University of Paderborn
Faculty of Business and Economics
Depatment of Information Systems
Decision Support & Operations Research Lab

Working Paper

WP1401

Solving the Pre-Marshalling Problem to Optimality with A* and IDA*

Kevin Tierney, Dario Pacino and Stefan Voß

Paderborn, Germany
March, 2014

1. Introduction

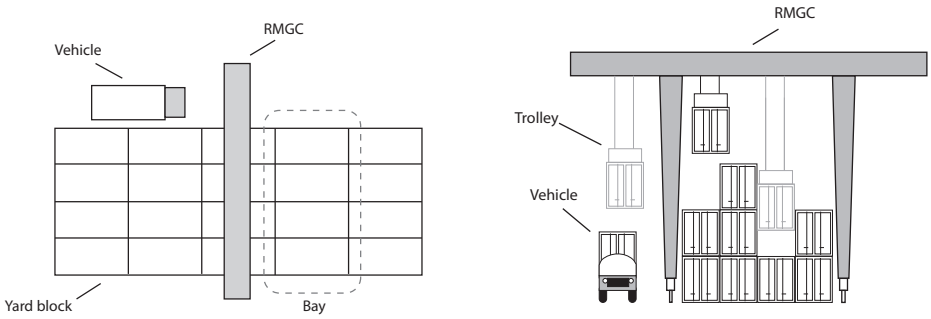
International trade is increasingly being conducted with containers, which are large, metal boxes in standardized sizes in which cargo can be secured during transit (UNCTAD 2012). Containers are inter-modal, meaning they can be easily transferred between different modes of transportation, such as trucks, trains and ships. Each year, millions of containers are transferred between different transportation modes at the world's ports and hinterland handling yards (Rodrigue et al. 2009). Delays in the transfer of containers are expensive, causing trucks, trains and ships to be delayed leaving ports, and avoiding such delays is a primary concern of container terminal operators.

A number of factors affect the speed of container transfer operations, such as intra-terminal container handling (see Stahlbock and Voß (2008)), inter-terminal transportation (e.g., Tierney et al. (2013)) and, our focus in this paper, the container arrangement in the yard. In the yard, containers are stored in *bays*, which are two-dimensional portions of the yard. Each export container is assigned a priority corresponding to its expected loading time and loading sequence. Due to the high level of uncertainty related to when containers should leave the yard, it is often not possible to arrange the containers optimally at the time they enter the yard, and, thus, containers are stacked on top of each other such that cranes must perform unproductive moves to access containers that are slated to leave the yard. We call this situation for a single container a *mis-overlay*.

More specifically, a mis-overlay occurs when a container with a lower priority is stacked on top of one with a higher priority, i.e., a container that has to leave later is stacked on top of one that needs to leave sooner. Mis-overlays imply extra, unnecessary, handling operations, and thus longer handling times for vehicles and ships. To alleviate this situation, terminals sometimes perform *pre-marshalling* in order to find a configuration of containers such that no mis-overlays occur. The goal of the pre-marshalling problem is to find the minimal number of container moves necessary to remove all mis-overlays from a bay.

This paper focuses on the pre-marshalling operations of container terminals that use certain types of gantries like Rail Mounted Gantry Cranes (RMGCs). Figure 1 shows cranes installed over yard blocks at a container terminal. The cranes are able to reach the containers on top of the stacks. Due to security reasons, RMGCs are often not allowed to move between bays while lifting a container, thus container marshalling between bays is seldom performed as it might require the use of other vehicles. It is also noteworthy that it is unusual to mix containers of different lengths (e.g. 20ft and 40ft) within the same bay. It is due to these considerations, and in accordance with the literature, that the pre-marshalling problem can be considered for a single bay.

Despite numerous heuristic methods for solving the pre-marshalling problem (see, e.g., Lee and Hsu (2007), Lee and Chao (2009), Caserta and Voß (2009)), only few approaches have been proposed for solving the pre-marshalling problem to optimality. The first, using integer programming was



(a) Top view of an RMGC.

(b) Front view of an RMGC.

Figure 1 A Rail Mounted Gantry Crane (RMGC) over a yard block.

proposed in Lee and Hsu (2007) and, the second, proposed more recently, is an A* algorithm from Expósito-Izquierdo et al. (2012). Solving the pre-marshalling problem to optimality can allow ports to make their operations even more efficient, and is an important research goal for quantifying the performance of the numerous heuristic approaches.

To this end, we present a novel solution technique for solving pre-marshalling problems to optimality using the A* and IDA* algorithms (see, e.g., Russell and Norvig (2010)) combined with specialized branching rules, symmetry breaking and strong lower bounds from the literature. Our contributions are as follows. We provide

1. two classes of novel symmetry breaking rules,
2. a novel branching rule based on preventing transitive moves,
3. a novel use of the Bortfeldt and Forster lower bound heuristic as an A*/IDA* cost estimation heuristic.

We evaluate our A* and IDA* approaches on all three well-known pre-marshalling datasets from the literature and are able to solve 568 instances more than the state-of-the-art A* approach, which represents nearly twice as many instances solved to optimality. Additionally, we see significant runtime speedups on a majority of instances solvable by both our approach and the state-of-the-art.

The remainder of the paper is organized as follows. First, we introduce the pre-marshalling problem in Section 2 and review the literature in Section 3. We describe A* and IDA* algorithms for the pre-marshalling problem in Section 4 and our branching rules in Section 5. Finally, we provide computational results in Section 6 and conclude in Section 7.

2. The Pre-Marshalling Problem

Given an initial layout of a bay, the goal of the pre-marshalling problem is to find the minimal number of container movements (or *rehandles*) necessary to eliminate all mis-overlays in the bay. Formally, a bay contains S stacks which are at most T tiers high. Each container in the bay is assigned a priority, $p_{st} \in \mathbb{N}_0$ $(s \in S, t \in T)$. We set $p_{st} = 0$ if there is no container at the position s, t.

Containers with a smaller priority value are retrieved first, meaning they must be above containers with a larger priority value in a configuration with no mis-overlays. Thus, a bay has no mis-overlays iff $p_{st} \leq p_{s,t+1}$ for all $s \in S$, $1 \leq t < |T|$. As previously mentioned we focus on a single bay, thus no movements between bays are allowed and all containers are assumed to be of the same size.

Consider the simple example of Figure 2(a), which shows a bay composed of three container stacks where containers can be stacked at most four tiers high. Each container is represented by a box with its corresponding priority.[1] This is not an ideal layout as the containers with priority 2, 4 and 5 will need to be relocated in order to retrieve the containers with higher priority (1 and 3). That is, containers with priority 2, 4 and 5 are mis-overlaid. Consider a container movement (f, t) defining the relocation of the container on top of the stack f to the top position of the stack t. The containers in the initial layout of Figure 2 (a) can reach the final layout (d) with three relocation moves: $(2, 3)$ reaching layout (b), $(2, 3)$ reaching layout (c) and $(1, 2)$ reaching layout (d) where no mis-overlays occur.

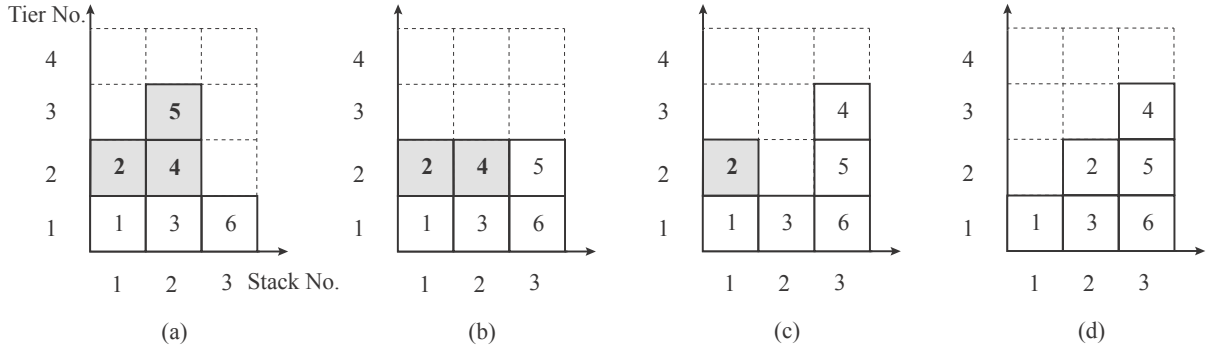


Figure 2 **An example solution to the pre-marshalling problem. Mis-overlays are indicated by the highlighted containers.**

The pre-marshalling problem can be seen as a problem arising in tactical and operational decision making at a yard; see, e.g., Dekker et al. (2006), Caserta et al. (2011). Following Choe et al. (2011), Park et al. (2009), Kang et al. (2006b), the unproductive movement of containers in the different phases of the container management process, i.e., rehandling, is perceived as the major source of inefficiency in most container terminals. On a tactical level, pre-marshalling refers to the pre-arrangement or housekeeping policy to be applied when re-arranging a terminal so as to ease, e.g., the future loading of a vessel. On the operational level, the problem arises when rehandling of containers within the yard becomes necessary. It should be noted that during re-marshalling and pre-marshalling the total number of containers in a specific stacking area is kept unchanged.

[1] We note that multiple containers may have the same priority, but in order to make containers easily identifiable, in this example we have assigned a different priority to each container.

As pointed out by many authors, e.g., Park et al. (2009), Kang et al. (2006b), Kim and Bae (1998), even when stacking policies aimed at minimizing the number of rehandling moves are used, later rehandling cannot be avoided altogether. Some of the reasons why this occurs are, e.g., that containers being shipped with different vessels are stored together due to the limited space capacity of the yard, that the precise information about the weight of containers is not available until shortly before the loading operations begin, or that the loading plan is not yet determined when a container arrives at the yard. In addition, trucks or ships might be delayed, etc. In an early paper by Kim (1997) influencing this area of research, various stack configurations and their influence on the expected number of rehandles are investigated in a scenario of loading import containers onto outside trucks with a single transfer crane.

3. Literature Review

A survey of storage and stacking logistics at container terminals is provided in Steenken et al. (2004) and Stahlbock and Voß (2008). A more specific overview of some of the most relevant optimization problems related to stacking at container terminals is given in Dekker et al. (2006), which also proposes a number of alternative policies for stacking containers in a yard. However, no details about reshuffling rules for containers are provided. The most comprehensive literature review on the pre-marshalling problem as well as its "brothers" (i.e., the blocks relocation problem and the re-marshalling problem) is provided by Caserta et al. (2011).

There are two main types of marshalling activities. Intra-block re-marshalling refers to containers that are rearranged into designated bays within the same (yard-)block. On a smaller scale, pre-marshalling refers to intra-bay operations in which containers within the same bay are reshuffled. In both cases, the goal is to minimize the number of future unproductive moves. That is, "re-marshalling refers to the task of relocating export containers into a proper arrangement for the purpose of increasing the efficiency of the loading operation" (Choe et al. 2011).

Intra-bay pre-marshalling may be motivated by means of different, mainly operational, arguments. When rail mounted gantry cranes are used as major container handling equipment as in, e.g., Lee and Chao (2009), Lee and Hsu (2007), it is assumed that marshalling problems need to be solved at the bay level. If side-loading is applied to access the blocks, moving gantries between bays while carrying a container is discouraged (see Figure 1 above).

There have been various approaches for solving the pre-marshalling problem. As the problem is NP-hard (see Caserta et al. (2011)), these approaches incorporate heuristics, metaheuristics as well as exact algorithms.

An optimization model for the pre-marshalling problem is proposed by Lee and Hsu (2007). More specifically, they consider an integer programming model based upon a time-space multi-commodity

network flow problem. A drawback of this approach is that the model contains a parameter T, defining the number of time periods necessary to complete the optimization. The optimal value of this parameter is not known a priori, and high values lead to long computation times. In order to overcome the limitations imposed by the size of the integer programming model of Lee and Hsu (2007), Lee and Chao (2009) propose a heuristic approach using neighborhood search and mathematical programming, in which the two approaches are alternated to find short chains of re-handles.

A heuristic based on the corridor method is presented in Caserta and Voß (2009). The central idea of the approach relies on iteratively solving smaller portions of the original problem to optimality. The algorithm consists of four different phases, in which ideas from the corridor method, roulette-wheel selection and local search techniques are intertwined to foster intensification around an incumbent solution. The algorithm is stochastic in nature and is based upon the use of a set of greedy rules that bias the behavior of the scheme toward the selection of the most appealing moves.

Additional approaches include using a tree search algorithm, see Bortfeldt and Forster (2012), as well as integer programming, see Lee and Lee (2010). Another idea is provided by Expósito-Izquierdo et al. (2012), who describe an A* algorithm which may be used as a blueprint for the more advanced approach described in this paper. Some comments on logical observations leading to a lower bound are provided in Voß (2012). Some of these ideas are also incorporated in the tree search algorithm of Bortfeldt and Forster (2012), and are used to enhance the capabilities of the A* approach.

Algorithms with direct heuristics have been developed by Huang and Lin (2012), Gheith et al. (2013), Salido et al. (2009). A neighborhood search heuristic can be found in Lee and Chao (2009). Incorporating these or similar heuristics into decision support systems or discrete event simulation studies is one idea regarding their evaluation; see, e.g., Salido et al. (2012), Klaws et al. (2011) Another paper on heuristics is Expósito-Izquierdo et al. (2012), who – besides presenting the A* approach mentioned above – develop a greedy algorithm based on a collection of rules, which are combined with a certain level of randomization to improve the quality of results. Based on this approach, Jovanovic et al. (2013) extend the specific greedy rules to replace the randomization and obtain considerably improved results.

One of the greedy rules incorporated in the approaches of Expósito-Izquierdo et al. (2012) and Jovanovic et al. (2013) deals with the storage position of reshuffled containers. One of the issues is to avoid deadlocks and infeasibility. Related stacking rules are also considered in Zhang et al. (2007).

Extensions of the pre-marshalling problem may be defined in various ways. As an example, one might consider the more practical situation of an online optimization problem. Rather than

explicitly defining such a problem, an interesting extension of the pre-marshalling problem can be found in Rendl and Prandtstetter (2013). Instead of defining priority values for the containers they assume a given range of priority values for them and redefine the pre-marshalling problem under this modified setting. They propose a constraint programming approach to solve the problem.

Related work in different fields Closely related to the pre-marshalling problem are the blocks relocation problem (see, e.g., Kim and Hong (2006)) and the re-marshalling problem (see, e.g., Choe et al. (2011), Kang et al. (2006a)). In particular, IDA* has been used to solve the blocks relocation problem Zhu et al. (2012), but the authors put their main emphasis on probing heuristics and a variety of lower bound techniques, which stands in contrast to our focus on branching rules. The pre-marshalling problem, however, can be imagined not only at container terminals but also in other situations. Some warehouses are organized following the stacking principle, by storing uniform items piled up on top of each other, where access is only granted to the uppermost item. While stacking operations in such warehouses follow similar rules as in container yards, more often we see situations differing from pre-marshalling when retrieving and receiving operations are performed in parallel; see, e.g., Nishi and Konishi (2010). Moreover, the physical properties of the items in a warehouse might differ from those of block-shaped containers. As an example, consider coils in the steel industry. The resulting storage setting might not be formed as stand-alone stacks, as each coil may be placed on top of two consecutive coils from the row below (see, e.g., Zäpfel and Wasner (2006)), which could happen if one would allow a 40 ft. container being stacked on two 20 ft. containers.

The handling of trains also involves stacking operations; see, e.g., Felsner and Pergel (2008). A train can be seen as a sequence of wagons. It might happen that the wagon sequence of a single train needs to be changed or that the wagons of several trains have to be *reshuffled* to new collections of trains. These operations are physically carried out on dead end sidings, where trains or parts of trains can be stored intermediately and taken away later on. Thus, on dead end sidings, trains can be "stacked" together and, moreover, rehandling of wagons is possible. Each of those dead end sidings relates to a stack in the container yard, where only the uppermost container/wagon is accessible.

A well-known problem in artificial intelligence is blocks world; see, e.g., Romero and Alquézar (2004), Gupta and Nau (1992). The blocks world problem consists of a "table" where blocks are stacked on top of each other. A typical blocks world instance consists of a given initial table state and a desired goal state. The task is to transform the initial state to the goal state with a minimum number of moves. Variants of blocks world incorporate limitations on the table size and different levels of given conditions for the goal state. A primary difference between blocks world and pre-marshalling is the lack of height constraints on the block stacks.

4. Solution Approach

In this work we propose two optimal approaches based on the well-known A* and IDA* algorithms, which perform a path-based search guided by a cost estimation heuristic (see, e.g., Russell and Norvig (2010)). Although these approaches are referred to as forms of heuristic search in the computer science community, they both find optimal solutions as long as the cost estimation heuristic does not overestimate the cost of finding a solution.

The main advantage of using a search procedure such as A* is the fact that, unlike in the multi-commodity flow formulation in Lee and Hsu (2007), A* builds the graph of moves as it searches, rather than trying to fit a large graph into memory at the start. Only the nodes explored during the search are kept in memory. IDA* takes this idea a step further and has only a constant sized memory footprint, as it iteratively runs a depth first search with a maximum search depth. However, IDA* must re-generate its search tree each time the search depth is increased.

4.1. A* and IDA* for pre-marshalling

We tailor A* and IDA* to the pre-marshalling problem as follows. The A* algorithm uses a priority queue to sort the search fringe, ensuring that when a configuration is found with no mis-overlays, it consists of a minimal number of moves. IDA*, on the other hand, uses a stack and conducts a depth first search to a particular depth (i.e., number of moves), β. If a solution is found, it is returned, otherwise β is incremented and the depth first search is performed again.

In both algorithms, the heuristic cost estimation is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the number of moves used to reach a particular configuration from the initial configuration, and the heuristic, $h(n)$ computes a lower bound on the number of moves necessary to find a configuration without mis-overlays. For $h(n)$, we use one of two cost estimation heuristics. The first heuristic, which we call the *direct* lower bound, is a simple lower bound given by the number of mis-overlays in the current node. This is the same lower bound used in the A* approach in Expósito-Izquierdo et al. (2012). The second cost estimation heuristic is provided in Bortfeldt and Forster (2012), which we refer to as the EMO (extended mis-overlay) lower bound.[2] Since both heuristic cost estimations are lower bounds, our A* and IDA* approaches will find optimal solutions.

At each step of the search, both algorithms use the following branching function. Given a bay configuration, we generate all possible successor configurations by enumerating all possible moves (f, t), $f, t \in S$ from a stack f to a stack t. As this enumerates every possible bay configuration, it will clearly find the optimal solution, if there is one. Since many moves result in symmetries or obviously dominated states, we prune the moves applied to a configuration using a number of branching rules, which we describe in the following section.

[2] A similar bound is provided in Voß (2012).

The EMO lower bound is able to take into account more than just the "obviously" mis-overlaid containers, and attempts to determine which containers that are currently not mis-overlaid will have to be rehandled in order to achieve a non-mis-overlaid bay. The EMO bound has not been previously used in any optimal approaches, and we note that the bound lacks the property of *consistency*. This means that given a sequence of moves, the EMO bound is not monotonically increasing across the configurations created through those moves. Thus, we implement a slight variation of the algorithm implemented in Bortfeldt and Forster (2012) in which the heuristic cost of the parent state is stored and used to enforce monotonicity.

For A*, we implement a tie breaking strategy to distinguish between configurations with the same lower bound. We look at the containers at the top of each stack, and distinguish between two configurations by the lowest priority container, as a configuration with the lowest priority container on top is more likely to be able to reduce the number of mis-overlays in the next move.

4.2. Implementation details

We briefly describe implementation details for A* and IDA* that were not present in previous approaches that help us achieve better performance.

Memory requirements for A* are an issue because the search expands an exponential number of nodes. Thus, we use *trailing*, a technique often used in constraint programming solvers (Schulte and Carlsson 2006), in order to reduce our memory overhead. The basic idea is to save the moves needed to reach a given node rather than the entire bay configuration. Such an adjustment allows the algorithm to scale further.

We also implement a memoization scheme for A* which stores information about the configurations visited in order to avoid exploring any configuration more than once. We note that using memoization essentially removes any gains in memory usage from using trailing.

We use a standard backtracking search to implement IDA*, which allows our IDA* to process a high number of nodes per second while maintaining a small memory footprint.

5. Branching Rules

In this section, we describe four classes of A* and IDA* branching rules for the pre-marshalling problem, three of which are novel contributions to the literature. Our branching rules analyze past moves to prune future moves from the search fringe that cause symmetries or lead to obviously dominated states.

We first describe the standard move reversal prevention rule, which has been used in the previous A^* approach by Expósito-Izquierdo et al. (2012). For the next two classes of rules, which prevent unrelated and transitive moves, we describe two variants covering the *directly successive* and *successive* cases, which refer to situations that occur in moves m_i and m_{i+1}, or moves m_i and m_j, $i < j$,

respectively. Although the directly successive case is simply a special case of successive moves, our directly successive rules can be detected and analyzed in constant time. In the case of successive moves leading to symmetries or dominated states, the previous move list must be examined at each search node. While not overly computationally expensive, examining the previous move list does present a trade-off in terms of pruning power and computational requirements worth investigating further. Finally, we provide a rule to avoid symmetries related to empty stacks.

5.1. Move reversal prevention

The first branching rule involves suppressing moves that reverse directly previous moves. We do not describe this rule in detail due to its simplicity, and the fact that it has also been used by Expósito-Izquierdo et al. (2012). Briefly formalized, given two directly successive moves $m_1 = (f_1, t_1)$ and $m_2 = (f_2, t_2)$, we do not apply move m_2 if $t_1 = f_2$ and $t_2 = f_1$, i.e., if move m_2 undoes m_1. If m_2 were to be performed, the resulting search node would have the same configuration as before m_1 was applied, but with a higher lower bound, which is clearly a dominated state.

5.2. Unrelated move symmetry breaking

We first target successive moves that do not share any from or to stacks in common. Starting from a given configuration, such moves can be ordered arbitrarily and the same resulting configuration is reached. We call such moves *unrelated* moves, and address the directly successive and successive cases as follows.

5.2.1. Directly successive moves We formally define the concept of unrelated moves as follows.

DEFINITION 1. Given two successive moves, $m_1 = (f_1, t_1)$ and $m_2 = (f_2, t_2)$, moves m_1 and m_2 are *unrelated* iff $f_1 \neq f_2 \neq t_1 \neq t_2$.

Figure 3 shows a symmetry caused by two unrelated moves performed in direct succession. Move 1 involves shifting container 3 from stack a to stack b, and move 2 consists of moving container 4

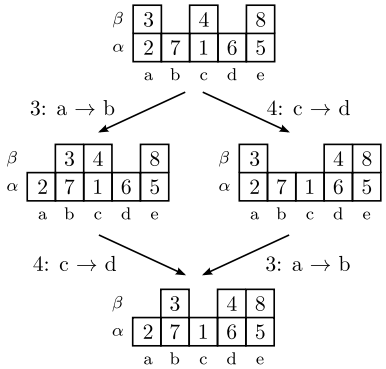


Figure 3 A symmetry caused by unrelated moves.

from stack c to stack d. From the original bay configuration, we perform move 1 on the left and move 2 on the right. Then, at the next level of the tree, we perform move 2 on the node on the left and move 1 on the node on the right. Both orderings of move 1 and move 2 result in the same final configuration, meaning without a symmetry breaking rule we will explore the same configuration twice. In order to break unrelated move symmetries, we examine directly successive moves and impose an ordering restriction on the from stack of each move. We formalize this notion as follows.

RULE 1. Given a move $m_1 = (f_1, t_1)$ and a directly successive unrelated move $m_2 = (f_2, t_2)$, m_2 is allowed only if $f_1 \diamond f_2$, where $\diamond \in \{<, >\}$ is fixed over the entire search.

In other words, a directly successive unrelated move is only allowed if its from stack is strictly less than (greater than) the from stack of the move preceding it. We note that $\diamond$ must remain consistent throughout the search tree in order for the rule to function. Furthermore, we only apply this rule to moves that are *directly successive*, i.e., they happen directly after each other.

PROPOSITION 1. *Both A* and IDA* remain complete when applying Rule 1.*

PROOF. Consider directly successive unrelated moves m_1 and m_2 with $f_1 \diamond f_2$ as described in Rule 1, and a search node n_1. We can apply m_1 and m_2 to n_1 in any order to achieve node n_2 because the moves are unrelated. By imposing an order $\diamond$ on to which unrelated moves are allowed, we now only accept a single ordering of m_1 and m_2. However, since both orderings of the moves generate the same node, no parts of the search tree are lost. Thus, the search remains complete. □

5.2.2. Successive moves In the case of successive unrelated moves, Rule 1 is an insufficient condition to detect symmetries. We now refer to two moves $m_i = (f_i, t_i)$ and $m_j = (f_j, t_j)$ with $i < j$. Rule 1 is insufficient because moves may have occurred between moves m_i and m_j that actually make m_i and m_j related. For example, assume there is a move from t_i to f_j ordered between m_i and m_j. In such a situation, preventing m_j from being carried out could cut off portions of the search tree that must be explored. We therefore keep our previous definition of unrelated moves, but only apply it in a strict set of situations, defined as follows.

RULE 2. Given a move $m_i = (f_i, t_i)$ and a successive unrelated move $m_j = (f_j, t_j)$, $i < j$, we allow m_j only if $f_i \diamond f_j$, where $\diamond \in \{<, >\}$ is fixed over the entire search, and $\forall_{i<k<j}\, m_k = (f_k, t_k)$, $\{f_k, t_k\} \cap \{f_i, t_i, f_j, t_j\} = \emptyset$.

The rule ensures that no move in between m_1 and m_2 modifies any of the stacks involved in the two moves. When this condition holds, we can be sure that applying Rule 2 will not cut off any parts of the search space that need to be explored to guarantee completeness.

PROPOSITION 2. *Both A* and IDA* remain complete when applying Rule 2.*

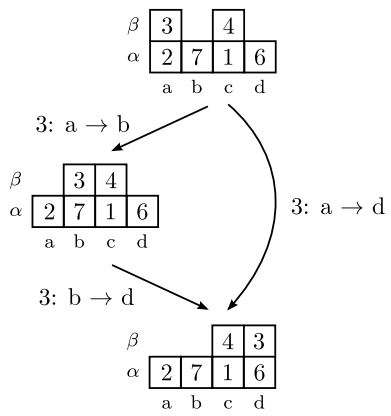


Figure 4 A dominated state caused by transitive moves.

PROOF. Rule 2 is only applied if it can be shown that no move between m_i and m_j modifies any of the stacks changed in m_i and m_j. Thus, the intermediate moves between m_i and m_j do not have any effect on the nodes generated by m_i and m_j. This means that Proposition 1 can be directly applied to Rule 2 and the search remains complete. □

5.3. Transitive move avoidance

Our second type of rule prevents moves that are clearly dominated, but cannot always be pruned purely by examination of the lower bound. In this situation, the same container is moved multiple times to achieve a configuration that can also be reached by simply moving the container a single time. We call such double moves *transitive moves*. These moves result in a symmetry-like situation, where the same configuration is reached twice, but with slightly different lower bounds.

5.3.1. Directly successive moves We formally define transitive moves as follows.

DEFINITION 2. Two successive moves $m_1 = (f_1, t_1)$ and $m_2 = (f_2, t_2)$ are *transitive* if $t_1 = f_2$.

Figure 4 shows an example situation in which a transitive move is dominated by a single move. The left path moves container 3 first from stack a to b and then from b to d, whereas the move on the right moves the container directly from a to d. Clearly using a single move dominates two moves as the cost of solving the configuration is the same between the two possibilities, but the number of moves required to reach the configuration is less when one move is used. Based on this definition, we can form the following rule to prevent transitive moves from being performed.

RULE 3. Given two directly successive moves $m_1 = (f_1, t_1)$ and $m_2 = (f_2, t_2)$, disallow m_2 if m_1 and m_2 are transitive moves.

We now show that we can use this rule and still be guaranteed to find the optimal solution.

PROPOSITION 3. *Both A* and IDA* remain complete when applying Rule 3.*

PROOF. Any search node n_3 that is reached by applying transitive moves m_1 and m_2 in direct succession to node n_1 can also be reached by applying some other move, $m_3 = (f_1, t_2)$. Since reaching

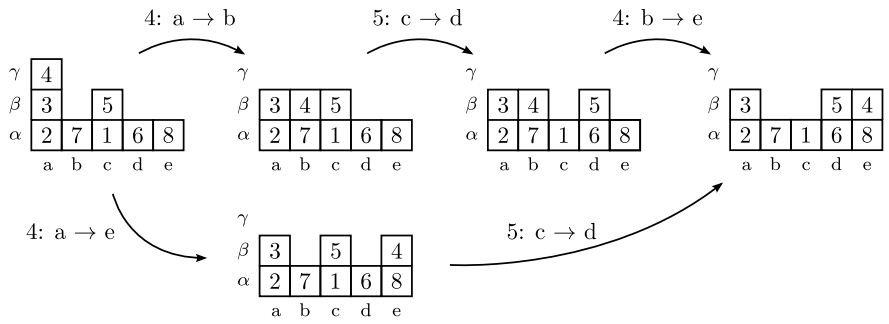


Figure 5 **Transitive moves that are not directly successive.**

n_3 using m_3 is cheaper than using m_1 and m_2, the sequence m_1 and m_2 is dominated and will never be in an optimal configuration. Since m_3 will always be applied to node n_1, applying m_2 after m_1 is not necessary to achieve a complete search space. □

5.3.2. Successive moves As in the case of unrelated move symmetries, our transitive move branching rule can also be extended to the case of successive moves. For successive moves, the pruning only applies to moves between stacks that have not been changed between the two transitive moves.

Figure 5 shows an example situation in which a transitive move is separated over several steps. The top sequence of moves contains the transitive moves $4 : a \to b$ and $4 : b \to e$, which is dominated by the bottom row's move $4 : a \to e$. Looking at the third configuration in the top row, directly after the move $5 : c \to d$, container 4 can be moved to stacks c and d without causing a transitive move, but cannot be moved to a or e.[3] We allow container 4 to be moved to stacks c and d because they have changed since container 4 was first moved from stack a to stack b. In order to retain completeness of the algorithm, we must allow such moves to be possible, as they may be necessary for finding an optimal move sequence.

RULE 4. Given two successive moves $m_i = (f_i, t_i)$ and $m_j = (f_j, t_j)$, $i < j$, disallow m_j if (i) m_i and m_j are transitive moves, and (ii) there is no move $m_k = (f_k, t_k), i < k < j$ such that $\{f_k, t_k\} \cap \{f_j, t_j\} \neq \emptyset$.

This rule prevents m_j from being performed if it is transitive with move m_i, no move has taken place changing the state of stacks f_j or t_j, and there is no other move that could qualify as transitive with m_j after m_i. When this is the case, we know that the move (f_i, t_j) was already carried out during the branching step when m_i was applied. We now prove this.

PROPOSITION 4. *Both A* and IDA* remain complete when applying Rule 4.*

[3] Moving container 4 to stack a is actually a move reversal as discussed in Section 5.1, which is a special case of successive transitive moves.

PROOF. If the intervening moves between m_i and m_j do not affect the search's completeness, then it suffices to apply Proposition 3 to show that the search is complete. Let n_j be the search node reached by applying m_j. In any case where m_j is banned, the search can also explore n_j by applying the move (f_i, t_j) and then moves $m_{i+1} \dots m_{j-1}$. Since moves $m_{i+1} \dots m_{j-1}$ do not change stacks f_j or t_j in any way, whether they are applied before or after m_j is irrelevant and has no bearing on the completeness of the search space. Since the only search node in question is n_j, and this node is visited even when banning m_j according to Rule 4, the search remains complete. □

5.4. Empty stack symmetry breaking

Symmetries can occur when a configuration has multiple empty stacks available at the same time. This is because when a container is moved into an empty stack, it does not matter which stack receives the container from the point of view of the lower bound. We note that breaking empty stack symmetries does not generally result in a significant performance gain, since empty stacks tend to be filled rather quickly with containers. Thus, not many nodes have empty stack symmetries present in the first place. In contrast to the previous rules we have presented, previous moves need not be examined in order to break empty stack symmetries. Rather, we impose an ordering over the empty stacks to ensure a container is only moved into a single empty stack at any branching step.

RULE 5. Given a search node n_1 with at least two empty stacks, represented by the set E, let $e^* = \min\{E\}$ be the empty stack with the lowest index. Disallow any move $m_1 = (f_1, t_1)$ applied to n_1 with $t_1 > e^*$.

The search remains complete under this rule since all empty stacks are equivalent, thus there is no need to branch on moving a particular container into multiple empty stacks.

6. Computational Experiments

We implemented both A* and IDA* for the pre-marshalling problem in C++11 (ISO/IEC 2011). We ran all experiments on AMD Opteron 6386 2.8 GHz processors for up to one hour and allowed up to 5 GB of RAM to be used. We first show the effectiveness of the unrelated symmetry breaking and transitive move avoidance rules on IDA*, and then provide results of IDA* versus A* and previous work. We only give results for our branching rules on IDA* since in general, the branching rules help both IDA* and A* a similar amount.

6.1. Datasets

We use three well-known datasets from the literature to test the effectiveness of our approach. The Caserta and Voß dataset (CV), from Caserta and Voß (2009), consists of 880 instances ranging in size from 3 stacks and 3 tiers up to 10 stacks and 10 tiers filled completely with containers. As

1. Algorithm	
A*, IDA*	
2. Unrelated move symmetries	
$S^<$	Single-level, less-than
$S^>$	Single-level, greater-than
$M^<$	Multi-level, less-than
$M^>$	Multi-level, greater-than
3. Transitive move avoidance	
T	Single-level
U	Multi-level
4. Empty stack symmetry breaking	
E	Empty stack symmetry breaking enabled
5. A* specific parameters	
O	State memoization enabled
I	Tie breaking enabled
6. Lower bound	
EMO	EMO lower bound Bortfeldt and Forster (2012)
D	"Direct" lower bound as described in Section 4

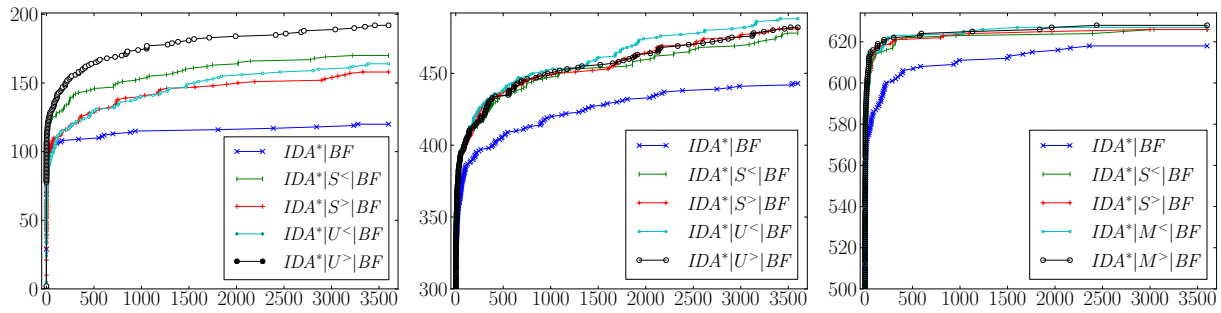
Figure 6 Overview of the parameterizations of the IDA* and A* algorithms.

in Caserta and Voß (2009), we add two empty tiers on top of each instance so that they can actually be solved. Each container receives its own priority, which is a unique feature of the CV dataset over other datasets. The Bortfeldt and Forster dataset (BF) (Bortfeldt and Forster 2012) consists of 721 instances, 40 of which are sourced from the CV dataset, and another 41 of which are variations of the instance used in Lee and Chao (2009). We remove the CV instances, but leave the instance from Lee and Chao (2009) and its variations, which are prefixed in our results tables with "LC". The BF instances are categorized into 32 different groups, which contain either 16 or 20 stacks, and either 5 or 8 tiers. The instances are not as densely filled as the CV instances, and the number of priorities is always less than the number of containers, meaning each priority contains multiple containers. Finally, we use the instance generator from Expósito-Izquierdo et al. (2012) to generate a dataset of 700 instances.[4] We generate a dataset similar to the one used in Expósito-Izquierdo et al. (2012), generating 25 instances with 4 tiers; 4, 7 or 10 stacks; a container fill percentage of 0.5, 0.75 or 1; and a "configuration" of either 0, 1, or 2, which describe container distributions. We call this dataset the EMM dataset.

6.2. Parameterizations

Since both our A* and IDA* algorithms have a number of parameters, we summarize our abbreviations for the parameterizations in Figure 6. For example, IDA*$|S^>E|EMO$ is IDA* with single level unrelated move symmetry breaking using the greater-than criterion, empty stack symmetry breaking, and the EMO lower bound.

[4] We note that we cannot use the exact same instances as in Expósito-Izquierdo et al. (2012) because the instances were unfortunately lost.



(a) BF (b) CV (c) EMM

Figure 7 **Number of instances solved versus time in seconds for the unrelated move symmetry breaking rule in the directly successive and successive cases using IDA*.**

6.3. Evaluation of unrelated move symmetry breaking

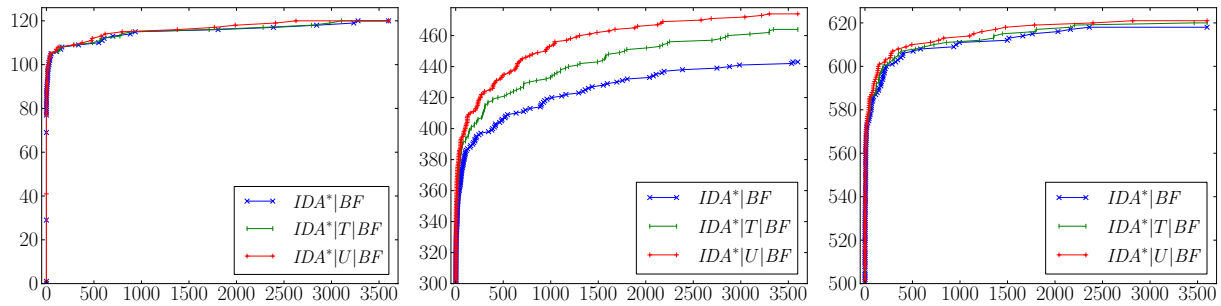
We investigate the effectiveness of unrelated move symmetry breaking on all three datasets in both the directly successive and successive cases on the IDA* algorithm. Figure 7 shows the number of instances solved on each dataset using the symmetry breaking rule with both the less-than and greater-than variants. On all three datasets using the symmetry breaking rule, even in the directly successive case, is significantly better than not using the rule at all. While the successive rule does not perform better than the directly successive rule on the CV or EMM datasets, it does greatly outperform other variants of the rule in the greater-than case for the BF dataset.

Interestingly, whether the rule is set up with a less-than or greater-than relation has a significant effect on the BF dataset, with the successive greater-than rule performing the best, followed by the directly successive less-than rule, and finally the successive less-than and directly successive greater-than rules perform roughly the same. We suspect that this is due to the exploration of multiple moves not being beneficial enough in the less-than case to warrant the extra work.

On the BF dataset, the successive greater-than rule solves 60% more instances than no rule at all, and 17% more instances than the nearest competitor, the directly successive less-than rule. The successive less-than rule solves 9% more instances on the CV dataset than not using any branching rules, but does not perform significantly better than any other unrelated symmetry breaking rule, while on the EMM dataset the unrelated symmetry breaking rule provides only minor gains in terms of total instances solved. However, the symmetry breaking rules do offer faster solutions than not using the rules, as can be seen at the left side of Figure 7(c). This means the rule is still beneficial on the EMM dataset.

6.4. Evaluation of transitive move avoidance

The effectiveness of our transitive move avoidance rules is shown in Figure 8. On the BF and EMM datasets, neither the directly successive or successive rules provide significant assistance over not using them. However it is worth noting that despite the extra work required to compute the rules,



(a) BF (b) CV (c) EMM

Figure 8 **Number of instances solved versus time in seconds for the transitive move avoidance rule in the directly successive and successive cases using IDA*.**

they do not negatively impact performance. On the EMM dataset there are small gains, with a few more instances being solved using the successive transitive move avoidance rule than without. Furthermore, some instance are solved faster, as can be seen by the "hump" in the figure between 1000 and 2000 seconds.

On the CV dataset, however, there are clear gains from using the rule, with successive transitive breaking outperforming the directly successive rule by roughly 10 instances. Both transitive move avoidance rules outperform not using them, both in terms of solution time and the number of instances solved. Since the CV instances are the most "dense" of all the instances in our datasets, we presume that this is the reason for the performance difference.

6.5. A* versus IDA*

The A* algorithm combined with the simple direct lower bound has been used for solving pre-marshalling problems to optimality in the literature in Expósito-Izquierdo et al. (2012). Since our A* and IDA* contain a number of engineering improvements over previous work, we implement the A* approach in Expósito-Izquierdo et al. (2012) within our code base.

6.5.1. BF Dataset The performance of IDA* and A* on the BF dataset is shown in Table 1. Each of the BF, LC2 and LC3 categories of instances contains 20 instances, and LC1 contains a single instance. We first note how effective both forms of the A* algorithm are at solving the LC1 instance from Lee and Chao (2009), which the authors target with heuristic approaches. The instance is easily solvable by every configuration of A* we test, with our novel extensions to IDA* and A* bringing the solution time down from 8.16 seconds to being practically instantaneous.

$A^*|M^>UOI|EMO$ is able to solve the most instances across the dataset, nearly twice as many as $A^*|EMO$ and over four times as many as $A^*|D$, showing the significant performance gains over the state-of-the-art that our branching rules and use of the EMO lower bound provide.

Although the number of instances solved by $IDA^*|M^>UE|EMO$ and $A^*|M^>UOI|EMO$ are relatively similar, $A^*|M^>UOI|EMO$ solves them significantly faster. We attribute this to the fact

Table 1 **Average CPU time in seconds (on instances solved) and number of instances solved on the various categories of the BF dataset.**

Category	$IDA^*\|M^>UE\|EMO$		$IDA^*\|EMO$		$IDA^*\|D$		$A^*\|M^>UOI\|EMO$		$A^*\|EMO$		$A^*\|D$	
BF1	49.17	18	365.67	15	0.00	2	0.73	12	27.83	14	0.04	2
BF2	99.25	15	311.65	10	280.13	10	3.69	17	1.07	15	0.73	8
BF3	365.11	12	183.19	7	13.21	2	10.19	10	1.31	9	0.02	1
BF4	13.51	18	17.15	14	69.18	5	22.78	15	-	0	0.03	4
BF5	264.61	8	423.34	11	-	0	34.61	10	-	0	-	0
BF6	311.36	1	-	0	-	0	36.73	13	-	0	-	0
BF7	470.07	12	997.00	15	-	0	60.99	7	-	0	-	0
BF8	146.93	3	966.73	3	-	0	45.39	5	-	0	-	0
BF9	902.10	7	684.51	6	-	0	60.85	1	-	0	-	0
BF10	130.00	3	330.56	2	-	0	-	0	0.08	1	-	0
BF11	633.76	4	567.99	4	-	0	92.15	2	-	0	-	0
BF12	2816.65	1	1246.81	1	-	0	-	0	-	0	-	0
BF13	-	0	-	0	-	0	-	0	-	0	-	0
BF14	-	0	-	0	-	0	-	0	-	0	-	0
BF15	-	0	-	0	-	0	-	0	-	0	-	0
BF16	-	0	-	0	-	0	-	0	-	0	-	0
BF17	371.73	7	24.70	8	0.00	3	15.70	14	0.07	12	0.05	3
BF18	7.74	16	55.77	13	212.24	14	6.99	14	0.39	16	0.14	12
BF19	1.05	11	545.90	8	0.00	4	20.72	11	12.12	10	0.05	4
BF20	8.66	14	152.22	12	132.00	10	6.72	8	0.08	14	11.82	9
BF21	701.75	7	376.76	5	-	0	21.57	8	9.02	2	-	0
BF22	-	0	2811.51	1	-	0	17.57	3	0.09	1	-	0
BF23	22.54	3	1723.35	3	-	0	23.58	8	119.67	2	-	0
BF24	-	0	-	0	-	0	173.55	2	-	0	-	0
BF25	-	0	922.24	1	-	0	109.61	1	-	0	-	0
BF26	-	0	-	0	-	0	482.29	1	-	0	-	0
BF27	268.91	2	1104.14	1	-	0	70.24	1	-	0	-	0
BF28	220.91	1	1868.77	1	-	0	-	0	-	0	-	0
BF29	-	0	-	0	-	0	-	0	-	0	-	0
BF30	-	0	-	0	-	0	-	0	-	0	-	0
BF31	-	0	-	0	-	0	-	0	-	0	-	0
BF32	-	0	-	0	-	0	-	0	-	0	-	0
LC1	0.00	1	0.00	1	8.16	1	0.00	1	0.00	1	56.78	1
LC2a	72.36	9	22.27	9	-	0	3.28	8	-	0	-	0
LC2b	388.93	7	184.35	7	-	0	25.26	6	-	0	-	0
LC3a	218.79	6	694.34	8	-	0	55.05	6	-	0	-	0
LC3b	296.42	7	404.58	7	-	0	97.12	5	-	0	-	0
Avg/Count	222.40	173	157.79	120	146.53	51	27.80	189	8.38	96	3.89	44

that A* need not spend time reconstructing the search tree each time it increases the lower bound of a problem like IDA*, thus allowing it to save time on the relatively large instances of the BF dataset. It is also possible that the tie breaking rules in A*, which do not have a direct translation into IDA*, provide better branching decisions in the final level of the search tree.

The LC2 and LC3 instances are variations of the LC1 instance that tend to be significantly more difficult than the original. The use of the EMO lower bound is critical to being able to solve these instances, although we note that our branching rules do not seem to help on these instances, with some instances solving faster using the branching rules, but others being slowed down. Using IDA* instead of A* allows several more instances to be solved.

6.5.2. CV Dataset Our comparison of approaches on the CV dataset is shown in Table 2. The instances are split into groupings of 40 instances each, where $|S|$ is the number of stacks and

$|T|$ is the number of tiers. The current state-of-the-art $A^*|D$ approach is only able to completely solve two classes of instances. Replacing the lower bound with the EMO lower bound allows us to completely solve an additional 4 classes, closing out the 3 tier problems. Including our branching rules, memoization and tie breaking results in an additional two classes completely solved by $A^*|M^>UOI|EMO$; however, the most classes are completely solved using $IDA^*|M^>UE|EMO$, which is able to solve all 3 and 4 tier problems to optimality. $IDA^*|M^>UE|EMO$ solves 1.8 times as many instances as $A^*|D$ and an additional 70 instances over $IDA^*|EMO$. $A^*|M^>UOI|EMO$ is also able to solve larger instances than any other approach, with the largest solved having 5 tiers and 10 stacks.

These results provide an interesting contrast to the BF dataset, where A* slightly outperformed IDA*. The CV instances are significantly denser than the BF instances, and have a one-to-one assignment of priorities to containers. Although it would seem like this small size would actually benefit A*, since less memory is required to store each search node, due to the efficient memory storage of nodes in our A* implementation, this benefit is rather small. For reasons that are unclear, the better performance of IDA* is likely due to its higher node throughput and potentially due to different branching decisions. IDA* evaluates between 10 and 74 times as many search nodes per second than A* on CV instances. Note, though, that some of these are evaluated multiple times, unlike in A*.

We note that although the overall average computation time required by the A* appears to be lower, this is only because larger instances are not solved.

6.5.3. EMM Dataset We present our results on the EMM dataset in Table 3, where $|S|$ is the number of stacks, p is the container density and C is the configuration type. All instances have 4 tiers. $IDA^*|M^>UE|EMO$ 1.4 times as many instances as $A^*|D$. In addition, on instances solved by both approaches $IDA^*|M^>UE|EMO$ is faster, with a number of categories, such as 4-1-1, 4-1-2 and 7-0.75-2 solving essentially instantly with $IDA^*|M^>UE|EMO$, but requiring significant CPU time with $A^*|D$. $IDA^*|M^>UE|EMO$ is almost able to solve all of the instances in the dataset, with the exception of several instances with 10 stacks and fill percentages of 0.75 and more, whereas $A^*|D$ begins to struggle with 7 stacks and a fill percentage of 0.75.

IDA* offers large performance gains over A*, and the use of the EMO lower bound also provides performance gains over the state-of-the-art A* approach. However, our branching rules only offer mild gains over not using them on the EMM dataset.

7. Conclusion

In this paper we have considered the container pre-marshalling problem, which is an important problem in maritime shipping for reducing the delay of inter-modal container transfers. We presented novel A* and IDA* algorithms, including several novel branching and symmetry breaking

Table 2 **Average CPU time in seconds (on instances solved) and number of instances solved on the various categories of the CV dataset.**

$\|T\|$	$\|S\|$	Moves	$IDA^*\|M^>UE\|EMO$		$IDA^*\|EMO$		$IDA^*\|D$		$A^*\|M^>UOI\|EMO$		$A^*\|EMO$		$A^*\|D$	
3	3	8.78	0.00	40	0.00	40	0.01	40	0.01	40	0.02	40	0.07	40
	4	9.03	0.00	40	0.00	40	0.03	40	0.01	40	0.02	40	0.41	40
	5	10.15	0.01	40	0.03	40	0.50	40	0.04	40	0.13	40	5.42	39
	6	11.28	0.02	40	0.07	40	4.60	40	0.07	40	0.17	40	17.28	38
	7	12.80	0.05	40	0.25	40	23.91	40	0.15	40	0.56	40	33.37	35
	8	13.53	0.11	40	0.55	40	33.63	39	0.17	40	0.56	40	58.58	32
4	4	15.82	1.22	40	12.46	40	114.69	40	6.99	40	83.69	35	459.55	24
	5	17.85	3.75	40	36.56	40	524.61	39	7.67	40	39.33	37	119.13	17
	6	19.30	24.26	40	179.41	39	563.10	26	77.71	38	72.65	25	51.85	9
	7	21.82	45.73	40	422.05	38	954.49	16	135.70	33	85.65	25	171.86	3
5	4	-	270.61	32	613.45	21	347.66	10	629.22	21	413.78	7	1071.42	3
	5	-	530.16	34	1035.71	14	884.22	2	870.33	11	253.07	3	-	0
	6	-	1148.75	16	1427.66	3	-	0	559.73	3	1052.07	1	-	0
	7	-	424.06	15	1153.23	7	-	0	223.93	6	429.96	2	-	0
	8	-	1762.02	10	889.66	1	-	0	121.25	1	-	0	-	0
	9	-	1592.46	3	-	0	-	0	-	0	-	0	-	0
	10	-	2261.26	3	-	0	-	0	-	0	-	0	-	0
6	6	-	-	0	-	0	-	0	-	0	-	0	-	0
	10	-	-	0	-	0	-	0	-	0	-	0	-	0
10	6	-	-	0	-	0	-	0	-	0	-	0	-	0
	10	-	-	0	-	0	-	0	-	0	-	0	-	0
Avg/Count			162.99	513	148.22	443	168.49	372	78.44	433	37.25	375	75.65	280

Table 3 **Average CPU time in seconds (on instances solved) and number of instances solved on the various categories of the EMM dataset.**

$\|S\|$	p	C	Moves	$IDA^*\|M^>UE\|EMO$		$IDA^*\|EMO$		$IDA^*\|D$		$A^*\|M^>UOI\|EMO$		$A^*\|EMO$		$A^*\|D$	
4	0.5	0	2.48	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25
		1	1.92	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25
		2	0.84	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25
	0.75	0	13.00	0.00	25	0.00	25	0.04	25	0.00	25	0.00	25	0.51	25
		1	5.76	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25
		2	1.96	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25
	1	0	19.40	0.15	25	0.51	25	14.76	25	0.43	25	2.17	25	539.11	15
		1	12.20	0.02	25	0.06	25	0.35	25	0.03	25	0.28	25	27.52	25
		2	9.36	0.03	25	0.12	25	0.58	25	0.05	25	0.31	25	89.87	25
7	0.5	0	7.28	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.02	25
		1	3.56	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25
		2	3.00	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25
	0.75	0	19.24	0.05	25	0.25	25	-	0	-	0	-	0	-	0
		1	10.52	0.03	25	0.82	25	17.09	25	0.14	25	25.13	25	2.10	24
		2	6.32	0.00	25	0.00	25	0.29	25	0.00	25	0.00	25	8.23	25
	1	0	28.20	1.22	25	4.63	25	-	0	-	0	-	0	-	0
		1	18.72	10.50	25	105.49	25	674.23	19	43.89	24	15.87	20	70.26	7
		2	15.56	8.13	25	132.33	25	534.91	19	93.97	23	18.98	18	182.73	4
10	0.5	0	9.32	0.00	25	0.02	25	0.91	25	0.01	25	0.14	25	0.35	24
		1	6.56	0.00	25	0.01	25	0.04	25	0.00	25	0.01	25	0.20	25
		2	3.56	0.00	25	0.00	25	0.00	25	0.00	25	0.00	25	0.03	25
	0.75	0	-	0.37	10	0.87	10	-	0	-	0	-	0	-	0
		1	-	0.17	10	3.09	10	235.32	8	0.02	10	9.38	10	46.79	6
		2	-	0.01	10	0.03	10	12.84	8	0.02	10	0.11	10	4.98	6
	1	0	37.76	60.19	25	28.23	24	-	0	-	0	-	0	-	0
		1	-	116.23	23	382.87	20	1139.05	3	-	0	-	0	-	0
		2	-	95.96	24	435.61	19	1430.33	3	47.92	21	15.17	12	24.16	1
Avg/Count				11.15	627	36.82	618	69.11	485	8.26	513	3.31	495	29.43	437

rules. Our approach significantly outperformed the state-of-the-art A* technique for solving pre-marshalling problems to optimality, solving 568 previously unsolved instances to optimality. Com-

binations of our approaches with (meta)heuristic techniques could be considered in future work, such as using a beam search or integrating IDA* within the corridor method. It is also possible that our branching rules are applicable to similar problems, such as the blocks relocation problem, and could be of assistance for solving this problem to optimality.

Bibliography

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

Caserta, M., S. Schwarze, S. Voß. 2011. Container rehandling at maritime container terminals. J.W. Böse, ed., *Handbook of Terminal Planning*, *Operations Research/Computer Science Interfaces Series*, vol. 49. Springer, New York, 247–269.

Caserta, M., S. Voß. 2009. A corridor method-based algorithm for the pre-marshalling problem. *Lecture Notes in Computer Science* **5484** 788–797.

Choe, R., T. Park, M.S Oh, J. Kang, K.R. Ryu. 2011. Generating a rehandling-free intra-block remarshaling plan. *Journal of Intelligent Manufacturing* **22**(2) 201–217.

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

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

Felsner, S., M. Pergel. 2008. The complexity of sorting with networks of stacks and queues. *Lecture Notes in Computer Science* **5193** 417–429.

Gheith, M.S., A.B. El-Tawil, N.A. Harraz. 2013. A proposed heuristic for solving the container pre-marshalling problem. E. Qi, J. Shen, R. Dou, eds., *The 19th International Conference on Industrial Engineering and Engineering Management*. Springer, Berlin, 955–964.

Gupta, N., D.S. Nau. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* **56**(2–3) 223–254.

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

ISO/IEC. 2011. Information technology – Programming languages – C++, Third Edition. ISO/IEC 14882:2011, International Organization for Standardization / International Electrotechnical Commission, Geneva, Switzerland.

Jovanovic, R., M. Tuba, S. Voß. 2013. A multi-heuristic approach for solving the pre-marshalling problem. Tech. rep., Institute of Information Systems, University of Hamburg, Hamburg, Germany.

Kang, J., M.S. Oh, E.Y. Ahn, K.R. Ryu, K.H. Kim. 2006a. Planning for intra-block remarshalling in a container terminal. *Lecture Notes in Artificial Intelligence* **4031** 1211–1220.

Kang, J., K.R. Ryu, K.H Kim. 2006b. Deriving stacking strategies for export containers with uncertain weight information. *Journal of Intelligent Manufacturing* **17**(4) 399–410.

Kim, K.H. 1997. Evaluation of the number of rehandles in container yards. *Computers & Industrial Engineering* **32** 701–711.

Kim, K.H., J.W. Bae. 1998. Re-marshalling export containers in port container terminals. *Computers & Industrial Engineering* **35**(3–4) 655–658.

Kim, K.H., G.-P. Hong. 2006. A heuristic rule for relocating blocks. *Computers & Operations Research* **33**(4) 940–954.

Klaws, J., R. Stahlbock, S. Voß. 2011. Container terminal yard operations - simulation of a side-loaded container block served by triple rail mounted gantry cranes. *Lecture Notes in Computer Science* **6971** 243–255.

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

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

Lee, Y., Y.-J. Lee. 2010. A heuristic for retrieving containers from a yard. *Computers & Operations Research* **37**(6) 1139–1147.

Nishi, T., M. Konishi. 2010. An optimisation model and its effective beam search heuristics for floor-storage warehousing systems. *International Journal of Production Research* **48** 1947–1966.

Park, K., T. Park, K.R. Ryu. 2009. Planning for remarshalling in an automated container terminal using cooperative coevolutionary algorithms. *SAC 2009 – Honolulu, Hawaii*. 1098–1105.

Rendl, A., M. Prandtstetter. 2013. Constraint models for the container pre-marshaling problem. G. Katsirelos, C.-G. Quimper, eds., *ModRef 2013: 12th International Workshop on Constraint Modelling and Reformulation*. 44–56.

Rodrigue, J.P., C. Comtois, B. Slack. 2009. *The Geography of Transport Systems*. 2nd ed. Routledge, Milton Park.

Romero, A.G., R. Alquézar. 2004. To block or not to block? *Lecture Notes in Computer Science* **3315** 134–143.

Russell, S.J., P. Norvig. 2010. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Salido, M.A., M. Rodriguez-Molins, F. Barber. 2012. A decision support system for managing combinatorial problems in container terminals. *Knowledge-Based Systems* **29** 63–74.

Salido, M.A., O. Sapena, M. Rodriguez, F. Barber. 2009. A planning tool for minimizing reshuffles in container terminals. *21st IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 567–571.

Schulte, C., M. Carlsson. 2006. Finite domain constraint programming systems. F. Rossi, P. van Beek, T. Walsh, eds., *Handbook of Constraint Programming*, *Foundations of Artificial Intelligence*, vol. 2. Elsevier, 29 – 83.

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

Steenken, D., S. Voß, R. Stahlbock. 2004. Container terminal operations and operations research – a classification and literature review. *OR Spectrum* **26**(1) 3–49.

Tierney, K., S. Voß, R. Stahlbock. 2013. A mathematical model of inter-terminal transportation. *European Journal of Operational Research.* doi:http://dx.doi.org/10.1016/j.ejor.2013.07.007. Available Online.

UNCTAD. 2012. *United Nations Conference on Trade and Development (UNCTAD), Review of maritime transport*.

Voß, S. 2012. Extended mis-overlay calculation for pre-marshalling containers. *Lecture Notes in Computer Science* **7555** 86–91.

Zäpfel, G., M. Wasner. 2006. Warehouse sequencing in the steel supply chain as a generalized job shop model. *International Journal of Production Economics* **104**(2) 482–501.

Zhang, Y., W. Mi, D. Chang, W. Yan. 2007. An optimization model for intra-bay relocation of outbound containers on container yards. *Proceedings of the IEEE International Conference on Automation and Logistics*. 776–781.

Zhu, W., H. Qin, A. Lim, H. Zhang. 2012. Iterative deepening A* algorithms for the container relocation problem. *IEEE Transactions on Automation Science and Engineering* **9**(4) 710–722.