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

Ning Wang[a], Bo Jin[b,*], Zizhen Zhang[c], Andrew Lim[d]

[a]*Department of Information Management, School of Management, Shanghai University, Shanghai, China*
[b]*Department of Management Sciences, City University of Hong Kong, Hong Kong*
[c]*School of Data and Computer Science, Sun Yat-Sen University, China*
[d]*Department of Industrial & Systems Engineering, National University of Singapore, Singapore*

Abstract

This paper addresses the container pre-marshalling problem (CPMP) which rearranges containers inside a storage bay to a desired layout. By far, target-driven algorithms have relatively good performance among all algorithms; they have two key components: first, containers are rearranged to their desired slots one by one in a certain order; and second, rearranging one container is completed by a sequence of movements. Our paper improves the performance of the target-driven algorithm from both aspects. The proposed heuristic determines the order of container rearrangements by the concepts of state feasibility, container stability, dead-end avoidance and tier-protection proposed in this paper. In addition, we improve the efficiency of performing container rearrangements by discriminating different task types. Computational experiments showcase that the performance of the proposed heuristic is considerable.

Keywords: container pre-marshalling problem, feasibility-based heuristic, tier-protection

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

*Corresponding author.
Email addresses: ningwang@shu.edu.cn (Ning Wang), msjinbo@cityu.edu.hk (Bo Jin), zhangzizhen@gmail.com (Zizhen Zhang), alim.china@gmail.com (Andrew Lim)

multiple transportation modes without directly handling the freight. Over the years, stringent requirements from consignors such as just-in-time operations have created challenges for the container transportation industry.

Container yards – the space 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 speaking, 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.

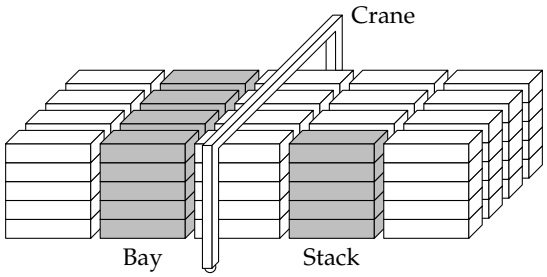


Figure 1: Yard block.

Containers in the same stack are operated in a last-in-first-out manner. To retrieve containers placed in lower tiers, containers on top of them must be relocated to other slots first. Such forced movements are known as container reshuffles or relocations. Container pre-marshalling addresses the reorganization of containers inside a storage area such that no reshuffle is further required when containers are retrieved. Hence, the aim of pre-marshalling is to improve terminals' performance level, including throughput rate per berth and turnaround time of vessels or road trucks [14].

The container pre-marshalling problem (CPMP) can be formally defined as follows. Containers are categorized into various groups, and each group is assigned with a group value. Given an initial layout of containers in a single bay, the CPMP aims at rearranging containers within

the bay, such that containers in each stack are placed in a descending order of group values from the bottom up. The objective of CPMP is to find the best container rearrangement plan with the fewest number of container movements. It is assumed that the future retrieval order of containers is known beforehand and no container arrives at or leaves from the bay during pre-marshalling.

This paper designs a constructive heuristic which can be embedded later into frameworks such as beam search, large neighborhood search (LNS) and the greedy randomized adaptive search procedure (GRASP). Our work makes three main contributions to the literature. The first contribution is the concept of state feasibility. A set of rules are proposed for checking the feasibility of the states during the pre-marshalling process. The feasibility of rearranging a certain container is checked before we actually conduct the rearrangement, which guarantees search efficiency. The time complexity of checking the feasibility is only $O(G)$; here G is the number of groups. Our paper is the first work that applies feasibility to prune branches in CPMP algorithms. The second contribution lies in the new techniques proposed: container stability, dead-end avoidance and tier-protection indicator, which foster better heuristic decisions. The third contribution is the improvement of single container rearrangement which is achieved based on the relationship between the number of available slots and that of the blocking containers. This improvement avoids unnecessary container movements in certain situations.

The remainder of this paper is structured as follows. Section 2 reviews existing approaches in the literature. Section 3 formally describes the problem and lists the notation used throughout this paper. The concepts of state, state feasibility and container stability are introduced in Section 4. The proposed heuristic and the associated techniques are explained in Section 5. Section 6 illustrates the experimental results of benchmark instances, and Section 7 concludes this paper and identifies future research directions.

2. Recent work

According to recent reviews [3,17], problems related to or caused by container reshuffles at terminals include three major types of decision problems. The first problem is the container relocation problem [13,11] which minimizes the total operational cost in the container retrieval process. The operational cost is commonly measured by the number of relocations conducted or the total

operational time. The second is the container pre-marshalling problem that is addressed in this paper. The third problem is the container stacking problem which decides the storage locations or crane operations for arriving containers [6]. All of the three problems are closely related and the solution methodologies have some common merits.

To the best of our knowledge, works that study the solutions to the CPMP are rather limited compared to other problems related to terminals. For example, berth allocation problems and quay crane scheduling problems have been studied by more than 120 publications just since 2009 [1]. One reason of limited works is that the variants of the CPMP by far are few. Known CPMP variants were studied by Wang et al. [22] who considered truck lanes, Rendl & Prandtstetter [18] who allowed for group ranges rather than group values, and Huang & Lin [10] who required that containers of different groups be separately located in the final layout.

Lee & Hsu [16] developed an integer programming model for the CPMP. In their work, the problem was formulated as the 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 CPMP. The model provides an innovative viewpoint to see the problem, however, its performance is not good because the network is too large even for a small instance.

A neighborhood search was proposed by Lee & Chao [15], which repeatedly modifies the current solution until some termination condition is met. Unlike other existing solution-construction approaches, the neighborhood search is required to start with a pre-generated initial solution. A feasible solution is 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 new resultant solution is not always ensured.

Bortfeldt & Forster [2] described a tree search procedure for solving the problem. In the tree search procedure, 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. Tierney et al. [19] realized an A* algorithm with symmetry breaking rules, and van Brink

& van der Zwaan [20] presented an exact algorithm based on the branch and bound.

Caserta & Voß [4] provided a greedy heuristic named the corridor method for solving the problem. The heuristic selects the direction of movements in a randomized manner according to the attractiveness of available successors confined by the corridor. A local improvement procedure was also conducted to accelerate the heuristic process. Expósito-Izquierdo et al. [7] provided the first group-oriented heuristic for the CPMP. Their method iteratively handles containers in a descending order of group values. After handling all containers with a specific group value, a stack filling process was applied to reduce the number of disorderly containers in the bay. Jovanovic et al. [12] developed a new method which designs different heuristics for each of the four stages of Expósito-Izquierdo et al. [7]. Wang et al. [22] proposed a target-guided heuristic and two beam search algorithms. The processes of determining container handling sequence and handling particular containers are both improved. This work can solve instances of different densities well, especially dense instances with very few empty slots. Gheith et al. [8] proposed a rule-based heuristic procedure for solving the problem, and then developed a variable chromosome length genetic algorithm [9].

3. Problem description and notation

The CPMP is restricted to the bay size, or more precisely, the dimensions of the operating cranes. A problem 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 ($H \geq 2$) with E empty slots ($E = SH - N$, $E \geq 2$) left.

Every container is labeled a group value $g \in \{1, \ldots, G\}$. A container is ***orderly*** if it is supported directly by the ground or another orderly container with equal or larger group value; otherwise, it is ***disorderly***. Other phrases in the recent literature that have the same meaning of "orderly/disorderly" include "well/badly placed" [2], "well-/non-located" [7], and "clean/dirty" [22].

Figure 2 gives an example of a bay with $S = 5$, $H = 4$, and $N = 13$. Containers are represented by boxes with their group values marked inside. Specially, disorderly containers are highlighted by gray backgrounds. The objective of the CPMP is to find an optimized sequence with the fewest

container movements. By applying the obtained movement sequence to the initial layout, all the containers could be rearranged to be orderly.

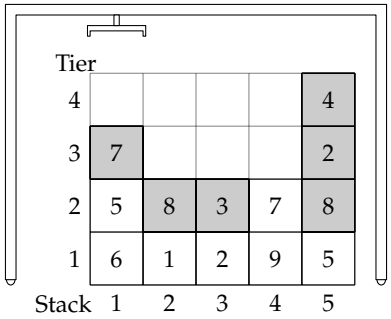


Figure 2: A container bay.

To better describe the solution to the CPMP, layout-related notation is introduced first. Stacks and tiers of the bay are labeled from left to right and bottom to top, respectively. 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 stack s. Note that the height of stacks should not exceed H. The orderly height (number of orderly containers) of stack s is denoted by $o(s)$. The slot positioned at the t-th tier of stack s is denoted by (s, t). The group value of a container c is denoted by $g(c)$. Moreover, the group value of a container located in slot (s, t) is denoted by $g(s, t)$.

4. State and state feasibility

Definition 1 (Fixed containers)**.** *A fixed container is an orderly container which has been locked to a certain slot. Fixed containers are not allowed to be moved in subsequent movements.*

According to the definition of fixed containers, fixed containers in the same stack are contiguous from the bottom up.

Definition 2 (State)**.** *A state* $(\mathsf{L}, \boldsymbol{f})$ *is a two-tuple composed of 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 of the layout L. The s-th element of $\boldsymbol{f}$ is denoted by $f(s)$.

Figure 3 illustrates two examples of state, where fixed and unfixed containers are separated by ***skylines*** (bold lines). In Figure 3(a), $\boldsymbol{f} = (2, 3, 1)$; in Figure 3(b), $\boldsymbol{f} = (2, 3, 0)$.

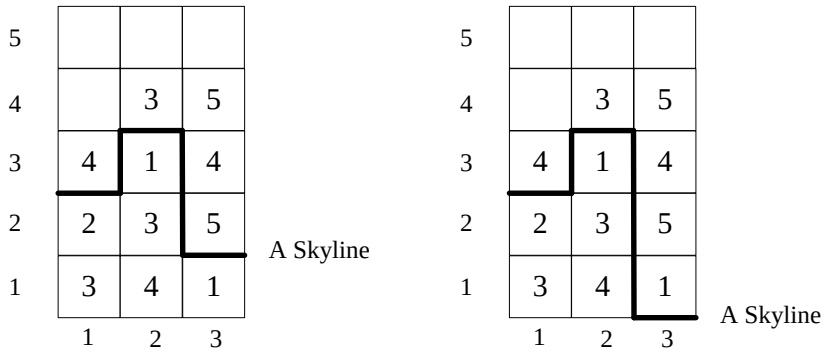


Figure 3: State, fixed vector and skyline.

It is noteworthy that if two states have the same layout but different fix vectors, then they are two different states, just like Figures 3(a) and 3(b).

Definition 3 (Feasibility of a state)**.** *A state* $(\mathsf{L}, \boldsymbol{f})$ *is feasible if there exists a movement sequence that can convert* L *to an orderly layout without moving fixed containers indicated by* $\boldsymbol{f}$.

The necessary condition for state feasibility has been talked about in Wang [21]. To make the content self-contained, we briefly explain it in this paper. Containers above the skyline can be moved, therefore slots and containers above the skyline are separately considered.

For a stack s in state $(\mathsf{L}, \boldsymbol{f})$, suppose that the smallest group value under the skyline is g, then there are $H - f(s)$ slots above the skyline which can only be filled with containers c such that $g(c) \leq g$. In such a situation, we say stack s has $H - f(s)$ slots with capability g. If the skyline of stack s coincides with the ground, then stack s has H slots with capability G. For example, in Figure 3(b), stack 1, 2 and 3 have 3, 2 and 5 slots with capabilities 2, 1 and 5, respectively. The total number of slots with capability g in the whole bay is denoted by $r(g)$ (resource for group g). Similarly, the number of unfixed containers with group value g is denoted by $d(g)$ (demand of group g).

In a feasible state, the slots available for group g (i.e., slots with capabilities no smaller than g) must exceed the demand of group g. That is,

$$\sum_{i=g+1}^{G} r(i) - \sum_{i=g+1}^{G} d(i) + r(g) \geq d(g),$$

which is equivalent to

$$\sum_{i=g}^{G} r(i) \geq \sum_{i=g}^{G} d(i).$$

Denote $R(g) = \sum_{i=g}^{G} r(i)$ and $D(g) = \sum_{i=g}^{G} d(i)$, then $\boldsymbol{R}$ and $\boldsymbol{D}$ are the two G-dimensional vectors with elements $R(g)$ and $D(g)$, respectively. The above condition can be hence expressed as $\boldsymbol{\Delta} = \boldsymbol{R} - \boldsymbol{D} \geq \boldsymbol{0}$. Here, $\boldsymbol{\Delta}$ is called the ***surplus vector***.

Proposition 1 (Necessary condition for state feasibility)**.** $\boldsymbol{\Delta} = \boldsymbol{R} - \boldsymbol{D} \geq \boldsymbol{0}$ *is a necessary condition for a feasible state* $(\mathsf{L}, \boldsymbol{f})$.

Take the two states in Figures 3(a) and 3(b) as examples. Table 1 shows the surplus vectors for both states. State a is infeasible since $\Delta(g) < 0$ for $g = 2, \ldots, 5$. It is intuitively seen that none of the stacks in Figure 3(a) can accommodate containers with group value 5, and this is the reason why state a is infeasible. The feasibility of state b is not sure yet because $\boldsymbol{\Delta} \geq \boldsymbol{0}$ is only a necessary condition. Note that the time complexity of checking $\boldsymbol{\Delta} \geq \boldsymbol{0}$ is $O(G)$.

Table 1: Computation of the surplus vector.

State a					
g	$d(g)$	$D(g)$	$r(g)$	$R(g)$	$\Delta(g)$
1	0	5	6	9	3
2	0	5	3	3	-2
3	1	5	0	0	-5
4	2	4	0	0	-4
5	2	2	0	0	-2

State b					
g	$d(g)$	$D(g)$	$r(g)$	$R(g)$	$\Delta(g)$
1	1	6	2	10	4
2	0	5	3	8	3
3	1	5	0	5	0
4	2	4	0	5	1
5	2	2	5	5	3

Definition 4 (Container stability)**.** *Consider a given feasible state. Disorderly containers are unstable. For any orderly container c in slot (s, t), try to fix container c and those underneath. If the resultant state has $\boldsymbol{\Delta} \geq 0$, then c is stable; otherwise, c is unstable.*

According to the definition, an unstable container is not allowed to be fixed in the current slot even if it is orderly, because the resultant state would be infeasible. Unstable containers must be moved in the future. Denote the stable height (number of stable containers) of stack s by $sh(s)$, then we have $\boldsymbol{f} \leq \boldsymbol{sh} \leq \boldsymbol{o} \leq \boldsymbol{h}$. If a container c could become stable by moving it onto stack s, we say that stack s can ***stabilize*** c.

Figure 4 gives another example of a state. Fixed containers are labeled with solid squares and unstable containers are highlighted in gray backgrounds. Notice that the container in slot $(2, 1)$ with group value 6 is orderly but unstable.

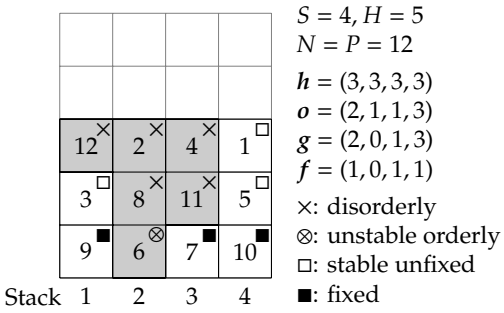


Figure 4: Stability of containers.

An orderly but unstable container indicates that its orderliness is valueless because it definitely requires at least one movement. Using the concept of container stability instead of container orderliness is more precise in evaluating the movements of blocking containers during the pre-marshalling process. For example, when deciding the next movement of a blocking container, if the container becomes orderly but unstable after being moved to a new stack, the attractiveness of such a movement should be reconsidered. The stability of containers should be recomputed after any container is fixed.

Definition 5 (Extreme state)**.** *A state* $(\mathsf{L}, \boldsymbol{f})$ *is called an extreme state if* $|\{s \in \mathbb{S} : f(s) = H\}| = S - 2$.

The definition of extreme state is to emphasize the special case where $S - 2$ stacks are fully fixed. In such a case, unfixed containers can only be moved between the other two stacks, say a and b. An extreme state is feasible if and only if one of the following conditions is satisfied:

1. Both stacks a and b are orderly;

2. Stack a is orderly, and stack b is disorderly. Stack a is still orderly after all the disorderly containers of stack b are moved to stack a.

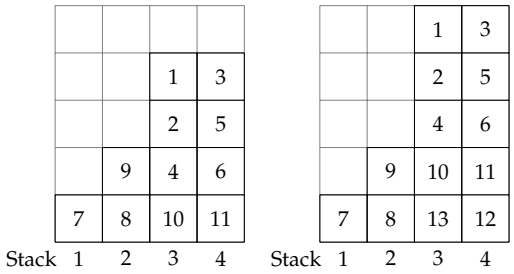
Definition 6 (Dead-end state)**.** *A state is called a dead-end state if it is an infeasible extreme state.*

We have the following proposition.

Proposition 2 (Existence of dead-end states)**.** *In extremely dense instances such that $E < 2(H-1)$, there exist possibilities of generating a dead-end state, while in instances with $E \geq 2(H-1)$, dead-end states will never be generated.*

When $E \geq 2(H-1)$, it has $N \leq SH-2H+2$. When fewer than $S-2$ stacks are fully occupied, the state is not a dead-end state; when $S-2$ stacks are fully occupied, there are at most two extra containers in the left two stacks. The state is not a dead-end state, either. Figure 5(a) gives an example of an instance with $E \geq 2(H-1)$, which has no chance to generate any dead-end state.

For instances with $E < 2(H-1)$, there exist possibilities of generating a dead-end state. Figure 5(b) gives an example with $E < 2(H-1)$; when $\boldsymbol{f} = (0,0,5,5)$, the corresponding state is a dead-end state.



(a) (b)

Figure 5: Existence of dead-end states.

5. Feasibility-based heuristic

In this section, a feasibility-based heuristic (FBH) is developed for solving the CPMP. The proposed heuristic is a target-driven algorithm. Target-driven algorithms [7,22] are efficient among

all existing algorithms. Generally speaking, containers (targets) are rearranged to certain slots one by one, and each rearrangement is achieved by a sequence of movements. The heuristic can be implemented independently as a greedy algorithm, or be a major component embedded in meta-heuristic frameworks. In this paper, we optimize the order of container rearrangements as well as the sequence of movements in one rearrangement.

The optimization is partially due to the concept of state feasibility. This is also the reason why our algorithm is called feasibility-based heuristic. Before a certain container is rearranged, the feasibility of the resultant state is always ensured. In other words, any rearrangement that leads to an infeasible state is eliminated. Due to the state feasibility check, we can explore larger space without sacrificing efficiency. The order of container rearrangements is dynamically determined as the algorithm goes on.

5.1. Heuristic framework

According to Wang [21], in dense instances with $E < H$, only the top $\sum_{i \neq s} e(i)$ containers of a stack s can be moved. The immovable number of containers (unreachable tiers) in any stack is $H - E$, i.e., containers in the lowest $H - E$ tiers of the bay are unreachable. In loose instances such that $E \geq H$, all the containers are movable. We define $U = \max\{H - E, 0\}$ as the number of unreachable tiers for any instance.

The feasibility-based heuristic firstly constructs the initial state according to the unreachable tiers. In the initial state, L^0 is the initial layout and containers in unreachable tiers are naturally fixed, i.e., the initial state is $(\mathsf{L}^0, U \cdot \mathbf{1})$, where $\mathbf{1}$ is an S-dimensional all-ones vector.

Starting from the initial state, the heuristic repeatedly fixes a target container c^* to an aim stack s^*. Since stack s^* already has $f(s^*)$ fixed containers, c^* will be fixed at tier $f(s^*)+1$. The algorithm continues until all the containers are fixed. The procedure of the proposed heuristic is concisely described in Algorithm 1.

The procedures of selecting the next task $c^* \rightarrow s^*$ and rearranging c^* to stack s^* will be explained in Sections 5.2 and 5.3, respectively.

The advantage of fixing containers one by one lies in twofold.

1. Movements are more target-oriented. Some works such as Bortfeldt & Forster [2] move a

Algorithm 1: Feasibility-based heuristic.

```
begin
    (L, f) := (L^0, U · 1);
    repeat N − S U times do
        (c* → s*) := the selected valid task;
        L := resultant layout after rearranging c* to stack s*;
        f(s*) := f(s*) + 1;
```

container each time, but they don't have a clear aim. Hence, it is difficult to measure the benefit of movements. Moreover, movements are blind: a container can be unnecessarily moved from stack a to b and then moved from b to a in later movements.

2. Unfixed containers are allowed to be moved, but fixed ones are not. The "fixed/unfixed" status can distinguish fixed containers and unfixed containers, especially unfixed orderly containers. In works that do not distinguish fixed and unfixed containers, containers that have been rearranged ("fixed" in our context) still get involved in later movements.

5.2. Task selection

5.2.1. Valid task

At every step of the heuristic, an unfixed container and a destination stack are determined as the target container and the aim stack, respectively. Most target-driven algorithms select target containers according to a predetermined order, such as the descending order of group values [7]. In this research, the order to rearrange containers is not fixed beforehand, but rather dynamically decided as the algorithm goes on. This innovation undoubtedly enables the algorithm to explore larger solution space, but one may say that the search efficiency is reduced. To this end, the concept of state feasibility is applied to avoid useless solution space. Our paper is the first work that applies state feasibility to prune branches in existing CPMP algorithms.

A container–stack pair $c \to s$ is a valid task for the current state $(\mathsf{L}, \boldsymbol{f})$ only if

- c is unfixed,
- $f(s) < H$,
- $g(c) \leq g(s, f(s))$, and

- $\Delta(\varphi) \geq H - f(s)$, for $\varphi \in (g(c), g(s, f(s))]$.

The first condition is easy to understand. The second condition ensures that stack s has at least one slot to accommodate c. The third condition makes sure that the group value of c is not larger than $g(s, f(s))$. The last condition ensures the non-negativity of the resultant surplus vector. If c is moved to stack s, then $d(g(c))$ (i.e., the demand of group $g(c)$) decreases by 1, that is, the demand of group $g(c)$ in the resultant layout is $d'(g(c)) = d(g(c)) - 1$, while other elements of the demand vector are not changed. Meanwhile, for the resource vector, $r'(g(s, f(s))) = r(g(s, f(s))) - (H - f(s))$, $r'(g(c)) = r(g(c)) + H - f(s) - 1$, and other elements are not changed. Then for $\varphi \in (g(c), g(s, f(s))]$, the feasibility condition in the resultant layout is $D'(\varphi) = \sum_{g \geq \varphi} d'(g) = D(\varphi) \leq R'(\varphi) = \sum_{g \geq \varphi} r'(g) = R(\varphi) - (H - f(s))$, which can be further simplified to $\Delta(\varphi) \geq H - f(s)$.

Container–stack pairs that satisfy the above conditions compose a set $\mathbb{T}$. However, the above conditions are only necessary conditions. The feasibility of the resultant state is still not ensured, due to the possibility of dead-end states.

There are three methods for resolving the issue of dead-end states.

1. Prevent from entering an extreme state when deciding the next task;

2. Design an ideal task accomplishment procedure that guarantees the feasibility of resultant states;

3. Allow entering a dead-end state and relabel a fixed container as unfixed to escape from the dead-end state.

Our previous work [22] adopts the last method listed above. When the algorithm enters a dead-end state, the algorithm relabels a fixed container as unfixed and moves it to a temporary slot. After taking a sequence of movements, the relabeled container returns to the slot where it is originally fixed. The disadvantage of such a method is that escaping from the current dead-end state causes unnecessary movements. A better choice is to avoid resulting in dead-end states in advance when choosing valid tasks.

Definition 7 (Pre-extreme state)**.** *A feasible state is a pre-extreme state if* $|\{i \in \mathbb{S} : f(i) = H\}| = S - 3$, $|\{i \in \mathbb{S} : f(i) = H - 1\}| \geq 1$ *and* $\sum_{i \in \mathbb{S}} f(i) < N - 2$.

The proposed heuristic only explores states $(\mathsf{L}, \boldsymbol{f})$ which satisfy $\boldsymbol{\Delta} \geq \mathbf{0}$. When $S-3$ stacks are fully fixed ($|i \in \mathbb{S} : f(i) = H| = S-3$), and at least one stack has only one available slot left ($|i \in \mathbb{S} : f(i) = H-1| \geq 1$), let us investigate the validity of performing a task $c \rightarrow s$ with $f(s) = H-1$ based on the number of remaining unfixed containers.

1. If $\sum_{i \in \mathbb{S}} f(i) = N$, all the containers are fixed. The algorithm terminates with an orderly layout.

2. If $\sum_{i \in \mathbb{S}} f(i) = N-1$, it means that only one container c remains unfixed. As $\boldsymbol{\Delta} \geq \mathbf{0}$, and $D(g(c)) = d(g(c)) = 1$, we have $R(g(c)) \geq 1$, that is, there is at least one slot which can make c orderly. At most one movement is needed to convert the layout to an orderly layout.

3. If $\sum_{i \in \mathbb{S}} f(i) = N-2$, two containers remain unfixed. Suppose the last two unfixed containers are a and b, and the valid task is $a \rightarrow s$ with $f(s) = H-1$. Because $a \rightarrow s$ is a valid task, so the fourth condition ($\Delta(\varphi) \geq H - f(s)$, for $\varphi \in (g(c), g(s, f(s))]$) is satisfied. Therefore, the resultant layout satisfies $\boldsymbol{\Delta}' \geq \mathbf{0}$ and only b remains unfixed in the resultant layout. Hence, performing task $a \rightarrow s$ does not result in a dead-end state.

4. If $\sum_{i \in \mathbb{S}} f(i) \leq N-3$, at least 3 containers remain unfixed. If the next task is to fix a container to a stack s with $f(s) = H-1$, the resultant layout might be a dead-end state. Taking Figure 6 as an example. Boxes with a solid square represent fixed containers. The task $1 \rightarrow 3$ is valid since the surplus vector of the resultant state is non-negative, however, the resultant state is a dead-end state.

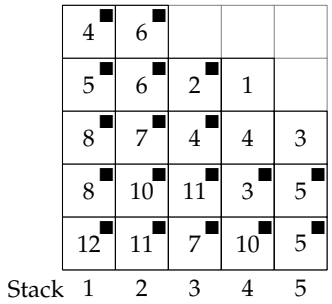


Figure 6: Pre-extreme state.

Based on the above discussion, we can see that when $\sum_{i\in\mathbb{S}} f(i) < N-2$, there is danger to enter a dead-end state.

When the current state is a pre-extreme state, we immediately eliminate those container–stack pairs $c \to s$ with $f(s) = H - 1$ from $\mathbb{T}$, so that the algorithm is able to avoid dead-end states. This technique is called "dead-end avoidance".

5.2.2. Task evaluation

Every valid task $c \to s$ in $\mathbb{T}$ is evaluated by a tuple with six elements:

1. the tier-protection indicator;
2. the number of movements required by the speedy task accomplishment procedure;
3. the number of stable containers that need to be moved from the aim stack;
4. the affected demand;
5. the fixed height of aim stack;
6. the negative of $g(c)$;

The tuples of valid tasks are compared lexicographically, and the task with the minimum tuple is selected as the next task. The six elements in the tuple will be explained in the following.

The tier-protection indicator is to balance the trade-off between the freedom of task selection and the surplus loss. In principle, a container can be moved to any valid stack. However, if a container with small group value occupies a relatively empty stack, then the loss of surplus vector is large: 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 $\varphi \in (g(c), g(s, f(s))]$. The larger the gap between $g(c)$ and $g(s, f(s))$ is, the more the surplus vector will lose. If a task $c \to s$ satisfies $f(s) \le P1$ and the affected demand $\sum_{\varphi=g(c)+1}^{g(s,f(s))} d(\varphi) \ge P2$, the task is then deleted from $\mathbb{T}$. As two customizable parameters, the number of tiers protected $P1$ and the threshold value $P2$ can be adjusted optionally.

The number of moves required by the speedy task accomplishment procedure can be precisely computed in the proposed speedy task accomplishment procedure, which is an improvement of

the estimation in Wang et al. [22]. The process to compute the number of needed movements is introduced in Section 5.3.

The number of stable containers that need to be moved from the aim stack is $sh(s) - f(s)$. The affected demand is $\sum_{\varphi=g(c)+1}^{g(s,f(s))} d(\varphi)$. The fixed height of aim stack $f(s)$ favors low tiers, which makes the fixed height of stacks even. The last element $-g(c)$ prefers a target container with a larger group value.

5.3. Speedy task accomplishment procedure

After a task is selected, a speedy task accomplishment procedure (STAP) is carried out to accomplish the task, resulting in a new state. The movement sequence of the STAP is based on task types.

Let c^* denote the target container located in (s^+, t^+) and s^* denote the aim stack. The task is ***immediate*** if c^* is already on the highest fixed container of stack s^*, i.e., $s^+ = s^*$ and $t^+ = f(s^*)+1$; the task is ***internal*** if c^* is above but not directly on the highest fixed container of stack s^*, i.e., $s^+ = s^*$ and $t^+ > f(s^*) + 1$; and the task is ***external*** if c^* is not in stack s^* currently, i.e., $s^+ \neq s^*$.

5.3.1. Immediate task

An immediate task does not require any move because the target container is already located in the aim slot.

5.3.2. Internal task

For an internal task, let a denote the number of empty slots except those in stack s^+ and one highest non-full stack; that is, $a = E - e(s^+) - \min\{e(s) : e(s) > 0, s \neq s^+\}$. Let B_1 ($b_1 = |B_1|$) and B_2 ($b_2 = |B_2|$) denote the sets (numbers) of blocking containers above and below c^*, respectively. Blocking containers refer to containers which blocks the target container or the aim slot. The accomplishment procedure for an internal task is given in Algorithm 2.

The accomplishment procedure is carried out based on the relationship between a and b_2, which determines how containers are relocated.

- I1: $a \geq b_2$;

Algorithm 2: Accomplish an internal task.

target container: $c^*(s^+, t^+)$
aim slot: $(s^+, f(s^+)+1)$
slot supply: $a = E - e(s^+) - \min\{e(s) : e(s) > 0, s \neq s^+\}$
blocking above: $b_1 = h(s^+) - t^+$
blocking below: $b_2 = t^+ - f(s^+) - 1$

case I1: $a \geq b_2$
  repeat b_1 **times do**
    $\mathbb{R} \coloneqq \{s \neq s^+ : h(s) < H,\ \alpha(s^+, s) \geq b_2\}$;
    `Relocate`$(s^+, 1, \mathbb{R})$;
  $\mathbb{I} \coloneqq \{s \neq s^+ : h(s) < H,\ E - e(s^+) - e(s) \geq b_2\}$;
  $s^{\text{tmp}} \coloneqq$ `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:` $h(s^+) - f(s^+) + 1$ `movements`

case I2: $a < b_2$ & $|\{s \neq s^+ : h(s) < H\}| > 1$
  `Relocate`$(s^+, b_1, \mathbb{S} \setminus \{s^+\})$;
  $s_1^{\text{tmp}} \coloneqq$ `Interim`$(\mathbb{S} \setminus \{s^+\})$;
  `Move`$(s^+, 1, s_1^{\text{tmp}})$;
  $k_2 \coloneqq E - e(s^+) - e(s_1^{\text{tmp}}) - 1$;
  `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:` $h(s^+) - f(s^+) + 2$ `movements`

case I3: $a < b_2$ & $|\{s \neq s^+ : h(s) < H\}| = 1$
  Find s' s.t. $s' \neq s^+$ & $h(s') < H$;
  $s^{\text{tmp}} \coloneqq$ `InterimFull`$(\mathbb{S} \setminus \{s^+, s'\})$;
  `BiSender`$(s^{\text{tmp}}, 1, s^+, b_1, \{s'\})$;
  `Move`$(s^+, 1, s^{\text{tmp}})$;
  `Move`(s^+, b_2, s');
  `Move`$(s^{\text{tmp}}, 1, s^+)$;
  // `I3:` $h(s^+) - f(s^+) + 2$ `movements`

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

In case I1, b_1 containers above c^* are relocated one by one. $\mathbb{R}$ represents the set of available destination stacks for the top container of stack s^+. When relocating containers from B_1, reserving slots for containers from B_2 should be considered beforehand. Because c^* is above B_2, so c^* is relocated before containers of B_2. If some movements put c^* and B_2 in the same stack, then c^* needs additional movements to be retrieved.

When deciding whether a stack s^{dst} can be the destination of a container c from B_1, function α is used to compute how many slots will be left for B_2 after c is placed to stack s^{dst} . If the left slots are not enough for B_2, c cannot be moved to s^{dst}. The pseudo-code of function α is given in Algorithm 3.

If $e(s^{\text{dst}}) > e^{\min}$, that is, stack s^{dst} is not the highest non-full stack, $e^{\min}$ is reserved for c^*. If c is moved to stack s^{dst}, the number of left slots for B_2 is $E - e(s^+) - e^{\min} - 1$.

If $e(s^{\text{dst}}) = e^{\min} \geq 2$, that is, stack s^{dst} is the highest non-full stack and it has at least two empty

Algorithm 3: Function α.

```
1  function α(s⁺, s^dst)
       // s⁺ ≠ s^dst
2      e^min := min{e(s) > 0 : s ≠ s⁺};
3      e^sec := min{e(s) > e^min : s ≠ s⁺};
4      k^min := |{s ≠ s⁺ : e(s) = e^min}|;
5      if e(s^dst) > e^min then
6          return E − e(s⁺) − e^min − 1;
7      else if e^min ≥ 2 then
8          return E − e(s⁺) − e^min;
9      else if k^min = 1 then
10         return E − e(s⁺) − e^sec − 1;
11     else
12         return E − e(s⁺) − 2;
```

slots. If c is moved to stack s^{dst}, the stack can also accommodate c^*, hence, the number of left slots for B_2 is $E - e(s^+) - e^{\min}$.

If $e(s^{\text{dst}}) = e^{\min} = 1$ and $k^{\min} = 1$, that is, stack s^{dst} is the unique highest stack with only one empty slot. If c is moved to stack s^{dst}, the stack with the second highest height will be reserved for c^* in the next round. The number of slots left for B_2 becomes $E - e(s^+) - e^{\text{sec}} - 1$.

If $e(s^{\text{dst}}) = e^{\min} = 1$ and $k^{\min} \geq 2$, at least one stack with only one empty slot can accommodate c^* in the future. If c is moved to stack s^{dst}, the number of slots left for B_2 becomes $E - e(s^+) - 2$.

Function `Relocate(`$s^{\text{src}}, k, \mathbb{R}$`)` is illustrated in Algorithm 4, which relocates k containers from a sender stack s^{src} to $\mathbb{R}$. For each of the top k containers of the sender, the destination stack s^{dst} is properly selected from $\mathbb{R}$ according to the evaluation by function `EvalMove(`s^{src}, s`)` for $s \in \mathbb{R}$. Function `Move(`$s^{\text{src}}, k, s^{\text{dst}}$`)` simply moves the top k containers from s^{src} to s^{dst}.

Algorithm 4: Relocate containers.

```
1  function Relocate(s^src, k, ℝ)
2      repeat k times do
3          ℝ′ := {s ∈ ℝ : h(s) < H};
4          s^dst := arg min_{s∈ℝ′} EvalMove(s^src, s);
5          Move(s^src, 1, s^dst);
```

`EvalMove(`s^{src}, s`)` returns a tuple, the first element indicates the type of the penalty and the second element indicates the scale of the penalty of moving from stack s^{src} to stack s (refer to

Algorithm 5).

Algorithm 5: Evaluate movements.

```
function EvalMove(s^src, s)
    c := the top container of stack s^src;
    case sh(s) = h(s) & s can stabilize c
        return ⟨1, Σ_{φ=g(c)+1}^{q(s,h(s))} d(φ)⟩;
    case sh(s) < h(s) & g(c) ≥ m(s)
        return ⟨2, g(c) − m(s)⟩;
    case sh(s) < h(s) & g(c) < m(s)
        return ⟨3, m(s) − g(c)⟩;
    case sh(s) = h(s) & s cannot stabilize c
        return ⟨4, q(s, h(s))⟩;
```

Let us define the capability of an occupied slot (s, t) by $q(s, t) = g(s, t)$ if the container inside is orderly, otherwise $q(s, t) = 0$; specifically, the ground is regarded as an occupied slot at tier 0 with group value G. Define the ***messiness*** of stack s by $m(s) = \max_{sh(s)<t\leq h(s)} g(s, t)$, that is, the largest group value among the unstable containers in stack s.

As $s^{\text{dst}} = \arg\min_{s\in\mathbb{R}'} \texttt{EvalMove}(s^{\text{src}}, s)$, the preference of selecting s^{dst} for a blocking container c can be summarized as follows.

1. If stack s is entirely stable and can stabilize c, the stack with the minimum affected demand is preferred;

2. If stack s is not entirely stable and $g(c) \geq m(s)$, the stack with the minimum gap between $m(s)$ and $g(c)$ is preferred;

3. If stack s is not entirely stable and $g(c) < m(s)$, the stack with the minimum gap between $g(c)$ and $m(s)$ is preferred;

4. If stack s is entirely stable but cannot stabilize c, the stack with the minimum $q(s, h(s))$ 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. Larger messiness implies a higher urgency of reshuffles. 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 by function `Interim(𝕀)` 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 group value of orderly containers. The most unattractive stack for receiving blocking containers is selected as the interim stack. In cases I3 and E4 (in Section 5.3.3), the interim stack is selected by the minimum group value of the top containers of full stacks, which is function `InterimFull(𝔽)`.

Algorithm 6: Interim and InterimFull functions.

```
1  function Interim(𝕀)
2      𝕀₁ := {s ∈ 𝕀 : sh(s) < h(s) < H};
3      𝕀₂ := {s ∈ 𝕀 : sh(s) = h(s) < H};
4      if 𝕀₁ ≠ ∅ then
5          return arg max_{s∈𝕀₁} m(s);
6      else
7          return arg min_{s∈𝕀₂} q(s, h(s));
8  function InterimFull(𝔽)
9      𝔽₁ := {s ∈ 𝔽 : f(s) ≤ sh(s) < h(s) = H};
10     𝔽₂ := {s ∈ 𝔽 : f(s) < sh(s) = h(s) = H};
11     if 𝔽₁ ≠ ∅ then
12         return arg min_{s∈𝔽₁} g(s, H);
13     else
14         return arg min_{s∈𝔽₂} g(s, H);
```

Function `BiSender(`$s_1^{\text{src}}, k_1, s_2^{\text{src}}, k_2, \mathbb{R}$`)` performs relocations from two senders s_1^{src} and s_2^{src} to a receiver set $\mathbb{R}$, and the respective relocation quantities are k_1 and k_2 (Algorithm 7). The two top containers from two senders are compared and the one with a smaller tuple is moved first. In Algorithm 7, $\vec{v}_1$ and $\vec{v}_2$ are tuple results of function `EvalMove`.

Likewise, function `BiReceiver(`$s^{\text{src}}, k_1, \mathbb{R}_1, k_2, \mathbb{R}_2$`)` in the external case relocates k_1 and k_2 containers from one sender s^{src} to two receiver sets $\mathbb{R}_1$ and $\mathbb{R}_2$, respectively.

The total number of movements needed in case I1 is $h(s^+) - f(s^+) + 1$, and the numbers of movements in case I2 and case I3 are $h(s^+) - f(s^+) + 2$. All are noted in the comments of Algorithm 2.

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 in $\mathbb{S} \setminus \{s^+, s_1^{\text{tmp}}\}$. This can be modified so that c^* is 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 containers in B_2. 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.

Algorithm 7: BiSender and BiReceiver functions.

```
1  function BiSender(s_1^src, k_1, s_2^src, k_2, ℝ)
2      while k_1 + k_2 > 0 do
3          if k_1 = 0 then
4              Relocate(s_2^src, k_2, ℝ);
5              k_2 := 0;
6          else if k_2 = 0 then
7              Relocate(s_1^src, k_1, ℝ);
8              k_1 := 0;
9          else
10             ℝ' := {s ∈ ℝ : h(s) < H};
11             s_1^dst := arg min_{s∈ℝ'} EvalMove(s_1^src, s);
12             v⃗_1 := EvalMove(s_1^src, s_1^dst);
13             s_2^dst := arg min_{s∈ℝ'} EvalMove(s_2^src, s);
14             v⃗_2 := EvalMove(s_2^src, s_2^dst);
15             if v⃗_1 ≤^lex v⃗_2 then
16                 Move(s_1^src, 1, s_1^dst);
17                 k_1 := k_1 − 1;
18             else
19                 Move(s_2^src, 1, s_2^dst);
20                 k_2 := k_2 − 1;

21 function BiReceiver(s^src, k_1, ℝ_1, k_2, ℝ_2)
22     while k_1 + k_2 > 0 do
23         if k_1 = 0 then
24             Relocate(s^src, k_2, ℝ_2);
25             k_2 := 0;
26         else if k_2 = 0 then
27             Relocate(s^src, k_1, ℝ_1);
28             k_1 := 0;
29         else
30             ℝ'_1 := {s ∈ ℝ_1 : h(s) < H};
31             ℝ'_2 := {s ∈ ℝ_2 : h(s) < H};
32             s_1^dst := arg min_{s∈ℝ'_1} EvalMove(s^src, s);
33             v⃗_1 := EvalMove(s^src, s_1^dst);
34             s_2^dst := arg min_{s∈ℝ'_2} EvalMove(s^src, s);
35             v⃗_2 := EvalMove(s^src, s_2^dst);
36             if v⃗_1 ≤^lex v⃗_2 then
37                 Move(s^src, 1, s_1^dst);
38                 k_1 := k_1 − 1;
39             else
40                 Move(s^src, 1, s_2^dst);
41                 k_2 := k_2 − 1;
```

5.3.3. 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 ($b_1 = |B_1|$) and B_2 ($b_2 = |B_2|$) denote the sets (numbers) of blocking containers above the target container c^* and the aim slot $(s^*, f(s^*))$, respectively. The pseudo-code describing the accomplishment procedure for an external task is given in Algorithm 8. Four situations are considered.

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

Algorithm 8: Accomplish an external task.

target container: $c^*(s^+, t^+)$
aim slot: $(s^*, f(s^*)+1)$
blocking above target container: $b_1 = h(s^+) - t^+$
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:` $b_1 + b_2 + 1$ `movements`

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:` $b_1 + b_2 + 2$ `movements`

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:` $2b_1 + b_2 - a + 3$ `movements`

case E4: $a = 0$
    $s^{\text{tmp}} \coloneqq$ `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:` $2b_1 + b_2 + 4$ `movements`

6. Computational results

In this section, the proposed FBH is compared to four benchmark approaches. Three of them are heuristics. They are: the target-guided heuristic (TGH) proposed in our previous work [22], an implementation of the largest priority first heuristic (LPFH) proposed by Expósito-Izquierdo et al. [7], and an implementation of the multi-heuristic proposed by Jovanovic et al. [12]. The last benchmark approach is called BS-B, that is a beam search framework consolidating TGH as a component [22]. By far, BS-B has the best performance on the CPMP. It is necessary to evaluate the quality level of FBH with the best known values.

Our algorithm is tested on the CVS data set and the BF data set, which are initially presented by Caserta et al. [5] and Bortfeldt & Forster [2], respectively. Since LPFH and the multi-heuristic approach do not report results on the BF data set, we reimplemented LPFH and the multi-heuristic approach to compute results of the CVS and BF.

The original LPFH involves randomness. We developed a deterministic version instead, referred to as the largest group value first heuristic (LGVFH). The LGVFH selects the next target container from unfixed ones with the largest group value, and then accomplish it by the STAP. The pseudo-code for the LGVFH is given in Algorithm 9.

Algorithm 9: Largest group value first heuristic.

begin
  $(\mathsf{L}, \boldsymbol{f}) \coloneqq (\mathsf{L}^0, U \cdot \mathbf{1})$;
  foreach $g = G, \ldots, 1$ **do**
    $\mathbb{C}_g \coloneqq$ containers with group value g;
    $\mathbb{A}_g \coloneqq \varnothing$; // `set of aim stacks`
    while $\mathbb{C}_g \neq \varnothing$ **do**
      $(c^*, s^*) \coloneqq \arg\min_{c \in \mathbb{C}_g, f(s) < H}$ `MoveNeed(`c, s`)`;
      $\mathsf{L} \coloneqq$ resultant layout after rearranging c^* to s^*;
      $f(s^*) \coloneqq f(s^*) + 1$;
      $\mathbb{C}_g \coloneqq \mathbb{C}_g \setminus \{c^*\}$;
      $\mathbb{A}_g \coloneqq \mathbb{A}_g \cup \{s^*\}$;
    foreach $s \in \mathbb{A}_g$ **do**
      `Fill(`s`)`;
function `Fill(`s`)`
  while $h(s) < H$ **do**
    $\mathbb{S}' \coloneqq \varnothing$;
    foreach $i \in \{i \neq s : sh(i) < h(i)\}$ **do**
      $c \coloneqq$ the top container of stack i;
      if s can stabilize c **then**
        $\mathbb{S}' \coloneqq \mathbb{S}' \cup \{i\}$;
    if $\mathbb{S}' \neq \varnothing$ **then**
      $s' \coloneqq \arg\max_{i \in \mathbb{S}'} g(i, h(i))$;
      `Move(`$s', 1, s$`)`;
    else
      break while;

Here, function `MoveNeed`(c, s) is the actual number of moves needed by the STAP to accomplish the task $c \to s$. Function `Fill`(s) is achieved by fulfilling stack s with unstable containers in other stacks which can be stabilized by s.

Experiments were conducted on a computer with Intel Core i7 CPU clocked at 3.40 GHz and Windows 7 operating system. All the algorithms were written in Java.

6.1. Configuration of P1 and P2

This section shows the effect of the tier-protection indicator. We tested six settings of parameters $(P1, P2)$: $(0, N)$, $(\frac{1}{4}H, \frac{1}{2}N)$, $(\frac{1}{3}H, \frac{1}{4}N)$, $(\frac{1}{2}H, \frac{1}{6}N)$, $(\frac{2}{3}H, \frac{1}{8}N)$, and $(H, \frac{1}{10}N)$. We made $P1$ and $P2$ proportional to H and N, respectively, because $f(s)$ and the affected demand are related to the maximum height and the number of containers, respectively.

The six settings are in a loose-intermediate-tight order; algorithms using latter settings will filter out more tasks when selecting the next task. The computational results are shown in Table 2. The first column indicates the particular values of $(P1, P2)$. The second and third columns give the average movements of the CVS and BF data sets, respectively. Parameter $(0, N)$ means that there is no tier-protection indicator so no task is filtered out. As the setting becomes tighter, the performance on both CVS and BF is improved, until a peak is achieved. When the setting becomes extremely tight, the performance decreases. When the setting is $(H, \frac{1}{10}N)$, the algorithm even fails to find solutions to some CVS instances, so that we mark the corresponding cell with "Nil". Because the BF instances are relatively loose (the empty slots are sufficient), solutions can still be found under the last setting, but the performance decreases dramatically.

6.2. Results of the CVS instances

Caserta et al. [5] present the complete CVS data set (named after the authors' surnames, Caserta, Voß and Sniedovich) originally for the container relocation problem. The CVS instances are classified into 21 groups, each consisting of 40 instances. For any instance, the heights of piled containers in each stack are the same, denoted by K, hence $N = SK$. It is worth noting that the height limitation of each stack is not specified in the original data. Researchers add two extra tiers above the initial layout in order to make the data suit for the CPMP; that is, $H = K + 2$.

Table 2: Effects of different $(P1, P2)$.

$(P1, P2)$	CVS	BF
$(0, N)$	52.15	71.69
$(\frac{1}{4}H, \frac{1}{2}N)$	50.43	71.21
$(\frac{1}{3}H, \frac{1}{4}N)$	45.82	69.05
$(\frac{1}{2}H, \frac{1}{6}N)$	45.32	69.51
$(\frac{2}{3}H, \frac{1}{8}N)$	46.33	71.23
$(H, \frac{1}{10}N)$	Nil	73.44

The CVS instances can be considered as typical dense CPMP instances. Table 3 illustrates the computational results of the CVS instances by the TGH, the LGVFH, the multi-heuristic approach, the FBH, and the BS-B. The values under the "moves" heading represent the average numbers of moves for every CVS group, whereas the values under the "time (ms)" heading are the average runtime in millisecond. The values under the "improvement" heading are the improvements of the FBH compared to the TGH, the LGVFH, and the multi-heuristic in percentage. The results showcase that the FBH surpasses the TGH and the LGVFH. The improvement gets larger as N increases. Compared to the multi-heuristic approach, generally speaking, the FBH is better when N is large while the multi-heuristic is better when N is small.

Table 3: Results of the CVS instances.

CVS K-S	TGH		LGVFH		multi		FBH		BS-B		improvement		
	moves	time (ms)	moves	time (ms)	moves	time (ms)	moves	time (ms)	moves	time (ms)	TGH	LGVFH	multi
CVS 3-3	12.95	0.43	11.25	0.45	10.23	0.50	11.28	0.53	9.35	7.05	12.93%	−0.22%	−10.27%
CVS 3-4	12.18	0.20	12.23	0.15	10.98	0.50	10.80	0.18	9.45	7.83	11.29%	11.66%	1.59%
CVS 3-5	12.78	0.10	13.45	0.20	12.23	0.75	12.08	0.18	10.45	12.55	5.48%	10.22%	1.23%
CVS 3-6	14.38	0.18	14.88	0.33	13.98	0.50	12.98	0.23	11.58	19.18	9.74%	12.77%	7.16%
CVS 3-7	16.00	0.13	16.58	0.23	15.83	0.50	14.75	0.15	13.13	31.63	7.81%	11.01%	6.79%
CVS 3-8	16.55	0.15	17.08	0.28	16.93	1.25	15.65	0.38	13.90	46.88	5.44%	8.35%	7.53%
CVS 4-4	23.35	0.05	21.93	0.08	20.38	1.53	21.88	0.15	16.98	61.35	6.32%	0.23%	−7.36%
CVS 4-5	26.73	0.10	26.48	0.18	23.78	0.75	23.08	0.18	18.90	112.08	13.66%	12.84%	2.94%
CVS 4-6	27.58	0.08	27.20	0.23	25.90	0.75	24.75	0.48	20.25	191.50	10.24%	9.01%	4.44%
CVS 4-7	29.93	0.08	31.23	0.38	29.63	0.75	27.63	0.10	23.15	331.18	7.69%	11.53%	6.75%
CVS 5-4	44.83	0.08	35.83	0.18	31.78	2.25	35.08	0.08	26.43	221.13	21.75%	2.09%	−10.39%
CVS 5-5	42.40	0.05	36.50	0.25	33.63	2.50	35.33	0.23	27.38	394.60	16.69%	3.22%	−5.06%
CVS 5-6	50.63	0.13	43.08	0.23	40.73	2.75	39.88	0.23	32.08	996.90	21.24%	7.43%	2.09%
CVS 5-7	48.83	0.13	46.95	0.73	44.35	3.50	41.68	0.28	34.20	1507.13	14.64%	11.24%	6.03%
CVS 5-8	56.68	0.20	51.83	0.63	50.48	6.00	47.50	0.48	38.63	3030.35	16.19%	8.35%	5.89%
CVS 5-9	57.50	0.08	55.65	1.08	55.28	7.00	50.45	0.90	42.15	4615.30	12.26%	9.34%	8.73%
CVS 5-10	62.80	0.53	60.88	1.30	59.60	7.00	54.63	0.80	44.85	7455.93	13.02%	10.27%	8.35%
CVS 6-6	74.33	0.13	57.85	0.28	54.00	4.25	55.23	0.50	43.80	2511.33	25.70%	4.54%	−2.27%
CVS 6-10	88.63	0.43	79.73	1.88	79.58	14.75	75.60	1.65	60.55	23423.80	14.70%	5.17%	5.00%
CVS 10-6	332.25	0.30	173.95	1.08	150.28	18.28	140.63	0.85	116.18	23010.70	57.57%	19.16%	6.42%
CVS 10-10	302.90	0.90	190.50	5.05	181.58	59.78	179.23	3.65	150.18	287920.50	40.83%	5.92%	1.29%
Average	64.48	0.21	48.81	0.72	45.77	6.47	44.29	0.58	36.36	16948.04	16.44%	8.29%	2.23%

6.3. Results of the BF instances

Bortfeldt & Forster [2] introduce 32 groups of CPMP instances (referred to as the BF instances); each group consists of 20 instances. In the 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 groups G 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 as typical loose CPMP instances. Table 4 illustrates the computational results of the BF instances by the five approaches. The values under the "moves" heading represent the average numbers of moves of every BF group solved by different algorithms, whereas the values under the "time (ms)" heading are the average runtime in millisecond. The values under the "improvement" heading are the improvements of the FBH compared to the TGH, the LGVFH and the multi-heuristic approach in percentage.

Table 4: Results of the BF instances.

BF	S	H	N	G	B	TGH		LGVFH		multi		FBH		BS-B		improvement		
						moves	time (ms)	moves	time (ms)	moves	time (ms)	moves	time (ms)	moves	time (ms)	TGH	LGVFH	multi
1	16	5	48	10	29	29.10	2.85	29.55	5.05	30.20	6.00	29.15	4.80	29.10	0.80	−0.17%	1.35%	3.48%
2	16	5	48	10	36	36.00	1.35	36.60	2.90	37.10	8.50	36.00	3.65	36.00	< 0.01	0.00%	1.64%	2.96%
3	16	5	48	20	29	29.45	0.85	30.90	2.65	31.20	4.50	29.35	3.30	29.10	23.45	0.34%	5.02%	5.93%
4	16	5	48	20	36	36.00	0.55	37.20	1.55	37.50	8.55	36.15	1.60	36.00	< 0.01	−0.42%	2.82%	3.60%
5	16	5	64	13	39	48.50	1.55	53.00	2.90	47.95	9.50	46.30	2.65	41.35	4341.50	4.54%	12.64%	3.44%
6	16	5	64	13	48	57.55	0.95	62.85	3.45	56.40	13.50	55.50	2.65	50.15	12658.20	3.56%	11.69%	1.60%
7	16	5	64	26	39	53.55	0.85	57.15	3.10	50.95	11.00	49.95	2.25	43.05	10554.05	6.72%	12.60%	1.96%
8	16	5	64	26	48	60.00	0.80	66.90	2.90	58.55	18.00	57.60	2.40	51.15	18552.70	4.00%	13.90%	1.62%
9	16	8	77	16	47	60.35	1.55	62.15	4.35	59.55	18.00	56.30	3.90	50.40	7752.60	6.71%	9.41%	5.46%
10	16	8	77	16	58	62.15	7.90	69.50	4.65	65.55	21.50	61.55	3.85	58.75	3353.75	0.97%	11.44%	6.10%
11	16	8	77	31	47	61.25	1.15	63.70	4.20	61.50	15.50	55.00	3.35	51.15	8739.95	10.20%	13.66%	10.57%
12	16	8	77	31	58	63.45	1.10	68.50	4.15	68.15	22.00	61.45	3.30	58.65	3884.90	3.15%	10.29%	9.83%
13	16	8	103	21	62	107.45	1.75	110.85	6.45	90.95	34.00	96.50	5.55	75.40	204259.90	10.19%	12.95%	−6.10%
14	16	8	103	21	78	124.75	2.55	134.90	5.95	106.80	44.00	116.05	4.75	93.10	3578554.40	6.97%	13.97%	−8.66%
15	16	8	103	42	62	110.60	2.90	110.40	9.45	96.30	36.00	99.45	6.50	78.70	212239.40	10.08%	9.92%	−3.27%
16	16	8	103	42	78	133.35	1.55	137.40	7.40	112.10	51.50	115.45	5.85	93.55	331990.90	13.42%	15.98%	−2.99%
17	20	5	60	12	36	36.50	0.90	37.35	2.30	39.00	9.50	36.50	2.15	36.25	9.40	0.00%	2.28%	6.41%
18	20	5	60	12	45	45.00	0.80	45.00	2.15	46.35	15.00	45.20	1.90	45.00	< 0.01	−0.44%	−0.44%	2.48%
19	20	5	60	24	36	36.80	0.75	38.50	2.90	39.90	7.50	36.75	1.95	36.45	5.45	0.14%	4.55%	7.89%
20	20	5	60	24	45	45.00	2.00	45.70	3.00	47.00	13.10	45.10	2.80	45.00	0.75	−0.22%	1.31%	4.04%
21	20	5	80	16	48	61.65	1.65	65.65	4.10	60.35	18.60	56.55	3.25	51.55	24430.10	8.27%	13.86%	6.30%
22	20	5	80	16	60	67.90	1.30	74.50	4.10	70.50	25.55	65.55	3.25	61.80	16871.30	3.46%	12.01%	7.02%
23	20	5	80	32	48	61.10	1.40	65.75	4.75	61.65	18.00	55.25	4.05	50.95	16706.40	9.57%	15.97%	10.38%
24	20	5	80	32	60	70.95	1.95	76.65	5.80	72.65	27.00	68.00	4.25	62.05	28704.25	4.16%	11.29%	6.40%
25	20	8	96	20	58	69.80	3.90	73.60	8.60	73.30	28.00	66.00	6.75	61.50	14390.80	5.44%	10.33%	9.96%
26	20	8	96	20	72	74.35	3.60	81.75	7.55	81.10	36.50	75.75	7.15	72.35	2737.45	−1.88%	7.34%	6.60%
27	20	8	96	39	58	71.85	2.95	73.65	8.50	73.35	26.00	65.65	6.70	61.85	18240.95	8.63%	10.86%	10.50%
28	20	8	96	39	72	76.30	2.10	83.55	8.35	83.55	38.50	76.50	6.25	72.65	6564.95	−0.26%	8.44%	8.44%
29	20	8	128	26	77	118.65	3.60	128.65	11.15	113.20	59.00	115.85	8.80	92.05	477551.30	2.36%	9.95%	−2.34%
30	20	8	128	26	96	143.05	3.60	155.15	11.05	128.75	88.50	129.60	9.75	110.25	713773.30	9.40%	16.47%	−0.66%
31	20	8	128	52	77	128.15	3.30	128.80	15.15	116.90	54.50	115.85	10.85	93.95	529949.50	9.60%	10.05%	0.90%
32	20	8	128	52	96	147.30	2.70	157.00	13.60	133.20	89.00	134.10	10.45	111.80	778493.80	8.96%	14.59%	−0.68%
Average						72.75	2.08	76.96	5.75	70.36	27.38	68.44	4.71	60.66	118894.88	5.92%	11.08%	3.72%

Table 5 shows the computational results based on the dimensions of instance density and height limitation. It shows that the instance density N/SH (or bay utilization) and the height of the bay H are key factors for the number of moves needed in pre-marshalling. In other words, denser or higher instances are more difficult to solve. The computational results of the BF instances also prove that the performance of the FBH is considerable; in most instances, the FBH performs the best.

Table 5: Summary on the BF instances.

	TGH		LGVFH		multi		FBH		BS-B	
density	$H = 5$	8	5	8	5	8	5	8	5	8
0.6	36.73	67.44	37.60	72.05	38.53	70.76	36.78	64.78	36.61	60.91
0.8	60.15	126.66	65.31	132.89	59.88	112.28	56.84	115.36	51.51	93.60

7. Conclusions

The CPMP deals with how to rehandle containers in a bay so that the containers are placed in a pre-determined order. By far, works talking about solutions to this problem are few. In this paper, we present a feasibility-based heuristic to solve the CPMP. The proposed heuristic can be implemented solely or combined with other frameworks. The main innovation of this paper is the concept of state feasibility, which checks the feasibility of states before really searching these states. Due to this concept, the algorithm is able to search larger solution space compared with the existing methods. In the meanwhile, the search efficiency is guaranteed. Other techniques such as the container stability, dead-end avoidance and tier-protection indicator are also proposed. Numerical experiments on two commonly used data sets show that the proposed method performs well.

Our algorithm can be inserted into meta-heuristic frameworks such as beam search, the LNS, and the GRASP. How these combinations work and to what extent they work are worthy of study in the future. A major challenge raised in the feasibility-based heuristic is the trade-off between the freedom of selecting target containers and the waste of available slots. In this paper, the low-tier protection technique is proposed to balance the trade-off. In future works, more efforts should be dedicated to balance the trade-off better.

In this paper, we assume that the containers in a bay are deterministic and no container comes in or goes out during the pre-marshalling process. In the future work, it is interesting to discuss how to handle dynamic yards in which containers come in or go out from time to time. To some extent, the new problem setting is similar to the container stacking problem. The relationship of the CPMP and the container stacking problem is rather close, so the methods proposed for the CPMP may provide a new perspective for the container stacking problem.

Acknowledgments

This research was partially supported by the National Natural Science Foundation of China (No. 71371114, 71301102), Guangdong Natural Science Funds (No. 2014A030310312), and Fundamental Research Funds for the Central Universities (No. 15LGPY37). .

References

[1] Bierwirth, C. & Meisel, F. (2015). A follow-up survey of berth allocation and quay crane scheduling problems in container terminals. *European Journal of Operational Research*, 244(3), 675 – 689.

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

[3] 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.

[4] 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.

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

[6] Dayama, N. R., Krishnamoorthy, M., Ernst, A., Narayanan, V., & Rangaraj, N. (2014). Approaches for solving the container stacking problem with route distance minimization and stack rearrangement considerations. *Computers & Operations Research*, 52, Part A, 68 – 83.

[7] 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.

[8] Gheith, M., Eltawil, A., & Harraz, N. (2014). A rule-based heuristic procedure for the container pre-marshalling problem. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)* (pp. 662–666).

[9] Gheith, M., Eltawil, A., & Harraz, N. (2015). Solving the container pre-marshalling problem using variable length genetic algorithms. *Engineering Optimization*, 47, 1–19.

[10] Huang, S.-H. & Lin, T.-H. (2012). Heuristic algorithms for container pre-marshalling problems. *Computers & Industrial Engineering*, 62(1), 13 – 20.

[11] 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.

[12] Jovanovic, R., Tuba, M., & Voß, S. (2014). A multi-heuristic approach for solving the pre-marshalling problem. *Central European Journal of Operations Research*, (pp. 1–28).

[13] Jovanovic, R. & Voß, S. (2014). A chain heuristic for the blocks relocation problem. *Computers & Industrial Engineering*, 75, 79 – 86.

[14] 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.

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

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

[17] 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.

[18] Rendl, A. & Prandtstetter, M. (2013). Constraint models for the container pre-marshaling problem. *ModRef*, 2013, 12th.

[19] Tierney, K., Pacino, D., & Voß, S. (2014). Solving the pre-marshalling problem to optimality with A* and IDA*. In *Conference of the International Federation of Operational Research Societies.*

[20] van Brink, M. & van der Zwaan, R. (2014). A branch and price procedure for the container premarshalling problem. In A. Schulz & D. Wagner (Eds.), *Algorithms - ESA*, volume 8737 of *Lecture Notes in Computer Science* (pp. 798–809). Springer Berlin Heidelberg.

[21] Wang, N. (2014). *Optimization Study on Container Operations in Maritime Transportation*. PhD thesis, City University of Hong Kong, Hong Kong.

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