X_i^{(n)}| \leq \begin{cases} \frac{1}{k}[(k-n)\rho_1 + n\rho_0] & \text{if } n \leq k, \\ \lambda w_i & \text{otherwise,} \end{cases} \quad (33)$$

$$|y_i - Y_i^{(n)}| \leq \begin{cases} \frac{1}{k}[(k-n)\rho_1 + n\rho_0] & \text{if } n \leq k, \\ \lambda l_i & \text{otherwise.} \end{cases} \quad (34)$$

This heuristic is not implemented inside of the CP model; instead, it is intended to be implemented outside of the model by iteratively adding items to the set of items to be packed and setting the bounds on all items packed in the previous iteration.

5. Illustrative computational experiments with packing heuristics

In this section, we will explore some typical runtime executions of the packing heuristics with a few computational experiments. This does not constitute a rigorous benchmarking of the performance of the model, solver, or heuristics. We simply present a few prototypical examples in order to give a general indication of the computational performance of the CP model and solver in concert with the sequential packing and aggregation heuristics. All computational experiments were run on a 3.16 GHz dualcore desktop with 4.00 GB of RAM running Windows XP.

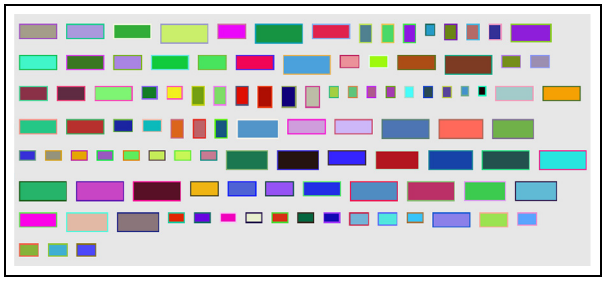

Figure 8. The list of rectangular items to be packed for the computational experiments (108 in total).

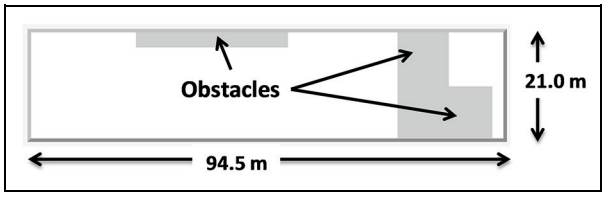

Figure 9. Example Deck *B* with three rectangular obstacles.

For our computational experiments, we will consider a single list of 108 rectangular items to be packed, as shown in Figure 8. Each rectangle in Figure 8 has been augmented with an artificial border (buffer region) of 200 mm. In order to illustrate the design concept of the developed packing heuristics, we will not give consideration to the mass of the rectangular items in the computational experiments that follow.

We refer to two different decks in our experiments. Example Deck *B* has dimensions 21.0 meters $\times$ 94.5 meters, with three rectangular obstacles, as shown in Figure 9 (note: the aggregated obstacle on the right-hand side of Example Deck *B* is made up of two rectangular components). Example Deck *C* has the same dimensions as Example Deck *A*, namely, 15.8 meters $\times$ 93.5 meters, but does not contain any obstacles.

5.1. Packing with obstacles – Example Deck B

We present a number of computational test runs using Example Deck *B* (Figure 9). We will firstly examine the effect of applying varying levels of aggregation to the item list (Figure 8) before passing the updated (aggregated) list to ILOG's CP solver. We will then consider a sequential packing heuristic (according to decreasing rectangle size) with constrained movement.

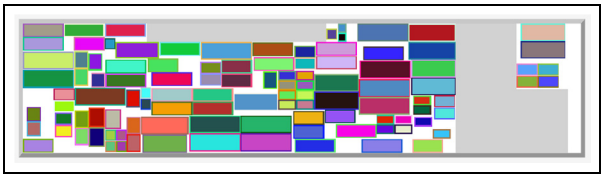

Figure 10. Solution returned from the solver after using three rounds of aggregation on the item list of Figure 8.

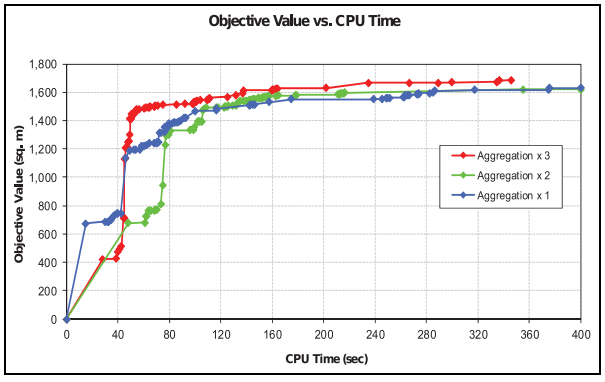

Figure 11. Objective value found by the solver versus CPU time for various levels of aggregation.

5.1.1. Solver only with aggregation. We run the aggregation heuristic once, twice, and three times before passing the updated (newly aggregated) item list to the solver for packing onto Example Deck *B*. A time out limit of 400 seconds was used in each of the three cases. Using three runs of the aggregation heuristic, the solver was able to return an optimal solution in approximately 346 seconds of CPU time. See Figure 10 for an illustration of the packing solution.

Using one run and two runs of the aggregation heuristic, an optimal solution could not be returned by the solver. In both cases, the solver timed out with an objective value of approximately 96% of the optimal objective value.

The optimal value of the objective function was returned by the solver for the $3\times$ aggregation case. The optimal objective value was then used to determine the optimality gap for the $1\times$ and $2\times$ aggregation cases. We can be sure of optimality, because in this case all items were packed.

The chart in Figure 11 shows the value of the objective function returned by the solver versus the elapsed CPU time for each of the three aggregation runs. The markers indicate the times at which the solver found a better objective value. In all cases, it can be seen that only very small improvements in the objective function are returned by the solver as the objective value gets close to the optimal solution.

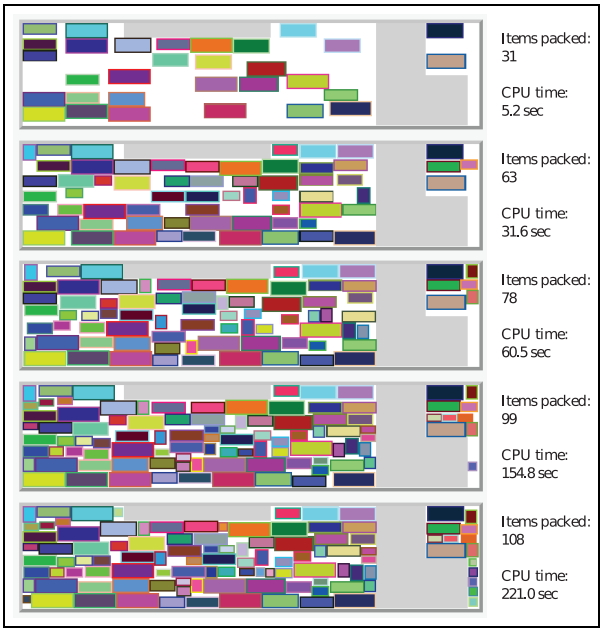

Figure 12. Time-lapse of the sequential packing (by decreasing order of size) routine for Example Deck *B* and the items of Figure 8.

We note that while aggregation can speed up solution times, like any heuristic, it may have the side effect of excluding optimal solutions. Further testing of this heuristic's value could be included in future work.

5.1.2. Sequential packing by decreasing order of size. In this example, we attempt to pack all the rectangles in Figure 8 onto Example Deck *B* by using a sequential packing heuristic with constrained movement. The ordering rule for the placement of rectangles onto the deck is by decreasing order of rectangle size (surface area), that is, the largest rectangle is packed first, and so on. (This is a design choice for the heuristic and has not been implemented with any operational reason in mind.) The first 30 rectangles placed onto the deck are allowed to move by up to 100 meters from their current position, with an allowable movement of 40% of the rectangle size applied thereafter.

A time-lapse of the sequential packing routine, showing the number and positions of rectangles packed onto the deck, can be seen in Figure 12. The final result is that all 108 rectangles in the list are able to be packed onto the deck in approximately 221 seconds of CPU time. This shows significantly better performance than applying aggregation three times to the item list and then passing the list to the solver (that is, the solution given in Figure 10, which took approximately 346 seconds of CPU time).

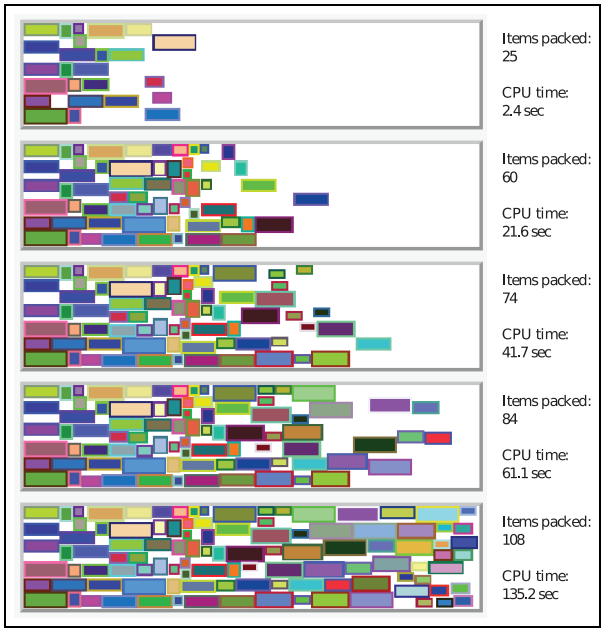

Figure 13. Time-lapse of the sequential packing (by priority ordering) routine for Example Deck *C* and the items of Figure 8.

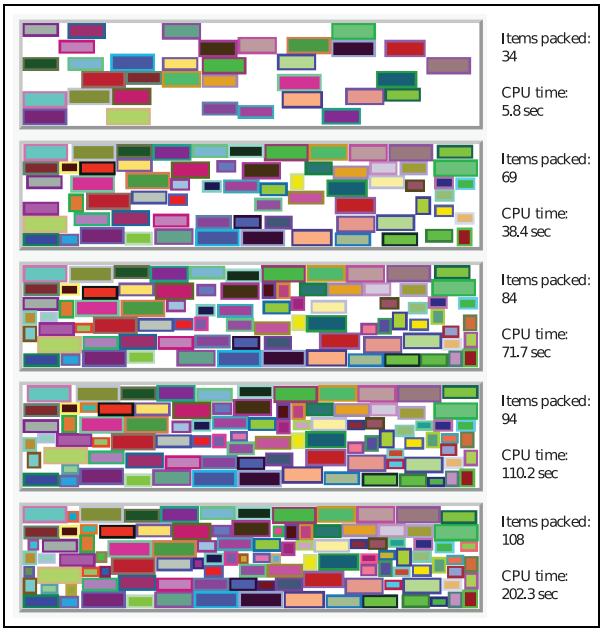

Figure 14. Time-lapse of the sequential packing (by decreasing order of size) routine for Example Deck *C* and the items of Figure 8.

5.2. Packing without obstacles - Example Deck C

We take the rectangle list of Figure 8 and we increase the buffer size from 200 to 400 mm. We will consider how this item list can be packed onto Example Deck *C*, which contains no obstacles, and has dimensions of 15.8 meters $\times$ 93.5 meters.

5.2.1. Sequential packing by priority ordering. We will show the results of a sequential packing routine with constrained movement, where the rectangles must be packed in the order in which they appear in Figure 8, that is, firstly read across the page from left to right, and secondly, top to bottom. The sequential packing routine implements the priority ordering rule by trying to place higher priority rectangles closer to the bottom side of the deck. Hence, the time-lapse of the solver's progress (as seen in Figure 13) shows the rectangles being pushed toward the bottom side. The allowable movement constraint on packed rectangles for this example is given by 80% of the rectangle size. An optimal solution is successfully returned by the solver after approximately 135 seconds of CPU time.

5.2.2. Sequential packing by decreasing order of size. We now consider how to pack items onto Example Deck *C* by decreasing order of size. In the first 40 iterations of the sequential packing routine, packed rectangles are allowed to move by up to 10 meters from their current positions, with the allowable movement altered to 40% of the rectangle size for iterations greater than 40. The solver returns an optimal solution in approximately 202 seconds of CPU time, which is slower than the priority ordering routine (approximately 135 seconds of CPU time). The time-lapse of the solver's progress can be seen in Figure 14.

6. Development of a test rig Graphical User Interface

The heuristics described in the previous sections have been implemented in a Java-based software package called the Constraint Optimization Packing Tool (COmPacT). The heuristics are underpinned by the CP solver from the IBM ILOG CPLEX Optimization Studio. Ultimately, the COmPacT model is intended to be incorporated into a deployable tool for use by operators with expertise in amphibious operations planning. However, such a tool may have many other amphibious command and control modules. Therefore, to work independently of the constraints of a such a tool, we have developed a test rig Graphical User Interface (GUI) to perform concept exploration with COmPacT.

6.1. Why develop a test rig?

A test rig GUI allows us to experiment with the applicability and functionality of the COmPacT model. In particular, the GUI's visualization features provide a framework for

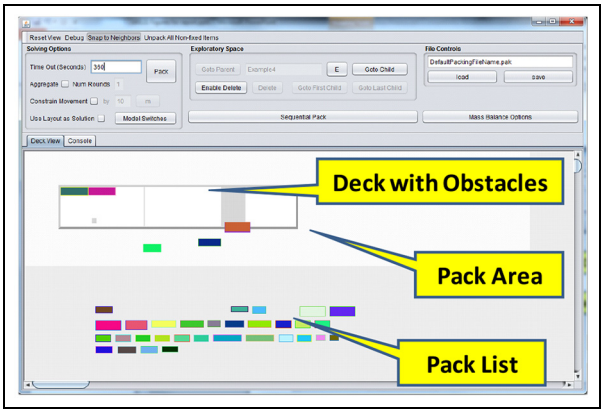

Figure 15. The Graphical User Interface, illustrating the Pack Area and Pack List.

the packing algorithms to work in concert with an operator's expertise and judgement. Furthermore, as our heuristic approaches have significant tuneable parameters, a GUI provides an environment to configure these to a range of appropriate settings.

Primarily, the GUI has been developed to provide a mechanism by which rectangular items can be drawn, dragged, and dropped onto and around a ship deck. In addition, the GUI has been developed for the following ancillary reasons.

1. For timely development of the model's code base, which allows ease of testing and exploration of how the model could be used.
2. To enable familiarity with the code base to test concepts quickly.
3. For the freedom to experiment with code concepts.
4. To explore methods for user interaction. A GUI allows experimentation with switches and interaction mechanisms. An example of an interaction mechanism is the ''tree-based exploration'' discussed below.
5. To permit rapid updating of a self-contained tool that can be compiled, run, set-up, saved, recompiled, and rerun quickly with as few overheads as possible.
6. To create a clear interface between the model and the surrounding GUI.

6.2. Interaction concepts

One of the goals of creating the GUI is to explore how an operator might interact with the model. Our exploration has come up with a number of concepts that have been implemented in the final version of the GUI and these are presented here.

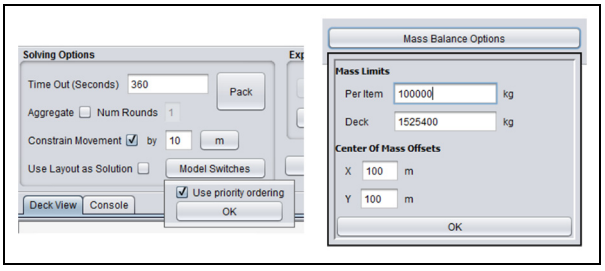

Figure 16. Solving options of the Graphical User Interface.

6.2.1. The Pack Area. There may be far too many items available to pack in a single execution of the solver and, in this case, it is likely that the operator will want to select the items that they wish to ensure are presented to the solver. Also, if there are too many items presented to the solver, the problem is likely to be too complex and the solver will fail to return an acceptable feasible solution in a timely manner.

To allow the operator to specify which items are desirable for packing, we introduce the concept of a *Pack Area* onto which ''packable'' items can be dragged or otherwise placed. Only the items in the Pack Area will be sent to the solver for placement on the deck. Note that the items in the Pack Area define a subset of items to be packed, which is distinct from the priority ordering of all items. The priority ordering will tell us the sequence in which to pack this subset.

The Pack Area is illustrated in Figure 15, which shows the deck (complete with obstacles) that needs to be packed, and a *Pack List* showing all the items that could be packed. Some of the items have been removed from the pack list and placed on the deck. The Pack Area consists of the lighter gray area and includes the deck itself. If the solver was executed at this point, it would be given five items to pack: the two items already on the deck, the one item on the boundary of the deck, and the two items outside the deck but within the Pack Area.

Note that COmPacT has the flexibility to move the two items already on the deck. For convenience, the GUI has a button that will unpack all items and place them back in the Pack List.

6.2.2. Model switches and parameters. The GUI gives access to the following switches and parameters that control the execution of the solver. These options are shown in Figure 16 and are described below.

1. *The time out limit for the solver*. For lists with a large number of items, the solver will take too long to find an optimal solution. The time out limit truncates these long calculations and returns any partial solutions found.

2. *Whether to execute the aggregation heuristic and, if so, the number of rounds of aggregation to be performed*. The default setting is to aggregate items top-to-bottom on the first iteration and then to alternate with a side-by-side aggregation.
3. *Whether to constrain the movement of items, and the degree to which this should be carried out*. Recall that this seeks to limit the distance a packed item can move from its current position. While the solver will look at the constraints on each individual item to be packed, this switch works by turning on/off the constraint and setting universal limits on the positions of all items already placed on the deck. Items that are outside the deck but within the Pack Area are free to move to any feasible location on the deck. The limits on movement can be set as either an absolute distance from the current location, or by a fixed percentage of the size of the item. Again, these are passed to the solver as fixed distances for each item.
4. *Using the current layout as a first solution (warm-start)*. With this switch the solver will identify the current set of packed items as a potential first solution, effectively allowing it to commence its search for a better solution by looking at the packing layout already given to the solver.
5. *Model switches*. The test rig queries the packing model to ask about any other boolean switches that it may have and then presents these to the user for modification. COmPacT presents only one switch, **Use priority ordering**, which seeks to break symmetry by setting the relative locations of items based on the priority ordering.
6. *Mass balance options*. These options set mass limits on each rectangular item and the deck as a whole. They also allow the user to specify how far off the center of the deck the center of mass of all packed items is allowed to be. In the examples shown in Figure 16, the values chosen are so large that the mass balance constraints are effectively nullified.

6.2.3. Tree-based exploration of possible solutions. As an operator explores different options for packing items onto a deck, it is likely that some configurations will not work, and the operator may want to go back to earlier packing configurations. Also, after being issued a packing command, the solver may come up with a solution that, while feasible, is undesirable to the operator.

To address this issue, the test rig implements a tree-based exploration mechanism. The control system for this mechanism is shown in Figure 17. At any point, the user can spawn off a ''child'' of the currently displayed packing configuration. Also, when **Pack** or **Sequential Pack** is executed, the resulting distribution of all items is placed into a new child node. In this way, the operator can go back to earlier packing configurations by navigating around the tree of designated states. **Goto Parent**, **Goto First Child**, and **Goto Last Child** buttons make it easy to rapidly click through the tree.

Figure 17. The control interface for the tree-based packing exploration system.

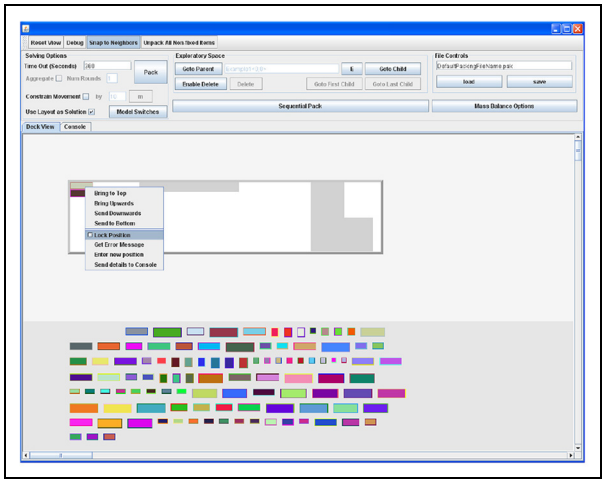

Figure 18. Example Deck *B* with two items manually packed. These items are in the process of being locked in place.

7. Exploration of the test rig Graphical User Interface

In this section, we will demonstrate the use of COmPacT and the GUI. We will use Example Deck *B* (Figure 9), which includes a number of obstacles, and the set of rectangular items from Figure 8. Note that COmPacT receives each item from the GUI as a rectangle of larger size, namely, the original width and length plus 400 mm to accommodate this mandatory border. We can then manually place a few items onto the deck and fix them in place by ''right-clicking'' on them and choosing **Lock Position**, as in Figure 18. After the items are locked in place, each one is marked with a diagonal line to signify a locked

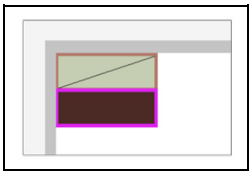

Figure 19. Two slightly overlapping rectangular items with the upper item locked in place.

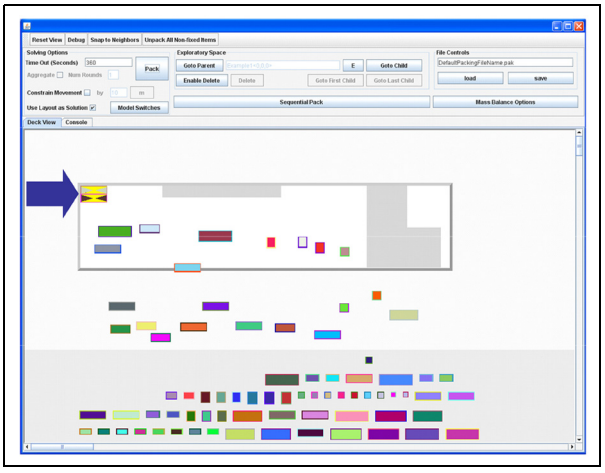

Figure 20. The visual error shown when trying to pack two items that overlap (arrow added; color online only).

status. Until such time as these items are unlocked, the model will treat them as obstacles.

To illustrate how the GUI treats inconsistencies, we can move the lower item in Figure 18 so that it slightly overlaps with the upper one. To do this we need to unlock the lower item and temporarily turn off the automatic **Snap to Neighbors** system, which is calibrated to place items hard up against any adjacent items. Figure 19 shows these two items just before we lock the lower one in place.

If we lock the lower item in place, and then add a few other items to the Pack Area and press the **Pack** button, we will get an error. The overlapping locked items will show up with a yellow cross pattern, as in Figure 20.

If we correct the overlapping issue and hit **Pack** again, all the items in the Pack Area (that is, all items on the deck but not locked into place and all items in the light gray area surrounding the deck) will be passed to COmPacT for packing. This is a very simple packing problem and so COmPacT returns quickly with the packing given in Figure 21. Notice that by pressing **Pack** we have caused a new name Example1 $< 0,0,0,2 >$ to appear in the **Exploratory Space** controls. This signifies that we have added a new child to our exploration tree. If we wanted to

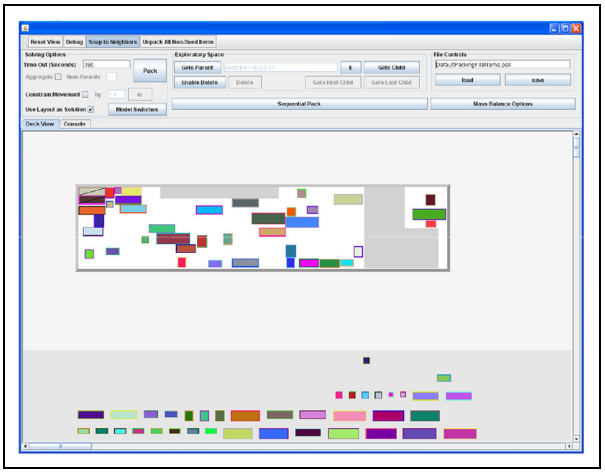

Figure 21. The result of packing the items in the Pack Area of Figure 20, where the slight overlap has been corrected.

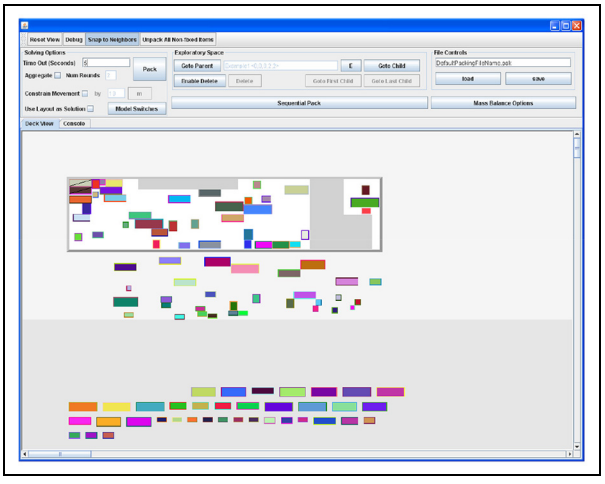

Figure 22. Setting up a slightly harder problem with a 5 second time out.

go back to our previous packing configuration, we would simply need to click **Goto Parent**.

Now let us move a few more items into the Pack Area (Figure 22) and see if these can be packed quickly. We will set the time out limit to 5 seconds and see if it is successful. It turns out that when we press **Pack** with only a 5 second time out, the solver cannot even get to the point where it finds its first solution before it times out. In this case it does not move anything and returns a pack state identical to the one it started with.

As the problem is starting to get harder for the solver, we can activate the aggregation heuristic with two rounds of aggregation. Setting the time out to 8 seconds, we find

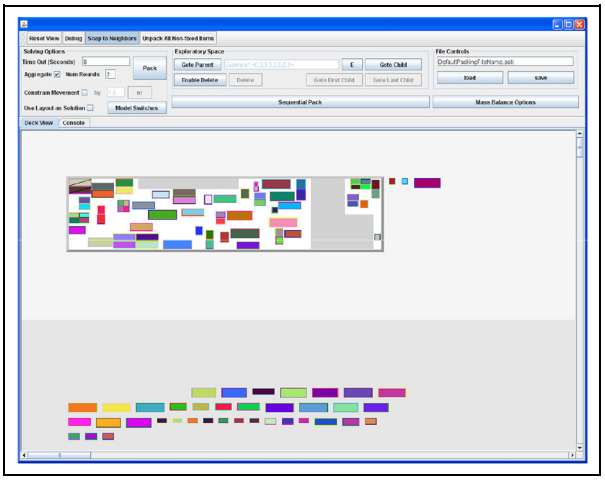

Figure 23. Example run with aggregation, but with too short a time out limit to find a solution that can pack everything. The unpacked items are placed in a list on the right-hand side of the screen.

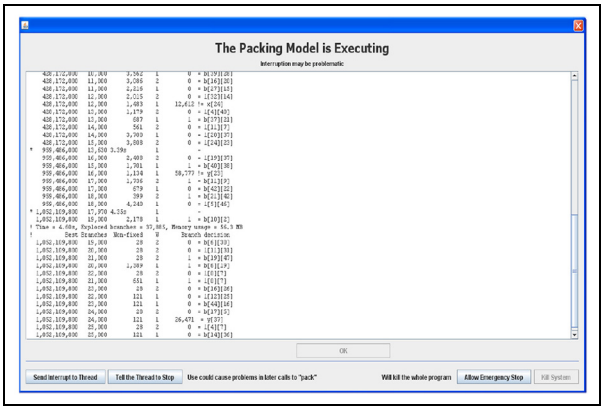

Figure 24. The packing model dialog that shows the current constraint programming solver status.

that the solver fails to find an optimal solution (Figure 23). In this case it returns a partial solution. So instead of leaving everything in its original position, it packs what it can, and places the unpacked items in a list on the right-hand side of the screen, but still within the Pack Area. Note in Figure 23 how some items are packed in 2×1 and 2×2 grids as a result of the aggregation.

We can now try again, but this time with a sufficiently large time out limit to ensure that the model can pack everything in the Pack Area. The pack time is between 9 and 15 seconds and, as this is greater than 3 seconds, a dialog box opens that displays the output of the packing model, in this case, the output of the solver from the ILOG CP Optimizer (Figure 24).

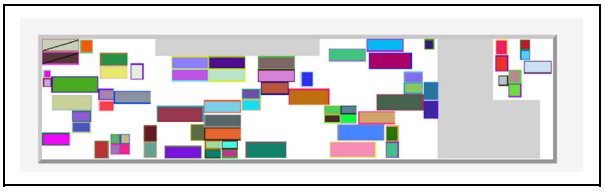

Figure 25. The result of packing the items in the Pack Area of Figure 23, given sufficient time.

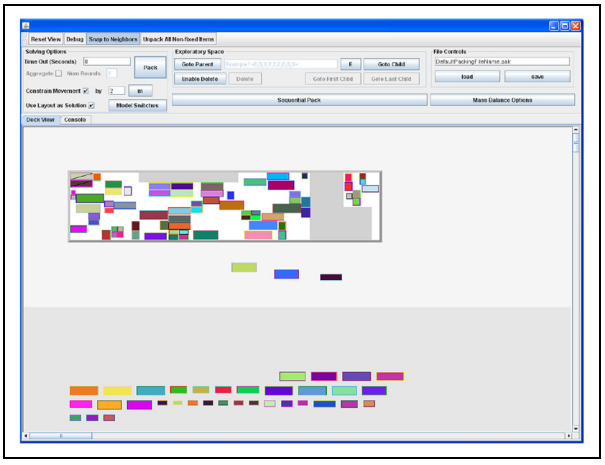

Figure 26. The solution of Figure 25 with three more items added to the Pack Area. The solving options are configured so that movement on the deck is constrained by 2 meters and the solver will recognize the current (partial) solution as a starting point for the search.

After pressing **OK** on the pack model execution dialog, we are able to see the packing solution that results, as shown in Figure 25.

We now wish to add another three items to the Pack Area to see if these can be packed onto the deck (Figure 26). In order not to ''throw away'' the work accomplished thus far, we can stipulate that the model constrains movement, that is, keeps items at or near their current position on the deck, and uses the current layout as an initial solution (*warmstart*), so that the solver does not have to start searching for a feasible solution from scratch. Pressing **Pack** now quickly results in a successful packing, as shown in Figure 27.

We now drag an additional six items from the Pack List to the Pack Area and use this as a seed to start a sequential packing routine (Figure 28). We also activate the option to constrain the movement of each item to within 2 meters of its current position, and to use the current solution as a starting point for the search. When we press **Sequential Pack**, the dialog box shown in Figure 29 appears.

The top part of the dialog of Figure 29 lists all the conditions that will be applied to the first iteration of the

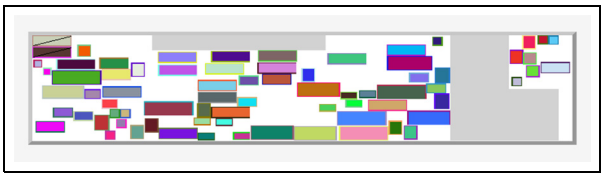

Figure 27. The packing solution resulting from the configuration of Figure 26.

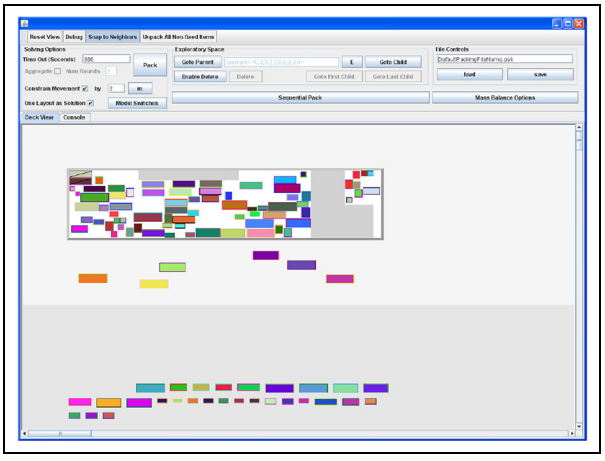

Figure 28. The result of adding six more items to the Pack Area, starting with the configuration of Figure 27.

solving sequence. This iteration will try to pack everything in the Pack Area in the same way as the **Pack** operation. The test rig has taken the conditions directly from the main window. In this case, it has set the mass balance constraints to the high default values, it will constrain movement to 2 meters, it will use the current solution as a starting point in its search, and it will not invoke the priority ordering constraints.

The lower part of the dialog of Figure 29 gives the conditions that will be varied in each iteration of the sequential packing routine. The model will be given one extra item to pack in each iteration, starting from the first item in the list. The conditions of the first iteration will be varied as follows:

1. the maximum allowable movement distance of a packed item will start at 2 meters;
2. the allowable movement distance will then linearly decrease to 1 meter over the course of five iterations;
3. for the next 23 iterations, the maximum allowable movement distance along the $x(y)$-axis for each item will be 30% of the width (length) of that item;
4. the time out for each iteration will be 60 seconds.

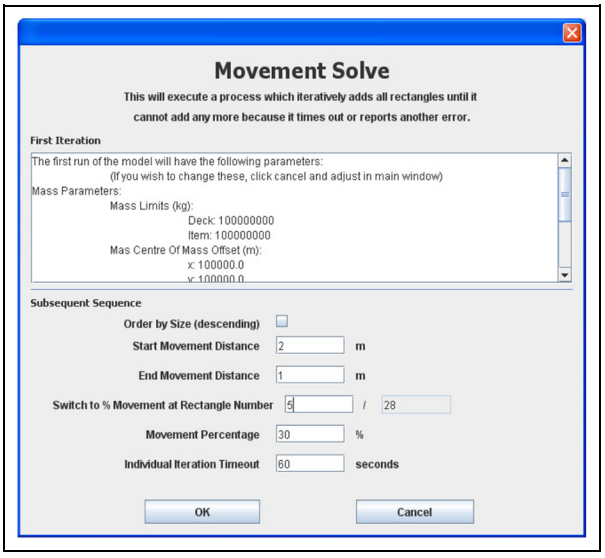

Figure 29. The dialog that controls the sequential packing routine.

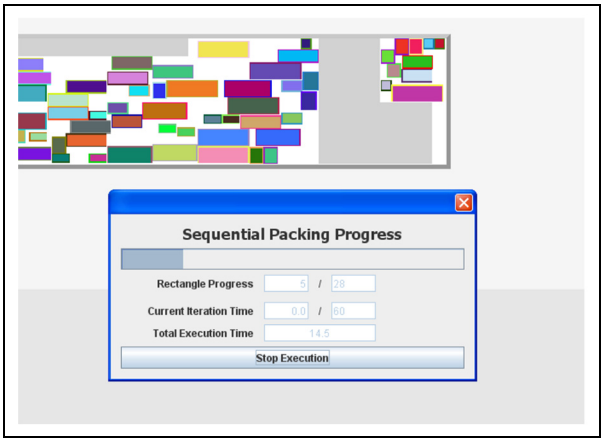

Figure 30. Progress dialog during a sequential packing routine using the starting state of Figure 28 and the options of Figure 29.

When we execute the model with the aforementioned settings, we get the **Sequential Packing Progress** dialog shown in Figure 30. This dialog shows the following:

- the current iteration number, both in the progress bar and as an integer value;
- the time within the current iteration, so that the user can see how long they have to wait for the iteration to time out (and the sequence stops) or discover a solution;
- the total elapsed time so far (in seconds).

In this particular example the solver times out, giving the exception report in Figure 31. Notice how the one

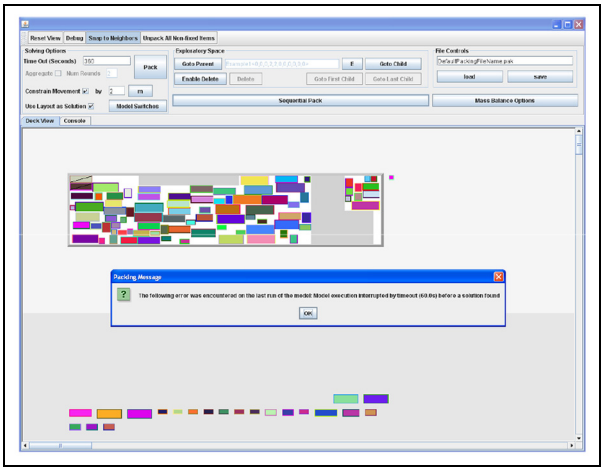

Figure 31. The result of a sequential packing routine showing the time out exception for premature termination of the solution process.

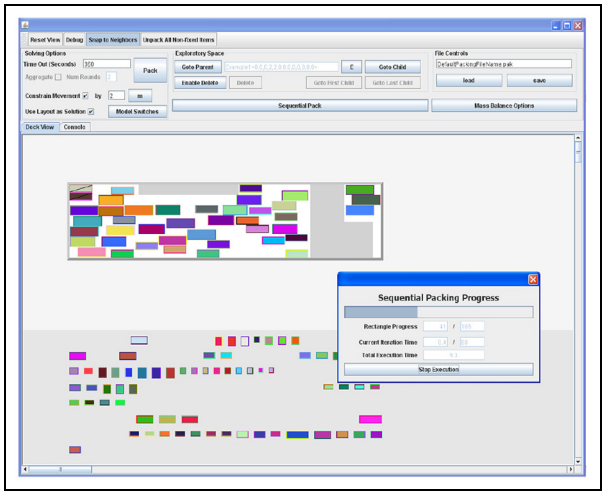

Figure 32. Screen-shot taken part way through the execution of the revised sequential packing routine for the configuration given in Figure 18.

unpacked item is placed beside the deck within the Pack Area. In this case, it turns out that the configuration of the currently packed items and the choice of constraints makes a difficult packing problem.

Instead of going back to earlier packing states using the **Exploratory Space** controls, we decide to go back to a completely unpacked state and see whether the sequential packing heuristic can pack all of the items in one go. Using the **Unpack All Non-fixed Items** button we backtrack to the initial state (Figure 18), with the exception that the locked items remain in their positions.

Choosing **Sequential Pack** presents us with another **Movement Solve** dialog like that of Figure 29. The upper part presents us with the conditions for the first iteration, which is to solve the packing problem for all items in the Pack Area. Since the Pack Area has no items in it, this iteration is trivial. Recall, however, that these conditions (such as the mass balance constraints) are the background conditions for all the iterations of the sequential packing routine.

The lower part of the **Movement Solve** dialog gives the conditions that will be varied in each iteration of the sequential packing routine. Here the model will be given one extra item in each iteration, starting from the largest item by size. The conditions of the first iteration will be varied as follows:

1. the maximum allowable movement distance of a packed item will start at 5 meters;
2. the allowable movement distance will then linearly decrease to 1 meter over the course of 53 iterations;
3. for the next 53 iterations, the maximum allowable movement distance along the $x(y)$-axis for each rectangle will be 30% of the width (length) of that rectangle;
4. the time out for any iteration will be 60 seconds.

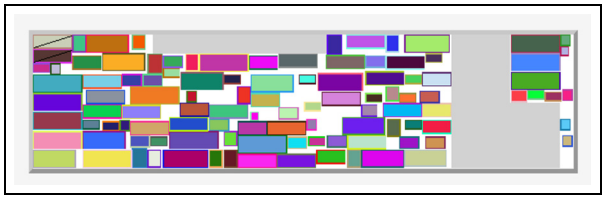

Figure 33. The final configuration after the execution of the revised sequential packing routine.

The sequential packing routine then proceeds as in Figure 32. Note that it is always possible to interrupt a sequential packing routine. However, this will prevent the next iteration from executing rather than interrupting the current iteration.

In this case, all items can be packed onto the deck and the packing configuration of Figure 33 results after a total execution time of less than 5 minutes.

8. Comparison with an open-source solver

Due to the licensing costs associated with the IBM ILOG CPLEX Optimization Studio, we investigated the option and viability of incorporating an open-source Java-based CP solver for COmPacT. This section will outline the process and results of that investigation.

8.1. Investigation process

The process of our investigation was structured as follows:

1. identify three widely used open-source Java-based CP solvers;
2. conduct a superficial appraisal of each solver, based on user-guide and application programming interface (API) documentation, and online user-support forums;
3. make the selection of a solver, based on a comparative assessment of the appraisals;
4. re-write the existing packing model using the class libraries of the selected solver and assess its functionality and performance on a number of test problems using the test rig GUI.

8.2. Choice of open-source solver

We identified three potential open-source Java-based CP solvers: *CHOCO, Cream*, and *JaCoP*. At the time we conducted this investigation, it was apparent that CHOCO had the most extensive user-guide and API documentation of the three options. We also noted that the user-guide for CHOCO contained a number of examples and tutorial problems. We considered this to be an especially attractive feature, since it would make the implementation of the unfamiliar class libraries much easier. On the whole, CHOCO had a number of desirable features that the remaining two solvers did not possess. Consequently, we decided to work with CHOCO as a candidate solver to replace the ILOG CP Optimizer libraries.

8.3. Implementation

We were able to state nearly all of the variables and constraints for the packing problem with CHOCO's class libraries without too many problems. (In most cases, there was a pretty straightforward way to translate between ILOG and CHOCO.)

Re-writing the mass balance constraints was not so straightforward. This entailed a complete overhaul of the former code and required the introduction of new variables to the problem. Expressing complex constraints such as these appears to be one drawback of moving away from ILOG to CHOCO.

A significant roadblock was encountered when we attempted to re-write the ''warmstart'' constraints. ILOG's CP solver has a specific class for this, whereby the solver's branch-and-bound algorithm is initialized through the following routine:

```
cp.SetStartingPoint(IloSolution);
```

where `cp` is an instance of `IloCP`. We were not able to locate a comparable routine for warmstart in CHOCO so, instead, we implemented a pseudo-warmstart procedure. For a sequential packing routine, the pseudo-warmstart procedure requires that rectangles remain permanently packed once they have been placed on the deck (no swapping out permitted). This procedure does not allow the same flexibility for rectangle movement on the deck as the ILOG CP Optimizer warmstart procedure (that is, limited *jiggling* around of rectangles in comparison). The pseudo-warmstart procedure was implemented with a greedy search heuristic that aims to pack the next rectangle in the sequence as close as possible to the deck's x-axis.

8.4. Performance

CHOCO was not able to handle measurements in millimeters for the rectangle and deck dimensions (the integers involved in area calculations were too large for CHOCO to handle). Hence, the dimensions were changed to decimeters. This entailed a slight decrease in packing position accuracy.

Packing a set of rectangles (in a single iteration) with CHOCO was problematic and slow, even for smaller sized sets. We assessed the runtime performance to be significantly slower than the ILOG CP Optimizer, but of course, more detailed testing needs to be performed in order to quantify the performance gap.

The mass balance constraints did not work consistently on the variety of problems tested. Sometimes they worked and other times they caused errors for the solver. At this initial stage, we do not know why this is the case, but it is possible that it could be related to the multiplication of large integers in the expression of the mass balance constraints.

The greedy search heuristic with the pseudo-warmstart procedure has shown that it can produce superficially complete packing solutions on some of the example problems configured in our GUI. The quality of these solutions can vary depending on the number, ordering, and type of rectangles that are to be packed. It should be noted that the same greedy search heuristic can be implemented (without too much effort) with the class libraries of the ILOG CP Optimizer package.

8.5. Summary

The CHOCO solver can be used as an adequate tool to perform greedy sequential packing, and from preliminary testing, this seems to perform well in cases where the set of rectangles (to be packed) has some homogeneity (that is, all rectangles have similar dimensions and the same orientation). The CHOCO solver performs poorly when trying to pack a set of rectangles in a single iteration (in this case, it is significantly slower than the ILOG CP Optimizer).

More rigorous testing is required in order to quantify the performance gap on a broader range of test problems.

9. An important implementation lesson

Early on in our work with the GUI, we found that repeated application of the solver to warmstart packing configurations caused subtle inconsistencies. Originally, we allowed the position of rectangular items to take any of a continuous set of values, which we represented as doubles. Because the model considered the precise placement of one item right up against another item, the round-off inherent in double precision arithmetic would yield a small overlap that returned an inconsistency from the solver. This problem appeared when we executed the solver, fixed a few items, and executed the solver a second time.

For example, suppose that four items of size 1.1 meters are placed next to each other starting at position (0, 0), as illustrated in Figure 34.

According to Figure 34, B would be positioned at (1.1, 0), C at (2.2, 0), and D at (3.3, 0). As a binary number, 1.1 is represented as 1.001100110011 . . ., and as a double this number is rounded up ever so slightly. Thus, the sum $1.1 + 1.1 + 1.1$ calculates as 3.300000000000 00030 in double precision arithmetic.

Therefore, if the solver was presented with A, B, C, D positioned as above with x-ordinates 0, 1.1, 2.2, and 3.3, it would return an inconsistency since it determines that rectangle C must ever so slightly overlap with rectangle D. Our solution to this problem was to move away from double precision arithmetic to integer arithmetic. This explains why we have measured everything in millimeters, thus allowing whole numbers only.

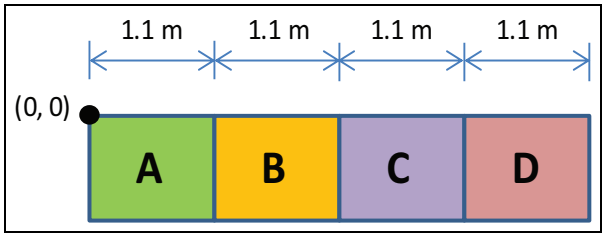

Figure 34. Illustration of round-off error.

10. Summary, conclusions, and future work

In this paper, we have reported on a number of packing heuristics that have been implemented within a Java-based software package called COmPacT. The heuristics are based on a CP paradigm and are implemented in COmPacT with the modeling and solver libraries of the IBM ILOG CPLEX Optimization Studio. The developed heuristic techniques complement and leverage off the modeling and solving capabilities of the ILOG CP Optimizer. They are beneficial to use in cases where an individual packing problem is too large to be handled by the CP solver alone. We also developed an interactive GUI, which demonstrates how manual and automated methods can be combined to pack a set of rectangular items onto a ship's deck.

The GUI allows an operator to specify a time limit to solve a packing problem. Some problems are still too complex for the CP search engine to solve (i.e., find a provably optimal solution) within acceptable timeframes. Nevertheless, if the CP search engine fails to find an optimal solution before the time limit has been reached, then the best partial solution will be returned. If a packing run in COmPacT does happen to terminate with a sub-optimal feasible solution, the user is permitted to call the CP solver again using the returned intermediate solution as a warmstart for another run. In this sense, COmPacT can be used as an iterative packing tool, with decisions pertaining to the automated and manual packing of items left to the discretion of the operator.

Finally, we suggest some avenues for future research and propose some extensions to the work presented in this article. (1) For deck configurations that can be decomposed into mutually exclusive subregions (for example, the left- and right-hand sides of Figure 9), a packing solution might be obtained by solving a number of interdependent sub-problems. It is important to note that the sub-problems will not be independent, as packing an item in one subregion precludes the possibility of it being packed in another subregion. In addition, each subregion may possess its own set of constraints for prohibited items, bounds, and mass balance. For example, if we consider the smaller right-hand subregion of the deck in Figure 9, we could naturally prohibit the packing of large items that exceed the subregion's physical dimensions. (2) More sophisticated implementations of the mass balance constraints in concert with the sequential packing heuristics could be investigated. For example, just as the movement tolerance of individual items can be made arbitrarily large at the beginning of a sequential packing routine and gradually decreased as more items are packed, the center of mass tolerance might be handled in a similar way. (3) In order to provide a detailed quantitative performance comparison of the ILOG CP Optimizer with CHOCO, a range of test problems could be designed to execute the developed packing heuristics with each solver. (A detailed quantitative comparison is beyond the intent and scope of this paper. However, such a study would constitute a natural extension to the present work.)

Acknowledgments

The authors would like to thank the anonymous reviewers for providing critical feedback, which has improved the quality of the manuscript.

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

ORCID iD

Paul A. Chircop https://orcid.org/0000-0001-8823-7966

References

1. Johnson DS, Demers A, Ullman JD, et al. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J Comput* 1974; 3: 299–325.
2. Jansen K and Prädel L. New approximability results for two-dimensional bin packing. *Algorithmica* 2016; 74: 208–269.
3. Lodi A, Martello S and Monaci M. Two-dimensional packing problems: a survey. *Eur J Oper Res* 2002; 141: 241–252.
4. Lodi A, Martello S, Monaci M, et al. Two-dimensional bin packing problems. In: Vangelis Th. Paschos (eds.) *Paradigms of combinatorial optimization*. Wiley-Blackwell, 2014, pp.107–129.
5. Lodi A, Martello S and Vigo D. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J Comput* 1999; 11: 345–357.
6. Lodi A, Martello S and Vigo D. Approximation algorithms for the oriented two-dimensional bin packing problem. *Eur J Oper Res* 1999; 112: 158–166.
7. Pisinger D and Sigurd M. Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS J Comput* 2007; 19: 36–51.
8. Martello S, Pisinger D, Vigo D, et al. Algorithm 864: general and robot-packable variants of the three-dimensional bin packing problem. *ACM Trans Math Software* 2007; 33: 7.
9. Côté JF and GendreauMand Potvin JY. An exact algorithm for the two-dimensional orthogonal packing problem with unloading constraints. *Oper Res* 2014; 62: 1126–1141.
10. Gendreau M, Iori M, Laporte G, et al. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks* 2008; 51: 4–18.
11. da Silveira JLM, Miyazawa FK and Xavier EC. Heuristics for the strip packing problem with unloading constraints. *Comput Oper Res* 2013; 40: 991–1003.
12. Dechter R. *Constraint processing*. Morgan Kaufmann, 2003 San Francisco, CA, USA.
13. Milano M and Trick M. Constraint and integer programming. In Milano M (ed.) *Constraint and integer programming: toward a unified methodology*. Boston, MA: Springer US, 2004, pp.1–31.
14. Smith BM. Modelling for constraint programming. In: Krzysztof R. Apt, Michela Milano, Francesca Rossi (eds.), *lecture notes for the first international summer school on constraint programming*, Acquafredda di Maratea, Italy, 11–15 September 2005, pp. 1–18. Available at: http://www.math.unipd.it/~frossi/cp-school/
15. Onodera H, Taniguchi Y and Tamaru K. Branch-and-bound placement for building block layout. In: Richard Newton A (eds.), *proceedings of the 28th ACM/IEEE design automation conference*, San Francisco, CA, 17–21 June 1991, pp.433–439. New York, NY, USA: ACM.
16. Chen CS, Lee SM and Shen QS. An analytical model for the container loading problem. *Eur J Oper Res* 1995; 80: 68–76.
17. Gilmore PC and Gomory RE. Multistage cutting stock problems of two and more dimensions. *Oper Res* 1965; 13: 94–120.
18. Pisinger D and Sigurd M. The two-dimensional bin packing problem with variable bin sizes and costs. *Discrete Optimiz* 2005; 2: 154–167.
19. Bettinelli A, Ceselli A and Righini G. A branch-and-price algorithm for the variable size bin packing problem with minimum filling constraint. *Ann Oper Res* 2010; 179: 221–241.
20. Polyakovskiy S and M'Hallah R. A hybrid feasibility constraints-guided search to the two-dimensional bin packing problem with due dates. *Eur J Oper Res* 2018; 266: 819–839.
21. Lustig IJ and Puget JF. Program does not equal program: constraint programming and its relationship to mathematical programming. *Interfaces* 2001; 31: 29–53.
22. Van Hentenryck P and Michel L. *Constraint-based local search*. Cambridge, MA, USA: The MIT Press, 09.

Author biographies

Paul A Chircop joined the Defence Science and Technology Group in 2006 after completing a Bachelor of Science (Advanced Mathematics) with First Class Honours at the University of Sydney. Since that time, he has developed mathematical models and provided analysis for a variety of maritime capability studies. He completed a PhD in Operations Research at the University of New South Wales in 2017. His doctoral research focused on the application of column generation algorithms to patrol asset scheduling with complete and maximum coverage requirements.

Timothy J Surendonk completed a PhD in Mathematical Logic at the Australian National University in 1998. Shortly after, he joined the Defence Science and Technology Group to work at Headquarters Australian Theatre. Subsequently, he has supported the Australian Defence Force (ADF) and wider National Security communities with model development and analysis, including studies of air lift operations, submarine cable protection, maritime patrol, and the Automated Identification System (AIS) for ships. He served as Staff Officer Science for the Headquarters Air Lift Group in the Royal Australian Air Force (RAAF) from 2010 to 2012. He now works in the field of Maritime Mathematical Sciences.