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

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

[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*
[c]*School of Mobile Information Engineering, Sun Yat-Sen University*

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: 1) containers are rearranged to their desired slots one by one in a certain order; and 2) rearranging one container is completed by a sequence of movements. Our algorithm improves the performance from the aforementioned two components. Instead of rearranging containers in the descending order of their group labels (traditional idea), this paper determines the order as the algorithm goes on by an evaluation function. To efficiently explore the enlarged search space, we are the first one to come up with a concept of state feasibility to cut branches. In addition, we improve the sequence of movements by techniques such as pre-extreme avoidance and tier protection. Computational experiments showcase that the performance of our proposed heuristic is better than other heuristics.
Keywords: container pre-marshalling problem, feasibility-based heuristic

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. Tel: +86 18930400865
Email addresses: msjinbo@cityu.edu.hk (Bo Jin), alim.china@gmail.com (Andrew Lim), ningwang@shu.edu.cn (Ning Wang), zhangzizhen@gmail.com (Zizhen Zhang)
[1]Andrew Lim is currently on no pay leave from City University of Hong Kong.

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.

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 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.

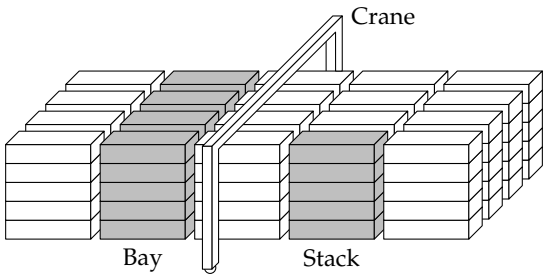


Figure 1: Yard block.

Containers in the same stack are organized in a last-in-first-out matter. To retrieve containers placed in lower levels, containers on top of them must be relocated to other places (other slots or the yard) first. Such forced movements are known as container reshuffles. Container premarshalling problem (CPMP) addresses the reorganization of containers inside a storage area such that no reshuffling is further required when containers are retrieved. Hence, CPMP is to improve terminals' performance level, including their throughput rate per berth and the turnaround time of vessels or road trucks [13].

The CPMP can be formally defined as follows: containers are initially placed in several stacks.

These stacks form a bay. Each container is marked with a group label. CPMP is to rearrange the containers within the bay, so that containers in each stack are placed in a descending order of group labels from bottom to top. The objective is to carry out the rearrangement with the fewest number of container movements. This paper investigates the CPMP, and it is assumed that the retrieval order of the containers is known beforehand and no container arrives at or leaves from the bay during pre-marshalling.

This paper makes two main contributions to the literature. One contribution is the idea behind the proposed heuristic. In the traditional algorithms, containers are allocated according to a certain order, such as the descending order of their groups [8]; in this research, the order to allocate containers is not fixed beforehand, but rather that it is decided as the algorithm goes on. This innovation undoubtedly enlarges the search space and decreases the search efficiency. To avoid efficiency decrease, we propose the concept of state feasibility to check the feasibility of allocating a certain container before we actually allocate it. The time complexity of check feasibility is only $O(G)$, here G is the number of groups. Our paper is the first work that uses feasibility to cut branches in CPMP algorithms. Another contribution is that the proposed heuristic performs outstandingly on both dense and loose instances, compared with existing heuristics. The better performance is thanks to the techniques used, such as state stability, dead-end states avoidance and tier-protection indicator which will be explained in Section 5.

The remainder of this paper is structured as follows. Section 2 reviews existing approaches in the literature. Section 3 gives a mathematical description of the problem and lists the notation used throughout this paper. The concept of state, state feasibility and stable state are introduced in Section 4 Comprehensive description of the heuristic and associate techniques are explained in Sections 5. Section 6 illustrates the results of our computational experiments, and the last section concludes this paper and identifies future research directions.

2. Recent work

According to recent reviews [3,16], problems related to or caused by container reshuffles at terminals include three major kinds of decision problems. The first problem is the container relocation problem [12] 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 addressed in this paper. The third problem is container stacking problem which decides the selection of storage locations or sequences the movements of cranes for arriving containers [6].

To the best of our knowledge, only seven works study the solutions to the pre-marshalling problem, which is rather few compared with other problems in ports, such as berth allocation problems and quay crane scheduling problems. The aforementioned two problems have been studied in more than 120 publications just since 2009 [1]. One reason behind is that the variants of pre-marshalling problem by far are few. Known variants appear in Wang et al. [18] which considers truck lanes and in Huang & Lin [10] which requires containers of different groups be separately located in the final layout. In the following, the only seven works will be discussed in sequence.

Lee & Hsu [15] 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 a small instance.

Caserta & Voß [4] provide a greedy heuristic, 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. 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 is proposed by Lee & Chao [14], 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 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 resultant new solutions are not always ensured.

Bortfeldt & Forster [2] 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. [8] provide the first group-oriented heuristic for the CPMP. Their method iteratively handles containers in a descending order of container group values. After handling all containers with a specific group value, a stack filling process is applied to reduce the number of disorderly containers in the bay.

Huang & Lin [10] solved two variants of CPMP by a heuristic algorithm. One variant is commonly seen which allows containers of different groups to mix within a bay. The other variant is newly proposed, which requires containers of different groups be separately located in the final layout.

Wang et al. [18] propose a target-guided heuristic (TGH) and two beam search algorithms. The TGH also handles containers in a descending order of the container group values. The method to determine container handle order and the way to handle a container are improved. This work can solve instances of different densities well, especially dense instances with fewer empty slots.

3. Problem description and notations

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

Each container is labeled with a group value $g \in [1, G]$. 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 which have the same meaning of "orderly/disorderly" in recent literature include "well/badly placed" [2], "well-/non-located" [8], and "clean/dirty" [18].

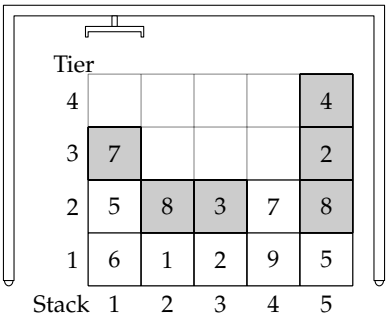


Figure 2: A container bay.

Figure 2 gives an example of a bay for which $S = 5$, $H = 4$, and $N = 13$. Containers are represented by boxes with their group values marked inside. In addition, boxes with gray background are disorderly containers. The objective of the CPMP is to find an optimized sequence with the fewest container movements, by which all containers are rearranged to be orderly. The movement sequence is called "applied to" the CPMP case.

To better describe the solution to CPMP, we represent layout-related items mathematically. The tier and stack numbers of the bay are labeled from bottom to up and left to right, 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 s. Note that the height of stacks should not exceed H. The orderly height (number of orderly containers) of s is denoted by $o(s)$.

The slot positioned at t-th tier of stack s is denoted by (s, t). The group value of a container c is denoted by $g(c)$. Likewise, 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 and not allowed to be moved in the following movement sequence.*

If a container is a fixed container, then the containers under it must be fixed containers.

Definition 2 (State)**.** *A state* $(\mathsf{L}, \boldsymbol{f})$ *is a pair composed of a layout* L *and a fix vector* $\boldsymbol{f}$.

The fix vector $\boldsymbol{f}$ indicates the number of fixed containers (fixed height) in each stack of L. $\boldsymbol{f}(s)$ is the sth element. Fixed and unfixed containers are separated by a line (a skyline), as shown in Figure 3. In Figure 3(a), $\boldsymbol{f} = (2, 3, 1)$; in Figure 3(b), $\boldsymbol{f} = (2, 3, 0)$.

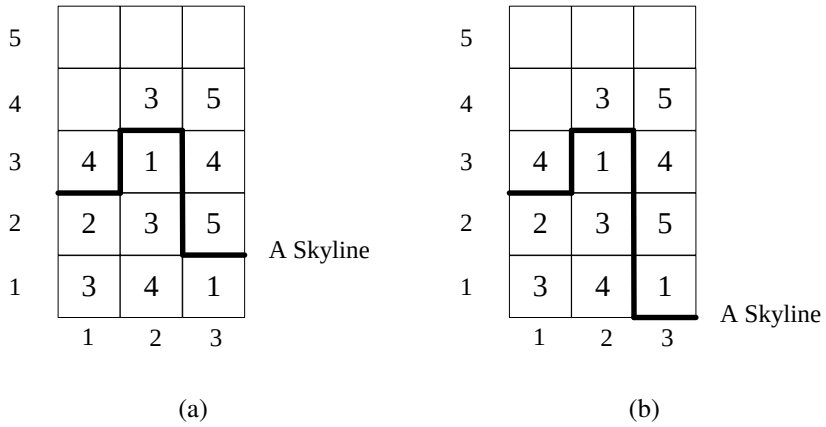


Figure 3: Fix vector and skyline

It is noteworthy that if two states have the same layout but different fix vectors, then they are two different states, just like Figure 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* f.

The necessary condition for state feasibility has been talked about in Wang [17]. 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 now are separately considered. In one stack s, suppose the smallest group label under the skyline is g, then $H - f(s)$ slots above the skyline can only be placed with containers with group label no larger than g, or the stack is disorderly. In such a situation, we say stack s has $H - f(s)$ slots with group label g. If the skyline of stack s is the ground , then s has H slots with group label G. For example, in Figure 3(b), stack 1, 2, 3 have 3, 2, 5 slots with group label 2, 1, 5, respectively. The total number of slots with group label g in the whole bay is denoted as $r(g)$ (resource of group g). Similarly, the number of unfixed containers with group label g is denoted as $d(g)$ (demand of group g). For a feasible state, the slots

available for group g must be more than the demand of group g. That is,

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

which is equivalent to

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

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

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

Take the layouts in Figure 3(a) and 3(b) as an example, and Table 1 shows the surplus vectors for both cases. Case a is infeasible as only $\Delta(1) \geq 0$; while the feasibility of Case b is not sure since the above conditions are necessary conditions. Intuitively, it can be seen that none of the three stacks in Figure 3(a) can accommodate containers with group label 5 orderly, and this is the reason why Case a is infeasible.

Table 1: Computation for the surplus vector.

Case a						Case b					
g	$d(g)$	$D(g)$	$r(g)$	$R(g)$	$\Delta(g)$	g	$d(g)$	$D(g)$	$r(g)$	$R(g)$	$\Delta(g)$
1	0	5	6	9	3	1	1	6	2	10	4
2	0	5	3	3	-2	2	0	5	3	8	3
3	1	5	0	0	-5	3	1	5	0	5	0
4	2	4	0	0	-4	4	2	4	0	5	1
5	2	2	0	0	-2	5	2	2	5	5	3

It is noteworthy that if only one container c is unfixed in a state $(\mathsf{L}, \boldsymbol{f})$, and the surplus vector $\boldsymbol{\Delta} \geq 0$, then at most one movement is need to convert the layout to a feasible layout. As $\boldsymbol{\Delta} \geq 0$, and $D[g(c)] = d[g(c)] = 1$, we have $S[g(c)] > 1$, i.e., there is at least one slot which can accommodate c orderly. We just need c to the slot.

Definition 4 (Container stability)**.** *Given a feasible state, a disorderly container is unstable. For any orderly container c in slot* (s,t)*, try to fix container c and those underneath it; if the resultant state remains feasible, c is stable. Otherwise, c is unstable.*

According to the definition, an unstable container cannot be fixed in the current slot even if it is orderly, because the resultant state will be infeasible. Unstable must to 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 is stable in stack s, we say stack s ***stabilizes*** container c.

Figure 4 gives an example of a state. Fixed containers are labeled with a solid square. Unstable containers are highlighted with gray background. Notice that the container $(2,1)$ with group value 6 is orderly but unstable.

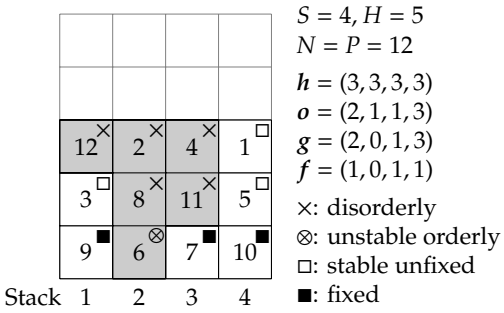


Figure 4: Stability of containers.

An orderly but unstable container indicates that its orderliness is valueless because it definitely requires at least one reshuffle. Using the concept of container stability instead of container orderliness is more precise. For example, when selecting the placement of a blocking container, if the container is orderly but unstable after moving to a stack, then the attractiveness of such a movement should be reconsidered.

Note that 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 especial case where $S - 2$ stacks are fully fixed. In such a case, unfixed containers can only be moved between two stacks, say a and b; the case is feasible if and only if one of the following conditions is satisfied:

1. Both stacks a and b are orderly;

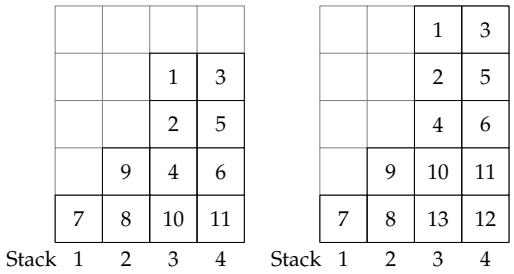
2. Stack a is orderly, and stack b is disorderly. Disorderly containers in b can be allocated to a and a is still orderly after the allocation.

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

The dead-end states are infeasible, therefore we should avoid generating them. We have the following proposition:

Proposition 2 (Existence of dead-end states)**.** *No dead-end state can be generated whatever movements and fixed vectors are applied to cases with* $E \geq 2(H-1)$*. For cases with* $E < 2(H-1)$*, there exist possibilities that the algorithm falls into a dead-end state.*

When $E \geq 2(H-1)$, it has $N \leq SH - 2H + 2$, even though all containers are fixed, there are at most $S - 2$ stacks fully occupied with fixed containers: 1) if there are fewer than S-2 stacks fully occupied, the state is not a dead-end state; 2) if there are S-2 stacks fully occupied, there are at most two extra containers in the left two stacks. The state is not a dead-end state, either. For cases with $E < 2(H-1)$, there exist possibilities that the algorithm falls into a dead-end state. As shown in Figure 5(a), it has $E \geq 2(H-1)$, and it has no chance to generate dead-end states. In Figure 5(b) with $E < 2(H-1)$, when $\boldsymbol{f} = (0, 0, 5, 5)$, the 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 have appeared in other works[8,18]: generally, 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 solely, then it is a greedy algorithm; the heuristic can also be the major component of other frameworks, such as GRASP and beam search. Target-driven algorithms are effective among all existing algorithms, and our algorithm in this work is more effective. We optimize the order of rearranging containers as well as the sequence of movements in one rearrangement.

The optimization is due to the concept of state feasibility. This is why our algorithm is called feasibility-based heuristic. Before really rearranging a certain container, the feasibility of rearrangement is checked. If the rearrangement is infeasible, then we give up this rearrangement. Due to the feasibility, we can explore larger space without sacrificing efficiency: the order to rearrange containers is not fixed beforehand, but rather that it is decided as the algorithm goes on.

5.1. Heuristic framework

According to Wang [17], in dense instances such that $E < H$, for a stack s, only the top $\sum_{i \neq s} e(i)$ containers can be moved to other stacks. The immovable number of containers (unreachable tiers) for any stack is $U = H - \sum_i e(i) = H - E$. Containers in the bottom $H - E$ tiers of the bay are unreachable, that is, no sequence can move these containers. In loose instances such that $E \geq H$, all containers in a stack are movable. Let $U = \max\{H - E, 0\}$ denotes the number of unreachable tiers for any instances.

The feasibility-based heuristic first constructs the initial state according to the unreachable tiers. In the initial state, L is the initial layout L^0 and containers in unreachable tiers are fixed, i.e., $\boldsymbol{f} = U \cdot \mathbf{1}$, where $\mathbf{1}$ is a S-dimension all-ones vector.

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

Algorithm 1: Feasibility-based heuristic.

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

The procedure of choosing a target container c^* and a target stack s^* and the procedure of moving c^* to s^* will be explained in the following section.

The advantage of fixing containers one by one lies in twofold: 1) movements are more target-oriented. Some works such as [2] move a container each time, but they don't have a clear aim. Hence, it is difficult to measure the benefit of moves and moreover, moves are blind: a container can be unnecessarily moved from stack a to b in a round and then moved from stack b to a after several rounds. 2) fixed containers are not allowed to be moved. But unfixed orderly containers are allowed to be moved. This can distinguish fixed and unfixed orderly containers. Even though both are orderly, the statuses of both kinds of containers are totally different. In works which does not distinguish fixed and unfixed containers, some containers which do not need moved (fixed containers in our context) are involved in the movements.

5.2. Valid task

At every step of the heuristic, a task determines an unfixed container (target container) and a destination stack (target stack). 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
- $\Delta(\varphi) \geq H - f(s), \varphi \in (g(c), g(s, f(s))]$.

The first condition is easy to understand. The second condition ensures that s has a slot to accommodate c. The third condition makes sure that the group label of c is no larger than that of

the highest fixed container of s, here, $g(c)$ and $g(s, f(s))$ denotes the group labels of c and highest fixed container of s, respectively. The last condition ensures the non-negativity of the resultant surplus vector. If c is moved to s, then the $g(c)$th element of demand vector $d[g(c)]$ decreases 1, i.e., $d'[g(c)] = d[g(c)] - 1$, while other elements do not change. In the meanwhile, for the resource vector, $r'[g(s, f(s))] = r[g(s, f(s))] - (H - f(s))$ while $r'[g(c)] = r[g(c)] + H - f(s) - 1$, and other elements do not change. Then for $\varphi \in (g(c), g(s, f(s))]$, the feasible condition in the new layout is $D'(\varphi) = \sum_{g \geq \varphi} d'(g) = D(\varphi) \leq R'(\varphi) = \sum_{g \geq \varphi} r'(g) = R(\varphi) - (H - f(s))$, i.e., $\Delta(\varphi) \geq H - f(s)$.

Pairs satisfy the above conditions compose a set $\mathbb{T}$. However, the above conditions are only necessary conditions, so the feasibility of the resultant state is not sufficiently ensured, due to the possibility of dead-end states.

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

1. Prevent from entering an extreme state when deciding the next task;
2. Design an ideal task accomplishment procedure that makes the resultant extreme state feasible;
3. Allow entering a dead-end state and relabel a fixed container as unfixed to run away from the dead-end state.

Our previous work [18] adopts the last method listed above. When the algorithm enters into a dead-end state, the algorithm relabels a fixed container as unfixed and moves it to a temporary slot; then the state becomes non-dead-end. After taking a sequence of movements, the relabeled container is recovered to its original slot. The disadvantage of such a choice is that running away from existing dead-end states costs many movements and these movements are unnecessary per se. A more choice is to avoid resulting in dead-end states when choosing valid tasks.

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

The algorithm only explores state $(\mathsf{L}, \boldsymbol{f})$ which satisfy the necessary condition for state feasibility (see Proposition 1). When only three stacks have available slots while the other $S - 3$ stacks

are fully fixed ($|s \in \mathbb{S} : f(s) = H| = S - 3$), and at least one stack has only one available slot left $|s \in \mathbb{S} : f(s) = H - 1| \geq 1$, 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 $\sum_{s\in\mathbb{S}} f(s) = N$, all the containers are fixed, the algorithm terminates with an orderly layout.

2. If $\sum_{s\in\mathbb{S}} f(s) = N - 1$, only one container remains unfixed. Performing the task will make the layout feasible.

3. If $\sum_{s\in\mathbb{S}} f(s) = N-2$, two containers remain unfixed. Suppose the last two unfixed containers are a and b, and the valid task is $a \rightarrow s$ with $f(s) = H - 1$. When generating valid tasks, the fourth condition $\Delta(\varphi) \geq H - f(s), \varphi \in (g(c), g(s, f(s))]$ must be satisfied, i.e., the resultant layout L' after task $a \rightarrow s$ satisfies $\mathbf{\Delta} \geq 0$ and only b is unfixed. Hence, L' can be converted into a feasible layout. Performing task $a \rightarrow s$ does not result in a dead-end state.

4. If $\sum f(s)_{s\in\mathbb{S}} \leq N - 3$, 3 or more containers remain unfixed. If the next task is to fix a container to the stack s such that $f(s) = H - 1$, the resultant may be a dead-end state. Taking Figure 6 as an example. Containers with a solid square are fixed containers. For the task $1 \rightarrow 3$, it is valid as the surplus vector of the resultant state is no lee than zero. But the resultant state becomes dead-end. Based on the above discussion, we can see that when $\sum_{s\in\mathbb{S}} f(s) < N - 2$, there

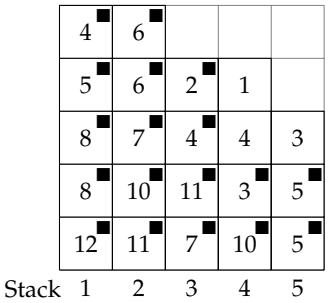


Figure 6: Pre-extreme state.

is a chance to generate dead-end state.

When the current state is a pre-extreme state, we eliminate the container–stack pairs $c \rightarrow s$ such that $f(s) = H - 1$ from $\mathbb{T}$, so that the algorithm avoids entering a dead-end state.

5.3. Task evaluation

Every valid task $c \rightarrow s \in \mathbb{T}$ is evaluated by a tuple with six elements: the tier-protection indicator, the number of moves required by the STAP, the number of stable containers that need to be moved from the aim stack $sh(s) - f(s)$, the affected demand, fixed height of the aim stack $f(s)$, $-g(c)$. 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. But if a container with small group value occupies an (relatively) empty stack, then the loss of the surplus vector is large: after the target container c is fixed to the target 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 spread between $g(c)$ and $g(s, f(s))$ is, the more the surplus vector will lose. For example, in Figure 6, if container 1 (container with group value 1) is moved on container 3 (stack 5 tier 3), the supply of group 3 is reduced by 2 as 1 becomes the highest orderly container of stack 5. The cumulative supplies $G(3)$ and $G(2)$ are reduced by 2 accordingly.

If a pair $c \rightarrow s$ satisfies $f(s) < P1$ and affected demand $\sum_{\varphi=g(c)+1}^{g(s,f(s))} d(\varphi) \geq P2$, then the task cannot be carried out and is deleted from the evaluation list. As customizable parameters, the number of tiers protected $P1$ and the threshold value $P2$ can be adjusted optionally.

The number of moves required by the STAP is the movements for moving c to s, the number is precise while in [18] the number is just an estimation. The process to calculate the number is introduced in Section 5.4. The affected demand is $\sum_{\varphi=g(c)+1}^{g(s,f(s))} d(\varphi)$.

5.4. Speedy task accomplishment procedure

After the task is decided, a speedy task accomplishment procedure (STAP) is carried out to accomplish the task, resulting in a new state. STAP is a move sequence which is based on task types.

Let the target container be denoted by c^* (located in (s^+, t^+)) and the target stack by s^*. The task is ***immediate*** if c^* is just on the highest fixed container in s^*, i.e., $s^+ = s^*$ and $t^+ = f(s^*) + 1$;

internal if c^* is above but not on the highest fixed container in s^*, i.e., $s^+ = s^*$ and $t^+ > f(s^*) + 1$; and ***external*** if c^* and s^* are in different stacks, i.e., $s^+ \neq s^*$.

5.4.1. Immediate task

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

5.4.2. Internal task

For an internal task, let a denote the number of empty slots except those in s^+ and 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. Blocking containers refer to those above target containers or target slots. The accomplishment procedure of an internal task is given in Algorithm 2.

Algorithm 2: Accomplish an internal task.

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

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:` $h(s^+) - f(s^+) + 2$ `moves`

case I3: $a < b_2$ & $|\{s \neq s^+ : h(s) < H\}| = 1$
  Find s' s.t. $s' \neq s^+$ & $h(s') < H$;
  $s^{\text{tmp}} \coloneqq$ `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:` $h(s^+) - f(s^+) + 2$ `moves`

The accomplishment procedure is carried out based on the relationship between available empty slots and the number of blocking containers. The relationship determines how containers are relocated.

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

In case I1, b_1 containers above c^* are relocated one by one. $\mathbb{R}$ represents the set of available destination stack for the top container which blocks c^*. A stack s is available if $s \neq s^+, h(s) < H, \alpha(s^+, s) \geq b_2$. Here α function in Algorithm 3 indicates how many slots are left for b_2.

Algorithm 3: Alpha function.

```
function α(s⁺, s^dst)
    // s⁺ ≠ s^dst
    e^min := min{e(s) > 0 : s ≠ s⁺};
    e^sec := min{e(s) > e^min : s ≠ s⁺};
    k^min := |{s ≠ s⁺ : e(s) = e^min}|;
    if e(s^dst) > e^min then
        return E − e(s⁺) − e^min − 1;
    else if e^min ≥ 2 then
        return E − e(s⁺) − e^min;
    else if k^min = 1 then
        return E − e(s⁺) − e^sec − 1;
    else
        return E − e(s⁺) − 2;
```

In the process of relocating containers from b_1, it need consider reserving slots for containers from b_2. Because c^* is above b_2. After retrieving b_1, c^* is relocated before containers in b_2. If the relocation is to put c^* and b_2 in the same stack, then c^* needs additional moves to be retrieved. Hence, we reserve the stack with the minimum empty slots $s_{\min}$ for c^*, and the slots from $\{s \in S : s \neq s^+, s \neq s_{\min}\}$ for b_2. Of course, containers in b_1 can also occupy $s_{\min}$ and $\{s \in S : s \neq s^+, s \neq s_{\min}\}$, and definitely $|\{s \in S : s \neq s^+\}| \geq b_1 + b_2$. But the number of containers from b_1 which occupy $\{s \in S : s \neq s^+, s \neq s_{\min}\}$ needs computation, in order to satisfy b_2 first. Hence, when deciding whether a stack s^{dst} can be the destination of a container c from b_1, α function is used to compute if c is placed to s^{dst} how many slots left for b_2. And if the left slots are not enough for b_2, c cannot be moved to s^{dst}.

If $e(s^{\text{dst}}) > e^{\min}$, that is, s^{dst} is not the highest non-full stack. $e^{\min}$ is reserved for c^*. Slots of s^+ cannot be used. If c is moved to s^{dst}, left slots for b_2 is $E - e(s^+) - e^{\min} - 1$.

If $e(s^{\text{dst}}) = e^{\min} \geq 2$, that is s^{dst} is the highest non-full stack and it has at least 2 empty slots. If c is moved to $e(s^{\text{dst}})$, the stack can also accommodate c^*, hence, 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, s^{dst} is the unique highest stack with only one empty slot. If c is moved to $s^{s^{\text{dst}}}$, the stack with the second highest height will reserve for c^* in the next round. The slots left for b_2 becomes $E - e(s^+) - e^{\text{sec}} - 1$.

If $e(s^{\text{dst}}) = e^{\min} = 1$ and $k^{\min} \geq 2$, another stack with the one empty slot accommodates c^* in the future. If c is moved to $s^{s^{\text{dst}}}$, the slots left for b_2 becomes $E - e(s^+) - 2$.

The container relocation 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 evaluations by function `EvalMove`$(s^{\text{src}}, s^{\text{dst}})$. Function `Move`$(s^{\text{src}}, k, s^{\text{dst}})$ performs k relocations from stack s^{src} to stack s^{dst}. The function `EvalMove`$(s^{\text{src}}, s^{\text{dst}})$ and `Move`$(s^{\text{src}}, k, s^{\text{dst}})$ are also illustrated in Algorithm 4.

Algorithm 4: Relocation function.

```
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);
6  function Move(s^src, k, s^dst)
7      Move k containers from stack s^src to stack s^dst;
8  function EvalMove(s^src, s^dst)
9      c := (s^src, h(s^src));
10     g := g(c);
11     q := q(s^dst, h(s^dst));
12     case sh(s^dst) = h(s^dst) & stack s^dst can stabilize c
13         return ⟨1, Σ_{φ=g+1}^{q} d(φ)⟩;
14     case sh(s^dst) < h(s^dst) & g ≥ m(s^dst)
15         return ⟨2, g − m(s^dst)⟩;
16     case g(s^dst) < h(s^dst) & g < m(s^dst)
17         return ⟨3, m(s^dst) − g⟩;
18     case g(s^dst) = h(s^dst) & stack s_2 cannot stabilize c
19         return ⟨4, q⟩;
```

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, regarding the ground as an occupied slot at tier 0 with group value G. The evaluations of moves are represented by numerical tuples, which are lexicographically comparable. `EvalMove`$(s^{\text{src}}, s^{\text{dst}})$ returns the penalty of the move from stack s^{src} to s^{dst}. 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. 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 $g(c) \geq m(s^{\text{dst}})$, the minimum gap between $m(s^{\text{dst}})$ and $g(c)$ is preferred;

3. If stack s^{dst} is not entirely stable and $g(c) > m(s^{\text{dst}})$, the minimum gap between $g(c)$ 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 group value of the unstable containers in a stack; larger messiness implies a higher urgency of reshuffles. The last preference indicates that an entirely stable stack should be protected from being ruined.

Function `EvalMove` returns a tuple, the first element indicates the type of the value and the second element indicates the scale of the value.

In cases I1 and I2, an interim stack is selected to temporarily store the target container (function `Interim(`$\mathbb{I}$`)`). The selection prefers stacks that are not entirely stable, with the largest messiness, then entirely stable stacks with the smallest group of orderly containers. The most unattractive stack for receiving blocking containers is selected as the interim stack. In cases I3 and E4 (in Section 5.4.3), the interim stack is selected by the minimum group value of the top containers of these full stacks, which is function `InterimFull(`$\mathbb{F}$`)`.

Algorithm 5: Interim and InterimFull functions.

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

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 evaluation tuple. The one with smaller tuple is moved first. In Algorithm 6, $\vec{v}_1$ 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 stack s^{src} to two receiver sets, $\mathbb{R}_1$ and $\mathbb{R}_2$, respectively.

Algorithm 6: BiSender and BiReceiver functions.

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

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

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

5.4.3. 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 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 7, 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: $a = 0 < b_1 + b_2$.

Algorithm 7: 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`

6. Computational results

In this section, the proposed feasibility-based heuristic (FBH) is compared to two benchmark heuristic approaches. One is the target-guided heuristic (TGH) proposed in our previous work [18], and the other is a similar implementation to the LPFH proposed by [8]. 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 developed a substitute instead, referred to as the largest group value first heuristic (LGVFH). We implemented both TGH and LGVFH. Experiments were conducted on a PC with Intel Core i7 CPU clocked at 3.40 GHz with Windows 7 operating system. The code was written in Java.

The LGVFH selects the next target container from unfixed containers with the largest group value, and then accomplish it by the STAP with the same user-defined functions used as that in the FBH. Necessary pseudo-code for the LGVFH is given in Algorithm 8.

Algorithm 8: Largest group value first heuristic.

```
1  begin
2      (L, f) := (L^0, U · 1);
3      foreach g = G, ..., 1 do
4          C_g := containers with group value g;
5          A_g := ∅; // set of aim stacks
6          while C_g ≠ ∅ do
7              (c*, s*) :=
               arg min_{c∈C_g, f(s)<H} MoveNeed(c, s);
8              Accomplish (c*, s⁻) by the STAP;
9              f(s*) := f(s*) + 1;
10             C_g := C_g \ {c*};
11             A_g := A_g ∪ {s*};
12         foreach s* ∈ A_g do
13             Fill(s*);

14 function Fill(s)
15     while h(s) < H do
16         S' := ∅;
17         foreach i ∈ {i ≠ s : sh(i) < h(i)} do
18             c := (i, h(i));
19             if stack s can stabilize c then
20                 S' := S' ∪ {i};
21         if S' ≠ ∅ then
22             s' := arg max_{i∈S'} g(i, h(i));
23             Move(s', 1, s);
24         else
25             break while;
```

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

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 [2,4,8,18] and the CRP [5,7,9,11,12].

6.1. Configuration of P1 and P2

This section shows the effect of the tier-protection indicator. We selected six settings of the two parameters $(P1, P2)$, which is $(0, N), (H/4, N/2), (H/3, N/6), (2H/3, N/8), (H, N/10)$. We made $P1(P2)$ proportional to $H(N)$, since $f(s)$ (affected demand) is related to height (number of containers).

The six settings are in a loose-intermediate-tight order, and algorithms will filter out more tasks when using back settings. The result is shown in Table 2. The first column indicates the parameter setting. The second and third columns indicate the average movements of CVS and BF data sets, respectively. Parameter $(0, N)$ means that there is no tier-protection indicator as no task is filtered out. As the setting becomes tight, the performance of both CVS and BF is improved, until a peak is achieved. But when the setting becomes extremely tight, the performance decreases. When the setting is $(H, N/10)$, some instances of CVS even cannot find a solution; therefore, we make the unit with "Nil". The instances of BF are loose, that is, the empty slots are sufficient. Solutions can be found even if under setting $(H, N/10)$, but the performance is worse than that with intermediate setting.

Table 2: Configuration of $P1$ and $P2$

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

6.2. Results for CVS instances

Caserta et al. [5] 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 3 illustrates the computational results on CVS instances by the TGH, the LGVFH, and the FBH. The values under the "move" headings represent average number of moves for every CVS group, whereas the values under the "time (ms)" headings are the run time in ms. The values under the "improvement" heading are the improvements compared to TGH and LGVFH in percentage. The results showcase that the FBH surpasses the benchmark heuristics by more than $10\% \times N$ moves on most instance groups. Especially on the narrowest group CVS 10-6, the FBH outperforms the TGH and the LGVFH by 319.38% and 55.54%, respectively.

6.3. Results for BF instances

Bortfeldt & Forster [2] 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 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 typical loose CPMP instances. Table 4 illustrates the computational results on BF instances by the three heuristics. The values under the move headings represent average number of moves for every BF group, whereas the values under the time (ms) headings are the run time in ms. The values under the improvement heading are the improvements compared to TGH and LGVFH in percentage.

Table 5 shows the computational results based on the dimension of density and stack height. It 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 FBH outperforms these two benchmark heuristics.

Table 3: Results for CVS instances.

CVS K-S	TGH		LGVFH		FBH		improvement	
	move	time (ms)	move	time (ms)	move	time (ms)	TGH	LGVFH
CVS 3-3	12.95	0.43	11.25	0.45	11.28	0.53	12.93%	-0.22%
CVS 3-4	12.18	0.20	12.23	0.15	10.80	0.18	11.29%	11.66%
CVS 3-5	12.78	0.10	13.45	0.20	12.08	0.18	5.48%	10.22%
CVS 3-6	14.38	0.18	14.88	0.33	12.98	0.23	9.74%	12.77%
CVS 3-7	16.00	0.13	16.58	0.23	14.75	0.15	7.81%	11.01%
CVS 3-8	16.55	0.15	17.08	0.28	15.65	0.38	5.44%	8.35%
CVS 4-4	23.35	0.05	21.93	0.08	21.88	0.15	6.32%	0.23%
CVS 4-5	26.73	0.10	26.48	0.18	23.08	0.18	13.66%	12.84%
CVS 4-6	27.58	0.08	27.20	0.23	24.75	0.48	10.24%	9.01%
CVS 4-7	29.93	0.08	31.23	0.38	27.63	0.10	7.69%	11.53%
CVS 5-4	44.83	0.08	35.83	0.18	35.08	0.08	21.75%	2.09%
CVS 5-5	42.40	0.05	36.50	0.25	35.33	0.23	16.69%	3.22%
CVS 5-6	50.63	0.13	43.08	0.23	39.88	0.23	21.24%	7.43%
CVS 5-7	48.83	0.13	46.95	0.73	41.68	0.28	14.64%	11.24%
CVS 5-8	56.68	0.20	51.83	0.63	47.50	0.48	16.19%	8.35%
CVS 5-9	57.50	0.08	55.65	1.08	50.45	0.90	12.26%	9.34%
CVS 5-10	62.80	0.53	60.88	1.30	54.63	0.80	13.02%	10.27%
CVS 6-6	74.33	0.13	57.85	0.28	55.23	0.50	25.7%	4.54%
CVS 6-10	88.63	0.43	79.73	1.88	75.60	1.65	14.7%	5.17%
CVS 10-6	332.25	0.30	173.95	1.08	140.63	0.85	57.57%	19.16%
CVS 10-10	302.90	0.90	190.50	5.05	179.23	3.65	40.83%	5.92%
Average	64.48	0.21	48.81	0.72	44.29	0.58	16.44%	8.29%

Table 4: Results for BF instances.

BF	S	H	N	G	B	TGH		LGVFH		FBH		improvement	
						move	time (ms)	move	time (ms)	move	time (ms)	TGH	LGVFH
1	16	5	48	10	29	29.10	2.85	29.55	5.05	29.15	4.80	-0.17%	1.35%
2	16	5	48	10	36	36.00	1.35	36.60	2.90	36.00	3.65	0	1.64%
3	16	5	48	20	29	29.45	0.85	30.90	2.65	29.35	3.30	0.34%	5.02%
4	16	5	48	20	36	36.00	0.55	37.20	1.55	36.15	1.60	-0.42%	2.82%
5	16	5	64	13	39	48.50	1.55	53.00	2.90	46.30	2.65	4.54%	12.64%
6	16	5	64	13	48	57.55	0.95	62.85	3.45	55.50	2.65	3.56%	11.69%
7	16	5	64	26	39	53.55	0.85	57.15	3.10	49.95	2.25	6.72%	12.6%
8	16	5	64	26	48	60.00	0.80	66.90	2.90	57.60	2.40	4%	13.9%
9	16	8	77	16	47	60.35	1.55	62.15	4.35	56.30	3.90	6.71%	9.41%
10	16	8	77	16	58	62.15	7.90	69.50	4.65	61.55	3.85	0.97%	11.44%
11	16	8	77	31	47	61.25	1.15	63.70	4.20	55.00	3.35	10.2%	13.66%
12	16	8	77	31	58	63.45	1.10	68.50	4.15	61.45	3.30	3.15%	10.29%
13	16	8	103	21	62	107.45	1.75	110.85	6.45	96.50	5.55	10.19%	12.95%
14	16	8	103	21	78	124.75	2.55	134.90	5.95	116.05	4.75	6.97%	13.97%
15	16	8	103	42	62	110.60	2.90	110.40	9.45	99.45	6.50	10.08%	9.92%
16	16	8	103	42	78	133.35	1.55	137.40	7.40	115.45	5.85	13.42%	15.98%
17	20	5	60	12	36	36.50	0.90	37.35	2.30	36.50	2.15	0	2.28%
18	20	5	60	12	45	45.00	0.80	45.00	2.15	45.20	1.90	-0.44%	-0.44%
19	20	5	60	24	36	36.80	0.75	38.50	2.90	36.75	1.95	0.14%	4.55%
20	20	5	60	24	45	45.00	2.00	45.70	3.00	45.10	2.80	-0.22%	1.31%
21	20	5	80	16	48	61.65	1.65	65.65	4.10	56.55	3.25	8.27%	13.86%
22	20	5	80	16	60	67.90	1.30	74.50	4.10	65.55	3.25	3.46%	12.01%
23	20	5	80	32	48	61.10	1.40	65.75	4.75	55.25	4.05	9.57%	15.97%
24	20	5	80	32	60	70.95	1.95	76.65	5.80	68.00	4.25	4.16%	11.29%
25	20	8	96	20	58	69.80	3.90	73.60	8.60	66.00	6.75	5.44%	10.33%
26	20	8	96	20	72	74.35	3.60	81.75	7.55	75.75	7.15	-1.88%	7.34%
27	20	8	96	39	58	71.85	2.95	73.65	8.50	65.65	6.70	8.63%	10.86%
28	20	8	96	39	72	76.30	2.10	83.55	8.35	76.50	6.25	-0.26%	8.44%
29	20	8	128	26	77	118.65	3.60	128.65	11.15	115.85	8.80	2.36%	9.95%
30	20	8	128	26	96	143.05	3.60	155.15	11.05	129.60	9.75	9.4%	16.47%
31	20	8	128	52	77	128.15	3.30	128.80	15.15	115.85	10.85	9.6%	10.05%
32	20	8	128	52	96	147.30	2.70	157.00	13.60	134.10	10.45	8.96%	14.59%
Average						72.75	2.08	76.96	5.75	68.44	4.71	5.92%	11.08%

Table 5: Summary on BF instances.

	TGH		LGVFH		FBH	
density	$H = 5$	8	5	8	5	8
0.6	36.73	67.44	37.60	72.05	36.78	64.78
0.8	60.15	126.66	65.31	132.89	56.84	115.36

7. Conclusions

The container pre-marshalling deals with how to rehandle containers in a bay so that the containers are placed in a determined order. By far, only seven works talk about solutions to this problem. In this paper we present a feasibility-based heuristic to solve the container pre-marshalling problem. The proposed heuristic can be implemented solely or combined with other frameworks. The main innovation is the concept of state feasibility, which checks feasibility of states before really searching those states. Thanks to this concept, the search space is enlarged compared with extant methods, whereas the search efficiency is guaranteed. Numerical experiments on two commonly used data sets show that our proposed method outperforms other methods.

Our algorithm can be inserted into many heuristic frameworks such as GRASP, LNS and beam search. How these combinations work is 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 resource surplus. In this paper, the bottom tiers protection technique is proposed to balance the trade-off. As a future work, more efforts should be dedicated to better balance the trade-off. Moreover, the feasibility of a state is essentially based on a lower bound of current state. If tighter lower bound is found, the performance of the algorithm can be improved. Therefore, finding a better lower bound of current state is a good research question. In this paper, we assume that the containers in the bay are determined and no container comes in or goes out during the pre-marshalling. In the future work, it can be discussed that how to handle dynamic yards in which containers come in or go out from time to time. To some extent, the new setting is similar to container stacking problem, as the close relationship of container pre-marshalling and container stacking problem, the methods proposed for the container pre-marshalling problem may provide a new perspective for the container stacking problem.

Acknowledgments

This research was partially supported by the National Natural Science Foundation of China (Grant no. 71371114, 71301102).

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, J. M. (2014). A domain-specific knowledge-based heuristic for the blocks relocation problem. *Advanced Engineering Informatics*, 28(4), 327–343.

[8] 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.

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

[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., 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.

[12] 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.

[13] 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.

[14] Lee, Y. & Chao, S.-L. (2009). A neighborhood search heuristic for pre-marshalling export containers. *European*

Journal of Operational Research, 196(2), 468–475.

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

[16] 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.

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

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