A generic feasibility-based heuristic scheme for the container pre-marshalling problem

Bo Jin[a,*], Andrew Lim[a,1], Ning Wang[b]

[a]*Department of Management Sciences, City University of Hong Kong, Tat Chee Avenue, Kowloon Tong, Hong Kong*
[b]*Department of Information Management, School of Management, Shanghai University, Shanghai, China*

Abstract

This paper addresses the container pre-marshalling problem (CPMP), the problem of reorganizing containers inside a storage bay such that additional reshuffling is not needed in the subsequent retrieval process. Traditional heuristic approaches handle containers in the reverse order of future retrievals. We propose a generic feasibility-based heuristic scheme for the CPMP, which is a departure from the traditional heuristic strategy. An implementation of the heuristic scheme, the greedy and speedy heuristic, is also proposed, in which four well designed heuristic techniques are applied. Computational experiments are conducted to showcase the proposed heuristic's performance.

Keywords: container pre-marshalling problem, heuristic, logistics, feasibility-based heuristic scheme

1. Introduction

Since the commencement of containerization, the global use of standardized containers has dramatically improved international trade. Containers enable the smooth flow of goods between multiple transportation modes without directly handling the freight. Over the years, stringent requirements such as just-in-time operations from consignors have created additional challenges for the container transportation industry.

*Corresponding author. Tel: +852 3442 5296
Email addresses: msjinbo@cityu.edu.hk (Bo Jin), alim.china@gmail.com (Andrew Lim), ningwang@shu.edu.cn (Ning Wang)

[1]Andrew Lim is currently on no pay leave from City University of Hong Kong.

Container yards — the spaces dedicated to the transshipment, handover, loading, consolidation, maintenance, and storage of containers — are key components of maritime container terminals. Some yards act as exchange venues for container transfers between different transportation modes while others are used as caches for temporary storage or as warehouses for long-term storage.

Generally, a container yard is divided into several yard blocks, each of which consists of several parallel bays. A bay is formed by a row of stacks. Usually the containers stored in the same bay have the same dimensions. Equipments such as rubber tyre gantry cranes, rail-mounted gantry cranes, and reach stackers are frequently used in container yards. Figure 1 illustrates an example of a yard block.

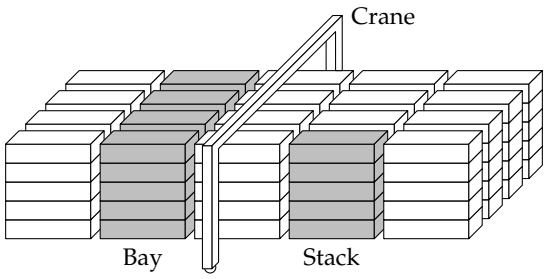

Figure 1: Yard block.

Because containers in the same stack are organized last-in-first-out, to retrieve lower containers, those on top must first be relocated. Such forced movements are known as container ***reshuffles***. According to recent reviews [2,13], terminals deal with three major decision problems related to container reshuffling. The first problem is the selection of storage locations for arriving containers. The second is the ***pre-marshalling*** problem, which addresses the reorganization of containers inside a storage area such that no reshuffling is further required. The last problem minimizes the total operational cost in the container retrieval process, measured by the number of relocations conducted or the total operational time. The aim is to improve terminals' performance levels, including their throughput rate per berth or the turnaround time of vessels or road trucks [10].

This paper investigates the ***container pre-marshalling problem*** (CPMP), in which the containers within a single bay are reshuffled before they are retrieved to facilitate retrieval without

subsequent reshuffling. It is assumed that the retrieval order of the containers is known beforehand and that no container arrives at or leaves from the bay during pre-marshalling.

This paper makes two main contributions to the literature. First, a generic feasibility-based heuristic scheme is proposed, based on a new state feasibility concept. The heuristic scheme breaks away from the traditional heuristic strategy, greatly enriching the methodology for the CPMP. The second contribution is an implementation of the generic scheme, the ***greedy and speedy heuristic*** (GASH), in which four well designed heuristic techniques are applied. The GASH performs outstandingly on both dense and loose instances, compared with existing heuristics.

The remainder of this paper is structured as follows. Section 2 reviews existing approaches in the literature. Section 3 lists the notation used throughout this paper and Section 4 gives a mathematical description of the problem. Comprehensive description of the heuristic scheme, four available techniques, and the proposed GASH can be found successively in Sections 5–7. Section 8 illustrates the results of our computational experiments, and the last section concludes this paper and identifies future research directions.

2. Recent work

Lee & Hsu [12] develop an integer programming model for the CPMP. In this work, the problem is formulated as a multi-commodity network flow problem. The overall network is divided into several subnetworks, with each subnetwork representing an intermediate layout. The nodes in a subnetwork correspond to the slots that accommodate containers, and the commodities correspond to the containers stored in the bay. Every valid flow in the network represents a solution to the pre-marshalling problem. The model provides an innovative viewpoint to the problem, however, its performance is limited because the network is too large even for small problem instances.

Caserta & Voß [3] provide a greedy heuristic, the corridor method, for solving the problem. The heuristic selects the direction of movement in a randomized manner according to the attractiveness of available successors confined by the corridor. The attractiveness is measured by an estimated number of additional relocations needed for the particular successor. A local improvement procedure is also conducted to accelerate the heuristic process.

A neighborhood search approach is proposed by Lee & Chao [11], which repeatedly modifies the current solution until some termination condition is met. Unlike the other existing solution construction approaches, the neighborhood search is required to start from a pre-generated initial solution. A feasible solution can be further improved by a four-step procedure, and the diversity of the neighborhood is raised by multiple subroutines. The main drawback of the approach is the unreliability of random solution modifications, i.e., the feasibility of the resulting new solutions are not always ensured.

Bortfeldt & Forster [1] describe a tree search procedure for solving the problem. In the tree search structure, solutions are constructed by compound moves instead of single moves. Moves are classified into four types, and only the most promising ones are adopted in the branching scheme.

There are two studies in the literature regarding heuristic approaches. They adopt the same heuristic strategy that containers are handled in reverse of their retrieval order.

Expósito-Izquierdo et al. [6] provide the first container-oriented heuristic for the CPMP, named the lowest priority first heuristic (LPFH). The LPFH iteratively handles containers in descending order of the container priority values. After handling all containers with a specific priority, a stack filling process is applied to reduce the number of disorderly containers in the bay.

Wang et al. [14] propose the target-guided heuristic (TGH) and two beam search algorithms. The TGH also handles containers in descending order of the container priority values. A gap utilization process is applied to enhance the heuristic performance. This work provides the first comprehensive analysis on every situation that the heuristic may face during the pre-marshalling process, especially for dense instances with fewer empty slots.

3. Notation

The notation for describing the problem is as follows.

I. Global constants

N	number of containers
S	number of stacks
H	height limitation of stacks

P	number of priorities
E	number of empty slots, $E = SH - N$
U	number of unreachable tiers, $U = \max\{H - E, 0\}$
$\mathbb{C}$	set of containers
$\mathbb{S}$	set of stacks, $\mathbb{S} = \{1, \ldots, S\}$

II. Common symbols and functions

L	layout
c	container
s	stack index
t	tier index
p	priority value
(s, t)	slot or the container located in
$h(s)$	height of stack s, vectorized as $\boldsymbol{h}$
$e(s)$	number of empty slots in stack s, $e(s) = H - h(s)$
$o(s)$	orderly height of stack s, vectorized as $\boldsymbol{o}$
$f(s)$	fixed height of stack s, vectorized as $\boldsymbol{f}$
$(\mathsf{L}, \boldsymbol{f})$	state
$p(c)$	priority value of container c
$p(s, t)$	priority value of container (s, t)
$q(s, t)$	capability of an occupied slot (s, t)
$q^f(s)$	capability of the top fixed slot in stack s, $q^f(s) = q(s, f(s))$

III. Resource-/demand-related functions

$d(p)$	order-p demand, i.e., the number of unfixed containers with priority p
$D(p)$	order-p accumulate demand, $D(p) = \sum_{\varphi=p}^{P} d(\varphi)$, vectorized as $\boldsymbol{D}$
$r(p)$	order-p resource, $r(p) = \sum_{q^f(s)=p}(H - f(s))$
$R(p)$	order-p accumulate resource, $R(p) = \sum_{\varphi=p}^{P} r(\varphi)$, vectorized as $\boldsymbol{R}$
$\Delta(p)$	order-p surplus, $\Delta(p) = R(p) - D(p)$, vectorized as $\boldsymbol{\Delta}$

The following is the extra notation used to describe the proposed algorithms.

IV. Task-related symbols

c^*	target container
s^+	stack of the target container
t^+	tier of the target container
s^-	aim stack
b	number of blocking containers
a	number of slots available for blocking containers

V. Extra symbols

s^{src}	source stack (sender) of a move
s^{dst}	destination stack (receiver) of a move
s^{tmp}	interim stack for temporarily storing the target container
$\vec{v}$	evaluation tuple of a move, lexicographically comparable
$\mathbb{R}$	receiver set
$\mathbb{I}$	set of potential interim stacks
$\mathbb{F}$	set of potential interim stacks which are currently full
$\mathbb{T}$	set of valid tasks

VI. Extra functions

$g(s)$	stable height of stack s, vectorized as $\boldsymbol{g}$
$m(s)$	messiness (largest unstable priority value) of stack s, $m(s) = \max_{g(s)<t\leq h(s)} p(s,t)$
$\alpha(s^+, s)$	number of slots available after the move from stack s^+ to stack s
$\delta(p,q)$	demand between p exclusive and q inclusive, $\delta(p,q) = \sum_{\varphi=p+1}^{q} d(\varphi)$

4. Problem description

The CPMP is restricted to the bay size, or more precisely, the dimensions of the operating crane. An instance (problem input) includes an initial layout of N containers, which are distributed in a single bay with S stacks ($S \geq 3$) and H tiers ($H \geq 2$), leaving E empty slots ($E = SH - N$, $E \geq 2$).

In dense instances such that $E < H$, the bottom $H - E$ tiers of the bay are unreachable. Let $U = \max\{H - E, 0\}$ denote the number of unreachable tiers. As a simple fact, containers in the reachable tiers can be permuted into any wanted arrangement.

Let $\mathbb{S} = \{1, \ldots, S\}$ be the set of stacks. Hereafter, for simplification of description, when a stack s is mentioned without declaring its domain, it is assumed that $s \in \mathbb{S}$. The height of stack s is denoted by $h(s)$ and $e(s) = H - h(s)$ denotes the number of empty slots in this stack. Note that the height of stacks should not exceed H.

The containers in the bay are categorized into P groups, determined by the container stowage plan which is based on specific constraints such as shipment destination, weight distribution. Each group is assigned a priority value from 1 to P, such that a smaller priority value indicates an earlier retrieval order (loading order to the vessel). The t-th slot (from the bottom up) of stack s or the container located inside is denoted by a pair (s, t). The priority values of a container c and the container located in slot (s, t) are denoted by $p(c)$ and $p(s, t)$, respectively.

To prevent confusion in the following expression, hereafter we only use "small/large priority value" to describe a container's retrieval order, instead of "high/low priority", which is commonly used in the literature.

A container is ***orderly*** if it is supported directly by the ground or another orderly container with equal or larger priority value; otherwise, it is ***disorderly***. The orderly height (number of orderly containers) of stack s is denoted by $o(s)$. Other phrases referring to "orderly/disorderly" in the recent literature include "well/badly placed" [1], "well-/non-located" [6], and "clean/dirty" [14]. Table 2 gives an example of a bay for which $S = 5$, $H = 4$, and $N = 13$. Containers are represented by boxes with their priority values marked inside. In addition, boxes with gray background are the disorderly containers.

Let us define the capability of an occupied slot (s, t) by $q(s, t) = p(s, t)$ if the container inside is orderly, otherwise $q(s, t) = 0$; specifically, regarding the ground as an occupied slot at tier 0 with priority value P. Thus, we can easily verify the placement of a container (s, t) by checking whether $p(s, t) \leq q(s, t - 1)$.

The objective of the CPMP is to find an optimized sequence with the fewest container moves capable of converting the initial layout into a final layout in which all of the containers are orderly.

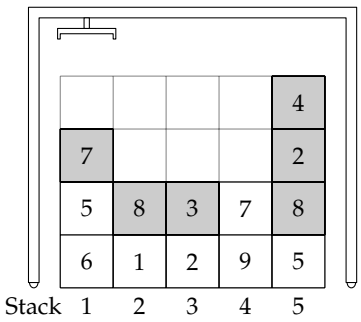

Figure 2: Container bay.

5. Generic feasibility-based heuristic scheme

In this section, a generic feasibility-based heuristic scheme is developed for solving the CPMP. The proposed heuristic scheme breaks away from the traditional heuristic strategy, contributing to the methodology for the problem. Before describing the heuristic scheme, the concepts of state and state feasibility are introduced first.

5.1. State and state feasibility

A state $(\mathsf{L}, \boldsymbol{f})$ is a pair comprising a layout L and a fix vector $\boldsymbol{f}$. The fix vector $\boldsymbol{f}$ indicates the fixed height (number of fixed containers) of each stack in L. Further moving of the fixed containers is not allowed. The fixed status of the containers in a layout raises the issue of state feasibility.

Proposition 1 (Necessary conditions for state feasibility). *The necessary conditions for the feasibility of a state* $(\mathsf{L}, \boldsymbol{f})$ *are* $\boldsymbol{f} \leq \boldsymbol{o}$ *and* $\boldsymbol{\Delta} = \boldsymbol{R} - \boldsymbol{D} \geq \boldsymbol{0}$.

Proposition 1 gives the necessary conditions for state feasibility, where $\boldsymbol{R}$ and $\boldsymbol{D}$ are the accumulative resource and demand vectors, and $\boldsymbol{\Delta}$ is the surplus vector. For $1 \leq p \leq P$, $R(p)$ is the order-p accumulative resource described by the total number of slots which can support containers with priority value p (certainly, as well as less than p), and $D(p)$ is the order-p accumulative demand or the number of unfixed containers with priority values not less than p. Let $d(p)$ be the number of unfixed containers with priority p, and $r(p) = \sum_{q^f(s)=p}(H - f(s))$ where $q^f(s) = q(s, f(s))$, we have $D(p) = \sum_{\varphi=p}^{P} d(\varphi)$ and $R(p) = \sum_{\varphi=p}^{P} r(\varphi)$. As a result, the non-negativity of the surplus vector $\boldsymbol{\Delta} = \boldsymbol{R} - \boldsymbol{D}$ should be maintained in a feasible state.

For those states where fewer than $S-2$ stacks are fully fixed, the necessary conditions are sufficient to ensure the feasibility of the stack. Taking the layout in Figure 2 and $\boldsymbol{f} = \mathbf{1}$ as an example, Table 1 illustrates the computation for the surplus vector. The example state is infeasible due to $\Delta(7) < 0$; that is, the containers with priority values 7, 8, and 9 cannot be well accommodated by enough unfixed slots.

Table 1: Computation for the surplus vector.

p	$d(p)$	$D(p)$	$r(p)$	$R(p)$	$\Delta(p)$
1	0	8	3	15	7
2	1	8	3	12	4
3	1	7	0	9	2
4	1	6	0	9	3
5	1	5	3	9	4
6	0	4	3	6	2
7	2	4	0	3	−1
8	2	2	0	3	1
9	0	0	3	3	3

Definition 1 (Extreme state). *A state* $(\mathsf{L}, \boldsymbol{f})$ *is an extreme state if* $|\{s \in \mathbb{S} : f(s) = H\}| \geq S - 2$.

However, in an ***extreme*** state, i.e., $S-2$ or more stacks are fully fixed, an additional condition is needed. Considering an extreme state, except for $S-2$ fully fixed stacks, if the remaining two stacks cannot be converted into orderly stacks by simply reshuffling unfixed containers from one to the other, we say it is a ***tricky*** state.

Proposition 2 (Existence of tricky states). *A tricky state exists if* $E < 2(H-1)$.

5.2. Feasibility-based heuristic

The feasibility-based heuristic first constructs the initial state according to the unreachable tiers. Starting from the initial state, the heuristic repeatedly fixes a target container c^* to a properly chosen slot $(s^-, f(s^-)+1)$, until all of the containers are fixed. The procedure of the heuristic scheme is concisely described in Algorithm 1.

Algorithm 1: Feasibility-based heuristic scheme.

```
begin
    (L, f) := (L^0, U · 1);
    repeat N − S U times do
        (c*, s⁻) := the chosen valid task;
        Accomplish (c*, s⁻) on L;
        f(s⁻) := f(s⁻) + 1;
```

5.3. Initialization

The initial state is generated with the initial layout L^0 and the initial fix vector $U \cdot \mathbf{1}$. It is a natural fact that when $S \geq 3$, the reachable (upper $H - U$) tiers can be permuted into any wanted placement. Meanwhile, the bottom U tiers are unreachable. With the help of the state feasibility concept, we can check the solubility of an instance by checking the feasibility of the initial state.

Proposition 3 (Instance solubility). *A CPMP instance is solvable if the induced initial state is feasible.*

5.4. Valid task

At every step of the generic heuristic scheme, a task is determined and then accomplished through a sequence of moves. The task is an assignment that aims to fix a chosen target container to the lowest unfixed slot of a chosen aim stack.

A container–stack pair (c, s) is a valid task for the current state $(\mathsf{L}, \boldsymbol{f})$ only if (necessary but not sufficient)

- c is unfixed,
- $f(s) < H$,
- $p(c) \leq q^f(s)$, and
- $\Delta(\varphi) \geq H - f(s)$, for $p(c) < \varphi \leq q^f(s)$.

The last condition ensures the non-negativity of the resulting surplus vector, however, the feasibility of the resulting state is not sufficiently ensured, due to the possibility of tricky state.

There are three methods for resolving the tricky state issue.

1. Prevent from entering a tricky state when deciding the next task;

2. Design an ideal task accomplishment procedure that makes the resulting extreme state feasible;

3. Allow entering a tricky state and resolve the regression issue.

Our previous work [14] adopts the last method listed above. The regression issue involves temporarily moving an already fixed container during the task accomplishment, and moving it back to its originally fixed slot.

In Section 6.2, we introduce a tricky state avoidance technique that inhibits the heuristic from entering a tricky state, saving unnecessary moves when solving dense instances.

6. Available techniques

In this section, four available techniques applicable to the feasibility-based heuristic are introduced.

6.1. Container stability

Considering an orderly container (s, t) in a feasible state $(\mathsf{L}, \boldsymbol{f})$, we say it is ***stable*** if the revised state $(\mathsf{L}, \boldsymbol{f}')$ is feasible such that $f'(s) = t$ and $f'(i) = f(i)$ for $i \neq s$. Denoting the stable height (number of stable containers) of stack s by $g(s)$, we have $\boldsymbol{f} \leq \boldsymbol{g} \leq \boldsymbol{o} \leq \boldsymbol{h}$. If the top container of some stack, named c for example, will be stable after being moved to a non-full stack s, we say stack s can ***stabilize*** container c.

Figure 3 gives an example of a state, where unstable containers are labelled with gray background. Notice that the container $(2, 1)$ with priority value 6 is orderly but unstable.

An orderly but unstable container indicates that its orderliness is valueless because it definitely requires reshuffling. Using the concept of container stability instead of container orderliness is more precise, as an orderly but unstable container has no opportunity to be fixed to its slot. For example, when a blocking container is becoming orderly but unstable onto a destination stack, the attractiveness of such a move should be reconsidered.

Note that the stability of containers should be recomputed after any container is fixed.

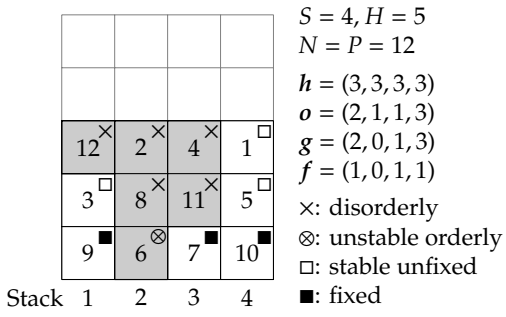

Figure 3: Stability of containers.

6.2. Tricky state avoidance

When talking about selecting the next task, the tricky state issue is inevitable. If a container–stack pair leads to an extreme state, the feasibility of the resulting state is unpredictable unless a complicated accomplishment procedure is designed. If unfortunately, a tricky state results, the remainder of the pre-marshalling work must consider the regression issue.

Definition 2 (Pre-tricky state). *A state is a pre-tricky state if* $|\{s \in \mathbb{S} : f(s) = H\}| = S - 3$, $|\{s \in \mathbb{S} : f(s) = H - 1\}| \geq 1$ *and* $\sum_{s\in\mathbb{S}} f(s) < N - 2$.

With the introduction of ***pre-tricky*** state by Definition 2, the tricky state avoidance technique can be summarized as follows. When the current state is pre-tricky, we eliminate the container–stack pairs (c, s) such that $f(s) = H - 1$ to avoid entering a tricky state.

6.3. Bottom tiers protection

In the feasibility-based heuristic scheme, the target container can be freely chosen as long as the task is valid, not subject to the largest priority value. After the target container c^* is fixed to the aim slot $(s^-, f(s^-) + 1)$, the surplus $\Delta(\varphi)$ is reduced by $H - f(s^-)$, for $p(c^*) < \varphi \leq q^f(s^-)$.

The bottom tiers protection technique balances the trade-off between the target freedom and the surplus loss, by eliminating the container–stack pairs (c, s) such that $f(s) < 2$ and $\delta(p(c), q^f(s)) \geq S$, where $\delta(p, q) = \sum_{\varphi=p+1}^{q} d(\varphi)$. The affected demand quantity $\delta(p(c), q^f(s))$ should not reach the threshold S in the bottom two tiers that are protected.

As customizable parameters, the number of tiers protected and the threshold value can be adjusted optionally.

6.4. *Speedy task accomplishment procedure*

After the task is decided by specific rules, it is accomplished by a sequence of moves, resulting in a new state. The speedy task accomplishment procedure (STAP) provides a structured framework to finish the given task with the fewest moves, providing customizable interfaces with several user-defined functions.

6.4.1. *Common functions*

The pseudo-code for the common functions of the STAP is given in Algorithm 2.

Algorithm 2: Common functions.

```
1  procedure Move(s^src, k, s^dst)
2      Move k containers from stack s^src to stack s^dst;
3  procedure Relocate(s^src, k, ℝ)
4      repeat k times do
5          ℝ′ := {s ∈ ℝ : h(s) < H};
6          s^dst := arg min_{s∈ℝ′} EvalMove(s^src, s);
7          Move(s^src, 1, s^dst);
8  procedure BiSender(s_1^src, k_1, s_2^src, k_2, ℝ)
9      i := k_1, j := k_2;
10     repeat k_1 + k_2 times do
11         if j = 0 then
12             Relocate(s_1^src, i, ℝ);
13             i := 0;
14         else if i = 0 then
15             Relocate(s_2^src, j, ℝ);
16             j := 0;
17         else
18             ℝ′ := {s ∈ ℝ : h(s) < H};
19             s_1^dst := arg min_{s∈ℝ′} EvalMove(s_1^src, s);
20             v⃗_1 := EvalMove(s_1^src, s_1^dst);
21             s_2^dst := arg min_{s∈ℝ′} EvalMove(s_2^src, s);
22             v⃗_2 := EvalMove(s_2^src, s_2^dst);
23             if v⃗_1 ≤^lex v⃗_2 then
24                 Move(s_1^src, 1, s_1^dst);
25                 i := i − 1;
26             else
27                 Move(s_2^src, 1, s_2^dst);
28                 j := j − 1;
29 function MoveNeed(c, s)
30     if (c, s) is immediate then
31         return 0;
32     else
33         return actual number of moves needed by the
           STAP to accomplish the task (c, s);
34 procedure BiReceiver(s^src, k_1, ℝ_1, k_2, ℝ_2)
35     i := k_1, j := k_2;
36     repeat k_1 + k_2 times do
37         if j = 0 then
38             Relocate(s^src, i, ℝ_1);
39             i := 0;
40         else if i = 0 then
41             Relocate(s^src, j, ℝ_2);
42             j := 0;
43         else
44             ℝ′_1 := {s ∈ ℝ_1 : h(s) < H};
45             ℝ′_2 := {s ∈ ℝ_2 : h(s) < H};
46             s_1^dst := arg min_{s∈ℝ′_1} EvalMove(s^src, s);
47             v⃗_1 := EvalMove(s^src, s_1^dst);
48             s_2^dst := arg min_{s∈ℝ′_2} EvalMove(s^src, s);
49             v⃗_2 := EvalMove(s^src, s_2^dst);
50             if v⃗_1 ≤^lex v⃗_2 then
51                 Move(s^src, s_1^dst);
52                 i := i − 1;
53             else
54                 Move(s^src, s_2^dst);
55                 j := j − 1;
```

Function $\texttt{Move}(s^{\text{src}}, k, s^{\text{dst}})$ performs k relocations from stack s^{src} to stack s^{dst}. Function

`Relocate`$(s^{\text{src}}, k, \mathbb{R})$ performs k relocations from a sender stack s^{src} to a receiver set $\mathbb{R}$. For each of the top k containers of the sender, the destination stack s^{dst} is properly selected from $\mathbb{R}$ according to the evaluations by the user-defined function `EvalMove`$(s^{\text{src}}, s^{\text{dst}})$. The evaluations of moves are represented by numerical tuples, which are lexicographically comparable. Function `MoveNeed`(c, s) returns the actual number of moves performed by the STAP to accomplish the given task (c, s).

Function `BiReceiver`$(s^{\text{src}}, k_1, \mathbb{R}_1, k_2, \mathbb{R}_2)$ relocates k_1 and k_2 containers from one sender stack s^{src} to two receiver sets, $\mathbb{R}_1$ and $\mathbb{R}_2$, respectively. Similarly, function `BiSender`$(s_1^{\text{src}}, k_1, s_2^{\text{src}}, k_2, \mathbb{R})$ performs relocations from two senders, stacks s_1^{src} and s_2^{src}, to the same receiver set $\mathbb{R}$, and the respective quantities are k_1 and k_2. The moving order of the two top containers from two senders is determined by the smaller evaluation tuple.

6.4.2. User-defined functions

User-defined functions in the STAP include

- `EvalMove`$(s^{\text{src}}, s^{\text{dst}})$ returns the penalty of the move from stack s^{src} to s^{dst};
- `Interim`$(\mathbb{I})$ selects the interim stack from set $\mathbb{I}$; and
- `InterimFull`$(\mathbb{F})$ selects the interim stack from set $\mathbb{F}$, which consists of $S - 2$ full stacks.

6.4.3. Task types

Let the target container be denoted by $c^*(s^+, t^+)$ and the aim stack by s^-. The task is ***immediate*** if it is already located in the aim slot; ***internal*** if the aim slot is below the target container in the same stack; and ***external*** if the aim slot is in a different stack.

6.4.4. Immediate task

An immediate task does not require a move because the target container is already located in the aim slot; that is, `MoveNeed`$(c^*, s^-) = 0$ for an immediate task (c^*, s^-).

6.4.5. Internal task

For an internal task, let a denote the number of empty slots in $\mathbb{S} \setminus \{s^+\}$ with the exclusion of the highest non-full stack; that is, $a = E - e(s^+) - \min\{e(s) > 0 : s \neq s^+\}$. If multiple stacks are

with the maximum height, only one is excluded. Let b_1 and b_2 denote the numbers of blocking containers above and below c^* in stack s^+, respectively. The accomplishment for an internal tack is given in Algorithm 3, which is divided into three situations:

- I1: $a \geq b_2$;
- I2: $a < b_2$ & $|\{s \neq s^+ : h(s) < H\}| > 1$; and
- I3: $a < b_2$ & $|\{s \neq s^+ : h(s) < H\}| = 1$.

Algorithm 3: Accomplish an internal task.

target container: $c^*(s^+, t^+)$
aim slot: $(s^+, f(s^+) + 1)$
slot supply: $a = E - e(s^+) - \min\{e(s) > 0 : s \neq s^+\}$
function $\alpha(s^+, s^{\text{dst}})$
  $e^{\min} := \min\{e(s) > 0 : s \neq s^+\}$;
  $e^{\text{sec}} := \min\{e(s) > e^{\min} : s \neq s^+\}$;
  $k^{\min} := |\{s \neq s^+ : e(s) = e^{\min}\}|$;
  if $e(s^{\text{dst}}) > e^{\min}$ **then**
    return $E - e(s^+) - e^{\min} - 1$;
  else if $e^{\min} \geq 2$ **then**
    return $E - e(s^+) - e^{\min}$;
  else if $k^{\min} = 1$ **then**
    return $E - e(s^+) - e^{\text{sec}} - 1$;
  else
    return $E - e(s^+) - 2$;
case I1: $a \geq b_2$
  repeat b_1 **times do**
    $\mathbb{R} := \{s \neq s^+ : h(s) < H,\ \alpha(s^+, s) \geq b_2\}$;
    `Relocate`$(s^+, 1, \mathbb{R})$;
  $\mathbb{I} := \{s \neq s^+ : h(s) < H,\ E - e(s^+) - e(s) \geq b_2\}$;
  $s^{\text{tmp}} :=$ `Interim`$(\mathbb{I})$;
  `Move`$(s^+, 1, s')$;
  `Relocate`$(s^+, b_2, \mathbb{S} \setminus \{s^+, s^{\text{tmp}}\})$;
  `Move`$(s^{\text{tmp}}, 1, s^+)$;
  // `I1:` $b_1 + b_2 + 2$ `moves`
blocking above: $b_1 = h(s^+) - t^+$
blocking below: $b_2 = t^+ - f(s^+) - 1$
case I2: $a < b_2$ & $|\{s \neq s^+ : h(s) < H\}| > 1$
  `Relocate`$(s^+, b_1, \mathbb{S} \setminus \{s^+\})$;
  $s_1^{\text{tmp}} :=$ `Interim`$(\mathbb{S} \setminus \{s^+\})$;
  `Move`$(s^+, 1, s_1^{\text{tmp}})$;
  $k_2 := E - e(s^+) - e(s_1^{\text{tmp}}) - 1$;
  `Relocate`$(s^+, k_2, \mathbb{S} \setminus \{s^+, s_1^{\text{tmp}}\})$;
  Find s_2^{tmp} s.t. $s_2^{\text{tmp}} \notin \{s^+, s_1^{\text{tmp}}\}$ & $h(s_2^{\text{tmp}}) < H$;
  `Move`$(s_1^{\text{tmp}}, 1, s_2^{\text{tmp}})$;
  `Move`$(s^+, b_2 - k_2, s_1^{\text{tmp}})$;
  `Move`$(s_2^{\text{tmp}}, 1, s^+)$;
  // `I2:` $b_1 + b_2 + 3$ `moves`
case I3: $a < b_2$ & $|\{s \neq s^+ : h(s) < H\}| = 1$
  Find s' s.t. $s' \neq s^+$ & $h(s') < H$;
  $s^{\text{tmp}} :=$ `InterimFull`$(\mathbb{S} \setminus \{s^+, s'\})$;
  `BiSender`$(s^{\text{tmp}}, 1, s^+, b_1, \{s'\})$;
  `Move`$(s^+, 1, s^{\text{tmp}})$;
  `Move`(s^+, b_2, s');
  `Move`$(s^{\text{tmp}}, 1, s^+)$;
  // `I3:` $b_1 + b_2 + 3$ `moves`
  // `(regression issue not yet included)`
  Resolve regression issue (cf. Section 6.4.7);

In case I1, a new function $\alpha(s^+, s)$ is defined similar to a, representing the number of empty slots in $\mathbb{S}\setminus\{s^+\}$ with the exclusion of the highest non-full stack, after the topmost blocking container of stack s^+ is moved to stack s. When the blocking containers above the target container are being relocated, the detection of enough slots is performed, to prevent unnecessary additional moves.

Note that in case I2, the target container c^* is moved to the new interim stack s_2^{tmp} from the first interim stack s_1^{tmp} when there is only one empty slot remaining in $\mathbb{S} \setminus \{s^+, s_1^{\text{tmp}}\}$. This can be modified so that c^* can be moved to a new interim stack s_2^{tmp} earlier as long as the empty slots in $\mathbb{S} \setminus \{s^+, s_2^{\text{tmp}}\}$ are enough for the remaining blocking containers in stack s^+. Moreover, if there is a full stack in $\mathbb{S} \setminus \{s^+\}$ in case I2, the task can also be completed in a similar way as that used in case I3, with the same operational cost.

6.4.6. External task

For an external task, the number of empty slots in $\mathbb{S} \setminus \{s^+, s^-\}$ is denoted by a; that is, $a = E - e(s^+) - e(s^-)$. Let b_1 and b_2 denote the numbers of blocking containers above the target container c^* and the aim slot $(s^-, f(s^-))$, respectively. The pseudo-code describing the accomplishment for an external task is given in Algorithm 4, with four situations considered,

- E1: $a \geq b_1 + b_2$;
- E2: $b_1 + 1 \leq a < b_1 + b_2$;
- E3: $1 \leq a < b_1 + \min\{1, b_2\}$; and
- E4: $a = 0 < b_1 + b_2$.

6.4.7. Regression issue

In cases I3 and E4, if the top container of the full stack selected (cf. line 34 in Algorithm 3 and line 19 in Algorithm 4) is not fixed, the task is done without any side effect. However, if the top container has been fixed, the container should be returned to its previously fixed slot, requiring additional moves.

The regression issue only occurs in tricky states, and it can be avoided if the tricky state avoidance technique (cf. Section 6.2) is applied.

7. Greedy and speedy heuristic

The proposed greedy and speedy heuristic (GASH) is a realized implementation of the generic feasibility-based heuristic scheme. The GASH employs all the techniques introduced above, i.e.,

Algorithm 4: Accomplish an external task.

target container: $c^*(s^+, t^+)$
aim slot: $(s^-, f(s^-)+1)$
blocking above target: $b_1 = h(s^+) - t^+$
blocking in aim stack: $b_2 = h(s^-) - f(s^-)$
slot supply: $a = E - e(s^+) - e(s^-)$

case E1: $a \geq b_1 + b_2$
  `BiSender`$(s^+, b_1, s^-, b_2, \mathbb{S} \setminus \{s^+, s^-\})$;
  `Move`$(s^+, 1, s^-)$;
  // `E1:` $b_1 + b_2 + 1$ `moves`

case E2: $b_1 + 1 \leq a < b_1 + b_2$
  $k_2 \coloneqq a - 1 - b_1$;
  `BiSender`$(s^+, b_1, s^-, k_2, \mathbb{S} \setminus \{s^+, s^-\})$;
  Find s^{tmp} s.t. $s^{\text{tmp}} \notin \{s^+, s^-\}$ & $h(s^{\text{tmp}}) < H$;
  `Move`$(s^+, 1, s^{\text{tmp}})$;
  `Move`$(s^-, b_2 - k_2, s^+)$;
  `Move`$(s^{\text{tmp}}, 1, s^-)$;
  // `E2:` $b_1 + b_2 + 2$ `moves`

case E3: $1 \leq a < b_1 + \min\{1, b_2\}$
  $k_1 \coloneqq a - 1$;
  `BiReceiver`$(s^+, k_1, \mathbb{S} \setminus \{s^+, s^-\}, b_1 - k_1, \{s^-\})$;
  Find s^{tmp} s.t. $s^{\text{tmp}} \notin \{s^+, s^-\}$ & $h(s^{\text{tmp}}) < H$;
  `Move`$(s^+, 1, s^{\text{tmp}})$;
  `Move`$(s^-, b_1 - k_1 + b_2, s^+)$;
  `Move`$(s^{\text{tmp}}, 1, s^-)$;
  // `E3:` $2b_1 + b_2 - a + 3$ `moves`

case E4: $a = 0 < b_1 + b_2$
  $s^{\text{tmp}} \coloneqq$ `InterimFull`$(\mathbb{S} \setminus \{s^+, s^-\})$;
  `BiSender`$(s^{\text{tmp}}, 1, s^+, b_1, \{s^-\})$;
  `Move`$(s^+, 1, s^{\text{tmp}})$;
  `Move`$(s^-, b_1 + b_2 + 1, s^+)$;
  `Move`$(s^{\text{tmp}}, 1, s^-)$;
  // `E4:` $2b_1 + b_2 + 4$ `moves`
  // `(regression issue not yet included)`
  Resolve regression issue (cf. Section 6.4.7);

the container stability concept, the techniques of tricky state avoidance and bottom tiers protection, and the STAP. The procedure of the GASH and the user-defined functions for the STAP are given in Algorithm 5.

Function `ValidTasks()` returns $\mathbb{T}$, the set of valid tasks for the current state. Every task $(c, s) \in \mathbb{T}$ is then evaluated by function `EvalTask`(c, s). The evaluation of a container–stack pair (c, s) is a six-tuple, that starts with the key indicator, followed by the actual number of moves required by the STAP, the number of stable containers that need to be moved from the aim stack, the affected demand, fixed height of the aim stack, and the opposite of the target container's priority value. At every step of the GASH, the next task is determined by the lexicographically minimum evaluation six-tuple.

Define the ***messiness*** of stack s by $m(s) = \max_{g(s)<t\leq h(s)} p(s, t)$, that is the largest priority value among the unstable containers in stack s. The preferences of selecting a stack s^{dst} as the destination stack for a blocking container are as follows.

1. If stack s^{dst} is entirely stable and can stabilize c, the minimum affected demand is preferred;
2. If stack s^{dst} is not entirely stable and $p(c) \geq m(s^{\text{dst}})$, the minimum gap between $m(s^{\text{dst}})$ and $p(c)$ is preferred;

Algorithm 5: Greedy and speedy heuristic.

begin
  $(\mathsf{L}, \boldsymbol{f}) \coloneqq (\mathsf{L}^0, U \cdot \mathbf{1})$;
  repeat $N - S\,U$ **times do**
    $\mathbb{T} \coloneqq$ `ValidTasks()`;
    $(c^*, s^-) \coloneqq \arg\min_{(c,s)\in\mathbb{T}}$ `EvalTask`(c, s);
    Accomplish (c^*, s^-) by the STAP;
    $f(s^-) \coloneqq f(s^-) + 1$;

function `ValidTasks()`
  $\mathbb{T} \coloneqq \varnothing$;
  foreach $(c, s) \in \mathbb{C} \times \mathbb{S}$ **do**
    if c is unfixed
    & $f(s) < H$
    & $p(c) \le q^f(s)$
    & $\Delta(\varphi) \ge H - f(s)$, for $p(c) < \varphi \le q^f(s)$ **then**
      $\mathbb{T} \coloneqq \mathbb{T} \cup \{(c, s)\}$;
  return $\mathbb{T}$;

function `EvalTask`(c, s)
  $p \coloneqq p(c)$;
  $q \coloneqq q^f(s, f(s))$;
  $\chi \coloneqq 0$; // `key indicator`
  if $f(s) < 2$ & $\delta(p, q) \ge S$ **then**
    // `bottom tiers protection`
    $\chi \coloneqq \infty$;
  if $(\mathsf{L}, \boldsymbol{f})$ is pre-tricky & $f(s) = H - 1$ **then**
    // `tricky state avoidance`
    $\chi \coloneqq \infty$;
  $n \coloneqq$ `MoveNeed`(c, s);
  return $\langle \chi,\ n,\ g(s) - f(s),\ \delta(p, q),\ f(s),\ -p(c) \rangle$;

function `EvalMove`$(s^{\mathrm{src}}, s^{\mathrm{dst}})$
  $c \coloneqq (s^{\mathrm{src}}, h(s^{\mathrm{src}}))$;
  $p \coloneqq p(c)$;
  $q \coloneqq q(s^{\mathrm{dst}}, h(s^{\mathrm{dst}}))$;
  case $g(s^{\mathrm{dst}}) = h(s^{\mathrm{dst}})$ & stack s^{dst} can stabilize c
    return $\langle 1,\ \delta(p, q) \rangle$;
  case $g(s^{\mathrm{dst}}) < h(s^{\mathrm{dst}})$ & $p \ge m(s^{\mathrm{dst}})$
    return $\langle 2,\ p - m(s^{\mathrm{dst}}) \rangle$;
  case $g(s^{\mathrm{dst}}) < h(s^{\mathrm{dst}})$ & $p < m(s^{\mathrm{dst}})$
    return $\langle 3,\ m(s^{\mathrm{dst}}) - p \rangle$;
  case $g(s^{\mathrm{dst}}) = h(s^{\mathrm{dst}})$ & stack s_2 cannot stabilize c
    return $\langle 4,\ q \rangle$;

function `Interim`$(\mathbb{I})$
  $\mathbb{I}_1 \coloneqq \{s \in \mathbb{I} : g(s) < h(s) < H\}$;
  $\mathbb{I}_2 \coloneqq \{s \in \mathbb{I} : g(s) = h(s) < H\}$;
  if $\mathbb{I}_1 \neq \varnothing$ **then**
    return $\arg\max_{s\in\mathbb{I}_1} m(s)$;
  else
    return $\arg\min_{s\in\mathbb{I}_2} q(s, h(s))$;

function `InterimFull`$(\mathbb{F})$
  $\mathbb{F}_1 \coloneqq \{s \in \mathbb{F} : f(s) \le g(s) < h(s) = H\}$;
  $\mathbb{F}_2 \coloneqq \{s \in \mathbb{F} : f(s) < g(s) = h(s) = H\}$;
  $\mathbb{F}_3 \coloneqq \{s \in \mathbb{F} : f(s) = g(s) = h(s) = H\}$;
  if $\mathbb{F}_1 \neq \varnothing$ **then**
    return $\arg\min_{s\in\mathbb{F}_1} p(s, h(s))$;
  else if $\mathbb{F}_2 \neq \varnothing$ **then**
    return $\arg\min_{s\in\mathbb{F}_2} p(s, h(s))$;
  else // `cause regression issue`
    return $\arg\min_{s\in\mathbb{F}_3} p(s, h(s))$;

3. If stack s^{dst} is not entirely stable and $p(c) > m(s^{\text{dst}})$, the minimum gap between $p(c)$ and $m(s^{\text{dst}})$ is preferred;

4. If stack s^{dst} is entirely stable but cannot stabilize c, the minimum $q(s^{\text{dst}}, h(s^{\text{dst}}))$ is preferred.

The first preference indicates that stabilizing a blocking container reduces the total number of unstable containers in the bay. The second and third preferences consider the messiness of the destination stack. Messiness refers to the largest priority value of the unstable containers in a stack; larger messiness implies a higher urgency of reshuffling. The last preference indicates that an entirely stable stack should be protected from being ruined.

In cases I1 and I2, an interim stack is selected to temporarily store the target container. The selection prefers stacks that are not entirely stable, with the largest messiness, then entirely stable stacks with the smallest capability. The most unattractive stack for receiving blocking containers is selected as the interim stack. In cases I3 and E4, the interim stack is selected by the minimum priority value of the top containers of these full stacks.

Note that because the tricky state avoidance technique is applied, the regression issue would never happen in the GASH, although it is still considered in function `InterimFull(`$\mathbb{F}$`)` to complete the whole logic.

8. Computational results

In this section, the proposed greedy and speedy heuristic (GASH) is compared to two benchmark heuristic approaches. One is the target-guided heuristic (TGH) proposed in our previous work [14], and the other is a similar implementation to the LPFH proposed by [6]. Since the LPFH only discusses cases with enough empty slots (cases I1 and E1 in our statement) and involves random decisions, it is difficult to implement a deterministic version. Thus, we develop a substitute instead, referred to as the largest priority value first heuristic (LPVFH).

The LPVFH selects the next target container from unfixed containers with the largest priority value, and then accomplish it by the STAP with the same user-defined functions used as that in the GASH. Necessary pseudo-code for the LPVFH is given in Algorithm 6.

Algorithm 6: Largest priority value first heuristic.

```
1  begin
2      (L, f) := (L^0, U · 1);
3      foreach p = P, ..., 1 do
4          C_p := containers with priority value p;
5          A_p := ∅; // set of aim stacks
6          while C_p ≠ ∅ do
7              (c*, s⁻) :=
               arg min_{c∈C_p, f(s)<H} MoveNeed(c, s);
8              Accomplish (c*, s⁻) by the STAP;
9              f(s⁻) := f(s⁻) + 1;
10             C_p := C_p \ {c*};
11             A_p := A_p ∪ {s⁻};
12         foreach s⁻ ∈ A_p do
13             Fill(s⁻);
14 function Fill(s⁻)
15     while h(s⁻) < H do
16         S' := ∅;
17         foreach s ∈ {s ≠ s⁻ : g(s) < h(s)} do
18             c := (s, h(s));
19             if stack s⁻ can stabilize c then
20                 S' := S' ∪ {s};
21         if S' ≠ ∅ then
22             s' := arg max_{s∈S'} p(s, h(s));
23             Move(s', 1, s⁻);
24         else
25             break while;
```

The problem instance format of the CPMP is the same as that of the container relocation problem (CRP, a.k.a. the blocks relocation problem), the problem of minimizing the total number of container moves for retrieving containers from the initial layout. In the literature, researchers usually use the same data sets when solving the CPMP [1,3,6,14] and the CRP [4,5,7,8,9].

8.1. Results for CVS instances

Caserta et al. [4] present the complete CVS data set (named after the authors' surnames, Caserta, Voß and Sniedovich) originally for the CRP. The CVS instances are classified into 21 groups, each consisting of 40 instances. The stacks of the initial layout have the same height in a CVS instance. The number of containers per stack in the initial layout is denoted by K, hence $N = SK$. It is worth noting that the stack height limitation is not specified in the original data. The researchers add two extra tiers above the initial layout; that is, $H = K + 2$.

The CVS instances can be considered typical dense CPMP instances. Table 2 illustrates the computational results on CVS instances by the GASH, the TGH, and the LPVFH. The values under the "move" headings represent average number of moves for every CVS group, whereas the values under the "gap%" headings are the ratios of the difference between each benchmark heuristic and the GASH to the number of containers N. The results showcase that the GASH leads the benchmark heuristics by more than $10\% \times N$ moves on most instance groups. Especially on

the narrowest group CVS 10-6, the GASH outperforms the TGH and the LPVFH by 319.38% and 55.54%, respectively.

Table 2: Results for CVS instances.

CVS K-S	GASH move	TGH		LPVFH	
		move	gap%	move	gap%
CVS 3-3	11.28	12.95	18.61	11.25	−0.28
CVS 3-4	10.80	12.18	11.46	12.23	11.88
CVS 3-5	12.08	12.78	4.67	13.45	9.17
CVS 3-6	12.98	14.38	7.78	14.88	10.56
CVS 3-7	14.75	16.00	5.95	16.58	8.69
CVS 3-8	15.65	16.55	3.75	17.08	5.94
CVS 4-4	21.88	23.35	9.22	21.93	0.31
CVS 4-5	23.08	26.73	18.25	26.48	17.00
CVS 4-6	24.75	27.58	11.77	27.20	10.21
CVS 4-7	27.63	29.93	8.21	31.23	12.86
CVS 5-4	35.08	44.83	48.75	35.83	3.75
CVS 5-5	35.33	42.40	28.30	36.50	4.70
CVS 5-6	39.88	50.63	35.83	43.08	10.67
CVS 5-7	41.68	48.83	20.43	46.95	15.07
CVS 5-8	47.50	56.68	22.94	51.83	10.81
CVS 5-9	50.45	57.50	15.67	55.65	11.56
CVS 5-10	54.63	62.80	16.35	60.88	12.50
CVS 6-6	55.23	74.33	53.06	57.85	7.29
CVS 6-10	75.60	88.63	21.71	79.73	6.88
CVS 10-6	140.63	332.25	319.38	173.95	55.54
CVS 10-10	179.23	302.90	123.68	190.50	11.28
Average	44.29	64.48	38.37	48.81	11.26

8.2. Results for BF instances

Bortfeldt & Forster [1] introduce 32 groups of CPMP instances (referred to as BF instances), each group consisting of 20 instances. In BF instances, the bay size is $S = 16$ or 20 and $H = 5$ or 8. The number of containers N is either $0.6 \times SH$ or $0.8 \times SH$, the number of priorities P is either $0.2 \times N$ or $0.4 \times N$, and the number of disorderly containers B is either $0.6 \times N$ or $0.75 \times N$ in the initial layout.

The BF instances can be considered typical loose CPMP instances. Table 3 illustrates the computational results on BF instances by the three heuristics. The values under the "ratio%" headings are the relative gap between per heuristic and the number of disorderly containers initially

B, and the values under the "gap%" headings are the differences between the "ratio%" columns of the GASH and each benchmark heuristic.

Table 3: Results for BF instances.

BF	S	H	N	P	B	GASH		TGH			LPVFH		
						move	ratio%	move	ratio%	gap%	move	ratio%	gap%
1	16	5	48	10	29	29.15	0.52	29.10	0.34	−0.17	29.55	1.90	1.38
2	16	5	48	10	36	36.00	0.00	36.00	0.00	0.00	36.60	1.67	1.67
3	16	5	48	20	29	29.35	1.21	29.45	1.55	0.34	30.90	6.55	5.34
4	16	5	48	20	36	36.15	0.42	36.00	0.00	−0.42	37.20	3.33	2.92
5	16	5	64	13	39	46.30	18.72	48.50	24.36	5.64	53.00	35.90	17.18
6	16	5	64	13	48	55.50	15.63	57.55	19.90	4.27	62.85	30.94	15.31
7	16	5	64	26	39	49.95	28.08	53.55	37.31	9.23	57.15	46.54	18.46
8	16	5	64	26	48	57.60	20.00	60.00	25.00	5.00	66.90	39.38	19.38
9	16	8	77	16	47	56.30	19.79	60.35	28.40	8.62	62.15	32.23	12.45
10	16	8	77	16	58	61.55	6.12	62.15	7.16	1.03	69.50	19.83	13.71
11	16	8	77	31	47	55.00	17.02	61.25	30.32	13.30	63.70	35.53	18.51
12	16	8	77	31	58	61.45	5.95	63.45	9.40	3.45	68.50	18.10	12.16
13	16	8	103	21	62	96.50	55.65	107.45	73.31	17.66	110.85	78.79	23.15
14	16	8	103	21	78	116.05	48.78	124.75	59.94	11.15	134.90	72.95	24.17
15	16	8	103	42	62	99.45	60.40	110.60	78.39	17.98	110.40	78.06	17.66
16	16	8	103	42	78	115.45	48.01	133.35	70.96	22.95	137.40	76.15	28.14
17	20	5	60	12	36	36.50	1.39	36.50	1.39	0.00	37.35	3.75	2.36
18	20	5	60	12	45	45.20	0.44	45.00	0.00	−0.44	45.00	0.00	-0.44
19	20	5	60	24	36	36.75	2.08	36.80	2.22	0.14	38.50	6.94	4.86
20	20	5	60	24	45	45.10	0.22	45.00	0.00	−0.22	45.70	1.56	1.33
21	20	5	80	16	48	56.55	17.81	61.65	28.44	10.63	65.65	36.77	18.96
22	20	5	80	16	60	65.55	9.25	67.90	13.17	3.92	74.50	24.17	14.92
23	20	5	80	32	48	55.25	15.10	61.10	27.29	12.19	65.75	36.98	21.88
24	20	5	80	32	60	68.00	13.33	70.95	18.25	4.92	76.65	27.75	14.42
25	20	8	96	20	58	66.00	13.79	69.80	20.34	6.55	73.60	26.90	13.10
26	20	8	96	20	72	75.75	5.21	74.35	3.26	−1.94	81.75	13.54	8.33
27	20	8	96	39	58	65.65	13.19	71.85	23.88	10.69	73.65	26.98	13.79
28	20	8	96	39	72	76.50	6.25	76.30	5.97	−0.28	83.55	16.04	9.79
29	20	8	128	26	77	115.85	50.45	118.65	54.09	3.64	128.65	67.08	16.62
30	20	8	128	26	96	129.60	35.00	143.05	49.01	14.01	155.15	61.61	26.61
31	20	8	128	52	77	115.85	50.45	128.15	66.43	15.97	128.80	67.27	16.82
32	20	8	128	52	96	134.10	39.69	147.30	53.44	13.75	157.00	63.54	23.85
Average						68.44	19.37	72.75	26.05	6.67	76.96	33.09	13.71

The relative gap between the solution value and the number of disorderly containers in the initial layout is a good measure of the heuristic performance. Table 4 shows that the instance density N/SH (or bay utilization) and the height of the bay H are key factors in the number of moves needed for pre-marshalling. In other words, denser or higher instances are more difficult

to solve. The computational results on BF instances also prove that the GASH outperforms these two benchmark heuristics.

Table 4: Summary on BF instances.

	GASH ratio%		TGH gap%		LPVFH gap%	
density	$H = 5$	8	5	8	5	8
0.6	0.78	10.91	−0.10	5.18	4.85	10.50
0.8	17.24	48.55	6.97	14.64	19.65	14.81

9. Conclusions

In this paper we present a generic heuristic scheme based on new concepts of state and state feasibility for solving the container pre-marshalling problem. The proposed heuristic scheme is an innovative departure from the traditional heuristic strategy. Furthermore, a realized instance of the heuristic scheme, the greedy and speedy heuristic, is also proposed. Numerical experiments on two commonly used data sets are conducted, comparing the proposed feasibility-based heuristic to two benchmark heuristics.

A major challenge raised in the feasibility-based heuristic scheme is the trade-off between the freedom of selecting target containers and the waste of the resource surplus. In this paper, the bottom tiers protection technique is proposed to balance the trade-off. As a future work, efforts in better solving the difficulty can be considered to improve the performance of the feasibility-based heuristic. Moreover, new feasibility-based heuristics and meta-heuristics are also worthy exploring.

References

[1] Bortfeldt, A. & Forster, F. (2012). A tree search procedure for the container pre-marshalling problem. *European Journal of Operational Research*, 217(3), 531–540.

[2] Carlo, H. J., Vis, I. F. A., & Roodbergen, K. J. (2014). Storage yard operations in container terminals: Literature overview, trends, and research directions. *European Journal of Operational Research*, 235(2), 412–430.

[3] Caserta, M. & Voß, S. (2009). A corridor method-based algorithm for the pre-marshalling problem. In M. Giacobini, A. Brabazon, S. Cagnoni, G. A. Di Caro, A. Ekárt, A. I. Esparcia-Alcázar, M. Farooq, A. Fink, & P. Machado (Eds.), *Applications of Evolutionary Computing*, volume 5484 of *Lecture Notes in Computer Science* (pp. 788–797). Springer Berlin Heidelberg.

[4] Caserta, M., Voß, S., & Sniedovich, M. (2011). Applying the corridor method to a blocks relocation problem. *OR Spectrum*, 33(4), 915–929.

[5] Expósito-Izquierdo, C., Melián-Batista, B., & Moreno-Vega, J. M. (2014). A domain-specific knowledge-based heuristic for the blocks relocation problem. *Advanced Engineering Informatics*, 28(4), 327–343.

[6] Expósito-Izquierdo, C., Melián-Batista, B., & Moreno-Vega, M. (2012). Pre-marshalling problem: Heuristic solution method and instances generator. *Expert Systems with Applications*, 39(9), 8337–8349.

[7] Forster, F. & Bortfeldt, A. (2012). A tree search procedure for the container relocation problem. *Computers & Operations Research*, 39(2), 299–309.

[8] Jin, B., Lim, A., & Zhu, W. (2013). A greedy look-ahead heuristic for the container relocation problem. In M. Ali, T. Bosse, K. V. Hindriks, M. Hoogendoorn, C. M. Jonker, & J. Treur (Eds.), *Recent Trends in Applied Artificial Intelligence*, volume 7906 of *Lecture Notes in Computer Science* (pp. 181–190). Springer Berlin Heidelberg.

[9] Jin, B., Zhu, W., & Lim, A. (2015). Solving the container relocation problem by an improved greedy look-ahead heuristic. *European Journal of Operational Research*, 240(3), 837–847.

[10] Kim, K. H. & Lee, H. (2015). Container terminal operation: Current trends and future challenges. In C.-Y. Lee & Q. Meng (Eds.), *Handbook of Ocean Container Transport Logistics*, volume 220 of *International Series in Operations Research & Management Science* (pp. 43–73). Springer International Publishing.

[11] Lee, Y. & Chao, S.-L. (2009). A neighborhood search heuristic for pre-marshalling export containers. *European Journal of Operational Research*, 196(2), 468–475.

[12] Lee, Y. & Hsu, N.-Y. (2007). An optimization model for the container pre-marshalling problem. *Computers & Operations Research*, 34(11), 3295–3313.

[13] Lehnfeld, J. & Knust, S. (2014). Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. *European Journal of Operational Research*, 239(2), 297–312.

[14] Wang, N., Jin, B., & Lim, A. (2015). Target-guided algorithms for the container pre-marshalling problem. *Omega*, 53, 67–77.