Target guided algorithms for the container pre-marshalling problem

Ning Wang, Bo Jin*, Andrew Lim

Department of Management Sciences, City University of Hong Kong, 83 Tat Chee Ave, Kowloon Tong, Hong Kong

Abstract

We investigate the container pre-marshalling problem (CPMP), which aims to rearrange containers in a bay with the least movement effort, so as to find a final layout where containers are piled according to a predetermined order. Previous researchers, without exception, assume that all the stacks in a bay are functionally identical. We reexamine such a classic problem setting and propose a new problem – the CPMP with a dummy stack (CPMPDS). At terminals that use gantry cranes, a bay includes a row of ordinary stacks and a dummy stack. The dummy stack actually is the bay space reserved for trucks, via which containers can be shipped out of the bay. During the pre-marshalling process, the dummy stack temporarily stores containers, just like an ordinary stack. But it is required to restore emptiness at the end of the pre-marshalling process. In this paper, we propose target guided algorithms which guarantee termination to tackle both the classic CPMP and the new CPMPDS; the key idea is to fix containers at appropriately chosen positions one by one. We also devise an improved lower bound for the number of required movements. Experimental results in terms of the CPMP showcase that our algorithms surpass the state-of-the-art algorithm.

Key words: container pre-marshalling, target guided algorithms, lower bound, dummy stack

1. Introduction

Seaborne transportation is the cornerstone of international trade and undoubtedly an engine of global economic development. According to *Review of Maritime Transport* (2012) released by UNCTAD, around 80% of global trade by volume is carried by sea. Amongst different ship types, container ships account for about 62% of dry cargoes. Since the commencement of containerization, containers facilitate smooth flow of goods across multiple transportation modes without direct handling the freight during the course of shipping. Accompany with the rapid growth of container shipping, container terminals face unprecedented development opportunities. The role of container terminals lies in providing physical spaces for container exchanges between sea and land. At terminals, a large proportion of land spaces is reserved for storing

*Corresponding author. Tel: +852 34425296

Email addresses: wangning@cityu.edu.hk (Ning Wang), msjinbo@cityu.edu.hk (Bo Jin), lim.andrew@cityu.edu.hk (Andrew Lim)

containers temporarily, which is known as container yards. A yard acts as a cache that makes the loading and unloading process of ships in a fast pace. Basically, a conventional practice of exporting containers is to store them on the yard first. When the ship arrives, all the containers are transported by trucks to the quayside and then loaded onto the ship. The process is reverse for import containers.

High efficiency is a key factor for any successful terminals since shipping line managers prefer efficient terminals. Carriers or shippers bear the pressure of quick delivery from down stream customers. Moreover, high efficiency increases the turnover rate of terminals, therefore generates more profit for the shareholders. Container terminals have every reason to increase their operational efficiency as much as possible. Both terminal practitioners and OR researchers have paid considerable attention on improving the efficiency of quayside operations. Nonetheless, the overall terminal productivity will not increase dramatically if only quayside operations are optimized, leaving yard operations inefficient (Jiang et al., 2012).

In general, a yard is divided into several blocks and a block is composed of several parallel bays. Each bay is formed by several stacks aligned side by side. In each stack, containers are piled vertically. Figure 1 shows a diagram of a yard. The height of stacks is constrained by the height of operation equipments, i.e. gantry cranes. Generally speaking, one stack can store 3∼10 containers.

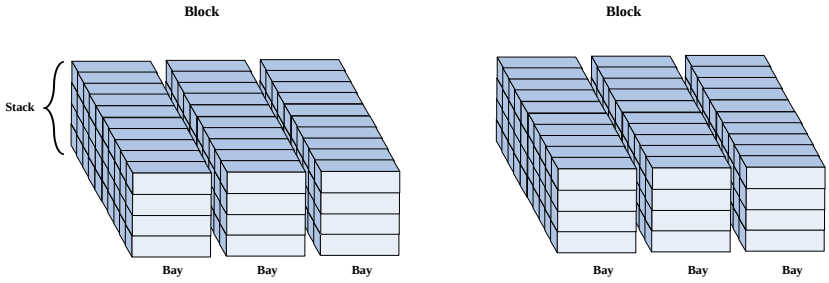


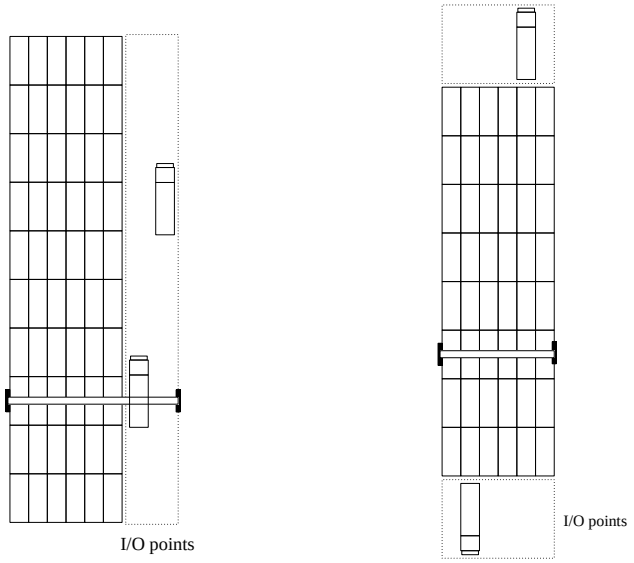
Figure 1: An example of a container yard

Containers in the same stack can only be retrieved in a "first in last out" manner. If the container to be fetched is not at the top of a stack, all the containers placed above it have to be relocated to other stacks before retrieving it. Such forced movements are known as rehandlings. Rehandling containers is a costly activity as this additional work decreases the efficiency of container retrieval, whereas consignors do not pay for this. In the ideal situation, containers are piled in an order consistent with the retrieval order, i.e., containers retrieved earlier are located at higher tiers. The realistic situation, however, goes by contraries. The placement order of containers is decided by inland transportation or consignors; while the retrieval order of containers is decided by the stowage planning or ship schedule. In most situations, the two orders are not exactly reverse. In reality, containers which arrive early do not necessarily depart late.

Prior to ship arrival, containers in a bay can be rearranged to comply with their retrieval order, hence the name pre-marshalling. The pre-marshalling work, though not billable to consignors, can reduce the berthing

time of ships and increase the turnover rate of terminals. How to carry out the container pre-marshalling with the least effort is the so called container pre-marshalling problem (CPMP).

By far, though the research community has studied the CPMP, it does not fully depict practical scenarios. For terminals using gantry cranes, there exist two configurations of I/O points where trucks and gantry cranes exchange containers Carlo et al. (2014). Figure 2 show the difference of two configurations. In the first configuration, I/O points form lanes (transfer lane) which are parallel to a block while in the second configuration, I/O points form bays which are at both ends of a block. In the first configuration, when no container is being retrieved, transfer lanes act as temporary stacks for the pre-marshalling work just like ordinary stacks. The only issue demanding attention is that transfer lanes should become empty again after the pre-marshalling work, which distinguishes themselves from ordinary stacks. We coin the transfer lanes as **dummy stacks**.



(a) Configuration with transfer lane (b) Configuration with transfer bay

Figure 2: Two configurations of I/O points

Existing algorithms for the CPMP have not taken dummy stacks into consideration, hence they cannot be implemented directly at terminals using gantry cranes. However, terminals using gantry cranes as yard operation equipments are rather popular across the world. Herein, it is imperative to fill this disparity between research and industry.

Our paper contributes to the literature on three aspects. First, we reexamine the CPMP and propose a new variant – the CPMP with a dummy stack (CPMPDS). Second, we devise a new lower bound on the calculation of movements, suitable for both CPMP and CPMPDS. The new lower bound dominates other existing lower bounds with respect to the CPMP. Third, we design three algorithms innovated by the idea of target-guided to solve both the classic CPMP and the new CPMPDS. The ideology of the algorithms is to fix containers at appropriately chosen positions and avoid moving them afterwards. This mechanism reduces the problem size

continuously until the problem is solved. Thus, our algorithms guarantee termination. Experimental results demonstrate that our algorithms are better than the state-of-the-art algorithm to the best of our knowledge. It is noteworthy that although this paper discusses the CPMP in the context of maritime terminals, it is also applicable to other scenarios such as warehouses and railway yards.

The remainder of this paper is structured as follows. Section 2 presents an overview of existing works related to the CPMP. After that, we formally define the CPMP/CPMPDS in Section 3 and present how to calculate lower bounds in Section 4. A heuristic algorithm and two advanced beam search algorithms are elaborated in Section 5 and Section 6, respectively. We conduct experiments which evaluate the effectiveness of our algorithms and report the results in Section 7. At last, we conclude our research in Section 8 with some closing remarks.

2. Literature review

In recent years, publications in press concerning maritime transportation, especially container transportation, have exploded. Vis & de Koster (2003) are the first who give an overview of the complex operations at container terminals and associated research problems. Then Steenken et al. (2004) provide a more extensive description on main logistics processes and operations at container terminals, together with a corresponding survey of optimization models and methods. It is followed by Stahlbock & Voß (2008) who trace new updates on container terminals. Collectively, these studies point out that the CPMP is an important topic and worthy of more research effort.

Most existing methods for the CPMP are heuristic approaches. Caserta & Voß (2009) provide a greedy heuristic based on the paradigm of corridor and roulette wheel. The corridor reduces the number of movement choices for a certain layout and the roulette wheel brings randomness when making choices. The probability of selecting an alternative is proportional to the corresponding attractiveness. Their algorithm first builds a corridor with respect to the current layout to determine the destination stacks of a specific misplaced container. Then it yields new layouts by conducting the movements in the corridor, and evaluates the attractiveness of each new layout by an estimated number of needed relocations. It also conducts a local improvement scheme to accelerate the search process. Expósito-Izquierdo et al. (2012) propose a multi-start heuristic to minimize the number of rehandlings. They select movements by a rule called "low priority first", so their method is more target-oriented than the one of Caserta & Voß (2009). In their work, the concept of corridor and roulette wheel are also used when building solutions. What is more, they give a method to generate instances and measure the difficulty of an instance.

Different from the above approaches that construct solutions by finding promising movements step by step, Lee & Chao (2009) deploy a neighborhood search where complete solutions are regarded as the units of

search. The neighborhood search obtains a new feasible solution by randomly modifying the current solution. Solutions are further shortened without disturbing the feasibility by a four-step procedure. In addition, three minor routines are devised to diversify the resultant solutions and further reduce the number of movements. As it is very likely to generate infeasible solutions by pure modification, their method needs a large number of iterations. Hence, it is not as efficient as the heuristic of Expósito-Izquierdo et al. (2012). After that, Huang & Lin (2012) solve two variants of the CPMP by a heuristic algorithm. One variant is commonly seen which allows containers of different groups to mix within a bay; while the other variant is newly proposed which requires containers of different groups separately located in the final layout.

Apart from heuristics, Lee & Hsu (2007) develop an integer programming model to tackle the CPMP. They convert the CPMP into a multi-commodity network flow problem. A network is composed of several subnetworks, with each representing an interim layout. The nodes in a subnetwork correspond to the slots of the bay while the containers are expressed as the commodities. A solution is expressed by a flow in the network. The disadvantage of their algorithm is that the scale of networks is giant even for small instances. Whereas, it provides us an innovative view to investigate the CPMP. Bortfeldt & Forster (2012) thoroughly describe a tree search procedure and a new lower bound for the CPMP. In the search tree, solutions are constructed by compound moves (several movements) instead of single movements, and the branches are classified by four movement types. Their performance, by far, is the best among all the extant methods.

As discussed in Caserta et al. (2011a), the counterparts of the pre-marshalling problem include the container re-marshalling problem and the container relocation problem. The container re-marshalling problem refers to reassigning containers scattered within a block into their designated bays. It is not a simple extension of the CPMP, as there are multiple cranes involved. The interference between multiple cranes, as well as the stacking positions of containers within a bay, is the difficulty of the problem. Caserta et al. (2011a), in their paper, prove that the container re-marshalling problem is NP-hard. Kim & Bae (1998); Kang et al. (2006); Park et al. (2009); Choe et al. (2011) have exploited the container re-marshalling problem and came up with some heuristic methods to cope with it. On the other hand, the container relocation problem aims to retrieve containers in a predetermined order with the fewest movements. It has been studied by researchers using various methods, ranging from meta-heuristics to simple heuristic algorithms (Kim & Hong, 2006; Yang & Kim, 2006; Caserta et al., 2009, 2011b; Lee & Lee, 2010; Caserta et al., 2012; Forster & Bortfeldt, 2012; Zhu et al., 2012).

3. Problem description

The container pre-marshalling problem (CPMP) can be summarized as follows: given a bay with S stacks, each stack is able to store up to H containers vertically. Tiers of the bay are indexed from 1 to H from the

bottom up. To make it consistent, the ground is considered as tier 0 in particular. There are N ($N \leq S \times H$) containers stored in the bay, forming the initial layout with a usage rate $N/(S \times H)$. Each container c is assigned with a group label $g(c) \in \{1, \ldots, G\}$, and the group label of different containers could be unique or duplicate. Figure 3 shows an example of a layout.

One container can be moved from the top of a stack (**origin**) to the top of another stack (**destination**) if the destination stack is not fully occupied. The CPMP is to find the shortest movement sequence which transforms the initial layout to a final layout with all the stacks **clean**. A stack is **clean** if its containers are piled in a non-increasing order of group labels from the bottom up, otherwise we say it is **dirty**. We say a stack s can **accommodate** a container c if and only if stack s is clean and is still clean if moving container c to its top. A container c in a certain layout is a clean container if and only if all the containers underneath are clean and their group labels are no smaller than $g(c)$. Otherwise, it is a dirty container.

6					3	
5			3		9	
4		5	8	4	7	
3		6	2	5	2	
2	4	2	8	4	7	
1	1	3	1	4	9	
ground 0	1	2	3	4	5	

Figure 3: An example of a layout

In the new variant – the container pre-marshalling problem with a dummy stack (CPMPDS), containers are only piled in the first $S - 1$ stacks, while the S-th stack (so called dummy stack) is empty in the initial layout. During the course of pre-marshalling, containers can be placed in the dummy stack. The objective of the CPMPDS is to transform the initial layout to a clean layout where the dummy stack is empty, with the least movement effort.

For ease of illustration, we define some notations in Table 1. All of the notations are used in the context of the current layout, therefore we do not explicitly point out the layout in notations.

4. Lower bound

In this section, we propose a method to calculate the lower bound for the number of movements necessary to reach a clean layout. We first illustrate how to calculate the lower bound for the CPMP; the lower bound for the CPMPDS only needs a small modification on the lower bound for the CPMP.

Table 1: Notations

Notations	Description
H	The height limitation of the bay
$e(s)$	The number of empty slots of stack s
$h(s)$	The number of containers piled at stack s
$d(s)$	The number of dirty containers of stack s
$uf(s)$	The number of unfixed containers of stack s
$sn(c)$	The stack in which container c is placed
$tn(c)$	The tier at which container c is placed
$g(c)$	The group label of container c
$o(c)$	The number of containers above c

4.1. Lower bound for the CPMP

The lower bound is obtained by estimating the number of movements that belong to different movement types.

1. Dirty–Clean movements

The number of dirty containers in a given layout is $\sum_{s=1}^{S} d(s)$. Each dirty container requires at least one movement to become clean; thus, the lower bound of the number of Dirty–Clean movements is $num_{DC} = \sum_{s=1}^{S} d(s)$.

2. Dirty–Dirty movements

If all stacks are dirty in a layout, at least one stack should become clean to accommodate containers. This step is done by moving dirty containers of a stack to other dirty stacks. The least number of such Dirty–Dirty movements is $num_{DD} = \min_{s=1,\ldots,S} d(s)$.

3. Clean–X movements

Before proceeding to explain the calculation, the concept of skyline is first introduced. A skyline is any vertical partition that separates a layout into higher and lower parts, where the lower part must be clean. The bold line in Figure 4 is an example of a skyline. A skyline can be represented by a vector SL such that each element SL_s represents the skyline segment across stack s. Segment SL_s is described by the container underneath; the tier and group numbers of SL_s are equal to those of the associated container. For example, in Figure 4, $tn(SL_2) = 3$ and $g(SL_2) = 1$.

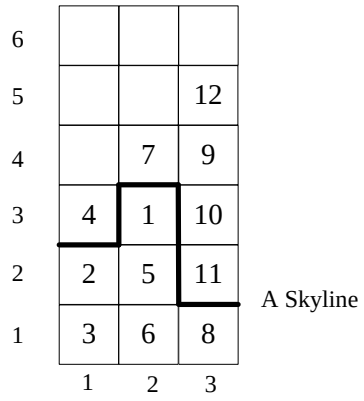


Figure 4: A skyline

Supply and demand vectors, denoted as $\mathbf{S}^{SL}$ and $\mathbf{D}^{SL}$, record the numbers of slots and containers above SL. The g-th element of $\mathbf{S}^{SL}$ is $\mathbf{S}^{SL}_g = \sum_{s:g(SL_s)=g} H - tn(SL_s)$, while the g-th element of $\mathbf{D}^{SL}$ is the number of containers above SL with group label g.

For ease of explanation, Clean–Dirty and Clean–Clean movements are merged as Clean–X movements. The number of moved clean containers is a lower bound of the number of Clean–X movements. Ideally, pre-marshalling work involves dirty containers only and leaves clean containers unmoved. In some cases, however, achieving a clean layout is impossible without moving clean containers. The number of clean containers that need to be moved can be counted by the following process: given a bay, all dirty containers and a set of clean containers C are removed from the layout, then slots are reassigned to removed containers. The resultant layout is required to be clean. Removed containers within a stack should be consecutive, i.e., if two containers from the same stack are removed, containers between them are also removed. The size of smallest C is the number of Clean–X movements num_{CX}. Finding the smallest C is equivalent to finding the skyline with the fewest clean containers in its higher part (smallest skyline) as well as non-negative surpluses (Equation 1).

$$\sum_{i \geq g} \mathbf{S}^{SL}_i - \sum_{i \geq g} \mathbf{D}^{SL}_i \geq 0, \quad g = 1, \ldots, G \tag{1}$$

To find the smallest skyline, a basic skyline SL that separates clean and dirty containers is drawn. Next a depth-first-search (DFS) is deployed to push a certain segment of SL one tier down in each branching. The leaves of the search tree are all the skylines with non-negative surpluses; the one with the fewest pushes is selected and the number of pushes is num_{CX}.

To conclude, the lower bound can be defined as $LB_{DFS} = num_{DC} + num_{DD} + num_{CX}$.

4.2. Clean–X movements by maximum knapsack

The lower bound LB_{DFS} is time-consuming because a DFS is used to calculate num_{CX}. A maximum knapsack method (MKM) is proposed to approximate num_{CX}. A basic skyline SL that separates clean and dirty containers is drawn, then the smallest skyline with non-negative surplus for a single group is obtained instead of for all groups. This search is repeated for each group, and the maximum skyline is selected amongst all obtained skylines.

For a given group label g, a knapsack-like problem is solved to find the smallest skyline with non-negative surplus in terms of g. Stacks in the bay are regarded as items, where each item $s \in \{1, \ldots, S\}$ has value v_s and cost c_s. The objective is to select a set of items with the lowest cost such that the total value selected is at least V. Every item can only be selected at most once. c_s is the number of clean containers in stack s to be moved to accommodate group g, while v_s is the resultant number of slots available for group g, i.e., $v_s = c_s + e(s)$. V is the number of dirty containers with group label $g(c) \geq g$. The decision variable x_s is 1 if stack s is selected; otherwise, $x_s = 0$. The integer programming model of this knapsack-like problem is show in Equation 2 and can be easily solved by dynamic programming.

$$\begin{aligned} \min \quad & \sum_{s=1}^{S} c_s x_s \\ \text{s.t.} \quad & \sum_{s=1}^{S} v_s x_s \geq V \\ & x_s \in \{0, 1\}, \quad s = 1, \ldots, S \end{aligned} \tag{2}$$

MKM is not as precise as DFS, but the solution gap is quite small and the computation speed is improved. The lower bound provided by MKM is denoted as LB_{MKM}.

4.3. Comparison with existing lower bound

The current state-of-the-art lower bound LB_{BF} was proposed by Bortfeldt & Forster (2012), which also consists of three parts: Dirty–Clean, Dirty–Dirty and Clean–X movements. The first two types are calculated in the same manner as LB_{DFS} and LB_{MKM} in this paper. For Clean–X movements, a basic skyline SL that separates clean and dirty containers is drawn, then they only take the group label g^* with the smallest surplus $min = \min_g \sum_{i \geq g} \mathbf{S}_i^{SL} - \sum_{i \geq g} \mathbf{D}_i^{SL}$. If min is negative, $\sum_{i \geq g^*} \mathbf{D}_i^{SL}$ surpasses $\sum_{i \geq g^*} \mathbf{S}_i^{SL}$ by $|min|$, which means that clean containers need to be moved to accommodate these $|min|$ dirty containers. The number of moved clean containers num_{CX} is estimated by Bortfeldt & Forster (2012) as follows: suppose one stack is able to (although in fact it may not be able to) accommodate H dirty containers with $g \geq g^*$, then $|min|$ dirty containers need at least $\left\lceil \frac{|min|}{H} \right\rceil$ stacks. As slots supply of stacks with $g(SL_s) \geq g^*$ have been calculated into $\sum_{i \geq g^*} \mathbf{S}_i^{SL}$, only stacks with $g(SL_s) < g^*$ (denote as set AS) can provide slots for $|min|$ dirty containers. Suppose c_s is the number of

clean containers in stack s to be moved to accommodate group g^*, sort c_s of stacks in set AS ascendingly. The first $\left\lceil \frac{|min|}{H} \right\rceil$ stacks are selected and $num_{CX} = \sum_{s=1}^{s=\left\lceil \frac{|min|}{H} \right\rceil} c_s$.

From the mathematical perspective, the method of Bortfeldt & Forster (2012) is equivalent to solve the following model, stacks in the bay are regarded as items, where each item $s \in \{1, \ldots, S\}$ has value v'_s and cost c_s. The objective is to select a set of items with the lowest cost such that the total value selected is at least V. Every item can only be selected at most once. c_s is the number of clean containers in stack s to be moved to accommodate group g^*, while v'_s is the resultant number of slots available for group g^*. According to Bortfeldt & Forster (2012) $v'_s = H$ if $g(SL_s) < g^*$, otherwise $v'_s = c_s + e(s)$. V is the number of dirty containers with group label $g(c) \geq g^*$. The decision variable x_s is 1 if stack s is selected; otherwise, $x_s = 0$. The integer programming model is show in Equation 3.

$$\begin{aligned} \min \quad & \sum_{s=1}^{S} c_s x_s \\ \text{s.t.} \quad & \sum_{s=1}^{S} v'_s x_s \geq V \\ & x_s \in \{0, 1\}, \quad s = 1, \ldots, S \end{aligned} \tag{3}$$

Compare Model 2 with Model 3, all parameters are the same except for v_s (v'_s). The solution to Model 3 is the solution to Model 2, while the reverse does not stand. The solution space of Model 3 is larger than Model 2, thus the optimal solution to Model 2 is no smaller than that to Model 3. Furthermore, Bortfeldt & Forster (2012) only solve Model 3 once for g^*, whereas this paper solves Model 2 for every group and then take the maximal result as num_{CX}. As a result, the lower bound LB_{MKM} dominates LB_{BF}: $LB_{BF} \leq LB_{MKM} \leq LB_{DFS}$.

4.4. Lower bound of the CPMPDS

The lower bound of the CPMPDS consists of three parts: Dirty–Clean, Dirty–Dirty, and Clean–X movements.

The lower bound of Dirty–Clean movements is equal to the number of dirty containers. For Dirty–Dirty movements $num_{DD} = \min_{s \neq s_d} d(s)$, where s_d is the dummy stack. If all stacks except for the dummy stack are dirty, then dirty containers from at lest one stack need to be relocated to other dirty stacks or the dummy stack. In both situations, these containers are still dirty after relocation.

The lower bound of Clean–X movements is the number of clean containers moved to make all containers clean and dummy stack empty, which is equivalent to assigning containers only to the first $S - 1$ stacks and ignoring the dummy stack.

Based on the analysis above, LB_{DFS} (LB_{MKM}) of a CPMPDS instance is equal to LB_{DFS} (LB_{MKM}) of the corresponding CPMP instance with the dummy stack removed.

5. Target guided heuristic

In this section, we present a target guided heuristic (TGH) for the CPMP/CPMPDS. This heuristic can be used as an independent algorithm, as well as a component of more complex algorithms in the next section. Before proceeding to the heuristic, we introduce the concept of movable (immovable) containers.

Definition 1. *A container is movable if it can be fetched and replaced to another stack.*

In a given layout, for any stack s, only the top $\sum_{i \neq s} e(i)$ containers are movable. Other containers in stack s, if any, are immovable; the numbers of immovable containers in every stack are the same, equal to $im = \max\left\{0, H - \sum_{s=1}^{S} e(s)\right\}$.

The main idea of the heuristic is: starting from the largest priority label, fix one container in a certain slot each time. After a container is fixed, it will not be relocated any more, except a special case. We resolve the special case by relocating fixed containers back. This will be explained later. As containers are fixed continuously, all the stacks become clean in the end. Hence, this heuristic guarantees to find a solution for any feasible instance. A complete description of the framework is displayed in Algorithm 1.

Algorithm 1 The target guided heuristic for the CPMP/CPMPDS

TGH (*inilay*)

```
1  curlay = inilay
2  mark all the immovable containers in curlay as fixed
3  mark all the movable containers in curlay as unfixed
4  g = G
5  while (g ≠ 0)
6      mark the clean containers with group label g as fixed
7      conList = the set of candidate containers
8      while (conList ≠ ∅)
9          stkList = the set of candidate stacks
10         select the target container and the target stack from conList and stkList, respectively
11         apply a giant move to fix the target container to the target stack
12     g = g − 1
```

In Algorithm 1, a movement sequence which fixes a target container to a target stack is named as a **giant move**. In contract, one movement operation is coined as a **baby move**.

The advantage of our heuristic is that containers are fixed one by one, therefore it is very easy to customize the heuristic for other CPMP variants, for example, variants which require the containers in a final layout adhere to special distribution characteristics. It only needs to modify the rules of candidate stacks selection, making the containers fixed according to the required distribution.

5.1. Candidate containers and stacks selection

When processing group g, the set of candidate containers is comprised of all the unfixed containers with group label g. Candidate stacks are selected based on the candidate containers. To be a candidate stack, a stack must be able to accommodate group g after relocating some of its containers. In the CPMPDS, the dummy stack cannot be a candidate stack. Among all the available stacks, we prefer stacks which can accommodate g without any relocation. If there does not exist such a stack, we select an available stack randomly.

5.2. Target pair selection

An evaluation scheme is designed to evaluate the attractiveness of fixing a candidate container to a candidate stack. Intuitively, we want the movements as few as possible. A function $f(c, s)$ is introduced to roughly estimate the movement cost for fixing container c to stack s. If container c is currently in stack s, $f(c, s) = uf(s)$; otherwise, $f(c, s) = uf(s) + o(c)$. Then we prefer to fix containers at lower tiers, i.e., stacks with smaller number of fixed containers $h(s) - uf(s)$ is preferred. This preference makes the fixed containers distributed evenly among stacks. In summary, 2-tuple $(f(c, s), h(s) - uf(s))$ evaluates the attractiveness.

When selecting the target pair, each container from candidate containers and each stack from candidate stacks make up of a candidate pair. Sort all the candidate pairs in the ascending lexicographical order of the evaluation values and the best one is selected as the target pair.

5.3. A giant move

Given the target container c^* and the target stack s^*, a giant move is executed to fix container c^* in stack s^*. The target slot of s^* for c^* is the slot just above the fixed containers of s^*. Hereafter, we do not explicitly point out the target slot. A giant move does not disturb the positions of already fixed containers. According to whether the origin of container c^* is stack s^* or not, the giant move includes two cases. We describe the concrete movements for the two cases separately.

5.3.1. Case 1: fixing to the same stack

In Case 1, the origin of container c^* is stack s^*. In the first step, all the containers above c^* are relocated; container relocation strategy will be explained in Section 5.4. After that, container c^* is at the top of stack s^* and it needs a stack for temporary storage. We consider the highest non-full stacks for temporary storage. Among such stacks, a dirty stack is preferred. Denote the temporary stack as stack ts.

Before moving container c^* to stack ts, we observe in advance where to relocate the unfixed containers under c^* so that stack s^* can accommodate c^*. Avoiding occupying stack ts and s^* is a straightforward idea. On top of this, denote the set of stacks except stack ts and s^* as AS, compute its available empty slots $nslot = \sum_{s \neq s^*, ts} e(s)$, and compare $nslot$ with $uf(s^*)$. According to the comparison results, three scenarios are raised.

1. $nslot \geq uf(s^*) - 1$

In this scenario, the empty slots of AS are enough to store the unfixed containers under c^*. We move c^* directly to stack ts and then relocate all the unfixed containers under c^* to the empty slots of AS. At last, c^* is moved back to stack s^*.

2. $0 < nslot < uf(s^*) - 1$

In this scenario, the empty slots of AS are not enough for the unfixed containers under c^*. Following moving c^* to stack ts, only the first $nslot - 1$ unfixed containers under c^* are relocated to the empty slots of AS, with an empty slot left. We occupy the last empty slot by moving c^* to it. Then the remained unfixed containers in stack s^* are relocated to stack ts. At last, c^* is moved back to stack s^*.

3. $nslot = 0$

In this scenario, all the stacks belonging to AS are full. We cannot move c^* directly to stack ts otherwise it will be buried soon by the unfixed containers in stack s^*. A top slot is needed to temporarily store c^*. To create such an empty slot, we seek a top container from all the top containers of stacks belonging to AS. The top container c is selected as follows.

If stack ts is a clean stack, the preference is:

1. Among dirty containers which stack ts can accommodate, select the one with the largest group label;
2. Among clean containers which stack ts can accommodate, select the one with the largest group label;
3. The dirty container with the smallest group label;
4. The clean container with the smallest group label.

If stack ts is a dirty stack, the preference is:

1. The dirty container with the smallest group label;
2. The clean container with the smallest group label.

After container c is moved to stack ts, its origin is occupied by c^*. In the following, all the unfixed containers of stack s^* are relocated to stack ts. At last, c^* is moved to stack s^*. If c is a fixed container, we should restore it to its origin. Suppose the origin of c is stack cs, relocate all the containers above c and move c back to stack cs.

5.3.2. Case 2: fixing to a different stack

In Case 2, the origin of container c^* is not stack s^*. If c^* is the topmost container and all the containers in stack s^* are fixed containers, move c^* to stack s^* directly. If not, in spirit of Case 1, we define AS as the set of all the stacks except stack s^* and $sn(c^*)$ (the origin of c^*) and compare least movements required $f(c^*, s^*)$ with available slots $nslot = \sum_{s \neq s^*, sn(c^*)} e(s)$. According to the comparison results, four scenarios are raised.

1. $nslot \geq f(c^*, s^*)$

In this scenario, relocate all the containers above c^* and unfixed containers in stack s^* to the empty slots of AS. When relocation, each time we relocate the top container from stack s^* or $sn(c^*)$ which has a larger group label. After that, container c^* is moved to stack s^*.

2. $o(c^*) + 1 \leq nslot < f(c^*, s^*)$

In this scenario, containers above c^* and the unfixed containers in stack s^* cannot be relocated to the empty slots of AS one-off. We choose to relocate all the containers above c^* together with a subset of unfixed containers in stack s^*, with a total of $nslot - 1$ containers, leaving one empty slot for c^*. Again, when relocation, select the topmost one with a larger group label each time. After relocating aforesaid containers, move c^* to the unique empty slot left in AS. In the following, relocate the remaining unfixed containers in stack s^* to stack $sn(c^*)$. At last, container c^* is moved to stack s^*.

3. $1 \leq nslot < o(c^*) + 1$

In this scenario, we only relocate $snlot - 1$ containers of stack $sn(c^*)$ to the empty slots of AS, and then relocate remaining containers above c^* to stack s^*. At present, c^* is on the top and moved to the unique empty slot in AS. Then all the unfixed containers in stack s^* including those originated from stack $sn(c^*)$ are relocated to stack $sn(c^*)$. At last, container c^* is moved to stack s^*.

4. $nslot = 0$

In this scenario, it is somehow like the third scenario in Case 1. We need to create an empty slot for c^* for temporary storage. The selection of a top container from a full stack is the same with that mentioned in Scenario 3 of Case 1. Denote the selected top container as c and its origin as stack cs. We first move c to stack s^*, and relocate all the containers above c^* to stack s^*. Then move c^* to stack cs for temporary storage. Next, all the unfixed containers of stack s^* including c are relocated to stack $sn(c^*)$. At last, c^* is moved to stack s^*. Similar to the issue raised in Scenario 3 of Case 1, container c may be a fixed container. If that is the case, we also have to restore c to stack cs.

5.4. Container relocation

Container relocation occurs when a container blocks the target container or it is an unfixed container in the target stack. Generally speaking, we can relocate a container c to any stack, excluding prohibited stacks such as the target stack, the origin of the target container, and sometimes the temporary stack for the target container. Whereas we would like c to be relocated at an appropriate stack, so that it will not block others or be blocked by others in the future. Suppose the target container in process is c^*. Apart from prohibited stacks and full stacks, we find the destination stack for relocating c according to the following preference.

1. A clean stack which can accommodate c. If there are several such stacks, prefer the one whose top container has the smallest group label;
2. A dirty stack whose dirty container with the largest group label $ldg \leq g(c)$. If there are several such stacks, prefer the one with the largest ldg;
3. A dirty stack whose dirty container with the largest group label $g(c) < ldg < g(c^*)$. If there are several such stacks, prefer the one with the smallest ldg;
4. A clean stack whose top container has the largest group label;
5. A dirty stacks whose dirty container with the largest group label $ldg = g(c^*)$.

When the selected destination can accommodate container c, we try **fulfillment** process. Denote the topmost container from the destination stack as dc. We are in hope of finding a top container fc from a dirty stack whose group label falls in the range $g(c) < g(fc) \leq g(dc)$. If there is more than one top container satisfying such criterion, select the one with the largest group label and move it to the destination stack. As the layout changes after one time of fulfillment, we give up continuing relocating c to the destination stack. Rather, we invoke the relocation procedure for c again.

The advantage of fulfillment is that slot space of clean stacks is fully utilized. This avoids the situation where a container with a small group label occupies the top of a stack, blocking the stack from accommodating dirty containers with larger group labels. Fulfillment is extremely efficient at the start of the heuristic when dirty containers with large group labels are quite many.

In our heuristic, every time the fulfillment is invoked, only one movement is carried out. After that the heuristic turns to relocate containers. This is different from that of Expósito-Izquierdo et al. (2012) which carries out movements as many as possible in one fulfillment. We prefer to perform one movement each time because our intention is to relocate c while fulfillment is only an auxiliary operation. Compared to a greedy fulfillment process, we are more eager to find whether a more suitable stack resulting from the fulfillment appears for relocating c. Performing fulfillment as much as possible, in the short run, deviates from our intention of relocating c. Moreover, over-fulfillment may cause the stacks fulfilled being emptied in later stages which brings more movements.

5.5. Termination analysis

Remind in Algorithm 1, we claim that our heuristic fixes containers one by one and therefore it guarantees to find a solution for any feasible instance. We do not mention the fact that a fixed container may be moved when fixing another container. If that situation happens, we can not guarantee that the heuristic will terminate because it may fall into cycling. Fortunately we successfully deal with moved fixed containers. The only chance that a fixed container is moved when fixing other containers is in the situation $nslot = 0$. In other situations, all of the moved containers are unfixed ones. Once a fixed container is moved, we recover it back to its fixed position (the last scenarios of Case 1 and Case 2). Undoubtedly, this recovery action brings more movements. But this avoids the heuristic cycling in vain and makes sure that the heuristic terminates. Besides our work, all the other published work concerning the CPMP fail to guarantee algorithm termination theoretically.

6. Beam search algorithms

Algorithm 1 is just a greedy heuristic, in which the core idea is to fix containers one by one. In that heuristic, how to select target containers and stacks is completely based on the evaluation scheme. Obviously, the precision of the scheme is very important; once it fails, the heuristic will be led to a wrong direction. To avoid this risk, we propose beam search algorithms in this section.

6.1. Beam search with giant moves

The first beam search algorithm maintains a beam of promising layouts and the number of such promising layouts in the beam is a user-defined parameter *bw* (short for beam width). Initially, the beam only contains the initial layout. Then the beam search process extends the layouts in the beam iteratively by giant moves. Hence, we call the algorithm beam search with giant moves (BS-G).

For each layout l in the beam, we perform a probing tree search with two parameters tree width *tw* and tree depth *td* to generate its children. Each child of l is generated by selecting a container-stack pair and conducting a giant move. As the number of container-stack pairs is numerous, we select only a few of them. First only pairs with container part in the candidate container list and stack part in the candidate stack list are considered. The way in which candidate containers and stacks are produced is as mentioned in Section 5.1. All of the candidate pairs are sorted based on the evaluation scheme values in Section 5.2 and the first *tw* pairs are selected; the corresponding giant moves are executed. To this end, we obtain *tw* children of l. Now the tree is at depth 1; l is regarded as at depth 0. The tree expansion continues until it arrives depth *td* with at most tw^{td} layouts. We call the layouts at depth *td* **leaves** of the tree although they are not necessarily clean layouts. Each leaf is further extended by invoking TGH in Section 5 until a clean layout is yielded. We mark

the scores of clean layouts by the number of movements executed from the initial layout. In the end, the score of a child of l (layout at depth 1) is the best score of its corresponding clean layouts.

Every layout in the beam generates at most tw children by a probing tree mentioned above and there are at most bw layouts in the beam. Therefore at most $tw \times bw$ children are generated. Sort them in a descending order of their scores, and select the first bw children into the beam for the next iteration. The pseudo-code is shown in Algorithm 2.

Algorithm 2 Beam search with giant moves for the CPMP/CPMPDS

```
BS-G (inilay)
1  beam = {inilay}
2  while (beam ≠ ∅)
3      canlaylist = ∅
4      for each lay in beam
5          generate up to tw children of lay and append the children to canlaylist
6      sort the layouts in canlaylist in a descending order of scores
7      beam = the first bw layouts in canlaylist
```

6.2. Beam search with baby moves

Inspired by the idea of BS-G, we modify its successor generation method and obtain another algorithm for the CPMP. In this algorithm, beam search is again deployed to maintain the diversity of layouts. The difference from BS-G is that the successors in the beam are generated by applying only a baby move, hence the name beam search with baby moves (BS-B). BS-B overcomes the shortcoming of BS-G that moving too far away (a giant move) each time. Apart from the successor generation method, other components in BS-B is the same with that of BS-G. Now we proceed to introduce how to generate successors in BS-B.

For each layout l in the beam, apply the probing tree search to l just as described in BS-G, generating tw children. The score of a child is the minimum score of the clean layouts expanded from that child. For each child *child* of l, BS-B records the first interim layout on the path from l to *child* and mark the score of this interim layout the same as the score of *child*. If an interim has multiple scores because multiple children have it as the first interim layout, then select the smallest score as its score. The successors of l in BS-B are such interim layouts, instead of the children of l.

6.3. Discussion on the CPMPDS

In this paper, we propose three interrelated algorithms: TGH, BS-G and BS-B. All these algorithms can be applied to solve the CPMP and the CPMPDS instances. Existing algorithms for the CPMP can not be applied directly to solve the CPMPDS instances because they cannot guarantee that the dummy stack is empty when the bay is clean. If we ignore the dummy stack and solve a CPMPDS instance by existing CPMP algorithms, there may not exist a feasible solution.

7. Experiments and analysis

To evaluate the effectiveness of our algorithms, we test our algorithms against the benchmark test data sets provided by Bortfeldt & Forster (2012). We first compare the performance of proposed lower bounds with that of Bortfeldt & Forster (2012). Then we conduct a series of experiments to evaluate the performance of the algorithm components – the evaluation scheme and search parameters. It is followed by the performance on the bench mark data. At the last of this section, a data set for the new CPMPDS is generated and the result is presented for further reference. All of our experiments were conducted on a PC with Intel Core i7 CPU clocked at 3.40 GHz. The operating system is Windows 7.

7.1. Lower bound comparison

Benchmark data are used to compare the performance of LB_{DFS}, LB_{MKM}, and LB_{BF}. In Table 2, columns DFS, MKM, and BF indicate average lower bounds of instances in corresponding data sets calculated by LB_{DFS}, LB_{MKM} and LB_{BF}, receptively. As such, columns tDFS, tMKM and tBF are the average computational time of three methods in seconds, which can be regarded as no difference. Table 2 shows that DFS and MKM have better performance than BF, but the improvement is tiny.

Table 2: Lower bound comparison

data set	DFS	MKM	BF	tDFS	tMKM	tBF
LC	29.780	29.659	29.659	< 1	< 1	< 1
CV	17.725	17.725	17.725	< 1	< 1	< 1
BF	57.239	57.233	57.231	< 1	< 1	< 1

7.2. Parameter configuration

In this section, we describe the experiment that reveals the effectiveness of the evaluation scheme as well as the experiments which show how the parameters affect the algorithm performance. These experiments were preliminary testing we performed in order to determine the final configuration of our BS-G and BS-B.

We select the data set *BF* as our test objective. The data set *BF* is generated by Bortfeldt & Forster (2012) and includes 32 groups of instances, denoted by BF1, . . . , BF32; each group of *BF* data contains 20 instances for a total of 640 *BF* instances. In their instances, the number of stacks is either 16 or 20, and the stack height is either 5 or 8. And they do not have dummy stacks. To avoid over-fitting our algorithms to the benchmark data, we only tested on the first 5 instances of each data group.

We first investigate the effectiveness of different evaluation schemes. As there are other parameters such as the probing tree width and depth, as well as the beam width, we select several combinations of such parameters to make the experiment more robust.

Three kinds of evaluation schemes were tested in our experiment. The first alternative is to evaluate candidate pair by $f(c, s)$ function plus randomness, denoted as `smallF-ran`. When the pairs have the same f function, randomly select one pair. The second alternative (named `smallF-lowT`) prefers small $(f(c, s), h(s) - uf(s))$ which is the one just used in TGH, BS-G and BS-B. The last alternative `smallF-largeS`, as implied by its name, prefers small $f(c, s)$ and then large *sum*. Here *sum* is the sum of group labels of containers above the target container if the target container and stack are within the same stack; or *sum* is the sum of group labels of containers above the target container, in conjunction with those unfixed containers in the target stack.

We tested the three evaluation schemes only against BS-G and set the probing tree width equal to the beam width. The result is shown in Figure 5. The column $w = 5$ & $d = 2$ means the probing tree and beam widths are set 5 while the probing tree depth is set 2. The meanings of other columns are similar. Each bar in the chart is the average movements of the test instances. From the experiment, we can see that `smallF-lowT` has the best performance, `smallF-ran` ranks the second, then `smallF-largeS`. The intention of `smallF-lowT` is to fix target containers lower. And the intention of `smallF-largeS` is to relocate containers with large group labels earlier. As the results with all the parameter combinations show `smallF-lowT` outperforms `smallF-largeS`, the former is indeed better. This can be explained as fixing containers in lower tiers makes the fixed heights among stacks even, namely, containers are fixed, to some extent, tier by tier. This can avoid containers with small group labels occupying short stacks, forcing the slots above them to store only dirty containers. On the contrary, relocating containers with large group labels when there exist larger groups unfixed is not that necessary, or even redundant as the performance is even worse than `smallF-ran`. The finding indirectly reflects that our "fix large group first" strategy is reasonable and effective.

We next consider the impact of the probing width and depth and the beam width on the algorithm performance. In this experiment, the evaluation scheme is set to be `smallF-lowT`. To reduce the number of parameter combinations, we once again set the probing tree width equal to the beam width, varying from 2 to 20. We neglect $w = 1$ because algorithms deteriorate to TGH when the width equals 1. In addition, the computational time increases exponentially when the probing depth tree increases, thus we only set the depth

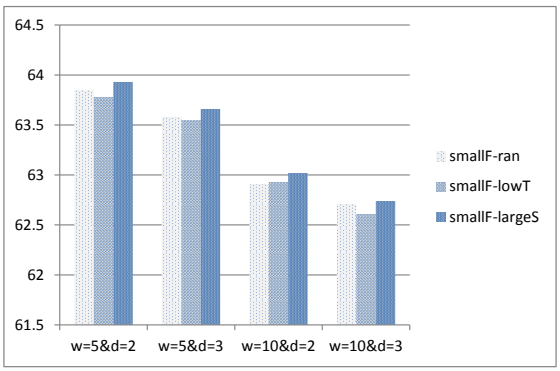
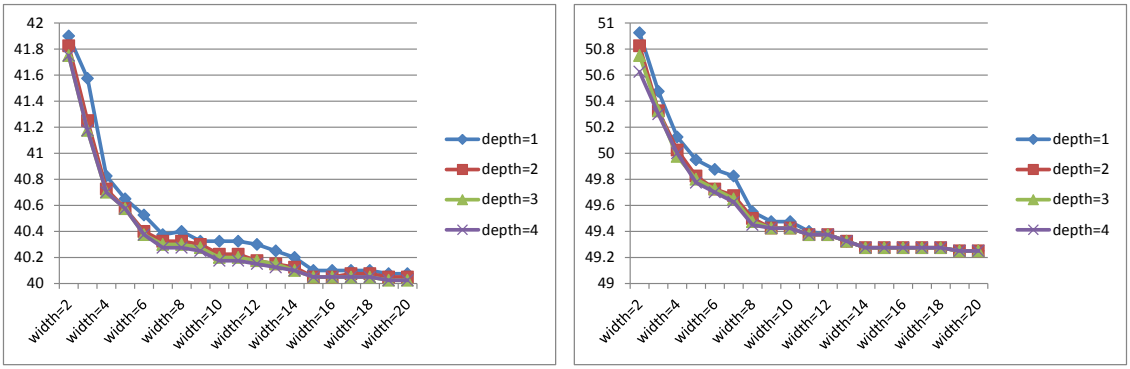


Figure 5: Effectiveness of evaluation schemes

to be 1, 2, 3, and 4. The data set *BF* is classified into four categories based on characteristics: the number of stacks and stack heights (refer to Table 5). The first category is the first eight data groups BF1~BF8 which have 16 stacks and 5 tiers in each instance. Then the second category is BF9~BF16 with 16 stacks and 8 tiers in each instance. In the following, the third category BF17~BF24 has 20 stacks and 5 tiers in each instance. At last, the fourth category BF25~BF32 has 20 stacks and 8 tiers in each instance.

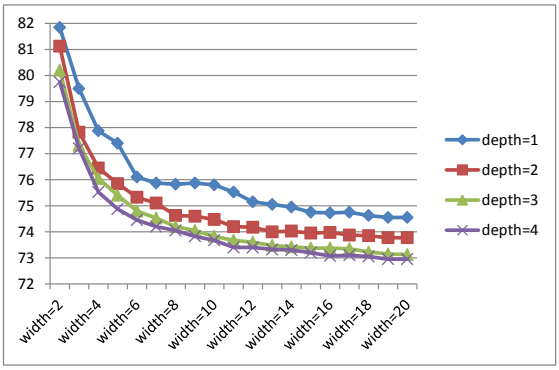
The performance of BS-G on the four categories is separately shown in Figure 6(a)~6(d). In each figure, it shows the average movements of the corresponding data category obtained by BS-G for different widths and depths. The results of four categories in general have the same trend. When the widths and depths increase, the performance increases, but the improvement becomes smaller and smaller, until in the end diminishes. Furthermore, the performance of $d = 2$, $d = 3$ and $d = 4$ are similar, even the same, indicating that our successor generation method is considerably precise. Compare Figure 6(a)~6(b) with 6(c)~6(d), it reveals that the performance difference by different parameter combinations for BF1~BF8 and BF17~BF24 is tiny while the performance difference by different parameter combinations for BF9~BF16 and BF25~BF32 is large. In essence, BF1~BF8 and BF17~BF24 are almost solved to the optimality even with a small depth and width. Remind that BF1~BF8 and BF17~BF24 are with $H = 5$ while BF9~BF16 and BF25~BF32 are with $H = 8$; all the categories have the same usage ratio. It suggests that the height limit affects the instance difficulty more than the number of stacks. This observation is understandable as containers piled vertically are much harder to be retrieved than those piled horizontally.

The performance of BS-B on the four categories are displayed in Figure 7; its trend is similar with that of BS-G in Figure 6, namely the algorithm becomes stable and the improvement is little as parameters enlarge. However, from Figure 7, it can be seen that sometimes the performance goes back when parameters enlarge, especially when the depth is small. For example, in Figure 7(d), the performance when $w = 8$ & $d = 1$ is better than the performance when $w = 10$ & $d = 1$. The explanation is when $d = 1$, the algorithm degenerates to evaluate successors by TGH. The smaller the depth, the weaker ability we have to evaluate

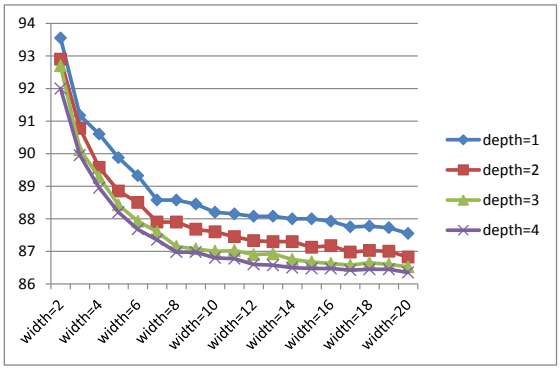


(a) BF1~8 average

(b) BF17~24 average



(c) BF9~16 average



(d) BF25~32 average

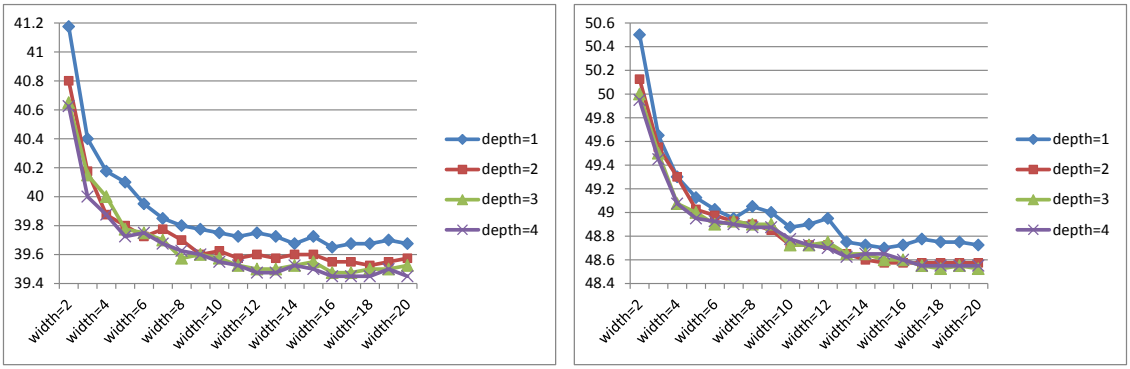
Figure 6: The effect of search depth and width on BS-G

successors, therefore more performance deterioration. When the depth increases, performance deterioration disappears. The conclusion is that the heuristic TGH alone is not enough to precisely evaluate successors, and incorporating probing tree search is effective.

7.3. Comparative analysis with CPMP benchmark data

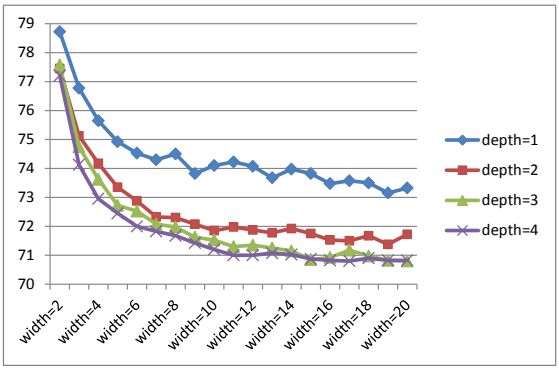
We compare our algorithms with the tree search algorithm of Bortfeldt & Forster (2012) (BF2012 for short) on the CPMP benchmark data. As far as we know, BF2012 has the best performance on the CPMP. The environment of BF2012 is Intel Core 2 Duo processor (P7350, 16 GFLOPS) with 2 GHz and 2 GB RAM under Mac OS X and their code is written in C and compiled with XCode under Mac OS X. Both the parameters of BS-G and BS-B are set $bw = tw = 20$ and $td = 3$, since as shown in Figure 6 and Figure 7, BS-G and BS-B become stable and the improvement is little if further enlarging the parameters. All the results show two decimal places.

To begin with, we show the result on the data set *LC* which is mentioned in Section 7.2 of Bortfeldt & Forster (2012). The data set *LC* is first generated by Lee & Chao (2009), but their concrete data set is not available. For this reason, Bortfeldt & Forster (2012) generate a new set of instances according to the description of Lee & Chao (2009). The result on data set *LC* is displayed in Table 3. The first five columns of Table 3 summarise the data information copied from Bortfeldt & Forster (2012), out of which the dirty ratio is

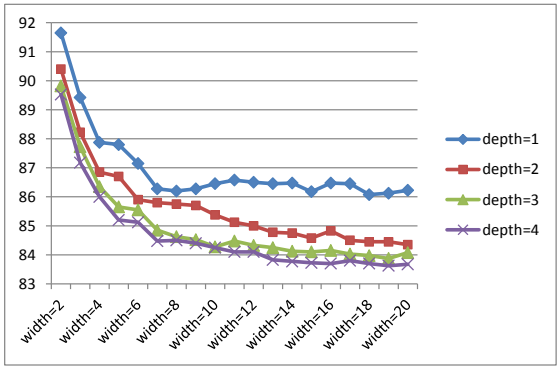


(a) BF1~8 average

(b) BF17~24 average



(c) BF9~16 average



(d) BF25~32 average

Figure 7: The effect of search depth and width on BS-B

the number of dirty containers in the initial layout divided by the number of containers. The column BF2012, TGH, BS-G and BS-B are the average movements on each data group by algorithms BF2012, our TGH, our BS-G and BS-B receptively. The average run time by BF2012, BS-G and BS-B are reported in column tBF2012, tBS-G and tBS-B, in unit of second. As the run time of TGH is less than 0.2s, we do not report its run time in the table. The column LB_{DFS} indicate the average lower bounds on each data group. In the last row of the table, we also give the mean movements on the whole *LC* data. The average time takes cells with "< 1" as 1s and cells with "< 0.1" as 0.1s. We highlight results on which our algorithms outperform BF2012 in bold. Our algorithm TGH and BS-G are worse than BF2012, while BS-B is better than BF2012, but the improvement is little. We explain this phenomenon as the size of the test instances is small and BF2012 has almost solved the instances to the optimality, which can be inferred from the spread between lower bounds and resultant solutions.

Then we compare our algorithms with BF2012 on data set *CV*. The data set *CV* is provided by Caserta & Voß (2009) and by far Bortfeldt & Forster (2012) has the best performance on this data set. Table 4 reports the comparative result on the data set *CV*. It can be seen that both BS-G and BS-B surpass BF2012.

At last we display the results on the data set *BF* by different algorithms in Table 5. The data set *BF* is generated by Bortfeldt & Forster (2012) and they claim that the instances are difficult enough to justify the

Table 3: Algorithms comparison on data set *LC*

group	stack	height	usage rate	dirty ratio	BF2012	tBF2012	TGH	BS-G	tBS-G	BS-B	tBS-B	LB_{DFS}
LC1	10	5	0.7	0.37	17	< 1	22	17	< 0.1	17	< 1	15
LC2a	12	6	0.69	0.38	22.6	1	26.9	23.2	< 0.1	**22.3**	< 1	21.1
LC2b	12	6	0.69	0.7	38.4	2.1	46.7	**38**	< 0.1	**37.9**	1.37	37.3
LC3a	12	6	0.75	0.39	23.7	1.4	29.1	23.9	< 0.1	23.7	< 1	22.3
LC3b	12	6	0.75	0.67	42.7	4.2	54.8	43.6	< 1	**42.3**	10.33	39.9
Ave	11.6	5.8	0.72	0.5	28.88	1.94	35.9	29.14	< 1	**28.64**	2.69	27.12

Table 4: Algorithms comparison on data set *CV*

group	stack	height	usage rate	dirty ratio	BF2012	tBF2012	TGH	BS-G	tBS-G	BS-B	tBS-B	LB_{DFS}
CV1	3	9	0.33	0.6	10.5	< 1	12.1	10.6	< 0.01	**10**	< 0.01	7.9
CV2	4	16	0.25	0.66	19.1	< 1	20.3	**17.8**	< 0.01	**17.2**	< 0.1	13.6
CV3	5	25	0.2	0.7	30.4	9.1	34	**27.3**	< 0.1	**26.4**	< 1	21.3
CV4	6	36	0.17	0.74	44.4	20	50	**37.1**	< 0.1	**35.9**	< 1	28.1
Ave	4.5	21.5	0.24	0.67	26.1	7.78	29.1	**23.2**	< 0.1	**22.38**	< 1	17.73

performance of algorithms. For TGH, its performance is worse than that of BF2012. This is understandable because BF2012 deploys a tree search algorithm whereas TGH is only a greedy heuristic. BS-G outperforms BF2012 by almost 5%, and BS-B outperforms BF2012 by almost 7%. What's more, the computational time of BS-G is less than that of BF2012. Although the computational time of BS-B is much more than that of BF2012 in Table 5, the results are still better than that of BF2012 if we decrease parameters to make run the time of BS-B close to the run time of BF2012.

7.4. New data

We generate a new data set for the CPMPDS. All the data is available at our website[1]. The new data set is composed of 36 groups of instances, with 30 instances in each group. The number of stacks is 6, 8, or 10 (dummy stack included) as in reality the number of stacks in a bay is in such a range. The maximum height limit is 5 or 8 as it is restricted by the handling equipments in practice. There are two alternatives with respect to the usage rate. One alternative is 60%~80%, and another alternative is 80% above. Following the spirit of Bortfeldt & Forster (2012), the number of different group labels in an instance is decided by the group ratio equal to the number of different groups divided by the number of containers. In our data set, the group ratio has three ranges: 100%, 60%~80%, 30%~50%. Particularly, group ratio 100% means that the group labels of containers are unique. The results by TGH, BS-G and BS-B are reported in Table 6.

[1] `http://www.computational-logistics.org/orlib/topic/CPMPDS/index.html`

Table 5: Algorithms comparison on data set *BF*

group	stack	height	usage rate	dirty ratio	BF2012	tBF2012	TGH	BS-G	tBS-G	BS-B	tBS-B	LB_{DFS}
BF1	16	5	0.6	0.6	29.1	< 1	29.1	29.1	< 0.01	29.1	< 0.01	29.1
BF2	16	5	0.6	0.75	36	< 1	36	36	< 0.01	36	< 0.01	36
BF3	16	5	0.6	0.6	29.1	< 1	29.45	29.15	< 0.01	29.1	< 0.1	29.1
BF4	16	5	0.6	0.75	36	< 1	36	36	< 0.01	36	< 0.01	36
BF5	16	5	0.8	0.61	41.6	8	48.25	42.05	< 0.1	**41.35**	3.73	40.75
BF6	16	5	0.8	0.75	49.4	4.9	57.65	51.1	< 0.1	50.15	10.87	48.6
BF7	16	5	0.8	0.61	42.9	26.3	53.5	44.75	< 0.1	43.05	8.95	41.3
BF8	16	5	0.8	0.75	50.6	24.7	60.05	51.95	< 0.1	51.15	15.75	49.05
BF9	16	8	0.6	0.61	51.8	30.8	60.35	**51.5**	< 0.1	**50.4**	6.87	49.35
BF10	16	8	0.6	0.75	59.8	21.9	62.05	**58.9**	< 0.01	**58.75**	2.99	58.4
BF11	16	8	0.6	0.61	52.2	44.6	61.15	52.3	< 0.1	**51.15**	7.75	49.25
BF12	16	8	0.6	0.75	60.2	28.2	63.45	**59.2**	< 0.1	**58.65**	3.47	58.2
BF13	16	8	0.8	0.6	84.6	60	107.45	**79.75**	2.67	**75.4**	167.50	65.95
BF14	16	8	0.8	0.76	105.6	60	125.25	**96.3**	4.62	**93.1**	293.56	81.25
BF15	16	8	0.8	0.6	95.5	60	110.9	**82.8**	2.66	**78.7**	176.51	65.85
BF16	16	8	0.8	0.76	109.8	60	133.3	**98**	4.28	**93.55**	275.25	81.75
BF17	20	5	0.6	0.6	36.3	3.5	36.45	**36.25**	< 0.01	**36.25**	< 0.01	36.25
BF18	20	5	0.6	0.75	45	< 1	45	45	< 0.01	45	< 0.01	45
BF19	20	5	0.6	0.6	36.5	< 1	36.8	36.5	< 0.01	**36.45**	< 0.01	36.45
BF20	20	5	0.6	0.75	45	< 1	45	45	< 0.01	45	< 0.01	45
BF21	20	5	0.8	0.6	51.7	31.5	60.8	52.45	< 0.1	**51.55**	20.31	50.65
BF22	20	5	0.8	0.75	60.9	7.6	68.9	62.85	< 0.1	61.8	14.47	60.65
BF23	20	5	0.8	0.6	51.5	25.5	60.85	52.4	< 0.1	**50.95**	13.90	50.35
BF24	20	5	0.8	0.75	61.3	18.2	70.95	63.25	< 0.1	62.05	24.45	60.65
BF25	20	8	0.6	0.6	62.8	45	69.95	**62.1**	< 0.1	**61.5**	12.79	60.35
BF26	20	8	0.6	0.75	74	30.8	74.35	**72.6**	< 0.1	**72.35**	2.53	72.1
BF27	20	8	0.6	0.6	64	55.9	71.8	**62.75**	< 0.1	**61.85**	16.12	60.35
BF28	20	8	0.6	0.75	74.9	44.7	76.1	**73**	< 0.1	**72.65**	5.82	72.35
BF29	20	8	0.8	0.6	106.6	60	120.55	**96.05**	8.5	**92.05**	396.10	81.6
BF30	20	8	0.8	0.75	128.5	60	143.05	**114.3**	11.4	**110.25**	581.61	99.65
BF31	20	8	0.8	0.6	115.2	60	128.15	**98.9**	9.17	**93.95**	429.09	81.15
BF32	20	8	0.8	0.75	132.3	60	147.3	**115.55**	8.32	**111.8**	612.55	99.2
Ave	18	6.5	0.7	0.68	65.02	29.35	72.81	**62.12**	1.64	**60.66**	96.97	57.24

Table 6: Results for the data set *CPMPDS*

group	stack	tier	usage rate	dirty ratio	TGH	BS-G	tBS-G	BS-B	tBS-B
CPMPDS1	6	5	0.70	0.57	29.33	21.13	< 0.1	19.97	< 1
CPMPDS2	6	5	0.72	0.62	33.10	23.30	< 0.1	21.97	< 1
CPMPDS3	6	5	0.69	0.53	23.87	17.83	< 0.1	17.37	< 1
CPMPDS4	6	5	0.82	0.68	62.13	37.37	< 0.1	34.33	< 1
CPMPDS5	6	5	0.82	0.63	55.03	33.23	< 1	30.43	1.16
CPMPDS6	6	5	0.82	0.61	53.57	30.70	< 1	28.90	1.98
CPMPDS7	6	8	0.68	0.73	72.97	43.73	< 1	41.00	1.23
CPMPDS8	6	8	0.69	0.74	74.47	45.40	< 1	42.27	2.28
CPMPDS9	6	8	0.69	0.73	60.80	41.37	< 1	39.63	2.28
CPMPDS10	6	8	0.80	0.78	142.73	72.00	< 1	66.93	4.05
CPMPDS11	6	8	0.81	0.78	157.57	72.83	< 1	68.63	6.09
CPMPDS12	6	8	0.81	0.75	138.43	66.43	< 1	62.60	11.27
CPMPDS13	8	5	0.71	0.60	32.47	25.30	< 0.1	24.33	< 1
CPMPDS14	8	5	0.72	0.59	33.67	25.93	< 1	25.03	1.17
CPMPDS15	8	5	0.68	0.55	25.40	21.17	< 0.1	20.43	< 1
CPMPDS16	8	5	0.84	0.69	73.50	44.30	< 1	41.50	3.86
CPMPDS17	8	5	0.83	0.64	64.67	39.00	< 1	37.47	4.85
CPMPDS18	8	5	0.83	0.61	53.83	35.30	< 1	33.77	5.23
CPMPDS19	8	8	0.69	0.71	73.20	48.90	< 1	46.30	3.49
CPMPDS20	8	8	0.70	0.72	76.43	51.97	< 1	49.43	9.31
CPMPDS21	8	8	0.69	0.72	70.23	49.10	< 1	47.20	9.88
CPMPDS22	8	8	0.84	0.78	173.37	87.50	1.44	81.70	18.77
CPMPDS23	8	8	0.85	0.78	176.23	90.37	2.31	85.73	28.55
CPMPDS24	8	8	0.84	0.78	153.87	82.07	3.32	78.67	48.31
CPMPDS25	10	5	0.71	0.58	35.77	28.07	< 1	27.17	1.46
CPMPDS26	10	5	0.69	0.58	31.20	25.77	< 1	24.87	1.44
CPMPDS27	10	5	0.69	0.56	30.43	25.23	< 1	24.47	2.19
CPMPDS28	10	5	0.86	0.63	83.30	50.40	1.33	48.20	12.90
CPMPDS29	10	5	0.85	0.64	68.70	46.30	1.18	44.17	13.09
CPMPDS30	10	5	0.86	0.63	71.67	44.90	2.64	42.93	22.31
CPMPDS31	10	8	0.69	0.72	82.07	59.00	< 1	55.87	11.78
CPMPDS32	10	8	0.70	0.71	84.63	58.40	1.12	55.97	19.15
CPMPDS33	10	8	0.70	0.73	77.73	57.50	1.50	55.13	23.76
CPMPDS34	10	8	0.87	0.77	221.83	114.23	5.61	110.37	79.28
CPMPDS35	10	8	0.85	0.77	171.73	97.70	5.49	93.03	79.78
CPMPDS36	10	8	0.85	0.77	172.83	95.90	9.44	91.90	158.40
Ave	8.00	6.50	0.77	0.68	84.52	50.27	1.18	47.77	16.46

8. Conclusions

The container pre-marshalling problem is an important topic at container terminals, and its efficient solution has a significant impact on efficient use of container yards, as well as competition power of container terminals. In particular, the CPMP with a dummy stack has been overlooked by the research community for a long time; whereas in reality terminals using gantry cranes are a lot. In this paper, we considered the classic container pre-marshalling problem. At the same time, we proposed a new variant CPMP with a dummy stack (CPMPDS). A new improved lower bound was devised to compute the number of movements. Three algorithms which guarantee termination were developed based on the idea of fixing containers one by one to solve both the classic CPMP and CPMPDS. Experiments showcase that our algorithms outperform existing methods in terms of the benchmark data of the CPMP, and BS-B can be regarded as the current best algorithm for the CPMP/CPMPDS. In addition, we constructed a new data set of the CPMPDS for future reference .

Through experimental analysis, we find that the idea of fixing containers from larger group labels is right as it narrows the solution space, and provides guidance for the algorithms. The heuristic rules in TGH are also critical as the efficiency of giant moves affects the efficiency of the whole beam search algorithms. The thought of combining giant moves with beam search is also applicable to other complex problems whose solution space are too large to be fully explored.

In the future, we can address which factors affect the difficulty of the CPMP instances and incorporate the CPMP with its upstream and downstream problems, such as stacking problems, cranes scheduling problems and stowage planning problems.

References

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

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

Caserta, M., Schwarze, S., & Voß, S., 2009. A new binary description of the blocks relocation problem and benefits in a look ahead heuristic. In C. Cotta, & P. Cowling (Eds.), *Evolutionary Computation in Combinatorial Optimization* (pp. 37–48). Springer Berlin Heidelberg volume 5482 of *Lecture Notes in Computer Science*.

Caserta, M., Schwarze, S., & Voß, S., 2011a. Container rehandling at maritime container terminals. In J. W. Böse (Ed.), *Handbook of Terminal Planning* (pp. 247–269). Springer New York volume 49 of *Operations Research/Computer Science Interfaces Series*.

Caserta, M., Schwarze, S., & Voß, S., 2012. A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research*, *219*, 96–104.

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

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

Choe, R., Park, T., Oh, M.-S., Kang, J., & Ryu, K. R., 2011. Generating a rehandling-free intra-block remarshaling plan for an automated container yard. *Journal of Intelligent Manufacturing*, *22*, 201–217.

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

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

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

Jiang, X., Lee, L. H., Chew, E. P., Han, Y., & Tan, K. C., 2012. A container yard storage strategy for improving land utilization and operation efficiency in a transshipment hub port. *European Journal of Operational Research*, *221*, 64–73.

Kang, J., Oh, M.-S., Ahn, E. Y., Ryu, K. R., & Kim, K. H., 2006. Planning for intra-block remarshalling in a container terminal. In M. Ali, & R. Dapoigny (Eds.), *Advances in Applied Artificial Intelligence* (pp. 1211–1220). Springer Berlin Heidelberg volume 4031 of *Lecture Notes in Computer Science*.

Kim, K. H., & Bae, J. W., 1998. Re-marshaling export containers in port container terminals. *Computers & Industrial Engineering*, *35*, 655–658.

Kim, K. H., & Hong, G.-P., 2006. A heuristic rule for relocating blocks. *Computers & Operations Research*, *33*, 940–954.

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

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

Lee, Y., & Lee, Y.-J., 2010. A heuristic for retrieving containers from a yard. *Computers & Operations Research*, *37*, 1139–1147.

Park, K., Park, T., & Ryu, K. R., 2009. Planning for remarshaling in an automated container terminal using cooperative coevolutionary algorithms. In *Proceedings of the 2009 ACM Symposium on Applied Computing* SAC '09 (pp. 1098–1105). ACM New York, NY, USA.

Stahlbock, R., & Voß, S., 2008. Operations research at container terminals: a literature update. *OR Spectrum*, *30*, 1–52.

Steenken, D., Voß, S., & Stahlbock, R., 2004. Container terminal operation and operations research - a classification and literature review. *OR Spectrum*, *26*, 3–49.

Vis, I. F. A., & de Koster, R., 2003. Transshipment of containers at a container terminal: an overview. *European Journal of Operational Research*, *147*, 1–16.

Yang, J. H., & Kim, K. H., 2006. A grouped storage method for minimizing relocations in block stacking systems. *Journal of Intelligent Manufacturing*, *17*, 453–463.

Zhu, W., Qin, H., Lim, A., & Zhang, H., 2012. Iterative deepening A* algorithms for the container relocation problem. *IEEE Transactions on Automation Science and Engineering*, *9*, 710–722.