A Branch and Bound Approach for Large Pre-marshalling Problems

Shunji Tanaka[a], Kevin Tierney[b], Consuelo Parreño-Torres[c,*], Ramon Alvarez-Valdes[c], Rubén Ruiz[d]

[a]*Institute for Liberal Arts and Sciences & Department of Electrical Engineering, Kyoto University, Kyotodaigaku-Katsura, Nishikyo-ku, 615-8510 Kyoto, Japan.*
[b]*Decision and Operation Technologies Group, Bielefeld University, Universitätsstraße 25 D-33615, Bielefeld, Germany.*
[c]*Department of Statistics and Operations Research, Valencia University, Doctor Moliner 50, Burjassot, 46100, Valencia, Spain.*
[d]*Grupo de Sistemas de Optimización Aplicada, Instituto Tecnológico de Informática, Ciudad Politécnica de la Innovación, Edificio 8G, Acc. B. Universitat Politècnica de València, Camino de Vera s/n, 46021, València, Spain.*

Abstract

The container pre-marshalling problem involves the sorting of containers in stacks so that there are no blocking containers and retrieval is carried out without additional movements. This sorting process should be carried out in as few container moves as possible. Despite recent advancements in solving real world sized problems to optimality, several classes of pre-marshalling problems remain difficult for exact approaches. We propose a branch and bound algorithm with new components for solving such difficult instances. We strengthen existing lower bounds and introduce two new lower bounds that use a relaxation of the pre-marshalling problem to provide tight bounds in specific situations. We introduce generalized dominance rules that help reduce the search space, and a memoization heuristic that finds feasible solutions quickly. We evaluate our approach on standard benchmarks of pre-marshalling instances, as well as on a new dataset to avoid overfitting to the available data. Overall, our approach optimally solves many more instances than previous work, and finds feasible solutions on nearly every problem it encounters in limited CPU times.

Keywords: container pre-marshalling, maritime applications, terminal operations

1. Introduction

Containerization has revolutionized seaborne trade since it was first introduced over 60 years ago. Nowadays, global containerized trade have reached volumes of around 140 million 20-foot

*Corresponding author

equivalent units (TEUs) and is forecasted to grow by 5% between 2017 and 2022 according to UNCTAD (2017). One of the main challenges faced by world container ports is the pressure to accommodate ever-larger ships, now exceeding 21,000 TEUs. The ships' size, together with their draft restrictions, physical features, and special handling requirements add pressure to berth and crane operations at ports. To quickly service the larger-sized ships, terminal operators use cranes over longer working hours and shifts, which leads to overutilization of port capacity on some days and underutilization on others (Drewry Maritime Research, 2016). The increasing competition between container ports makes improving container handling operations essential.

There are three stages in a container's life cycle within the yard. First, the container is placed on a stack, using assignment strategies, such as stacking containers destined for the same section of a ship together (Dekker et al., 2006). The goal in this stage is to avoid *misoverlaying* containers, which involves blocking the retrieval of containers by placing other containers above them. However, as container yards are dynamic environments with many containers entering and leaving, it is difficult to perfectly sort containers in advance. In the second stage, containers wait for extraction. In this period, the container pre-marshalling problem (CPMP) provides decision support on how to re-order containers that were or have become misoverlaid. In the third stage, containers are extracted.

Pre-marshalling is an important mechanism to prepare containers for their onward journey, since delays of trucks, trains or ships could result in misoverlays even in a previously well-planned layout. Solving the CPMP can soften the workload peaks, increasing the efficiency of retrieval operations. The objective is to compute a minimal sequence of container movements, such that no containers in a set of stacks are misoverlaid. We propose a new branch-and-bound based approach with the following novel components:

1. an extension of the IBF^1 bound from Tanaka and Tierney (2018),
2. two new lower bounds to complement IBF^1,
3. one new dominance rule and generalizations of the rules from Tanaka and Tierney (2018),
4. and a memoization-based heuristic for finding feasible solutions.

We evaluate our approach on standard pre-marshalling benchmarks, and on a new dataset to guard against overfitting our algorithm to the available instances. An extended computational study has been carried out, and it shows the better performance of our algorithm with respect to the state-of-the-art approaches. Our algorithm dramatically reduces the search space, finding 8% more optimal solutions and 6% more feasible solutions in the tested instances. Remarkably, it is able to optimally solve the vast majority of instances tested with 7 tiers and 9-10 stacks, which are considered difficult, real-world CPMP instances.

We first present an overview of the literature in Section 2, followed by a discussion of the CPMP and its assumptions in Section 3. Section 4 introduces the iterative deepening branch and bound algorithm, including new dominance rules. We formalize the new lower bounds in Section 5 and provide computational results in Section 6. In Section 7, we conclude and offer ideas for future work.

2. Literature review

(Re-)Stacking and marshalling problems have been studied extensively in the maritime transportation and warehousing/supply chain literature. Stacking problems are identified and categorized from a mathematical perspective in Lehnfeld and Knust (2014), which gives a wide overview of the area. We discuss first works directly related to pre-marshalling, and then other relevant problems.

Lee and Hsu (2007) present an integer programming model (IP) using a multi-commodity flow network. In de Melo da Silva et al. (2018), a unified model for the pre-marshalling and the container relocation problem (CRP) is proposed. The IP models discretize time and need an upper bound for the number of moves. Parreño-Torres et al. (2018) develop an iterative solution procedure making IP models independent from that bound and presents several alternative formulations for the problem. Even though this approach is not competitive with state-of-the-art techniques for the problem, it offers significantly more flexibility for adding new side constraints. van Brink and van der Zwaan (2014) introduce an abstracted IP model suitable for column generation and apply a branch-and-price technique (BP) to the model.

An A* approach is presented in Expósito-Izquierdo et al. (2012). Its main drawback is high memory usage due to a best-first branching strategy. They also introduce the lowest priority first heuristic (LPFH) for greedily solving the problem. The paper from Tierney et al. (2017) uses an iterative deepening A* (IDA*) that avoids storing solutions in memory, at the expense of extra CPU time spent on the deepening process. In Prandtstetter (2013), a dynamic programming

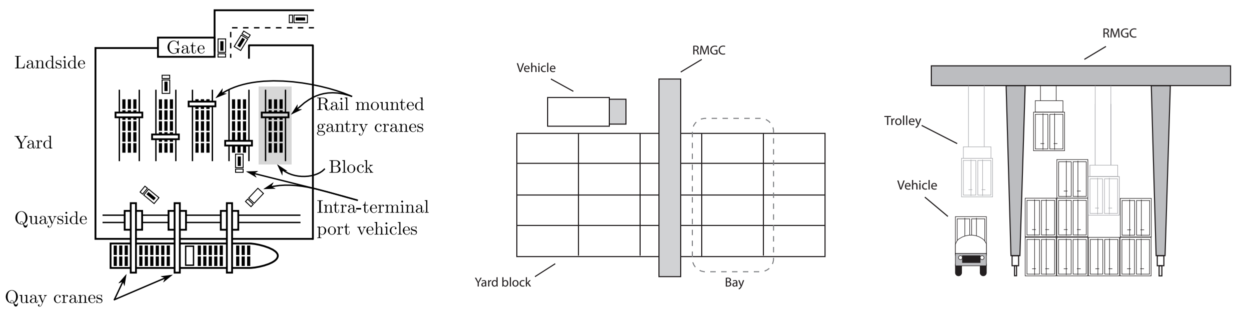
(DP) based branch and bound (BB) search is presented that embeds state memoization within the branch and bound search. Branch-and-bound approaches have been attempted in several works. Zhang et al. (2015) determine which branches are explored first based on a lower bound. A similar strategy is used in Tanaka and Tierney (2018), with a different branching rule, an extra dominance rule, and a new lower bound which extends the lower bound presented in Bortfeldt and Forster (2012).

The most recent heuristic method for solving the pre-marshalling problem is a deep learning tree search (Hottung et al., 2017), which uses a tree search guided with branching and bounding decisions made with deep neural networks. Further heuristic methods include a biased random-key genetic algorithm (Hottung and Tierney, 2016), target-guided approaches (Wang et al., 2015, 2017) and a heuristic tree search from Bortfeldt and Forster (2012). These methods are capable of finding solutions even to extremely large pre-marshalling problems, although we note that our exact method finds optimal solutions to nearly all problems of an industrially relevant size, making heuristic methods for pre-marshalling unnecessary in practice.

There is also extensive literature on related problems. We highlight the Container Relocation Problem (CRP) that just differs from the CPMP in that containers are successively removed from the bay as the CRP is solved (Tanaka and Takii, 2016; Tanaka and Mizuno, 2018). Other related problems are: the loading of vehicles on a "roll-on roll-off" ship (Hansen et al., 2017); the stacking of steel coils (Tang et al., 2015); stack loading problems for train cars and assembly lines (Boysen and Emde, 2016); and the work of Galle et al. (2018) that considers the CRP when deciding where to store containers in the yard. The blocks world problem from the AI community is also similar to the CPMP in that a set of blocks (containers) ought to be manipulated from above without height or stacks number constraints (Slaney and Thiébaux, 2001).

3. Container pre-marshalling

In a container terminal, containers are stored in a buffer area called the *yard*, in which they await onward transportation by truck, train, or ship, as shown in Figure 1a. Within the yard, containers are organized into *blocks* that consist of multiple rows of container *stacks*. Each of these rows is called a *bay*, as Figure 1b shows. A bay contains multiple stacks with a height restriction measured in *tiers* of containers, due to the height of the crane (see Figure 1c).



(a) Top view of a yard port. (b) Top view of an RMGC. (c) Front view of an RMGC.

Figure 1: Overview of a sea container terminal's operations in a yard with rail mounted gantry cranes (RMGCs) and a RMGC over a yard block, from Tierney (2015) and Tierney et al. (2017).

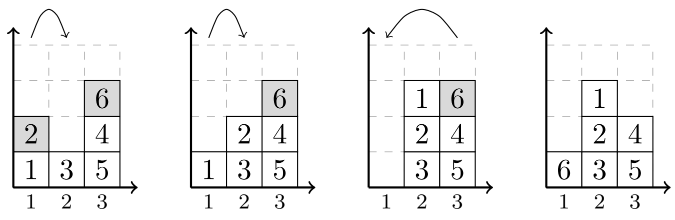


Figure 2: An example sequence of moves to solve the CPMP. Misoverlaying containers are shown in gray.

The objective of the CPMP is to eliminate all *misoverlays*[1] from the bay with a minimal number of container movements. Formally, C is the set of containers placed in the bay, S is the set of stacks of the bay, and T the set of tiers. The non-negative integer-valued function $group(s,t)$ gives the exit time of the container at stack s, tier t. It equals 0 if no container is located at position (s,t). Note that a bay has no misoverlays if $group(s,t) \geq group(s,t+1)$ for all $s \in S, t \in T \setminus \{|T|\}$. We represent a sequence of n moves as $(c_1, s_1, d_1), ..., (c_n, d_n, d_n)$, in which container c_i is moved from stack s_i to stack d_i in move i. Figure 2 shows an example of an optimal solution to the CPMP.

Although the CPMP is now common in the literature, we discuss the assumptions of the problem:

- *Single bay assumption.* The CPMP is solved for a single bay, as it is assumed that moving the crane between bays not only consumes time, but also potentially violates safety constraints due to crane-crane or human-crane interactions.
- *Uniform move cost assumption.* As only the number of moves is considered, all crane moves have an equivalent cost or duration that is not entirely true in practice, as some moves may be

[1] *Misoverlaying* containers are those blocking (i.e., on top of) other containers, *misoverlaid* containers are those containers that are blocked by others. Similarly, *Non-misoverlaying* are those containers that are not blocking any other container, and *non-misoverlaid* containers are not blocked by any container.

between stacks that are far apart.

- *Single crane assumption.* There is just one crane that can only move one container at one time and all containers are of the same type.
- *Full information assumption.* We assume that all the groups of the containers are known in advance. Since delays are common in port operations, in reality the groups of some containers might change, potentially causing new misoverlays even after pre-marshalling.

We adopt the notation of Tanaka and Tierney (2018) and add several new concepts. We employ different names for container moves as in Bortfeldt and Forster (2012). A *bad-good* (BG) move, is a move in which a misoverlaying container is moved to a stack where it is no longer misoverlaying. Similarly, a BB (*bad-bad*) move involves a container being transferred from a misoverlaying position to a stack where it is still misoverlaying. We can thus define GB and GG moves in a similar way. Furthermore, BX and GX refer to moves from a misoverlaid or non-misoverlaid stack, respectively, to any other stack. Table 1 shows the notation used in the following sections.

Table 1: Main notation employed throughout the paper.

S	Set of stacks.
T	Set of tiers.
G	Set of groups.
C	Set of containers.
S^{M}	Set of misoverlaid stacks.
S^{minM}	Set of misoverlaid stacks with the minimum misoverlay height. $S^{\mathrm{minM}} := \{s \in S \mid n_s^{\mathrm{M}} = h^{\mathrm{M}}\}$.
S^{N}	Set of non-misoverlaid stacks.
U	Set of stacks where the misoverlaying containers are "upside down" sorted, i.e., $\mathit{group}(s,t) \leq \mathit{group}(s,t+1)$ for all misoverlaying tiers t. $U := \{s \in S^{\mathrm{M}} \mid n_s^{\mathrm{BG}} = 1\}$
U'	Set of stacks that would be upside down if one misoverlaying container was removed. We assume $U \subset U'$.
n_s	Number of containers in stack s.
n_s^{M}	Number of misoverlaying containers in stack s.
n_s^{N}	Number of non-misoverlaying containers in stack s.
n^{M}	Total number of misoverlaying containers in the bay.
h^{M}	Minimum number of misoverlaying containers in misoverlaid stacks. ($h^{\mathrm{M}} := \min_{i \in S^{\mathrm{M}}} n_s^{\mathrm{M}}$).
g_s^{top}	Group of the topmost container in stack s. If stack s is empty, $g_s^{\mathrm{top}} := \lvert G \rvert$.
g_s^{sec}	Group of the second topmost container in stack s. If $n_s \leq 1$, $g_s^{\mathrm{sec}} := \lvert G \rvert$.
c_s^{top}	Topmost container in stack s. If stack s is empty, $c_s^{\mathrm{top}} = \varnothing$.
m_s	Largest group of the misoverlaying containers in stack $s \in S^{\mathrm{M}}$. If $s \in U$, $m_s = g_s^{\mathrm{top}}$.

m_s^i	The group of the ith largest misoverlaying group value in stack s, e.g., $m_s^1 = m_s$, m_s^2 is the second largest misoverlaying group, etc.
w_s	Smallest group of non-misoverlaying containers in stack s. If stack s is empty, $w_s := \|G\|$. If $s \in S^{\mathrm{N}}$, $w_s = g_s^{\mathrm{top}}$.
w_s^{sec}	Second smallest group of non-misoverlaying containers in stack s. If all the non-misoverlaying containers in stack i have the same group, $w_s^{\mathrm{sec}} := \|G\|$.
g_s^{X}	Minimum top group of the non-misoverlaid stack necessary for repairing stack $s \in U'$ with BG moves to the stack and one BX move to another stack.
n_s^{BG}	Minimum number of non-misoverlaid stacks necessary for repairing stack s with only BG moves.
g_{si}^{BG}	Minimum group of the topmost container of the ith non-misoverlaid stack for repairing stack s. Sorted as $g_{s1}^{\mathrm{BG}} \geq g_{s2}^{\mathrm{BG}} \geq \cdots \geq g_{sn_s^{\mathrm{c}}}^{\mathrm{BG}}$. We assume $g_{si}^{\mathrm{BG}} = 0$ for $i > n_s^{\mathrm{BG}}$.

Minimum/maximum values are computed over the set of stacks U. However, sometimes U is empty and we define $\min_{i \in U}$ ($\max_{i \in U}$) to be ∞ ($-\infty$), respectively. To determine the highest and second highest group values of misoverlaid stacks we do the following. First, we sort the containers from highest to lowest value and then take the first and second containers from this list to be the highest and second highest (lowest), respectively. This means m_s and m_s^2 could contain the same group value if multiple containers have the same value.

4. Improved iterative deepening branch and bound

An iterative deepening branch and bound for the CPMP is proposed in Tanaka and Tierney (2018). We extend this approach by adding tighter lower bounds, a new branching comparison algorithm, new dominance rules, as well as an initial memoization-based heuristic. In this section, we briefly describe the iterative deepening branch and bound approach and the heuristics used during the search. We next present our branching comparison algorithm, we formalize the new dominance rules used in our approach, and, finally, we provide a memoization-based initial upper bound heuristic.

4.1. Algorithm overview

Algorithm 1 shows the pseudocode of our approach. Given an initial layout, *bay*, the algorithm calculates its lower bound, *lb*, and tries to obtain a feasible solution, π, by using the greedy heuristic presented in Tanaka and Tierney (2018). If a feasible solution is obtained, u denotes its number

Algorithm 1 Iterative deepening branch and bound algorithm pseudocode.

```
1: function CPMP-IDBB($bay$)
2:     $lb \leftarrow$ LB($bay$)                                        ▷ Depth lower limit
3:     $u, \pi \leftarrow$ Greedy_Heuristic($bay$)                      ▷ Feasible solution, depth upper limit
4:     if $\pi = \varnothing$ then
5:         $u, \pi \leftarrow$ Memoization_Heuristic($bay$)
6:     $l \leftarrow lb$
7:     $M \leftarrow \varnothing$                                       ▷ Feasible solution to be found in the search
8:     while $l < u$ do
9:         $M \leftarrow$ BB-Recur($M, bay, l, u$)
10:        if $M \neq \varnothing$ then return $M$
11:        $l \leftarrow l + 1$
12:    return $\pi$
```

of moves. In a nutshell, this heuristic attempts to make a solution feasible using only BG and GG moves and stops when this is no longer possible. Although it is not guaranteed to find a solution, it is extremely quick. When a solution is not found, we apply the memoization-based algorithm presented later in Section 4.4 before starting the search.

The iterative deepening method uses a depth-first search (DFS) in which the search depth is limited by a parameter l. The complete tree is searched up to level l, starting at lb and ending at u. The optimality of a solution is guaranteed when l equals u. The function BB-Recur carries out the DFS up to level l. At each node, all possible non-dominated moves are performed and the branches are ordered according to our branching comparison algorithm. A move is dominated when one of the dominance rules is satisfied or the lower bound of the layout is larger than the current depth limit. We try to complete partial solutions by using the greedy heuristic algorithm. In this way, the value u is updated during the search every time a better solution is found. The search ends if a solution of value l is reached.

4.2. Branching comparison algorithm

The order in which the branches are explored is irrelevant for the correctness of the search. Nevertheless, a good branching strategy is critical for obtaining a solution quickly. We consider seven different tie-breaking criteria to order the branches, and they are evaluated in order until the tie is broken.

1. The branch with the lowest lower bound.

2. The largest difference d, given by equation (1):

$$d = \left(\sum_{s \in S^{\mathrm{N}}} g_s^{\mathrm{top}} \cdot (|T| - n_s) \right) - \left(\sum_{s \in S^{\mathrm{M}}} \sum_{1 \leq i \leq n_s^{\mathrm{M}}} m_s^i \right) \tag{1}$$

   On the one hand, the larger the group of the topmost container of a non-misoverlaid stack is, the wider the range of containers that can be well-placed (containers with a group less than or equal to that of the topmost container) above that stack. On the other hand, the lower the group of a misoverlaying container is, the easier it will be to move it to a good position.
3. The lowest sum of the groups of the misoverlaying containers.
4. The largest number of upside down containers that can be well-placed using only BG moves.
5. The largest difference between initial and final gaps. The initial gap of a move is the difference between the group of the topmost container of the origin stack after the move and the group of the container being moved. The final gap is the difference between the group of the container being moved and the group of the top container on the destination stack before the move.
6. The smallest final gap.
7. The largest group of the container being moved.

4.3. New dominance rules

Dominance rules for (*i*) unrelated, (*ii*) transitive, (*iii*) same group symmetry move avoidance, and (*iv*) empty stacks rules are introduced in Tanaka and Tierney (2018). We use rule (*iv*) and propose extensions of the rules (*i*) to (*iii*) by introducing the concept of an *invariant stack*. We also present a new rule that uses the current upper and lower bounds of a node. In this section, n_s refers to the number of containers in stack s at the end of a given move sequence $(c_1, s_1, d_1), (c_2, s_2, d_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1}), (c_n, s_n, d_n)$ and δ is the Kronecker delta: $\delta_{x,y} = 0$ if $x \neq y$ and $\delta_{x,y} = 1$ if $x = y$. The proofs of the propositions appear in Appendix A.

Definition 1. *A stack s is invariant to a sequence of moves $(c_1, s_1, d_1), \ldots, (c_n, s_n, d_n)$, i.e, from move 1 to n, if the layout before and after the sequence of moves is the same and $c_s^{\mathrm{top}} \notin \{c_1, \ldots, c_n\}$, the topmost container of stack s in the layout before the sequence is not moved by the sequence.*

Given stack s being invariant to $(c_1, s_1, d_1), \ldots, (c_n, s_n, d_n)$, the maximum number of temporary containers that are simultaneously assigned to s during the sequence involving from move 1 to n

is:

$$t_s^{1,n} := \max_{j \in \{1,\dots,n\}} \left\{ \sum_{i=1}^{j} (\delta_{s,d_i} - \delta_{s,s_i}) \right\}$$

The example in Figure 3 shows a sequence of moves $(3,1)(3,1)(3,2)(1,3)(1,3)$ for which stack 1 is invariant. It can be seen that $t_1^{1,5} = 2$.

4.3.1. Unrelated move symmetries

We consider the cases in which a move can be indistinctly placed before and after a sequence even though its source or destination stack has been temporarily used in that sequence.

Proposition 1. *A sequence* $(c_1, s_1, d_1), (c_2, s_2, d_2), \dots, (c_{n-1}, s_{n-1}, d_{n-1}), (c_n, s_n, d_n)$ *is disallowed if all of the following conditions are satisfied:*

1. $s_1 > s_n$
2. s_n *and* d_n *are invariant to the sequence* $(c_1, s_1, d_1), (c_2, s_2, d_2), \dots, (c_{n-1}, s_{n-1}, d_{n-1})$.
3. $t_{d_n}^{1,n-1} + n_{d_n} \leq |T|$

Proposition 1 is reduced to the case $s_1, d_1 \notin \{s_2, d_2, \dots, s_n, d_n\}$ in Tanaka and Tierney (2018). Note that condition $s \notin \{s_2, d_2, \dots, s_n, d_n\}$ is a particular case of s invariant to $(c_2, s_2, d_2), \dots, (c_{n-1}, s_{n-1}, d_{n-1}), (c_n, s_n, d_n)$, which satisfies $\sum_{i=2}^{n} \delta_{s_i,s} = 0$. Given the initial layout of Figure 4, a sequence $(7,2,1), (6,2,4), (1,2,3), (6,4,2), (7,1,2), (5,1,4)$ equivalent to $(5,1,4), (7,2,1), (6,2,4), (1,2,3), (6,4,2), (7,1,2)$ could be disallowed.

4.3.2. Transitive move avoidance

This kind of moves considers the relocation of a single container twice, once from one stack a to another stack x, and other from that stack x to another stack c. In some cases, the container

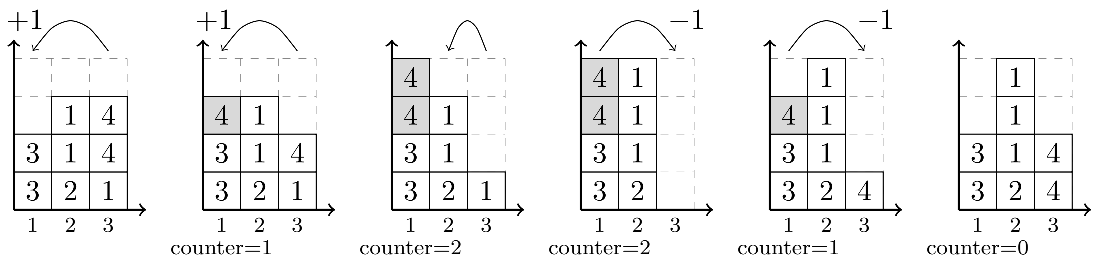


Figure 3: Example in which stack 1 is invariant to the sequence $(3,1)(3,1)(3,2)(1,3)(1,3)$.

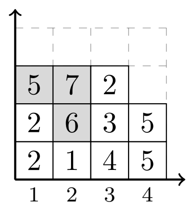


Figure 4: Initial layout for the unrelated move avoidance example.

can be moved directly from a to c. In other cases, there are several options for stack x and all but one of them can be disallowed.

Proposition 2. *A sequence of moves* $(c_1, s_1, d_1), (c_2, s_2, d_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1}), (c_1, d_1, d_n)$ *is disallowed if* $c_1 \notin \{c_2, \ldots c_{n-1}\}$ *and one of the following conditions is satisfied:*

1. *Stack* s_1 *is invariant to the sequence* $(c_2, s_2, d_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1})$ *and* $t_{s_1}^{2,n-1} + n_{s_1} + 1 \leq |T|$.
2. *Stack* d_n *is invariant to the sequence* $(c_2, s_2, d_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1})$ *and* $t_{d_n}^{2,n-1} + n_{d_n} \leq |T|$.
3. *There exists a stack* $s' < d_1$ *invariant to the sequence* $(c_2, s_2, d_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1})$ *that satisfies the condition* $t_{s'}^{2,n-1} + n_{s'} + 1 \leq |T|$.

4.3.3. Same group symmetries

Symmetries appear when two containers of the same group are relocated in two different moves. We identify some cases in which an equivalent sequence is obtained by changing the source stack or the destination stack.

Proposition 3. *A sequence of moves* $(c_1, s_1, d_1), (c_2, s_2, d_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1}), (c_n, s_n, d_n)$ *is disallowed if* $group(c_1) = group(c_n)$ *and one of the following condition is satisfied:*

1. $s_1 = d_n$, $t_{s_1}^{2,n-1} + n_{s_1} \leq |T|$, *and* $\{s_1, s_n\}$ *or* $\{s_1, d_1\}$ *are invariant from move* 2 *to* $(n-1)$.
2. $s_1 > s_n$, $t_{s_1}^{2,n-1} + n_{s_1} \leq |T|$, s_1 *and* s_n *are invariant from move* 2 *to* $(n-1)$.
3. $d_1 > d_n$, $t_{d_n}^{2,n-1} + n_{d_n} \leq |T|$, d_1 *and* d_n *are invariant from move* 2 *to* $(n-1)$.

4.3.4. BB and GB move avoidance

Let $\mathrm{LB}^{\mathrm{BF}}$ be the lower bound proposed by Bortfeldt and Forster (2012), l be the current depth in the algorithm, and $\textsc{Moves}(x)$ be the number of moves to reach node x. We only permit BG and GG moves when $l = \textsc{Moves}(x) + \mathrm{LB}^{\mathrm{BF}}$ and at least one stack is non-misoverlaid. This rule is validated by the fact that $\mathrm{LB}^{\mathrm{BF}}$ assumes only BG and GG moves. It is strengthened by only allowing BG moves if $l = \textsc{Moves}(x) + n^{\mathrm{M}}$ (or BX moves if $l = \textsc{Moves}(x) + n^{\mathrm{M}} + h^{\mathrm{M}}$ and all stacks in x are misloverlaid).

We also implement a tie-breaking mechanism within this rule. If a container can be moved by a BG move to either one of two non-misoverlaid stacks with the same height, we choose the one with the smaller topmost group. Further ties are broken by the stack number. Since this dominance rule conflicts with (3) of Proposition 3, we modify (3) so that it is applied only when the heights of stacks d_1 and d_n are different, at least one of the stacks d_1 and d_n is misoverlaid, or the group of the topmost non-misoverlaying container of stack d_1 is larger than or equal to that of stack d_n.

4.4. Memoization-based upper bound heuristic

In the definition of the CPMP, stacks are essentially interchangeable, i.e., given a layout of the bay, only the order in which containers are stacked vertically matters, and not the order of the stacks in the bay. For each arrangement of the bay, there can be up to $S!$ equivalent layouts. The idea of rearranging the bay to create an abstract configuration is proposed by Ku and Arthanari (2016) for the CRP and also used in Quispe et al. (2018) for the same problem. We build on this idea to create an efficient and simple upper bound heuristic.

We design a hash function that allows us to detect layouts corresponding to abstract configurations already visited during the search. First, nonempty stacks are ordered by non-increasing number of containers. In case of a tie, they are examined from top to bottom until a container of a different group is found at the same tier. In this case, the one with the higher group is ordered first. If two stacks have the same layout, it does not matter which one is ordered first. The key concatenates the groups of the containers of each stack in the ordered list from bottom to top.

Since the use of the memoization strategy during the complete iterative deepening has a very high computational cost, we use it just to obtain a feasible solution. The algorithm starts at the initial configuration, saving it in the hash table. It then creates all of the initial solution's children, pruning those already saved in the hash table and any nodes removed by our dominance rules. Just the best node according to the branching comparison algorithm is saved and the process is repeated from this node until a feasible solution is found or a maximum fixed level is reached.

5. Extended lower bound

In this section we describe our new bounds. First, we introduce $\mathrm{LB}^{\mathrm{BF}}$ proposed by Bortfeldt and Forster (2012), on which all the other bounds are based, and IBF^1 by Tanaka and Tierney (2018), which we also extend. Finally, we present our proposed lower bounds IBF^2, IBF^3, and IBF^4. The proofs of the propositions appear in Appendix B.

5.1. The Bortfeldt and Forster lower bound

The $\mathrm{LB}^{\mathrm{BF}}$ from Bortfeldt and Forster (2012) is formalized as $\mathrm{LB}^{\mathrm{BF}} := n_{\mathrm{BX}} + n_{\mathrm{GX}}$ where

$$n_{\mathrm{BX}} := \begin{cases} n^{\mathrm{M}} + h^{\mathrm{M}} & \text{if all stacks are misoverlaid,} \\ n^{\mathrm{M}} & \text{otherwise,} \end{cases}$$

and n_{GX} puts a lower bound on the number of non-misoverlaying containers that must be moved. First, a slot (s,t) is a *potential supply slot* if $t > n_s^{\mathrm{N}} > 0$ and $w_s = g$. Let $s^{\mathrm{P}}(g)$ be the number of potential supply slots of group g, and $S^{\mathrm{P}}(g) = \sum_{g' \in G} s^{\mathrm{P}}(g') + |T| n_{\mathrm{E}}$ be the cumulative potential supply of group g, where n_{E} is the number of empty stacks. The *demand* of group g is given by $d(g)$, which is the number of misoverlaying containers of group g. Furthermore, $D(g) = \sum_{g' \in G} d(g')$ is the *cumulative demand* of g. The *cumulative demand surplus* for group g is given by $D^{\mathrm{S}}(g) = D(g) - S^{\mathrm{P}}(g)$.

The item group $g^* = \operatorname{argmax}_g D^{\mathrm{S}}(g)$ is the item group with the largest cumulative demand surplus. All stacks s satisfying $w_s < g^*$ are called *potential GX* stacks, and there are $n_{\mathrm{GX}}^{\mathrm{PS}}$ of them. Assume $n_{\mathrm{GX}}^{\mathrm{PS}} \geq n_{\mathrm{GX}}^{\mathrm{S}} = \max(0, \lceil D^{\mathrm{S}}(g^*)/|T| \rceil$. The stacks are now sorted such that potential GX stacks come first and are then sorted according to the value $n^{g^*}(s)$ ascending, where $n^g(s)$ is the number of non-misoverlaying containers with a group $g' < g$ in stack s. Finally, we define $n_{\mathrm{GX}} := \sum_{s=1}^{n_{\mathrm{GX}}^{\mathrm{S}}} n^{g^*}(s)$ and get a lower bound on the number of GX moves. We refer to Bortfeldt and Forster (2012), for a proof of correctness of the bound.

5.2. IBF^1 lower bound

Tanaka and Tierney (2018) introduce the lower bounds IBF^0 and IBF^1. When non-full stacks are all misoverlaid, IBF^0 increases $\mathrm{LB}^{\mathrm{BF}}$ by their minimum misoverlay height. Moreover, IBF^1 increases IBF^0 by 1 in the following cases.

Case 1: All stacks are misoverlaid.

Case 2: One stack is not misoverlaid and it is not full (2a), or it is full (2b).

Case 3: Two stacks are not misoverlaid and at least one is not full.

Case 4: $n_{\mathrm{GX}} = 0$ and only one non-misoverlaid stack is not full.

5.3. Extended lower bound IBF^2

We prove that IBF^0 can be improved not by 1, but by 2 under some conditions in Cases 1, 2b, 3, and 4. This lower bound is denoted by IBF^2.

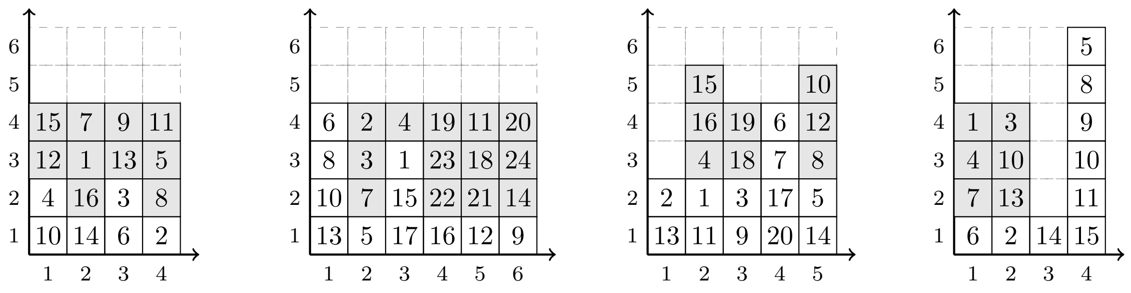
5.3.1. Extension of IBF^1, case 1: All stacks are misoverlaid

The bound IBF^1 improves IBF^0 by 1 through consideration of the moves required to make two stacks non-misoverlaid. Let IBF^1 := IBF$^0 + 1$ if it is impossible to repair a stack with the minimum number of misoverlaying containers (h^{M}) only using BB moves and then another stack only using BG moves. We now show that in certain situations we can further improve IBF^1 by 1. This is done by checking whether the second stack can be repaired under the assumption that a single additional GB or BB move is permitted.

Proposition 4. IBF^2 := IBF^1 + 1 *is a valid lower bound for the CPMP if all of the following conditions are true:*

1. *All stacks are misoverlaid*
2. IBF^1 := IBF$^0 + 1$
3. $\min\limits_{s'_1 \in U} m_{s'_1} > \max\limits_{s_1 \in \{s \in S \mid n^{\mathrm{M}}_s = h^{\mathrm{M}}+1\}} w_{s_1}$
4. $\min\limits_{s'_1 \in U} m_{s'_1} > \max\limits_{s_2 \in S^{\mathrm{minM}}} w^{\mathrm{sec}}_{s_2}$
5. $\min\limits_{s'_2 \in U'} g^{\mathrm{X}}_{s'_2} > \max\limits_{s_2 \in S^{\mathrm{minM}}} w_{s_2}$

Example 1. In this example, LB$^{\mathrm{BF}}$ = 13 ($n_{\mathrm{BX}} = 10 + 2 = 12$, $n_{\mathrm{GX}} = 1$) and IBF^1 = 14 for the CV4-4-14 instance in Figure 5a. Consider now solving the given bay in 14 moves. First, examine the option in which stack 2 (or 4) is repaired first, and then another stack is repaired only by BG moves. The next stack to repair would be stack 1, since it is upside down. However, the topmost container with value 15 is larger than the topmost non-misoverlaying container in stack 2 with a value of 14. Thus, only BG moves will not suffice. Repairing stack 4 first leads to a similar situation. Alternatively, consider repairing stack 1 (or 3) first. Then, the topmost container of



(a) CV4-4-14, Case 1. (b) CV-4-6-20, Case 2b. (c) CV-4-5-16, Case 3 (d) Case 4

Figure 5: Example instances showing the lower bound improvements, IBF^2.

stack 1 (resp. 3) is moved by a GB move and another stack must be repaired using only BG moves. Even if we move the container with group 4 from stack 1 with a GB move, we still cannot repair any other stack with only BG moves, as only stack 1 is upside down. Repairing stack 3 first also does not work, since even after removing container 3, the containers from stack 1 cannot be moved on top of container 6 without creating additional misoverlays. A final option to achieve a 14 move solution would be as follows. After repairing stack 1 (or 3), another stack is repaired with one BB move and BG moves. If we consider using BG moves for containers 7 and 1 from stack 2 to 1, the stack is upside down if we then use the BB move for container 16. However, $7 > 4$, thus we cause a misoverlay in stack 1. The same argument is valid for stacks 3 and 4. Similar arguments can be made when stack 3 is repaired first, thus $\text{IBF}^2 = 15$ is a valid lower bound.

5.3.2. Extension of IBF^1, case 2b: Only one stack is non-misoverlaid and it is not full

The bound $\text{IBF}^1 = \text{IBF}^0 + 1$ is valid when exactly one stack s is non-misoverlaid and not full, and $\min_{s' \in U} m_{s'} > w_s$. Note how this bound only examines the top container of the upside down stack and the non-misoverlaid stack. If we dig one container deeper, in certain situations we can improve IBF^1 by one move. There are two situations to consider, namely when $\text{IBF}^1 = \text{IBF}^0 + 1$ and when $\text{IBF}^1 = \text{IBF}^0$.

Proposition 5. $\text{IBF}^2 := \text{IBF}^1 + 1$ *is a valid lower bound for the CPMP if all of the following are true:*

1. *Only one stack s is non-misoverlaid and it is not full*
2. $\text{IBF}^1 = \text{IBF}^0 + 1$
3. $\min\limits_{s' \in U} m_{s'} > w_s^{\text{sec}}$

4. $\min_{s'' \in U'} g^{\mathrm{X}}_{s''} > w_s$

Proposition 6. $\mathrm{IBF}^2 := \mathrm{IBF}^1 + 1$ *is a valid lower bound for the CPMP if all of the following are true:*

1. *Only one stack s is non-misoverlaid and it is not full*
2. *Only one stack s' is upside down and $g^{\mathrm{top}}_{s'} \le w_s$.*
3. $\mathrm{IBF}^1 = \mathrm{IBF}^0$
4. $\forall s'' \in S^{\mathrm{M}} \setminus \{s'\}$, $n^{\mathrm{BG}}_{s''} > 2 \vee g^{\mathrm{BG}}_{s''1} > \max\{w'_s, w'_{s'}\} \vee g^{\mathrm{BG}}_{s''2} > \min\{w'_s, w'_{s'}\}$*, holds, where*

$$w'_{s'} := \begin{cases} w^{\mathrm{sec}}_{s'} & n_{\mathrm{GX}} > 0 \wedge n_s + n^{\mathrm{M}}_{s'} < |T|, \\ w_{s'} & \textit{otherwise.} \end{cases}$$

Example 2. Figure 5b shows an example instance where IBF^2 strengthens the lower bound. Stack 1 is (clearly) the non-misoverlaid, non-full stack, s. Stack 3 is the upside down stack s' that can be placed on s with only BG moves. The lower bound is $\mathrm{IBF}^1 = \mathrm{LB}^{\mathrm{BF}} = 15$ ($n_{\mathrm{BX}} = 13, n_{\mathrm{GX}} = 2$). Since $n_{\mathrm{GX}} > 0$, we can move the container with group 1 from stack 3 to stack 1. We now attempt to repair the other stacks with BG moves. Since three misoverlaying containers in stack 2 or 5 are in nondecreasing order of groups from top to bottom, at least three non-misoverlaid stacks are required for repairing them ($n^{\mathrm{BG}}_{s''} = 3$). Stack 4 needs two stacks that can accept its containers with groups 23 and 19 ($g^{\mathrm{BG}}_{41} = 23$ and $g^{\mathrm{BG}}_{42} = 19$). Similarly, stack 6 needs two non-misoverlaying stacks with topmost groups of at least 24 and 20. Thus, we cannot repair stacks 2, 4, 5 and 6, and the bound $\mathrm{IBF}^2 := \mathrm{IBF}^1 + 1 = 16$ is obtained.

5.3.3. Extension of IBF^1, case 3: Two stacks are non-misoverlaid, at least one is not full

Let the two non-misoverlaid stacks be stacks s_1 and s_2 and assume that $w_{s_1} \le w_{s_2}$ without loss of generality. Let $\mathrm{IBF}^1 := \mathrm{IBF}^0 + 1$ if we cannot repair any stack only by GG and BG moves.

IBF^2 tries to increase IBF^1 by 1 if $\mathrm{IBF}^1 = \mathrm{IBF}^0$. Let w''_{s_1} and w''_{s_2} be the topmost non-misoverlaying containers in stacks s_1 and s_2, respectively, after GG moves between them. If $n_{\mathrm{GX}} = 0$, no GG moves are permitted, so that $w''_{s_1} := w_{s_1}$ and $w''_{s_2} := w_{s_2}$. If $n_{\mathrm{GX}} = 0$ and either stack s_1 or s_2 is full, we consider the lower bound in case 4 rather than here in case 3. When $n_{\mathrm{GX}} > 0$, the following situations set the values of w''_{s_1} and w''_{s_2}

(S3.1) If $w_{s_1} < w_{s_2}$ and stack s_2 is full ($n_{s_2} = |T|$), then no GG moves are possible, only stack s_1 is available for repairing, and $w''_{s_1} = w_{s_1}, w''_{s_2} = -\infty$.

(S3.2) Otherwise.

$$w''_{s_1} := \begin{cases} w^{\text{sec}}_{s_1} & n_{s_2} < |T|, \\ w_{s_1} & \text{otherwise,} \end{cases} \qquad w''_{s_2} := \begin{cases} w^{\text{sec}}_{s_2} & n_{s_1} < |T| \land w_{s_1} = w_{s_2}, \\ w_{s_2} & \text{otherwise.} \end{cases} \tag{2}$$

Proposition 7. $\text{IBF}^2 := \text{IBF}^1 + 1$ *is a valid lower bound for the CPMP if all the following are true:*

1. *Two stacks are non-misoverlaid and at least one of these is not full*
2. $\text{IBF}^1 = \text{IBF}^0$
3. $\forall s' \in S^{\text{M}}$, $n^{\text{BG}}_{s'} > 2 \lor g^{\text{BG}}_{s'1} > \max\{w''_{s_1}, w''_{s_2}\} \lor g^{\text{BG}}_{s'2} > \min\{w''_{s_1}, w''_{s_2}\}$ *holds with values* w''_{s_1} *and* w''_{s_2} *set according to situations (S3.1) and (S3.2).*

Example 3. The instance in Figure 5c has $\text{IBF}^1 = \text{IBF}^0 = \text{LB}^{\text{BF}} = 9$ ($n_{\text{BX}} = 8$, $n_{\text{GX}} = 1$). According to our definition, $s_1 = 1$ and $s_2 = 4$. As $w_{s_1} < w_{s_2}$ and stack s_2 is not full, (S3.1) does not hold and we apply (S3.2) so the topmost groups are 13 and 6. We try to repair stacks 2, 3, or 5 with BG moves to stacks 1 or 4. Stacks 2 and 3 cannot be repaired as groups of the misoverlaying containers are larger than those of s_1 and s_2. Analyzing stack 5, despite container 12 not causing a problem ($g^{\text{BG}}_{51} = 12 \leq \max\{c_{s_1}, c_{s_2}\} = 13$), the top group of stack 3 is not large enough to support container 10 ($g^{\text{BG}}_{52} = 10 > \min\{c_{s_1}, c_{s_2}\} = 6$). Thus, no stack can be repaired and we obtain $\text{IBF}^2 := 9 + 1 = 10$.

5.3.4. Extension of IBF^1, case 4: $n_{\text{GX}} = 0$ and only one non-misoverlaid stack is not full

When $\text{IBF}^1 = \text{IBF}^0 + 1$, we further try to increase IBF^1 by 1. Let s be the non-full, non-misoverlaid stack, and s' the stack to be repaired. Since $n_{\text{GX}} = 0$, we only need to consider one GG or BB move in addition to $n_{\text{BX}}(= \text{IBF}^0)$ BG moves. There are two situations to consider, namely (S4.1) when we perform a GG move from a full, non-misoverlaid stack s'' to stack s and stack s' is repaired with only BG moves to s and s'', or (S4.2) when stack s' is repaired by BG moves with exactly one BB move before, during or after the BG move sequence. Excluding these two situations, $\text{IBF}^2 := \text{IBF}^1 + 1$.

Proposition 8. $\text{IBF}^2 := \text{IBF}^1 + 1$ *is a valid lower bound for the CPMP if all the following are true:*

1. Only one non-misoverlaid stack is not full

2. $n_{\mathrm{GX}} = 0$

3. $\mathrm{IBF}^1 = \mathrm{IBF}^0 + 1$

4. $\min_{s' \in U'} g^{\mathrm{X}}_{s'} > w_s$

Example 4. Figure 5d shows an example in which $\mathrm{IBF}^0 = \mathrm{LB}^{\mathrm{BF}} = 6$ ($n_{\mathrm{BX}} = 6$, $n_{\mathrm{GX}} = 0$) and $\mathrm{IBF}^1 = 7$. As neither stack 1 nor 2 becomes upside down by removing only one misoverlaying container, we cannot arrange the bay only with one BB move and BG moves. Therefore, $\mathrm{IBF}^2 = \mathrm{IBF}^1 + 1 = 8$.

5.4. Extended lower bound IBF^3

We now change our focus to dealing with cases where even IBF^2 cannot improve the $\mathrm{LB}^{\mathrm{BF}}$, i.e., $\mathrm{IBF}^2 = \mathrm{LB}^{\mathrm{BF}}$. If at least one non-misoverlaid stack is not full, IBF^3 tries to repair each stack in S^{M} with only BG moves to the stacks in S^{N}, without considering stack height limits. If $n_{\mathrm{GX}} > 0$, at best these *GX* moves empty $n^{\mathrm{S}}_{\mathrm{GX}}$ stacks. Considering this, we add $n^{\mathrm{S}}_{\mathrm{GX}}$ dummy empty stacks to S^{N}. If a misoverlaid stack can be repaired, all of its misoverlaying containers are removed from the bay, and it is moved from S^{M} to S^{N} for further repairs. When this process fails to repair all the stacks, at least one extra move will be necessary and we obtain $\mathrm{IBF}^3 := \mathrm{LB}^{\mathrm{BF}} + 1$. To speed up the process, we assume that at most three non-misoverlaid stacks are necessary for repairing a stack, even when more than three are required.

Proposition 9. IBF^3 *is a valid lower bound for the CPMP if* $\mathrm{IBF}^2 = \mathrm{LB}^{\mathrm{BF}}$.

5.5. Extended lower bound IBF^4

When IBF^3 fails to improve the lower bound over $\mathrm{LB}^{\mathrm{BF}}$, IBF^4 offers one last chance for improving the bound by one move. Recall that in $\mathrm{LB}^{\mathrm{BF}}$ the supply and demand of each group is computed. These values try to quantify how many places are available to accept containers of a particular group or are required for containers of a group, respectively. This bound takes advantage of the situation when at some $\hat{g}$ the cumulative demand surplus $D^{\mathrm{S}}(\hat{g}) = D(\hat{g}) - S^{\mathrm{P}}(\hat{g}) = 0$. When supply and demand are balanced at $\hat{g}$, it means that the demand with group $g < \hat{g}$ cannot be assigned to any supply of group $g \geq \hat{g}$. The groups are thus partitioned by balanced supply and demand and we examine if containers providing the demand can be moved to stacks providing supply using only BG moves. In certain situations in which this is not possible, $\mathrm{IBF}^4 = \mathrm{IBF}^3 + 1$.

We introduce some setup information and several terms to help us describe the IBF^4 bound. First, when $n_{\mathrm{GX}} > 0$, assume each $n^{\mathrm{S}}_{\mathrm{GX}}$ stack provides $|T|$ supply for the largest group. Let $[\check{g}^k, \hat{g}^k]$ be the kth subset of groups partitioned in the above manner, so that $D^{\mathrm{S}}(\hat{g}^k + 1) = 0$ holds. For each $\bar{g}$ satisfying $\check{g}^k \leq \bar{g} \leq \hat{g}^k$, we only consider from now on the demand and supply of groups g such that $\bar{g} \leq g \leq \hat{g}^k$. We refer to a stack having one or more containers with demand as a demand stack. Formally, a stack s is a demand stack if $\bar{g} \leq m^i_s \leq \hat{g}^k$ holds for some i. A stack that provides supply is referred to as a supply stack: A supply stack s' satisfies $\bar{g} \leq w_{s'} \leq \hat{g}^k$. We call a stack "*dirty*" when it is a demand and supply stack, and the demand and supply from a dirty stack *dirty demand* and *dirty supply*, respectively. The total number of dirty demands in a dirty stack with groups g satisfying $\bar{g} \leq g \leq \hat{g}^k$ is referred to as the *dirty height.* We refer to a demand stack and a supply stack as being *clean* when they are not dirty, and their demand and supply as *clean demand* and *clean supply.* If one of the following conditions is satisfied, IBF^3=LB$^{\mathrm{BF}}$ can be improved by 1:

1. The minimum dirty height is larger than the clean supply.
2. There is only one *clean supply stack*, there are no *dirty stacks*, and there exists a *clean demand stack* such that the group of the topmost demand container is smaller than the maximum group of the demand containers in that stack.
3. There is only one *clean supply stack* and the maximum group of the dirty demand containers is larger than the topmost dirty demand containers and the topmost non-misoverlaying containers in *dirty stacks.*

Proposition 10. *IBF^4 is a valid lower bound for the CPMP if* IBF^3 $=$ LB$^{\mathrm{BF}}$*.*

6. Computational experiments

We carry out an extensive computational analysis to assess the performance of the algorithm proposed for the CPMP, which we refer to as PT$^{\mathrm{A}}$. We test our approach and several algorithm variants that arise from different combinations of the contributions presented in this study, see Table 2. The letters U, T, G, and XB, refer to unrelated move, transitive move, same group, and BB/GB move avoidance dominance rules, respectively. "Gen." refers to the generalized rules and BCA^1 and BCA^2 to the branching comparison algorithm presented in Tanaka and Tierney (2018) and our modification in this work, respectively. These novel approaches are evaluated against the state-of-art exact algorithm from Tanaka and Tierney (2018), and obtain a significantly larger number of optimal solutions as well as feasible solutions of unsolved instances. We focus on answering the following research questions:

1. What effect do the improved lower bounds have on the performance of PT^{A}?
2. What effect do the dominance rules have on the performance of PT^{A}?
3. What is the effect of the new branching strategy?
4. Do the benefits of using the memoization-based upper bound heuristic outweigh the time invested?
5. Does PT^{A} improve the state-of-the-art exact algorithms?
6. Are we overfitting the well-known datasets?

We conduct all computational experiments on Intel Xeon X5650 CPUs at 2.67 GHz running CentOS 6.6, with each execution of the algorithms being given a maximum of 3 GB of RAM and an hour of CPU time.

Table 2: An overview of the algorithms directly compared in this section.

Algorithm	Lower bound	Branching	Dominance rules	Hash table
PT^{A}	IBF^2, IBF^3, IBF^4	BCA^2	Gen.(T, U, G), XB	Initial UB
PT^{D}	IBF^2, IBF^3, IBF^4	BCA^2	Gen.(T, U, G), XB	-
PT^{4+}	IBF^2, IBF^3, IBF^4	BCA^2	T, U, G	-
PT^{4}	IBF^2, IBF^3, IBF^4	BCA^1	T, U, G	-
PT^{3}	IBF^2, IBF^3	BCA^1	T, U, G	-
PT^{2}	IBF^2	BCA^1	T, U, G	-
TT^{A} (Tanaka and Tierney, 2018)	IBF^1	BCA^1	T, U, G	-
IDA* (Tierney et al., 2017)	$\mathrm{LB}^{\mathrm{BF}}$	Left	T, U	-

6.1. Test instances

We focus on instance sets from Bortfeldt and Forster (2012) (BF dataset) and Caserta and Voß (2009) (CV dataset) as they contain the most difficult instances in the literature. In Tanaka and Tierney (2018) three other datasets were used from van Brink and van der Zwaan (2014) (BZ dataset), from Expósito-Izquierdo et al. (2012) (EMM dataset), and from Zhang et al. (2015) (ZJY dataset); however, these instances are all solved to optimality in little runtime by TT^{A}, so we do not consider them again here. Finally, the CPMP has been thoroughly studied, and most of the state-of-art exact algorithms use the previous datasets to test their performance. Avoiding overfitting of the studied methods on these instances is a crucial concern. To ensure that we are not overfitting, we also consider a fresh dataset, CVX, from Hottung et al. (2017). Experimental results on this dataset are conducted at the end of the section, showing that our approach also performs well on new data and definitely does not overfit. We now briefly describe the datasets.

CV dataset. In these instances, all containers have different groups and stacks are filled to the same height. The instances range in size from 3 tiers by 3 stacks up to 6 tiers by 10 stacks full of containers. Two empty tiers are added above the top of each stack.

BF dataset. This dataset is made up of two groups (BF and LC). BF contains 32 categories, having either 16 or 20 stacks with a height of either 8 or 5 tiers. LC is divided into 5 categories whose instances are based on the instances by Lee and Chao (2009). They have either 10 or 12 stacks and either 5 or 6 tiers. Since the load capacity of these instances is under 100%, it is not necessary to add empty tiers as in the CV dataset. Moreover, unlike the CV dataset, there can be multiple containers of the same group.

CVX dataset. There are three groups of instances in the CVX dataset: CVX1, CVX2, and CVX3. Instances in CVX1 follow the structure of CV (all containers have different groups and stacks are filled to the same height). Stacks are also filled to the same height in CVX2 and CVX3, but in these cases, there are two and three containers of each group, respectively. The instances in CVX1-CVX3 range in size from 5 tiers by 7 stacks up to 5 tiers by 10 stacks full of containers, with two extra empty tiers are added as in the CV dataset.

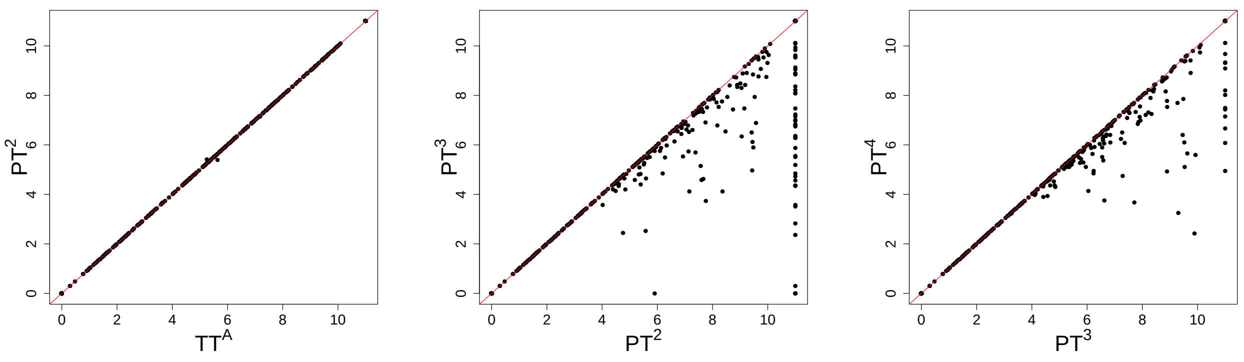
6.2. Improved lower bounds IBF^2, IBF^3, IBF^4

We evaluate the effectiveness of the improved lower bounds, IBF^2, IBF^3, and IBF^4, presented in this work. Since IBF^3 tries to improve the bound only if IBF^2 fails, and IBF^4, in turn, only tries to improve the bound only if IBF^3 fails, we assess their incremental effectiveness.

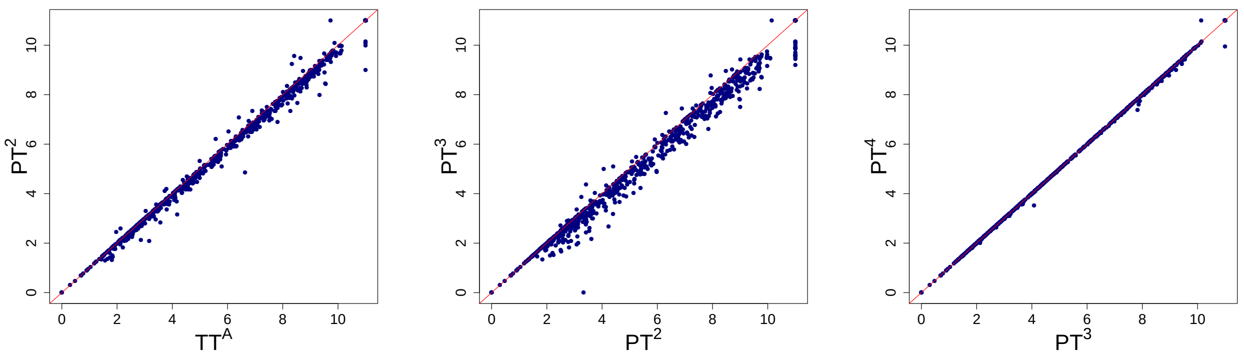
Root node analysis. Given the strength of IBF^1, it is not surprising that the benefits of the improved bounds are limited in the root node, except for IBF^2 on the CV dataset that increases the lower bound by 1 on 200 of 760 instances. Nevertheless, we now show that they are useful during the search.

BF Dataset. Figure 6a shows the logarithmic number of nodes searched for solving BF instances with and without the improved lower bounds. Points below the line indicate improved solving performance from the new lower bound, whereas points above the line indicate degraded performance.

The most effective bounds on the BF dataset are IBF^3 and IBF^4. We highlight that PT^3 solves 44 more instances than PT^2 (which solves the same as TT^A), as well as dramatically reduces the number of nodes in all the instances. PT^4 adds 13 more optimal solutions to that achieved by PT^3, finding 499 optimal solutions instead of the 442 of TT^A. The number of nodes decreases again significantly, especially on the most challenging instances. Considering just the 442 cases in which PT^4 and TT^A achieve optimal solutions, there are 8 categories in which the reduction percentage of the number of nodes exceeds 90%, and the CPU time is reduced on average by 76%.



(a) BF instances



(b) CV instances

Figure 6: Number of nodes necessary for finding an optimal solution on all BF and CV instances with the lower bound improvements (PT^2, PT^3, PT^4) and without (TT^A). The graphics uses a log-10 scale and unsolved instances are assigned a node count of 10^{11}.

CV Dataset. In general, PT^2 finds solutions in slightly fewer nodes than TT^A, see Figure 6b. PT^3 obtains a greater reduction in the number of nodes over PT^2 whereas the number of nodes explored in PT^3 and PT^4 is quite similar, with slight gains for PT^4. In regards to the number of optimal solutions, PT^2 solves 3 more instances to optimality than TT^A, while PT^3 and PT^4 solve 16 more than PT^2. PT^4 reduces the number of nodes on average by 65% and the CPU time by 71% across instances optimally solved by both PT^4 and TT^A.

6.3. Branching strategy

We next examine the performance of the branching comparison algorithm BCA^2 proposed in Section 4.2 against that used by TT^A, changing the branching strategy of PT^4 and calling this variant PT^{4+}. PT^{4+} solves 502 instances to optimality instead of 499 on the BF instances, and 681 instead of 679 cases for the CV instances. The number of feasible solutions goes from 592 to 599 on the BF dataset, and from 710 to 713 on the CV dataset.

6.4. New dominance rules

Figure 7 shows the logarithmic number of nodes explored for solving CV and BF instances with PT^{4+}, and with PT^D (including all the new dominance rules), clearly showing that the new dominance rules introduced in this paper reduce the number of nodes searched.

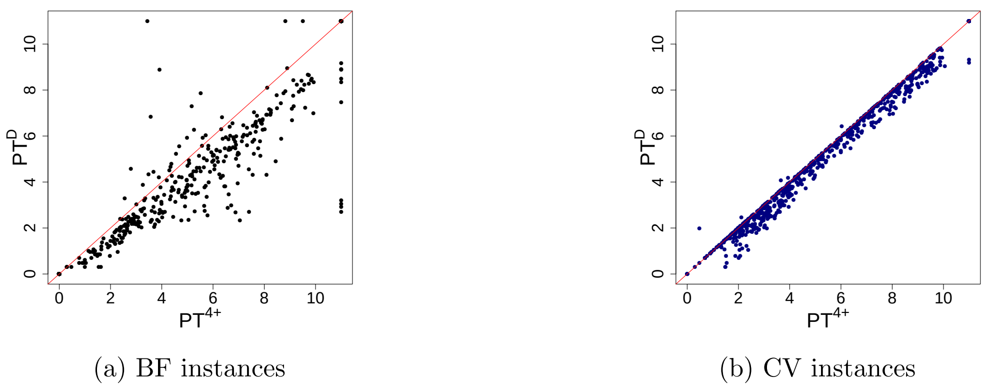
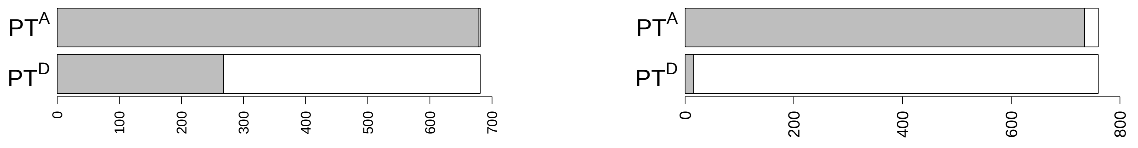


Figure 7: Number of nodes necessary for finding an optimal solution on all BF and CV instances with PT^{4+} and PT^D. The graphics use a log-10 scale and unsolved instances are assigned a node count of 10^{11}.

On the BF dataset, PT^D solves 7 more instances to optimality and cuts the number of nodes down on average by 93% and CPU time on average by 47% in those instances optimally solved by both versions. We obtain new optimal solutions in some of the most challenging categories such as BF13, BF15, and BF31. The average node count is reduced in almost all categories and the reduction percentage exceeds 90% in 19 of them. On the CV dataset, PT^D optimally solves 3 more instances and cuts the number of examined nodes down by 49%. Nevertheless, CPU time increases by 23%.

6.5. *Memoization-based upper bound heuristic*

The algorithm PT^{D} finds a feasible solution on the root node in 268 instances of 681 on the BF dataset and in 16 instances of 760 on the CV dataset. Using the memoization strategy, we obtain 411 and 719 more feasible solutions on the root node on BF and CV instances, respectively, as shown in Figure 8. This strategy spends an average CPU time higher than 1 second only in three categories: BF30 (1.03 seconds), BF31 (1.18 seconds), and BF32 (3.58 seconds). Our heuristic finds a feasible solution for all the instances of these categories while TT^{A} does not provide a solution in any of them. The average CPU time is still small on the CV dataset, only 0.02 seconds on average.



(a) BF instances (b) CV instances

Figure 8: The gray bars represent feasible solutions found on the root node with PT^{A} and PT^{D} on BF and CV datasets.

PT^{A} solves the same number of instances as PT^{D} but finds at least one feasible solution on 679 instances of BF instead of 611, and on 751 instances of CV instead of 712. The feasible solutions achieved during the iterative deepening are progressively closer to the lower bound than those obtained initially by the memoization strategy, but PT^{A} allows having feasible solutions in a more significant set of instances in an integrated algorithm.

6.6. *Comparison to the state-of-the-art*

BF Instances. Table 3 shows the performance of the state-of-the-art approaches Tierney et al. (2017) and Tanaka and Tierney (2018), together with the proposed approach PT^{A}. PT^{A} solves 15% more instances to optimality, 509 instead of 442. PT^{A} fails to provide a feasible solution in only two instances of the category BF16, in contrast with the 90 instances in which TT^{A} does not provide any solution. Furthermore, PT^{A} solves 61 more instances of the hardest categories, those

Table 3: Algorithm performance of the IDA* algorithm from Tierney et al. (2017), the TT^{A} algorithm from Tanaka and Tierney (2018) and PT^{D} and PT^{A} on the BF dataset from Bortfeldt and Forster (2012) grouped by categories (Cat).The column #Inst. contains the number of instances tested in each group. The average time and average log nodes include only instances for which an optimal solution was found.

		#	Avg. Time			Avg. Log Nodes			# Optimal			# Feasible		
Cat.	C	Inst.	IDA*	TT^{A}	PT^{A}	IDA*	TT^{A}	PT^{A}	IDA*	TT^{A}	PT^{A}	IDA*	TT^{A}	PT^{A}
BF1	48	20	121.77	0.00	0.11	11.05	0.00	0.00	18	20	20	18	20	20
BF2	48	20	175.81	0.00	0.04	8.73	0.00	0.00	15	20	20	15	20	20
BF3	48	20	271.40	0.01	0.03	11.10	0.48	0.39	11	20	20	11	20	20
BF4	48	20	11.81	0.00	0.05	9.91	0.12	0.12	18	20	20	18	20	20
BF5	64	20	567.62	209.51	19.83	16.57	12.02	7.77	8	18	20	8	20	20
BF6	64	20	504.27	165.47	0.52	20.16	14.39	9.18	1	18	20	1	20	20
BF7	64	20	237.97	8.18	2.74	17.74	13.24	10.49	10	20	20	10	20	20
BF8	64	20	845.54	138.56	53.25	19.28	15.49	11.86	3	20	20	3	20	20
BF9	77	20	2084.96	150.59	123.45	21.13	13.98	9.05	5	16	19	5	20	20
BF10	77	20	257.22	0.00	6.14	15.02	1.63	3.37	3	11	18	3	20	20
BF11	77	20	338.50	293.10	16.10	18.59	15.51	10.88	3	9	18	3	20	20
BF12	77	20	-	104.23	70.12	-	6.14	5.92	0	12	20	0	20	20
BF13	103	20	-	1013.68	546.06	-	21.80	18.21	0	3	5	0	13	20
BF14	103	20	-	-	-	-	-	-	0	0	0	0	7	20
BF15	103	20	-	-	1556.41	-	-	19.55	0	0	2	0	13	20
BF16	103	20	-	165.25	1067.33	-	20.21	19.43	0	1	1	0	6	18
BF17	60	20	0.00	0.01	0.09	6.85	0.66	0.49	6	20	20	6	20	20
BF18	60	20	12.83	0.01	0.19	8.19	0.00	0.00	16	20	20	16	20	20
BF19	60	20	2.55	0.01	0.10	9.87	0.60	0.29	11	20	20	11	20	20
BF20	60	20	24.57	0.01	0.09	9.00	0.00	0.00	14	20	20	14	20	20
BF21	80	20	727.45	145.63	11.54	18.11	14.00	8.25	6	18	20	6	20	20
BF22	80	20	-	212.79	0.87	-	11.87	8.59	0	16	19	0	20	20
BF23	80	20	37.60	186.99	87.24	15.06	12.14	8.82	3	17	20	3	20	20
BF24	80	20	-	566.22	9.44	-	17.34	11.47	0	16	18	0	20	20
BF25	96	20	-	378.14	258.79	-	8.64	7.28	0	8	12	0	20	20
BF26	96	20	-	0.01	22.85	-	1.82	2.58	0	16	19	0	20	20
BF27	96	20	1203.96	96.61	28.66	20.32	12.75	10.14	2	11	17	2	20	20
BF28	96	20	238.33	0.01	237.67	18.90	2.75	4.12	1	11	15	1	20	20
BF29	128	20	-	-	512.44	-	-	18.94	0	0	1	0	11	20
BF30	128	20	-	-	1538.85	-	-	19.77	0	0	2	0	7	20
BF31	128	20	-	-	1788.39	-	-	19.77	0	0	2	0	9	20
BF32	128	20	-	-	-	-	-	-	0	0	0	0	4	20
LC1	35	1	0.00	0.00	0.01	5.98	4.48	3.53	1	1	1	1	1	1
LC2a	50	10	208.33	2.67	0.86	13.50	10.32	7.82	9	10	10	9	10	10
LC2b	50	10	312.96	1.91	0.10	17.00	12.71	6.16	6	10	10	6	10	10
LC3a	54	10	331.53	1.27	2.95	18.15	13.24	9.90	6	10	10	6	10	10
LC3b	54	10	403.67	14.22	3.83	18.25	15.50	11.58	7	10	10	7	10	10
Total:		681	-	-	-	-	-	-	183	442	509	183	591	679

with 8 tiers (BF9 through BF16 and BF25 through BF32). Using PT^{A}, 15 categories through BF1-BF32 are completely solved to optimality, 5 more categories than TT^{A}. We note the large reduction in the number of examined nodes provided by the new contributions on categories where TT^{A} and PT^{A} achieve the same number of optimal solutions.

CV Instances. We refer now to Table 4. TT^{A} solves 660 of 760 instances, while PT^{A} solves 684. All of the instances ranging in size from 3 tiers by 3 stacks up to 5 tiers by 7 are solved by PT^{A}, which is two more categories than TT^{A}. We note again that the number of examined nodes and

Table 4: Algorithm performance of the IDA* algorithm from Tierney et al. (2017), the TT^A algorithm from Tanaka and Tierney (2018) and PT^A on the CV dataset from Caserta and Voß (2009).The column #Inst. contains the number of instances tested in each group. The average time and average log nodes include only instances for which an optimal solution was found.

	#			Avg. Time			Avg. Log Nodes			# Optimal			# Feasible		
$\lvert T\rvert$	$\lvert S\rvert$	C	Inst.	IDA*	TT^A	PT^A	IDA*	TT^A	PT^A	IDA*	TT^A	PT^A	IDA*	TT^A	PT^A
5	3	9	40	0.00	0.00	0.00	6.08	5.02	4.29	40	40	40	40	40	40
5	4	12	40	0.00	0.00	0.00	6.68	5.11	4.05	40	40	40	40	40	40
5	5	15	40	0.01	0.00	0.00	7.76	6.15	4.48	40	40	40	40	40	40
5	6	18	40	0.01	0.00	0.00	8.03	6.06	4.26	40	40	40	40	40	40
5	7	21	40	0.03	0.00	0.01	9.24	6.95	4.93	40	40	40	40	40	40
5	8	24	40	0.07	0.00	0.01	9.27	6.75	4.80	40	40	40	40	40	40
6	4	16	40	0.66	0.04	0.04	12.32	10.15	9.40	40	40	40	40	40	40
6	5	20	40	2.20	0.04	0.03	12.92	10.13	9.08	40	40	40	40	40	40
6	6	24	40	15.12	0.22	0.09	14.52	11.66	9.93	40	40	40	40	40	40
6	7	28	40	29.91	0.75	0.37	15.76	12.92	10.68	40	40	40	40	40	40
7	4	20	40	299.50	36.97	32.10	17.91	16.21	15.24	34	40	40	34	40	40
7	5	25	40	396.38	106.89	96.68	19.11	16.75	15.54	35	40	40	35	40	40
7	6	30	40	883.10	231.66	169.60	20.21	18.91	17.29	17	39	40	17	40	40
7	7	35	40	732.43	507.22	312.16	19.63	18.66	17.02	18	37	40	18	40	40
7	8	40	40	1332.14	376.94	222.61	20.91	19.83	17.77	10	35	38	10	40	40
7	9	45	40	2016.98	466.57	470.61	21.46	20.33	18.34	6	30	38	6	40	40
7	10	50	40	1901.81	360.27	327.51	21.46	19.99	17.85	3	32	36	3	39	40
8	6	36	40	-	1318.63	1796.14	-	22.21	21.72	0	5	8	0	13	34
8	10	60	40	-	1980.76	1928.22	-	22.64	20.77	0	2	4	0	16	37
		Total:	760	-	-	-	-	-	-	523	660	684	523	708	751

the CPU time dramatically decrease on the most challenging categories.

Running PT^A for a day of CPU time. We also tried solving the BF and CV instances using a day of CPU time to gauge whether some solutions were barely out of reach of the one hour timeout. On the BF dataset, our approach solves 28 more instances to optimality, 2 of them in BF14 and also 2 in BF32. Remarkably, on the CV dataset, PT^A is able to solve all the instances in the categories 5-8, 5-9, and 5-10 in 576.53, 1315.64, and 3076.00 seconds on average, respectively. Categories 5-9 and 5-10 represent the size of the largest RMGC systems we are aware of, meaning our algorithm is able to solve difficult real-world CPMP instances to optimality. Furthermore, it also solves 33 out of 80 instances of categories 6-6 and 6-10, which pose difficulty even to metaheuristic approaches.

CVX Instances. We now evaluate both algorithms on a "fresh" dataset to ensure we are not overfitting our approach to the existing instances. The results are shown in Table 5. On this dataset, our algorithm still outperforms TT^A, solving 441 instances to optimality instead of 397 in the category in which all containers have different priorities (CVX1), 468 instead of 442 in the category with at least two containers of each priority (CVX2), and 486 instead of 468 in the category with at least three containers of each priority (CVX3). A feasible solution is found for all the instances by PT^{A} while TT^{A} does not find any solution on 4 instances. PT^{A} cuts the number of nodes down by 91% and the CPU time by 65%. These data show the excellent performance of our approach on a set of instances different to that used for its development.

Table 5: Algorithm performance of the TT^{A} algorithm from Tanaka and Tierney (2018) and PT^{A} on the CVX dataset from Hottung et al. (2017). The column #Inst. contains the number of instances tested in each group.The average time and average log nodes include only instances for which an optimal solution was found.

				#	Avg. Time		Avg. Log Nodes		# Optimal		# Feasible	
	$\lvert T\rvert$	$\lvert S\rvert$	C	Inst.	TT^{A}	PP^{A}	TT^{A}	PP^{A}	TT^{A}	PP^{A}	TT^{A}	PP^{A}
CVX1	7	7	35	250	293.34	216.85	18.96	17.21	223	241	250	250
	7	10	50	250	750.33	414.07	20.58	18.15	174	200	247	250
			Total:	500	-	-	-	-	397	441	497	500
CVX2	7	7	35	250	198.24	107.78	18.22	16.28	242	248	250	250
	7	10	50	250	452.63	314.86	20.01	17.52	200	220	249	250
			Total:	500	-	-	-	-	442	468	499	500
CVX3	7	7	35	250	80.88	43.30	17.06	15.14	248	250	250	250
	7	10	50	250	318.79	212.28	19.22	16.67	220	236	250	250
			Total:	500	-	-	-	-	468	486	500	500

7. Conclusions and future research

We presented novel lower bounds and dominance rules for solving real-world sized pre-marshalling problems. Our approach extends the IBF^{1} lower bound from the literature, as well as introduces two new lower bounds, extended and new dominance rules, and a feasible solution heuristic. Our lower bounds and dominance rules are computationally inexpensive, allowing them to be implemented as part of an iterative deepening branch-and-bound procedure. Our approach finds feasible solutions for nearly every CPMP instance it encounters, and optimal solutions on between 80.0% and 99.2% of the instances in industrially relevant instance categories.

There are several potential avenues for future work. In particular, we intend to investigate the integration of crane movement costs into the CPMP, so that moves of different distances and heights take different amounts of time. Furthermore, robust variants of the problem (as in Tierney and Voß, 2016) offer a way of preventing the need for multiple iterations of the CPMP due to delays of outbound transportation.

Acknowledgements

The authors thank the Paderborn Center for Parallel Computation (PC^{2}) for the use of the Arminius cluster for the computational study in this work. This work has been partially supported by the Spanish Ministry of Science, Innovation, and Universities, FPU Grant A-2015-12849 and by the Spanish Ministry of Economy and Competitiveness, under projects DPI2014-53665-P and DPI2015-65895-R, partially financed with FEDER funds.

References

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

Boysen, N., Emde, S., 2016. The parallel stack loading problem to minimize blockages. European Journal of Operational Research 249 (2), 618 – 627.

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

de Melo da Silva, M., Toulouse, S., Calvo, R. W., 2018. A new effective unified model for solving the pre-marshalling and block relocation problems. In press at European Journal of Operational Research.

Dekker, R., Voogd, P., van Asperen, E., 2006. Advanced methods for container stacking. OR Spectrum 28 (4), 563–586.

Drewry Maritime Research, 2016. Global Container Terminal Operators Annual Review and Forecast 2016. Annual Review and Forecast.

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.

Galle, V., Barnhart, C., Jaillet, P., 2018. Yard crane scheduling for container storage, retrieval, and relocation. In press at European Journal of Operational Research.

Hansen, J. R., Fagerholt, K., Stålhane, M., 2017. A shortest path heuristic for evaluating the quality of stowage plans in roll-on roll-off liner shipping. In: International Conference on Computational Logistics. Springer, pp. 351–365.

Hottung, A., Tanaka, S., Tierney, K., 2017. Deep learning assisted heuristic tree search for the container pre-marshalling problem. arXiv preprint arXiv:1709.09972.

Hottung, A., Tierney, K., 2016. A biased random-key genetic algorithm for the container pre-marshalling problem. Computers & Operations Research 75, 83 – 102.

Ku, D., Arthanari, T. S., 2016. Container relocation problem with time windows for container departure. European Journal of Operational Research 252 (3), 1031–1039.

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

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

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.

Parreño-Torres, C., Alvarez-Valdes, R., Ruiz, R., 2018. Integer programming models for the pre-marshalling problem. In press at European Journal of Operational Research.

Prandtstetter, M., 2013. A dynamic programming based branch-and-bound algorithm for the container pre-marshalling problem. Tech. rep., Technical report, AIT Austrian Institute of Technology.

Quispe, K. E. Y., Lintzmayer, C. N., Xavier, E. C., 2018. An exact algorithm for the blocks relocation problem with new lower bounds. Computers & Operations Research 99, 206–217.

Slaney, J., Thiébaux, S., 2001. Blocks world revisited. Artificial Intelligence 125 (1), 119–153.

Tanaka, S., Mizuno, F., 2018. An exact algorithm for the unrestricted block relocation problem. Computers & Operations Research 95, 12–31.

Tanaka, S., Takii, K., 2016. A faster branch-and-bound algorithm for the block relocation problem. IEEE Transactions on Automation Science and Engineering 13 (1), 181–190.

Tanaka, S., Tierney, K., 2018. Solving real-world sized container pre-marshalling problems with an iterative deepening branch-and-bound algorithm. European Journal of Operational Research 264 (1), 165 – 180.

Tang, L., Liu, J., Yang, F., Li, F., Li, K., 2015. Modeling and solution for the ship stowage planning problem of coils in the steel industry. Naval Research Logistics 62 (7), 564–581.

Tierney, K., 2015. Optimizing Liner Shipping Fleet Repositioning Plans. Springer.

Tierney, K., Pacino, D., Voß, S., 2017. Solving the pre-marshalling problem to optimality with A* and IDA*. Flexible Services and Manufacturing Journal 29 (2), 223–259.

Tierney, K., Voß, S., 2016. Solving the robust container pre-marshalling problem. In: International Conference on Computational Logistics. Springer, pp. 131–145.

UNCTAD, 2017. United Nations Conference on Trade and Development (UNCTAD) Review of Maritime Transport.

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

Wang, N., Jin, B., Lim, A., 2015. Target-guided algorithms for the container pre-marshalling problem. Omega 53, 67–77.

Wang, N., Jin, B., Zhang, Z., Lim, A., 2017. A feasibility-based heuristic for the container pre-marshalling problem. European Journal of Operational Research 256 (1), 90 – 101.

Zhang, R., Jiang, Z., Yun, W., 2015. Stack pre-marshalling problem: A heuristic-guided branch-and-bound algorithm. International Journal of Industrial Engineering: Theory, Applications and Practice 22 (5), 509–523.

Appendix A. Proofs of the new dominance rules provided in Section 4.3

Proof of Proposition 1. The first condition $s_1 > s_n$ is just for tie-breaking. Since s_n and d_n are invariant to the sequence $(c_1, s_1, d_1), \dots, (c_{n-1}, s_{n-1}, d_{n-1})$, the same layout as the original sequence is obtained by the sequence $(c_n, s_n, d_n), (c_1, s_1, d_1), \dots, (c_{n-1}, s_{n-1}, d_{n-1})$ whenever it is feasible. It is only feasible when the third condition is satisfied, ensuring d_n to have enough space to allocate the maximum number of temporary containers assigned to it during the sequence $(c_1, s_1, d_1), \dots, (c_{n-1}, s_{n-1}, d_{n-1})$. With respect to s_n, temporary containers always fit into it. Since $t^{1,n-1}_{s_n} + n_{s_n} + 1 \le |T|$, therefore $t^{1,n-1}_{s_n} + n_{s_n} \le |T|$.

Proof of Proposition 2.

1. Since stack s_1 is invariant, the number of containers in s_1 before the second move is the same as n_{s_1}. The second term ensures that s_1 has enough space to move the container c_1 after the sequence $(c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1})$. Therefore, the same layout as that of the original sequence is obtained by another feasible sequence with one less move $(c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1}), (c_1, s_1, d_n)$ when $s_1 \neq d_n$ or by another feasible sequence with two less moves $(c_2, s_2, d_2), \dots, (c_{n-1}, s_{n-1}, d_{n-1})$ when $s_1 = d_n$.
2. Since stack d_n is invariant to the sequence $(c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1})$ and n_{d_n} is the number of containers stored in d_n after the move (c_1, s_1, d_n), d_n has $n_{d_n} - 1$ containers stored before and after that sequence. By the second term, d_n has enough space to store the maximum number of containers that are temporarily allocated to it during the sequence $(c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1})$, although it now has one less free slot. The same layout as that of the original sequence is obtained by the feasible sequence $(c_1, s_1, d_n), (c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1})$ when $s_1 \neq d_n$ or by $(c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1})$ when $s_1 = d_n$.
3. The sequence of moves $(c_1, s_1, s'), (c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1}), (c_1, s', d_n)$ is feasible since $c_1 \notin \{c_2, \dots c_n\}$ because s' is invariant to $(c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1})$, s' has enough space to store the temporarily moves of containers, and it provides the same layout as the original sequence. Condition $s' < d_1$ is just for tie-breaking. When $s' = d_n$, the sequence $(c_1, s_1, d_n), (c_2, s_2, d_2), ..., (c_{n-1}, s_{n-1}, d_{n-1})$ is feasible.

Proof of Proposition 3. Let $p = \mathit{group}(c_1) = \mathit{group}(c_n)$. It suffices to show that the same layout with respect to group values is obtained by another feasible sequence. It does not matter if the positions of containers c_1 and c_n of the same group may be interchanged.

1. In the sequence, the container c_1 of group p is moved from s_1 first and finally c_n (also of group p) is moved to $d_n = s_1$. Since s_1 is invariant to $(c_2, s_2, s_2), \dots, (c_{n-1}, s_{n-1}, d_{n-1})$, the number of containers in s_1 in the initial layout is equal to that of the final layout, n_{s_1}. First, assume that s_n is invariant and $s_n \neq d_1$, which implies $c_n \notin \{c_1, \dots, c_{n-1}\}$. Noting that c_n is on top of s_n in the initial layout, we consider a new sequence $(c_n, s_n, d_1), (c'_2, s_2, d_2), \dots,$

$(c'_{n-1}, s_{n-1}, d_{n-1})$, where $c'_k = c_n$ if $c_k = c_1$, and $c'_k = c_k$ otherwise, to move c_n in place of c_1 by the second and later moves. In this sequence, c_1 is not moved from s_1, which may block moves from s_1 and/or cause lack of spare space for moves to s_1. However, the former is not the case because s_1 is invariant, and the latter never occurs from $t^{2,n-1}_{s_1} + n_{s_1} \leq |T|$. Therefore, this sequence is feasible and yields the same layout as the original sequence except for the interchanged positions of c_1 and c_n. Next, assume that d_1 is invariant and $d_1 \neq s_n$. It implies that c_1, which is moved to d_1 by (c_1, s_1, d_1), is not moved by $(c_2, s_2, s_2), \ldots, (c_n, s_n, d_n)$ and thus $c_1 \notin \{c_2, \ldots, c_n\}$ holds. This condition together with the above argument on s_1 ensures the feasibility of a new sequence $(c_2, s_2, s_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1}), (c_n, s_n, d_1)$. It is not difficult to verify that it yields the same layout as the original sequence with respect to group values. Finally, assume that $s_n = d_1$ is invariant, therefore $c_1 = c_n$ and $c_1 \notin \{c_2, \ldots, c_{n-1}\}$ holds, thus d_1 remains the same at the end of the whole sequence. The same layout is obtained by the sequence $(c_2, s_2, d_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1})$ that is feasible because of the above argument on s_1.

2. Consider a sequence of moves $(c_n, s_n, d_1), (c'_2, s_2, d_2), \ldots, (c'_{n-1}, s_{n-1}, d_{n-1})$ where $c'_k = c_n$ if $c_k = c_1$ and $c'_k = c_k$ otherwise. As already shown in case 1, the sequence is feasible since s_1 and s_n are invariant and $t^{2,n-1}_{s_1} + n_{s_1} \leq |T|$ holds. The topmost container of s_1 after this sequence is c_1 as s_1 is invariant, so that the sequence $(c_n, s_n, d_1), (c'_2, s_2, d_2), \ldots, (c'_{n-1}, s_{n-1}, d_{n-1}), (c_n, s_1, d_n)$ is also feasible. Obviously, it yields the same layout as the original sequence with respect to the group values.
3. We consider a sequence of moves $(c_1, s_1, d_n), (c_2, s_2, d_2), \ldots, (c_{n-1}, s_{n-1}, d_{n-1})$. Since d_1 is invariant, $c_1 \notin \{c_2, \ldots, c_{n-1}\}$ holds. Stack d_n is also invariant and $t^{2,n-1}_{d_n} + n_{d_n} < |T|$ holds. These facts ensure the feasibility of the sequence. It yields the same layout as the original sequence with regard to group values since d_1 and d_n are invariant.

Appendix B. Proofs of the Extended lower bounds presented in Section 5

Proof of Proposition 4. The following situations are the only options for solving the CPMP in IBF^1 moves when Conditions 1 and 2 hold:

(S1.1) A stack s_1 with exactly $h^{\mathrm{M}} + 1$ misoverlaying containers is repaired first using BB moves, and then another stack s'_1 can be repaired using only BG moves.

(S1.2) A stack s_2 with exactly h^{M} misoverlaying containers is repaired using BB moves, and

(S1.2a) the topmost non-misoverlaying container in stack s_2 is moved by a GB move, and then another stack s'_2 is repaired only by BG moves, or (S1.2b) another stack s'_2 is repaired by exactly one BB move before, during or after a BG move sequence.

We now show that for each of these situations there is a condition when it does not hold, and when all of the conditions hold, there is no way to solve the CPMP in only IBF^1 moves. First consider Condition 3 as applied to situation (S1.1). We first incur $(h^{\mathrm{M}} + 1)$ BB moves to repair s_1. All remaining moves must be BG moves. However, Condition 3 stipulates there is no upside down stack that can be placed on top of the repaired stack. Thus, situation (S1.1) is not sufficient for solving the bay in just IBF^1 moves. Next, Condition 4 is applied to situation (S1.2a). According to the condition, even if we perform a GB move to remove the topmost non-misoverlaying container of s_2, we will not be able to flip the upside down stack anywhere without causing misoverlays. This would then require additional BB moves, meaning situation (S1.2a) is also not sufficient in just IBF^1 moves. Finally, Condition 5 is used in situation (S1.2b). To repair stack $s'_2 \in U'$, the top group of s_2 should be at least $g^{\mathrm{X}}_{s'_2}$, granted that a single BG move to another stack is permitted. Obviously, it is not possible when Condition 5 applies. Thus, when Conditions 3, 4 and 5 hold, the CPMP cannot be solved using situations (S1.1), (S1.2a) or (S1.2b), and $\mathrm{IBF}^2 := \mathrm{IBF}^1 + 1$ is valid.

Proof of Proposition 5. To solve the CPMP in $\mathrm{IBF}^1 = \mathrm{IBF}^0 + 1$ moves when Condition 1 holds, two options for repairing another stack are (S2b.1) the topmost non-misoverlaying container in stack s is moved by a GB move, and then another stack s' is repaired only by BG moves, and (S2b.2) another stack s'' is repaired by exactly one BB move before, during or after the BG move sequence. Since these are similar to (S1.2a) and (S1.2b) in the proof of Proposition 4, respectively, it is easy to show that (S2b.1) and (S2b.2) are not possible if Conditions 4 and 5 hold, respectively. Thus, $\mathrm{IBF}^2 = \mathrm{IBF}^1 + 1$ must hold.

Proof of Proposition 6. The only option for solving the CPMP in $\mathrm{IBF}^1 = \mathrm{IBF}^0 + 1$ moves is to repair stack s' with BG moves to stack s and then another stack $s'' \in S^{\mathrm{M}} \setminus \{s'\}$ with BG moves to stacks s and s'. When $n_{\mathrm{GX}} > 0$, we may perform GG moves from s' to s unless s becomes full. Note that GG moves from s to s' are not possible since s' was repaired through moves to s. These moves can change the top group of stack s to $w^{\mathrm{sec}}_{s'}$. If both the cases are taken into account, the top groups of stacks s and s' are given by w'_s and $w'_{s'}$, respectively. To repair stack s'' with only BG moves to stacks s and s', $n^{\mathrm{BG}}_{s''} \leq 2$, $g^{\mathrm{BG}}_{s''1} \leq \max(w'_s, w'_{s'})$, and $g^{\mathrm{BG}}_{s''2} \leq \min(w'_s, w'_{s'})$ should all be satisfied. Condition 4 ensures that no such stack s'' exists. Therefore, an extra move is required and IBF^2 is valid.

Proof of Proposition 7. First, we note that the situations (S3.1) and (S3.2) provide us with the largest possible group values on stacks s_1 and s_2 given the GG moves available. We must then repair one of the misoverlaid stacks using only BG moves to s_1 and s_2. However, when Condition 3 holds, there are no misoverlaid stacks that have containers that fit into s_1 and s_2 without requiring an additional move.

Proof of Proposition 8. First, consider situation (S4.1) in which we perform a GG move from s'' to s. Since we are only left with BG moves, we need to be able to place the misoverlaying containers from one of the misoverlaid stacks s' on top of s'' and s without any extra moves. Furthermore, stack s'' can accept only one container as it was full before the GG move. Thus, $s' \in U'$ and at least $g^{\mathrm{X}}_{s'} \leq w_s$ should hold to repair it. Next, for situation (S4.2), we perform a BB move when there is a stack that would be upside down if one container was removed from it. The same argument is true in this case, and $s' \in U'$ and $g^{\mathrm{X}}_{s'} \leq w_s$ are required. Thus, when $\min_{s' \in U'} g^{\mathrm{X}}_{s'} > w_s$, $\mathrm{IBF}^2 := \mathrm{IBF}^1 + 1$ is a valid lower bound.

Proof of Proposition 9. Suppose that the procedure in the proposition fails to repair all the stacks in S^{M}. Then, we will not be able to repair any of the remaining stacks in S^{M} with only BG moves even when all misoverlaying containers in the other stacks are removed from the bay. Since GG moves are taken into consideration by adding empty stacks to S^{N}, at least an additional move is needed to repair the bay.

Proof of Proposition 10.

1. If the minimum dirty height is larger than clean supply, none of dirty stacks can be repaired only by BG moves, so that IBF^3=LB$^{\mathrm{BF}}$ is improved by 1.
2. If there is only one clean supply stack and no dirty stack, clean demand containers are moved to the unique clean supply stack only by BG moves. It is not possible when there is a clean demand stack where demand containers are not upside down, therefore IBF^3 = LB$^{\mathrm{BF}}$ can be increased by 1. To speed up the check, we only search for such a clean demand stack that the group of the topmost demand container is smaller than the maximum group of demand containers in that stack.

3. If there is only one clean stack, at least one of the topmost dirty demand containers is moved first to that stack. Let $g^{\mathrm{D}}_{\max}$ be the maximum group of dirty demand containers. A container with group $g^{\mathrm{D}}_{\max}$ should also be moved to the clean supply stack if $g^{\mathrm{D}}_{\max}$ is larger than topmost non-misoverlaying containers in dirty stacks. However, a BG move is not possible if $g^{\mathrm{D}}_{\max}$ is larger than the topmost dirty demand containers, thus we can improve IBF^3 by 1.