



Solving the pre-marshalling problem to optimality with A* and IDA*

Tierney, Kevin; Pacino, Dario; Voß, Stefan

Published in:
Flexible Services and Manufacturing Journal

Link to article, DOI:
[10.1007/s10696-016-9246-6](https://doi.org/10.1007/s10696-016-9246-6)

Publication date:
2017

Document Version
Peer reviewed version

[Link back to DTU Orbit](#)

Citation (APA):
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. <https://doi.org/10.1007/s10696-016-9246-6>

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

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

Kevin Tierney · Dario Pacino · Stefan Voß

the date of receipt and acceptance should be inserted later

Abstract We present a novel solution approach to the container pre-marshalling problem using the A* and IDA* algorithms combined with several novel branching and symmetry breaking rules that significantly increases the number of pre-marshalling instances that can be solved to optimality. A* and IDA* are graph search algorithms that use heuristics combined with a complete graph search to find optimal solutions to problems. The container pre-marshalling problem is a key problem for container terminals seeking to reduce delays of inter-modal container transports. The goal of the container pre-marshalling problem is to find the minimal sequence of container movements to shuffle containers in a set of stacks such that the resulting stacks are arranged according to the time each container must leave the stacks. We evaluate our approach on three well-known datasets of pre-marshalling problem instances, solving over 500 previously unsolved instances to optimality, which is nearly twice as many instances as the current state-of-the-art method solves.

e

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 [49]. 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 [37]. 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.

K. Tierney
Decision Support & Operations Research Lab
University of Paderborn
33098 Paderborn, Germany
E-mail: kevin.tierney@uni-paderborn.de

D. Pacino
DTU Management Engineering
Technical University of Denmark
2800 Kgs. Lyngby, Denmark
E-mail: darpa@dtu.dk

S. Voß
Institute of Information Systems
University of Hamburg
20146 Hamburg, Germany
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso
Valparaíso, Chile
E-mail: stefan.voss@uni-hamburg.de

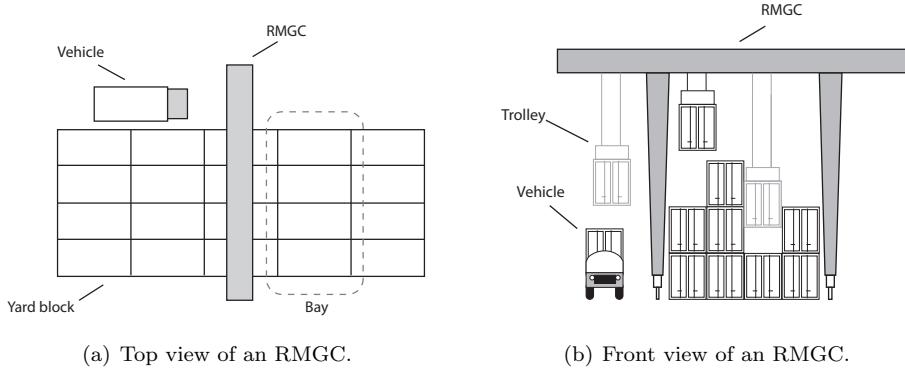


Fig. 1: A Rail Mounted Gantry Crane (RMGC) over a yard block.

A number of factors affect the speed of container transfer operations, such as intra-terminal container handling (see [45]), inter-terminal transportation (e.g., [48]) 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. Pre-marshalling is not to be confused with the general housekeeping where containers are repositioned along the entire yard (i.e., between blocks) to improve transhipment operations. See, for example, [?], in which a simulation study investigates housekeeping operations at an Italian container terminal without pre-marshalling operations. Most commonly, one may categorize container movements as follows: (i) pre-marshalling, if the movements are bay-oriented, that is, movements are done within a bay (ii) housekeeping, if the movements are yard-oriented, that is, they are done within the yard of a terminal and (iii) inter-terminal transport, if the movements are done between different terminals or locations within a port.

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 and move the containers on top of the stacks. [29] point out that pre-marshalling is often only performed within a bay due to the high costs of bay to bay moves. One can, however, still model bay to bay moves by creating a single bay formed of all the stacks and assigning different costs to moves between stacks in different bays. 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., [29, 28, 6]), only two approaches have been proposed for solving the pre-marshalling problem to optimality. The first, using integer programming, is proposed in [29] and the second is a basic A* algorithm from [11]. Optimal approaches, unless they can be executed within reasonable times, are often used to quantify the performance of heuristic methods and policies. The aforementioned exact approaches for the pre-marshalling problem are, however, only able to solve very small instances, making it hard for an interested terminal operator to evaluate the quality of their operations or the numerous published heuristic approaches.

To this end, we present a novel solution technique for solving pre-marshalling problems to optimality using the A* [15] and IDA* [27] algorithms combined with specialized branching rules, symmetry breaking

and strong lower bounds from the literature. The A* and IDA* algorithms are graph search methods that, similar to Dijkstra's shortest path algorithm [10], use a cost based search mechanism to find shortest paths through a graph. We model the pre-marshalling problem as a graph and apply A* and IDA* to this to find optimal solutions. 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 [2] 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, where tier 1 is the bottom row of the bay. We define a parameter $p_{st} \in \mathbb{N}_0$ ($s \in S, t \in T$) to be the priority of the container currently in stack s at tier t . We set $p_{st} = 0$ if there is no container at the position (s, t) . Containers with a smaller priority value must leave the bay earlier than those with a larger value, meaning they must be above containers with a larger priority value in a configuration with no mis-overlays. A bay has no mis-overlays iff $p_{st} \geq p_{s,t+1}$ 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.

We make several standard assumptions regarding pre-marshalling. The first is that the crane can only move a single container at a time. Second, we only perform pre-marshalling within a single bay. Given that the costs of moving between bays are much higher than moving within a bay [29] this assumption makes sense, as RMGCs can work through each bay one-by-one until they are sorted. While moving containers between bays might lead to less overall moves, the total time would be significantly higher. We further note, however, that if moves between bays are assumed to be low cost, our method can be applied to an instance consisting of the bays appended together. Our third assumption is that the exit times of the containers have a single deterministic value. Of course in reality it is possible that delays of ships or trucks could change the exit time of a container while it is in the stacks. We believe this does not affect enough containers to invalidate the approach, not to mention pre-marshalling can be re-run on a bay over several time periods as new information is gathered.

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.

The pre-marshalling problem can be seen as a problem arising in tactical and operational decision making at a yard; see, e.g., [9, 5]. Following [7, 33, 22], 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 and or housekeeping policies refer to the

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.

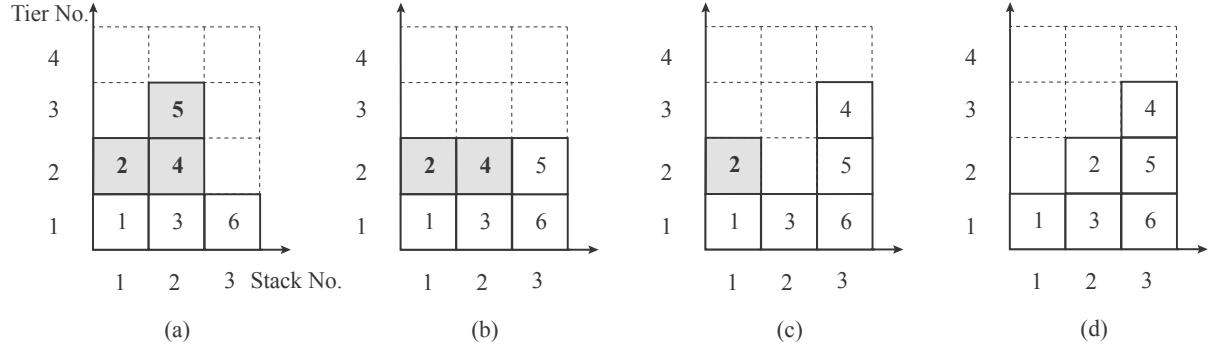


Fig. 2: An example solution to the pre-marshalling problem. Mis-overlays are indicated by the highlighted containers.

pre-arrangement 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.

As pointed out by many authors, e.g., [33, 22, 24], even when stacking policies aimed at minimizing the number of rehandling moves are used, later rehandling cannot be avoided altogether. There are several reasons why rehandling often cannot be prevented. First, containers being shipped with different vessels are stored together due to the limited space capacity of the yard. Furthermore, the loading plan for a vessel is not yet determined when a container arrives at the yard, and is often only finalized hours before a vessel arrives. In addition, trucks or ships might be delayed, etc. In an early paper by [23] 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 [46], [45], [5] and [3]. A more specific overview of some of the most relevant optimization problems related to stacking at container terminals is given in [9], 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 recent overview of loading, unloading and pre-marshalling problems is given in [31], which provides a classification and theoretical overview of a wide range of stowage problems.

We now describe the most relevant literature to the pre-marshalling problem, followed by an overview of work on similar problems, such as the blocks relocation problem (BRP), which is also known as the container relocation problem, terminal re-marshalling problems and steel coil stacking.

Related work in pre-marshalling

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” [7].

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., [28, 29], 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).

Article	Method	Lower bound heuristic
Lee and Hsu (2007) [29]	Mixed-integer linear programming	Standard LP-relaxation
Expósito-Izquierdo et al. (2012) [11]	A*	Direct lower bound
Prandtstetter (2013) [35]	Dynamic programming	-
Rendl and Prandtstetter (2013) [36]	Constraint programming	Bortfeldt & Forster lower bound [2]
Zhang et al. (2015) [55]	Heuristic guided branch-and-bound	Extended direct lower bound
This article	A*, IDA*	Multiple lower bounds; multiple symmetry breaking and branching rules

Table 1: A categorization of optimal pre-marshalling algorithms.

There have been various approaches for solving the pre-marshalling problem. As the problem is NP-hard (see [5]), these approaches incorporate heuristics, metaheuristics as well as exact algorithms.

An optimization model for the pre-marshalling problem is proposed by [29]. 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 [55], an approach similar to our work is presented using less general versions of our branching rules. Table 1 provides an overview of the optimal approaches for solving the pre-marshalling problem.

In order to overcome the limitations imposed by the size of the integer programming model of [29], [28] 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 [6]. 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 [2], as well as integer programming, see [30]. Another idea is provided by [11], who describe an A* algorithm which can be considered as a blueprint for the more advanced approach described in this paper. The approach by [11] does not include any symmetry breaking or branching rules, and only uses a simple lower bound. Some comments on logical observations leading to a lower bound are provided in [51]. Some of these ideas are also incorporated in the tree search algorithm of [2], and are used to enhance the capabilities of the A* approach. The A* algorithm is also often used in automated planning tools. A domain specific heuristic for pre-marshalling for use in an automated planning tool is provided in [38]. We note that our approach introduces domain specific symmetry breaking not present in the method of [38]. However, the domain independent planners used in their approach do have symmetry breaking capabilities. Nonetheless, as we point out later, the computational performance of these approaches is inferior to our custom built method.

Heuristic algorithms have been developed by [18, 13]. A neighborhood search heuristic can be found in [28]. Incorporating these or similar heuristics into decision support systems or discrete event simulation studies is one idea regarding their evaluation; see, e.g., [41, 26]. Another paper on heuristics is [11], 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, [20] extend the specific greedy rules to replace the randomization and obtain considerably improved results. One of the greedy rules incorporated in the approaches of [11] and [20] 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 [57].

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 [36]. 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

The most closely related problems to the pre-marshalling problem are the previously mentioned BRP (see, e.g., [25]), the re-marshalling problem (see, e.g., [7, 21]) and the assignment of inbound containers to stacks [50]. Simulation studies, such as [34], have also been used to investigate terminal stacking operations, although optimization routines for pre-marshalling and other problems are not used.

In the BRP, containers (called blocks) must be extracted from a bay such that the number of shifted mis-overlaid containers is minimized. In doing so, only containers relevant to the current extraction operation may be moved (i.e., pre-marshalling is not allowed) in a commonly solved, “restricted” variant of the problem. The removal of containers makes the BRP significantly easier to solve than pre-marshalling problems of similar size because the problem size is reduced over the course of finding a solution. Secondly, the restriction of which containers are allowed to be shifted also simplifies the problem, although variants exist where this restriction is lifted.

The most relevant works to our approach from the BRP are [54] and [58], in which IDA* is used to solve the BRP. The authors do not introduce any symmetry breaking or branch reduction heuristics, rather, they focus on lower bounds and probing heuristics. Aspects of their IDA* algorithm may sound similar to our approach, but it actually differs significantly, as the BRP allows a simplification of the search through container removals. In pre-marshalling, containers cannot be removed, thus we must define a much wider class of symmetries and branching rules to avoid repeated work. A branch-and-price algorithm is introduced in [53]; however, its reliance on container removals makes it impractical for pre-marshalling. The RLIDA* approach in [1] achieves state-of-the-art performance for the BRP by intelligently switching between different lower bound heuristics with differing levels of CPU time requirements and bound tightness. This approach is not yet relevant for pre-marshalling as there are only two lower bound algorithms, both with very low CPU time requirements. Furthermore, as we will discuss later, one lower bound computation is completely dominated by the other one, i.e., the other will always find at least as good a lower bound.

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., [32]. 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., [52] and [47]), 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., [12]. 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 the blocks world problem; see, e.g., [39], [14]. 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. Another similar problem from artificial intelligence is the Tower of Hanoi problem [16]. The main difference to pre-marshalling is that ordering constraints of the elements of the Tower of Hanoi problem are enforced throughout the problem, whereas in pre-marshalling containers may be temporarily mis-overlaid in order to sort the bay.

4 Solution Approach

In this work we propose two optimal approaches based on the well-known A* and IDA* algorithms, which both perform an exact path-based search guided by a cost estimation heuristic for the selection of the branches to explore first (see, e.g., [40]). A* builds a so-called *search fringe* that stores the shortest path found so far to explored nodes in a graph. The main advantage of using a search procedure such as A* is that, unlike in the multi-commodity flow formulation of pre-marshalling in [29], A* builds a graph of moves as it searches, rather than trying to fit a large graph into memory before solving even starts. A* is nearly equivalent to the very well known Dijkstra's algorithm for finding shortest paths in graphs [10], except that A* uses an improved cost estimation heuristic $f(x) = g(x) + h(x)$ for any node x where $g(x)$ is the number of nodes currently in the path to x , and $h(x)$ is a lower bound on the cost from x to the destination of the search. As long as $h(x)$ never overestimates the real cost of completing a path, A* always finds an optimal solution.

A downside to the A* algorithm is that it must save every node it explores in the graph along with the cost of getting to that node. For large graphs, such as in the case of the pre-marshalling problem, this does not fit in memory. Iterative deepening A* (IDA*) is a memory efficient version of A* in which a so-called *iterated deepening* search is used. This type of search iteratively increases a depth limit on the search. Each time the limit is increased, a depth first search is called and the search must re-generate all of its work from the previous iteration. While this provides a constant sized memory footprint, the computational cost of IDA* is higher than for A*. In practice, the time required for regenerating the search tree is actually rather low. Due to the exponential growth of the tree, most time is spent investigating leaf nodes. We refer the reader to [40] for a more in depth description of the general algorithm.

Note that although A* and IDA* 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.

Since A* and IDA* are graph search algorithms, we must define a graph for the pre-marshalling problem that can be searched. The graph is based on looking at a sequence of container moves as a path and searching through this space. A node in the graph is a configuration of the bay. Each node has an arc to every configuration that can be reached by moving exactly one container from the top of one stack to the top of a different stack.

We apply state-of-the-art A* and IDA* techniques for solving the pre-marshalling problem, and advance them with domain-specific rules. Although there has been significant work on domain independent symmetry breaking and heuristics (see, e.g., [8] or more recently [44]) for A* and automated planning algorithms, these techniques usually require a significant domain specific component. For pre-marshalling, we provide domain specific rules that are fine-tuned to the problem to get fast performance.

4.1 A* for pre-marshalling

Algorithm 1 provides pseudo-code for the A* algorithm, with the following parameters. The initial solution, n , provides the starting configuration of the bay. The functions g and h provide the cost to reach the current solution from the initial state and the estimated cost of completing the solution, respectively, and are described in a following subsection. Finally, the parameter *do-close* indicates whether or not the search should explicitly store already seen nodes in a form of state memoization (see, e.g., [40]). Turning this parameter off means that A* may visit some nodes more than once. There is no risk of entering loops or infinitely long paths since with each move the path length increases.

The priority queue *open* stores the unexplored nodes, called the *search fringe*, whereas the set *closed* is a set of all previously seen nodes. The search continues until a solution is found or the priority queue becomes empty, meaning no solution exists. On line 6 the closed queue is updated with the currently explored node (if the *do-close* parameter is set). When exploring all possible branches on line 7, the already explored nodes are removed from consideration. We refer to the closed list as *memoization* throughout this work.

The function BRANCHES creates a node for every possible move of a container in the current bay. That is, for each container at the top of a stack, we create a new node that moves the container to every other stack, subject to a number of branching rules and symmetry breaking introduced in the next section. As

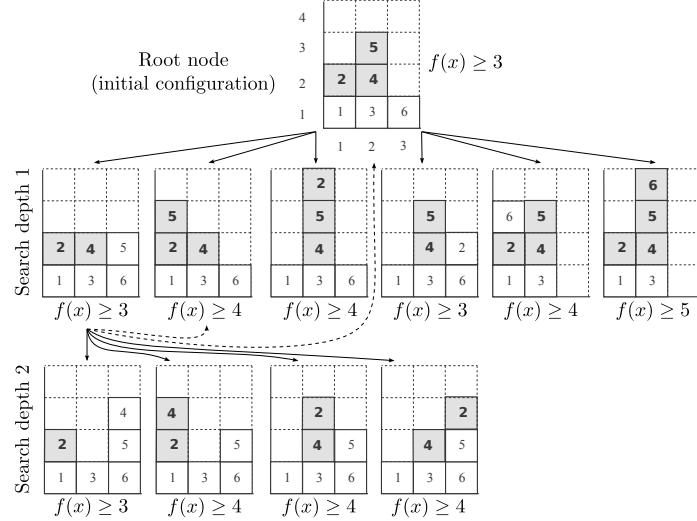


Fig. 3: Example search graph of A*.

Algorithm 1 An A* algorithm for the container pre-marshalling problem.

```

1: function PM-A*( $n, g, h, \text{do-close}$ )
2:    $\text{open} \leftarrow \text{PRIORITY-QUEUE}()$ 
3:    $\text{closed} \leftarrow \emptyset$ 
4:   while MIS-OVERLAYS( $n$ ) > 0 do
5:     if  $\text{do-close} = \text{true}$  then ▷ Memoization step
6:        $\text{closed} \leftarrow \text{closed} \cup \{n\}$ 
7:     for  $m \in \text{BRANCHES}(n) \setminus \text{closed}$  do
8:       PUSH( $\text{open}$ ,  $\langle g(m) + h(m), m \rangle$ )
9:     if  $|\text{open}| = 0$  then return no solution
10:     $n \leftarrow \text{POP}(\text{open})$ 
11:    return  $g(n)$ 

```

this procedure enumerates every possible bay configuration, our approach clearly finds the optimal solution, if there is one.

Since the algorithm uses a priority queue to sort the search fringe by objective function value, when a configuration is found with no mis-overlays, it must consist of a minimal number of moves.

Consider the example search graph shown in Figure 3. The root node of the A* search is the initial configuration of the bay, with outgoing arcs to every configuration that can be reached through a single move of a container. The value of the node, $f(x)$, is computed for each node that is opened. After opening search depth 1, a node must be chosen for the next iteration of A*. Any node with a minimal $f(x)$ value can be chosen for further exploration. We expand the leftmost node for illustrative purposes. Out of the six possible moves from this node, only 4 are actually opened. Two moves would lead to nodes we have already seen (dashed lines) and are pruned through branching rules we will describe later. In the next iteration, another node must be chosen to be opened. Again, any node with the minimal $f(x)$ value can be opened, in this case any node with $f(x) = 3$. When there is a tie between two nodes, as there is in this example, we open the node with the highest $g(x)$ value, as it is more likely to quickly lead to a solution. This is the case in this example, as opening the leftmost node at search depth 2 leads to an optimal solution.

4.2 IDA* for pre-marshalling

IDA* conducts a series of depth-limited depth first searches in order to avoid using as much memory as A*, at the expense of some extra computation time. Algorithm 2 provides the pseudo-code for our IDA* approach. The parameters n , g , and h are the same as in A*. The parameter k refers to the maximum

Algorithm 2 An IDA* algorithm for the container pre-marshalling problem.

```

1: function PM-IDA*( $n, g, h, k, k^{max}$ )
2:   if  $k \geq k^{max}$  then return no solution
3:    $m \leftarrow$  PM-IDA*-RECUR( $n, g, h, k$ )
4:   if  $m \neq \emptyset$  then
5:     return  $m$ 
6:   else
7:     return PM-IDA*( $n, g, h, k + 1, k^{max}$ )

8: function PM-IDA*-RECUR( $n, g, h, k$ )
9:   if MIS-OVERLAYS( $n$ ) = 0 then return  $n$ 
10:  for  $m \in$  BRANCHES( $n$ ) do
11:    if  $g(m) + h(m) \leq k$  then
12:       $r \leftarrow$  PM-IDA*-RECUR( $m, g, h, k$ )
13:      if  $r \neq \emptyset$  then return  $r$ 
14:  return  $\emptyset$ 

```

depth that the depth first search in IDA*-RECUR may search to, and k^{max} is the maximum k value of the algorithm. Too low a value leads to no solution being returned on instances that actually have solutions. We believe that most practical pre-marshalling instances either have a solution or are obviously unsolvable, meaning setting this value to (for example) five times the lower bound of the initial configuration suffices for finding solutions.

The algorithm starts by running a depth first search on line 3 to the given depth. If a solution is found, it is optimal and is returned. Otherwise, k is incremented and the depth first search is executed again. Within the depth first search IDA*-RECUR, all branches of a search node are explored in no particular order². The algorithm uses the same branching function as in the A* algorithm. When the cost of reaching a particular search node plus the lower bound on the number of moves necessary to complete that node's solution is larger than k , the branch is discarded. Otherwise, the branch is explored recursively.

Consider again the example in Figure 3 for the IDA* approach. First, it computes the $f(x)$ value of the root node, which is 3. This provides the initial depth limit of the search, which in this example is sufficient for finding the optimal solution. Note that this is not always the case. IDA* then performs a depth first search, which explores the “left most” node with the best $f(x)$ value over all branches. Thus, the leftmost node of search depth 1 is opened. Then, the leftmost node of search depth 2 with $f(x)$ is opened. An optimal solution is then found. In contrast to A*, IDA* must only open the left most nodes, rather than all of the nodes at each level of the search.

4.3 Lower bound and optimality

In both algorithms, the heuristic cost estimation is given by $f(n) = g(n) + h(n)$, where g is the number of moves used to reach a particular configuration from the initial configuration, and the heuristic h computes a lower bound on the number of moves necessary to find a configuration without mis-overlays. For both A* and IDA*, as long as $f(n)$ does not overestimate the cost of reaching the optimal solution given solution n and the branching at each node is complete, the algorithms are guaranteed to find optimal solutions.

We use one of two cost estimation heuristics for the function h : the “direct” heuristic and the lower bound presented in [2]. The first lower bound heuristic, which we call the *direct* lower bound, counts the number of mis-overlaid containers in the current search node. This is the same lower bound used in the A* approach in [11]. Figure 4 shows a sample bay in which the gray containers are mis-overlaid. The direct lower bound simply counts these gray containers. Thus, the direct lower bound in this example is 5. However, the optimal number of moves to solve this example is actually 8, meaning a tighter lower bound that also takes into account *indirect* moves is desirable.

Considering Figure 4 again, the lower bound can be increased by one since at least one of the highest priority containers (i.e., containers with priority 5) must be put in the bottom tier. In order to do this, one of the containers at the bottom must be removed, even though none of them are mis-overlaid. The second cost estimation heuristic we use is provided in [2] and generalizes this idea into notions of supply and demand,

² We note that this may be a fruitful direction for future work.

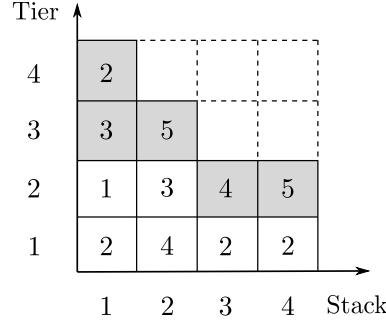


Fig. 4: A bay with mis-overlaid containers.

in which demand refers to high priority mis-overlaid containers needing to be moved (such as containers with priority 4 and 5 in the example), and supply describes areas where such containers can be placed (for example, stacks 3 and 4). We refer to this bound as the extended mis-overlay (EMO) lower bound, and refer to [2] for the algorithmic details. In performing this extra analysis the EMO bound is often able to provide tighter bounds than the direct lower bound at little extra computational cost.

In Figure 4, the EMO bound is 7, only one container short of the 8 moves necessary to solve the problem. This missing container is due to the fact that the EMO bound’s generalized notions of “flow” do not completely capture all the moves that must be performed. In particular, the consequences of performing indirect moves are often not completely captured, such as moving the containers with priority 5 into stacks 3 or 4. The EMO bound correctly sees that either stack 3 or 4 must be cleared, resulting in two extra moves. However, the container of priority 2 will be handled twice because there is nowhere it can be placed in this instance where it is not mis-overlaid. [51] discusses various situations in which the lower bound can be strengthened through such reasoning, but no general algorithm is provided for doing so. Thus, we do not include these ideas in our approach.

The EMO bound has not been previously used in any optimal approaches, and we note that while the bound is valid and will never overestimate the cost of a solution (i.e., it is *admissible*) the bound lacks the property of *consistency*. With each move added to a partial plan, the EMO bound is not guaranteed to monotonically increase. This means that moving a container could potentially cause the lower bound to go up, rather than down. This is a rare edge case of the lower bound, but since A* and IDA* require a completely consistent heuristic, we simply store the heuristic cost of a parent state and compare it against the bound of a child node. Should the bound of the parent be higher than that of the child, we use the parent’s value as the bound.

4.4 Tie breaking

During a search with A*, the search fringe is often full of thousands of nodes with the same f value. Correctly determining which node to open next is important for finding solutions quickly, as in A* open nodes are stored in a priority queue in which the node with the lowest f value is explored at each iteration in a best-first strategy. In contrast, IDA* does not store the search fringe as it uses a depth-limited depth-first search. Thus, no tie breaking is necessary.

We use two tie breaking rules sequentially to try to distinguish between search nodes that look the same. The first is to sort nodes based on g , ordering configurations with higher values of g first. Recall that g counts the number of moves in a partial solution, meaning higher values are solutions where less decisions still need to be made. This is an intuitive tie breaking rule that is a common strategy for A* algorithms with an admissible search heuristic.

Our second tie breaking rule distinguishes between two search nodes by comparing the lowest priority container that is located on the top of a stack. The idea is that nodes with a low priority container on top are potentially more sorted than other nodes, and once the node is expanded the number of mis-overlays in the next move will be low.

4.5 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 [43], 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 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. We further pay close attention to caching effects and avoiding unnecessary memory allocations in order to further increase the number of nodes processed per second.

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 [11]. 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. We use the notation $m = (f, t)$ to indicate the movement of the container on the top of stack f (the “from” stack) to the top of stack t (the “to” stack).

We note that this work is novel even in the context of advances in A* and IDA* search procedures. While work has been done to automatically detect move reversals [17], as well as remove duplicate states from the search using memoization, the branching rules we propose use domain specific insights to achieve pruning that cannot be accomplished through independent heuristics alone.

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 [11]. 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 5 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 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 onto 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. \square

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 $\forall_{i < k < j} m_k = (f_k, t_k), \{f_k, t_k\} \cap \{f_i, t_i, f_j, t_j\} = \emptyset$.

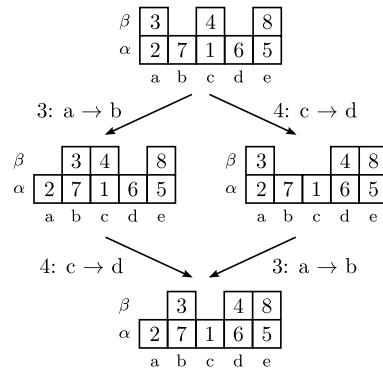


Fig. 5: A symmetry caused by unrelated moves.

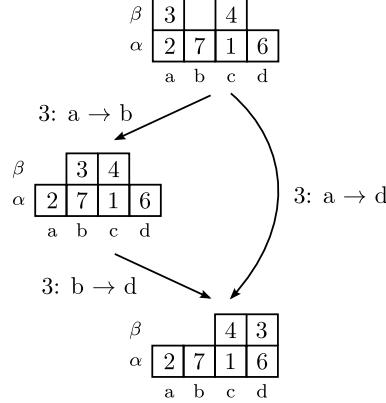


Fig. 6: A dominated state caused by transitive moves.

The rule ensures that no move 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.

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. \square

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 6 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 an optimal solution.

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

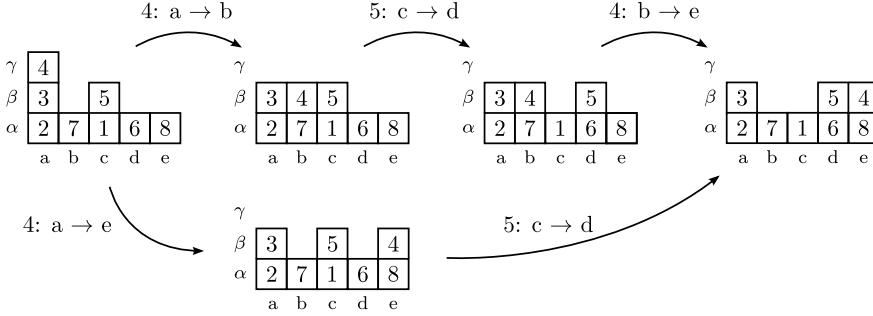


Fig. 7: Transitive moves that are not directly successive.

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 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. \square

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 7 shows an example situation in which a transitive move is separated over several steps. The top sequence of moves contains the transitive moves $4 : a \rightarrow b$ and $4 : b \rightarrow e$, which is dominated by the bottom row's move $4 : a \rightarrow e$. Looking at the third configuration in the top row, directly after the move $5 : c \rightarrow d$, 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.

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. \square

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.

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. Empty stack symmetries are also broken in [56] for the blocks relocation problem.

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 [19]. 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.

The state-of-the-art optimal approach is the A* with the “direct” lower bound presented in [11]. However, we only apply the lower bound from [2] for our tests of the various branching rules and heuristics since it dominates the “direct” lower bound. We further note that we forego a complete comparison with the automated planning approach of [42, 38] based on the results in their paper, which IDA* easily outperforms. In particular, they report needing 6 seconds to even find a first solution for 4 stack instances with 15 containers, whereas our approach solves 4 stack instances with 16 containers to optimality in an average time of 1.22 seconds on hardware that is roughly half as fast.

6.1 Datasets

We use three well-known datasets from the literature to test the effectiveness of our approach, summarized in Table 2. The table provides the number of instances; the minimum and maximum number of stacks, tiers and containers; and the average and standard deviation for the maximum priority, number of empty stacks, and “density” over all instances in each dataset. The density is computed as the number of containers in an instance divided by the total number of slots in the bay. Interestingly, the average density is almost the same for all datasets, even though the CV appear more dense due to their lack of empty stacks and because all stacks are filled to 2 slots below the maximum number of tiers.

The CV dataset, from [6], 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 in [6], 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 compared to other datasets.

The BF dataset [2] contains 721 instances, 40 of which are sourced from the CV dataset, and another 41 of which are variations of the instance used in [28]. We remove the CV instances (giving the numbers in Table 2 indicated by an asterisk), but leave the instance from [28] and its variations, which are prefixed in our results tables with “LC”. We note that while the minimum number of stacks of the BF instances is 10, this is only due to the group of LC instances, which are either 10 or 12 stacks wide. The non-LC BF instances are categorized into 32 different groups, which contain either 16 or 20 stacks, and either 5 or 8 tiers. In contrast to the CV instances, multiple containers are assigned the same priority. This can be seen in

Dataset	Insts.	Stacks		Tiers		Containers		Max. p_{st}		Empty		Density	
		Min	Max	Min	Max	Min	Max	\emptyset	σ	\emptyset	σ	\emptyset	σ
CV	880	3	10	3	10	9	100	32.8	20.7	0.0	0.0	0.7	0.1
BF	681*	10*	20	5	8	35	128	24.1	11.6	1.5	2.0	0.7	0.1
EMM	700	4	10	4	4	8	40	5.4	2.4	0.8	0.2	0.5	0.2

Table 2: Instance characteristics for each of the datasets used in our evaluation.

1. Algorithm	
A*, IDA*	
2. Unrelated move symmetries	
$S^<$	Directly successive, less-than
$S^>$	Directly successive, greater-than
$M^<$	Successive, less-than
$M^>$	Successive, greater-than
3. Transitive move avoidance	
T	Directly successive
U	Successive
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 [2]
D	“Direct” lower bound as described in Section 4

Fig. 8: Overview of the parameterizations of the IDA* and A* algorithms.

the maximum priority column of the table, which shows that BF instances have a lower maximum priority on average.

Finally, we use the instance generator from [11] to generate a dataset of 700 instances.⁴ The dataset is similar to the one used in [11], and has 25 instances each 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 describes the priority distributions for the containers. We call this dataset the EMM dataset. We refer to [11] for structural details regarding the instances.

6.2 Parameterizations

Since both our A* and IDA* algorithms have a number of parameters, we summarize our abbreviations for the parameterizations in Figure 8. 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. The parameterization A*| D is our own implementation of the state-of-the-art technique from [11], which has some implementation differences as discussed in Section 4.5.

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 both algorithmic approaches. Figure 9 shows the number of instances solved on each dataset using the symmetry breaking rule with both the less-than and greater-than variants. Table 3 shows the same information in a table with the percentage of instances that were solved for each dataset using A* and IDA* under the various parameterizations. The symmetry breaking rule is

⁴ We note that we cannot use the exact same instances as in [11] because their instances were unfortunately lost by those authors.

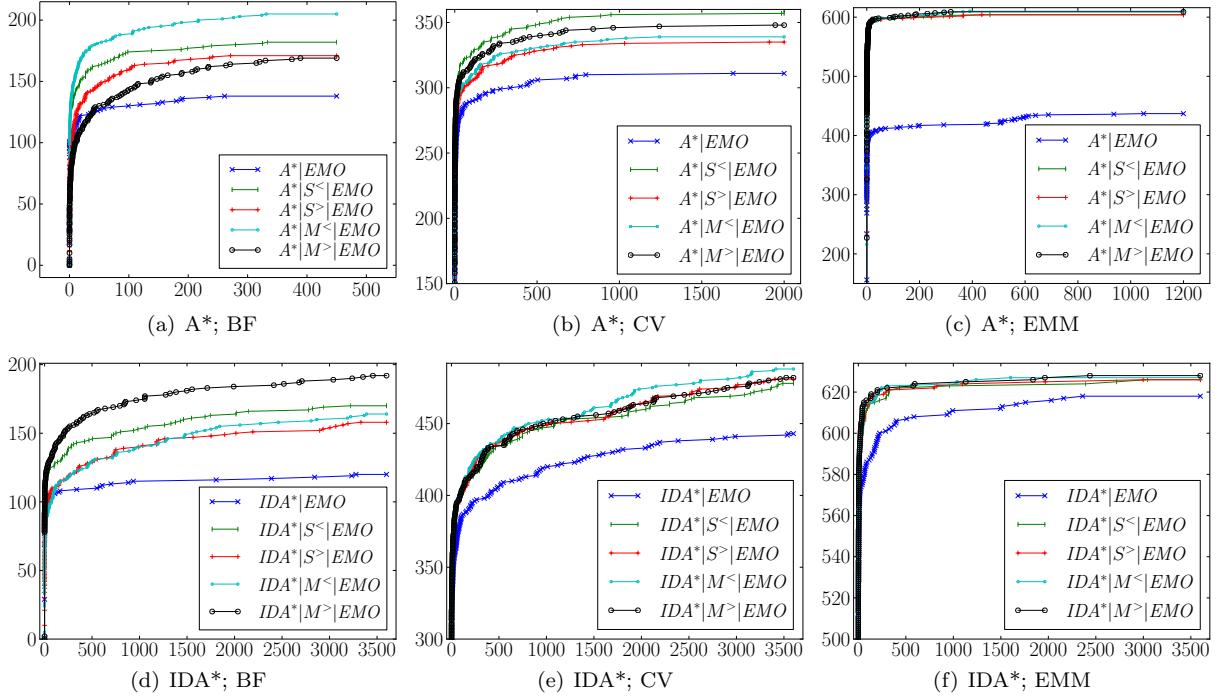


Fig. 9: Number of instances solved versus time in seconds for the unrelated move symmetry breaking rule in the directly successive and successive cases using A* and IDA*.

significantly better than not using the rule on all datasets, even in the directly successive case. Overall, the best performing symmetry breaking rule on each dataset is able to solve 127 more instances than not using the rule with IDA* and 286 more instances for A*.

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 for both algorithms. We suspect this is due to the high number of stacks in the BF dataset, which can lead to many unrelated move symmetries. When not many stacks are present, the extra overhead of looking back multiple moves simply does not pay off.

On the BF dataset for IDA*, 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 9((f)). This means the rule is still beneficial on the EMM 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. This could be an artifact of the random instance generation process, as from a theoretical standpoint the less than or greater than operators are equivalent.

6.4 Evaluation of transitive move avoidance

The effectiveness of our transitive move avoidance rules is shown in Figure 10 and in Table 4. The rules are highly effective in combination with A*, although the successive and directly successive rules have similar

Parameters	BF		CV		EMM	
	A*	IDA*	A*	IDA*	A*	IDA*
EMO	138 (20%)	120 (17%)	311 (35%)	443 (50%)	437 (67%)	618 (94%)
$S^{<} EMO$	182 (27%)	170 (24%)	357 (41%)	478 (54%)	604 (92%)	626 (96%)
$S^{>} EMO$	171 (25%)	158 (22%)	335 (38%)	481 (55%)	604 (92%)	626 (96%)
$M^{<} EMO$	205 (30%)	164 (23%)	339 (39%)	488 (55%)	610 (93%)	627 (96%)
$M^{>} EMO$	169 (25%)	192 (27%)	348 (40%)	482 (55%)	609 (93%)	628 (96%)

Table 3: Effectiveness of unrelated move symmetry breaking

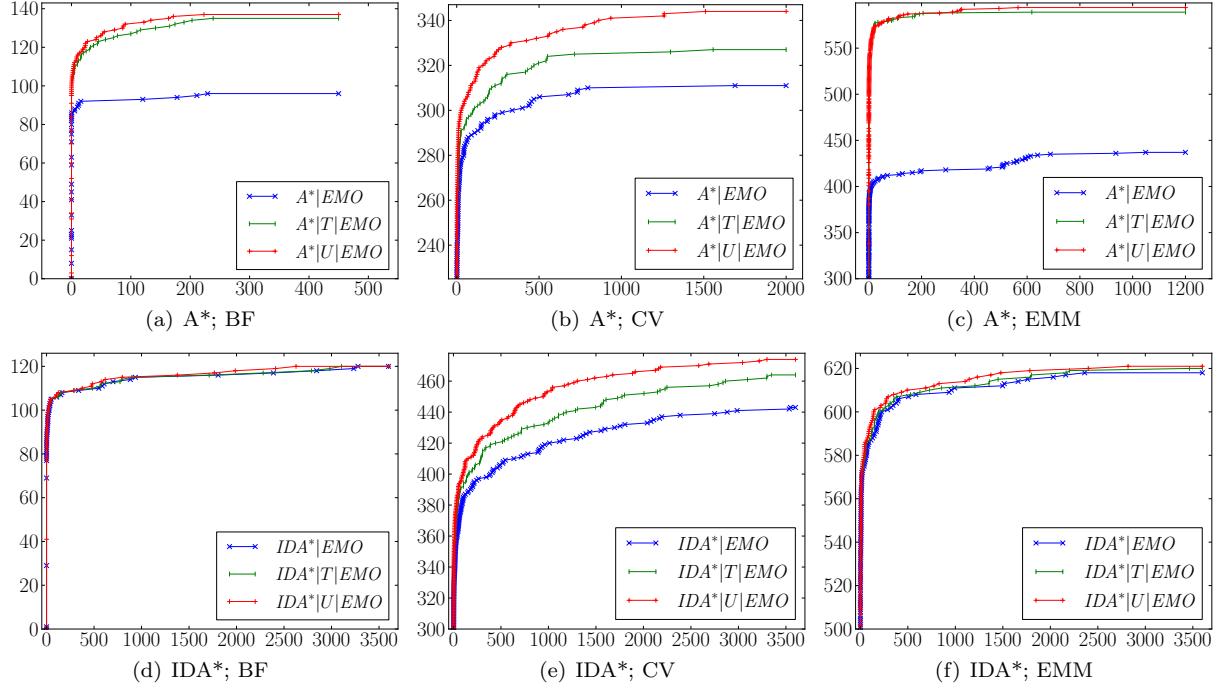


Fig. 10: Number of instances solved versus time in seconds for the transitive move avoidance rule in the directly successive and successive cases using IDA*.

results on the BF and EMM datasets. This contrasts with IDA*, where the transitive move avoidance rules only work well on the CV dataset, although there are some small gains on the EMM dataset.

This rule seems to help A* more than IDA* because it prevents A* from running out of memory as quickly. This allows A* to dig deeper into the tree. The rule shows particular good performance on the CV dataset, which could be due to the uniform stack heights and lack of empty stacks.

The successive rule only outperforms the directly successive on the CV dataset. Otherwise the performance is fairly comparable, although we note some slight gains for A* on BF and for IDA* on EMM. This result is surprising, because we would have thought the multi-level transitive avoidance scheme works best on instances with a high number of stacks, since more similar configurations will be created. One reason for this performance could be that in the CV instances, the lower bound tends to be unable to correctly estimate the number of rehandles necessary for single containers, some of which must be moved two or three times. The transitive move avoidance rule is therefore able to prevent similar configurations from being created through these rehandles that the lower bound cannot avoid.

6.5 Evaluation of empty stack symmetry breaking

We only provide this brief mention of the empty stack symmetry breaking because it is ineffective at improving the performance of either the A* or IDA* approach. The reason is that very few instances have enough empty

Parameters	BF		CV		EMM	
	A*	IDA*	A*	IDA*	A*	IDA*
EMO	138 (20%)	120 (17%)	311 (35%)	443 (50%)	437 (67%)	618 (94%)
$T EMO$	135 (20%)	120 (17%)	327 (37%)	464 (53%)	589 (90%)	620 (95%)
$U EMO$	137 (20%)	120 (17%)	344 (39%)	474 (54%)	594 (91%)	621 (95%)

Table 4: Effectiveness of transitive move avoidance.

stacks to matter. Even on large instances, we rarely see more than 10 empty stack symmetries broken on a single instance. In comparison, our other novel rules are invoked tens to hundreds of thousands (and sometimes millions) of times. We leave the empty stack symmetry breaking rule on in our evaluations, however, because it is inexpensive to use and does not hurt performance.

6.6 Evaluation of A* memoization and tie breaking

We now evaluate the effectiveness of the A*-specific memoization and tie breaking rules. Figure 11 shows the performance of adding these two approaches into a standard A* algorithm for the pre-marshalling problem. Memoization is helpful on the CV and EMM datasets, allowing a total of 177 more instances to be solved between the two datasets. That memoization works better on these two datasets than on BF is not so surprising, since smaller instances take up less memory, meaning A* does not run out of memory as fast as on the large BF instances. We note that memoization does not hurt performance on the BF instances, meaning we can recommend it always be used in combination with A*.

The tie breaking rule is mainly effective on the EMM instances, although small gains can be seen on both BF and CV instances. The tie breaking will only help A* solve more instances when the search fringe is cleared of all nodes that have an f value less than the optimal number of moves. We hypothesize that the effectiveness of the tie breaking rule on the EMM instances is due to their simplicity, as the search often reaches the f value with the optimal number of moves. Without the tie breaking, the size of the search fringe overwhelms the amount of memory available. Given more memory, A* would likely solve nearly the same number of instances as without the tie breaking rule, just not as quickly.

6.7 A* versus IDA*

We now show that both our A* and IDA* approaches outperform the state-of-the-art approach from the literature [11], parameterized as $A^*|D$. We note that we have reimplemented $A^*|D$ within our framework to ensure a fair comparison. We evaluate our approaches on each of the three datasets and observe that we are able to solve significantly more instances to optimality than was previously possible. We only show a selection of all of the possible parameterizations of the algorithms in our tables due to the effectiveness of the branching rules and symmetry breaking for both A* and IDA*, and the potency of the A* memoization and tie breaking shown in the previous subsections.

6.7.1 BF dataset

The performance of IDA* and A* on the BF dataset is shown in Table 5. 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 algorithms are at solving the LC1 instance from [28], which the authors target with heuristic approaches. The instance is easily solvable by every configuration 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 that A* does not need to spend time reconstructing the search tree each time it increases the lower bound of a problem like

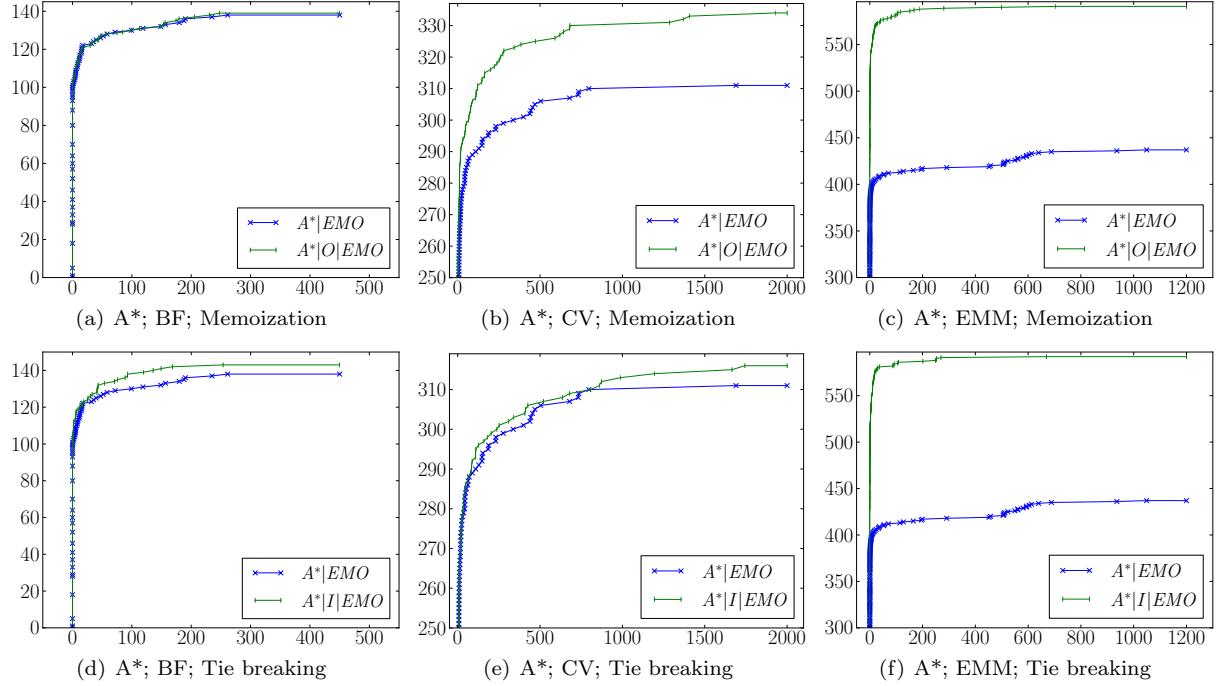


Fig. 11: Number of instances solved versus time in seconds for the transitive move avoidance rule in the directly successive and successive cases using IDA*.

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.

6.7.2 CV dataset

Our comparison of approaches on the CV dataset is shown in Table 6. 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 outperforms IDA*. The CV instances have less stacks 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

Table 5: 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
BF2	99.25	15	311.65	10	280.13	10
BF3	365.11	12	183.19	7	13.21	2
BF4	13.51	18	17.15	14	69.18	5
BF5	264.61	8	423.34	11	-	0
BF6	311.36	1	-	0	-	0
BF7	470.07	12	997.00	15	-	0
BF8	146.93	3	966.73	3	-	0
BF9	902.10	7	684.51	6	-	0
BF10	130.00	3	330.56	2	-	0
BF11	633.76	4	567.99	4	-	0
BF12	2816.65	1	1246.81	1	-	0
BF13	-	0	-	0	-	0
BF14	-	0	-	0	-	0
BF15	-	0	-	0	-	0
BF16	-	0	-	0	-	0
BF17	371.73	7	24.70	8	0.00	3
BF18	7.74	16	55.77	13	212.24	14
BF19	1.05	11	545.90	8	0.00	4
BF20	8.66	14	152.22	12	132.00	10
BF21	701.75	7	376.76	5	-	0
BF22	-	0	2811.51	1	-	0
BF23	22.54	3	1723.35	3	-	0
BF24	-	0	-	0	173.55	2
BF25	-	0	922.24	1	-	0
BF26	-	0	-	0	482.29	1
BF27	268.91	2	1104.14	1	-	0
BF28	220.91	1	1868.77	1	-	0
BF29	-	0	-	0	-	0
BF30	-	0	-	0	-	0
BF31	-	0	-	0	-	0
BF32	-	0	-	0	-	0
LC1	0.00	1	0.00	1	8.16	1
LC2a	72.36	9	22.27	9	-	0
LC2b	388.93	7	184.35	7	-	0
LC3a	218.79	6	694.34	8	-	0
LC3b	296.42	7	404.58	7	-	0
Avg/Count	222.40	173	157.79	120	146.53	51
					27.80	189
					8.38	96
					3.89	44

Table 6: 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
	4	9.03	0.00	40	0.00	40	0.03	40
	5	10.15	0.01	40	0.03	40	0.50	40
	6	11.28	0.02	40	0.07	40	4.60	40
	7	12.80	0.05	40	0.25	40	23.91	40
	8	13.53	0.11	40	0.55	40	33.63	39
	4	15.82	1.22	40	12.46	40	114.69	40
	5	17.85	3.75	40	36.56	40	524.61	39
4	6	19.30	24.26	40	179.41	39	563.10	26
	7	21.82	45.73	40	422.05	38	954.49	16
	4	-	270.61	32	613.45	21	347.66	10
	5	-	530.16	34	1035.71	14	884.22	2
	6	-	1148.75	16	1427.66	3	-	0
	7	-	424.06	15	1153.23	7	-	0
	8	-	1762.02	10	889.66	1	-	0
5	9	-	1592.46	3	-	0	-	0
	10	-	2261.26	3	-	0	-	0
	6	-	-	0	-	0	-	0
	10	-	-	0	-	0	-	0
	10	-	-	0	-	0	-	0
	Avg/Count	162.99	513	148.22	443	168.49	372	78.44
						433	37.25	375
							75.65	280

Table 7: 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	
		1	1.92	0.00	25	0.00	25	0.00	25	
		2	0.84	0.00	25	0.00	25	0.00	25	
	0.75	0	13.00	0.00	25	0.04	25	0.00	25	
		1	5.76	0.00	25	0.00	25	0.00	25	
		2	1.96	0.00	25	0.00	25	0.00	25	
	1	0	19.40	0.15	25	0.51	25	14.76	25	
		1	12.20	0.02	25	0.06	25	0.35	25	
		2	9.36	0.03	25	0.12	25	0.58	25	
7	0.5	0	7.28	0.00	25	0.00	25	0.00	25	
		1	3.56	0.00	25	0.00	25	0.00	25	
		2	3.00	0.00	25	0.00	25	0.00	25	
	0.75	0	19.24	0.05	25	0.25	25	-	-	
		1	10.52	0.03	25	0.82	25	17.09	25	
		2	6.32	0.00	25	0.00	25	0.29	25	
	1	0	28.20	1.22	25	4.63	25	-	-	
		1	18.72	10.50	25	105.49	25	674.23	19	
		2	15.56	8.13	25	132.33	25	534.91	19	
10	0.5	0	9.32	0.00	25	0.02	25	0.91	25	
		1	6.56	0.00	25	0.01	25	0.04	25	
		2	3.56	0.00	25	0.00	25	0.00	25	
	0.75	0	-	0.37	10	0.87	10	-	-	
		1	-	0.17	10	3.09	10	235.32	8	
		2	-	0.01	10	0.03	10	12.84	8	
	1	0	37.76	60.19	25	28.23	24	-	-	
		1	-	116.23	23	382.87	20	1139.05	3	
		2	-	95.96	24	435.61	19	1430.33	3	
Avg/Count			11.15	627	36.82	618	69.11	485	8.26	
							513	3.31	495	
								29.43	437	

nodes are evaluated multiple times, unlike in A^* . Although the overall average computation time required by the A^* appears to be lower, this is only because larger instances are not solved.

6.7.3 EMM dataset

We present our results on the EMM dataset in Table 7, where $|S|$ is the number of stacks, p is the container density and C is the configuration type. All instances have 4 tiers. $IDA^*|M>UE|EMO$ solves 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, probably because of how easy the vast majority of the EMM instances are.

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^* approaches, including several novel branching and symmetry breaking rules. Our approach significantly outperforms the state-of-the-art A^* technique for solving pre-marshalling problems to optimality, solving 568 previously unsolved instances to optimality. Combinations 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. Integrating pre-marshalling solution methods in other storage or seaside problems [4] could also make terminal operations more efficient. It is also possible that our branching rules are applicable to similar problems, such as the blocks relocation problem in the unrestricted case, and could be of assistance for solving this problem to optimality.

References

1. O. Betzalel, A. Felner, and S.E. Shimony. Type system based rational lazy IDA*. In *Eighth Annual Symposium on Combinatorial Search*, 2015.
2. A. Bortfeldt and F. Forster. A tree search procedure for the container pre-marshalling problem. *European Journal of Operational Research*, 217(3):531–540, 2012.
3. H.J. Carlo, I.F.A. Vis, and K.J. Roodbergen. Storage yard operations in container terminals: Literature overview, trends, and research directions. *European Journal of Operational Research*, 235(2):412–430, 2014.
4. H.J. Carlo, I.F.A. Vis, and K.J. Roodbergen. Seaside operations in container terminals: literature overview, trends, and research directions. *Flexible Services and Manufacturing Journal*, 27(2-3):224–262, 2015.
5. M. Caserta, S. Schwarze, and S. Voß. Container rehandling at maritime container terminals. In J.W. Böse, editor, *Handbook of Terminal Planning*, volume 49 of *Operations Research/Computer Science Interfaces Series*, pages 247–269. Springer, New York, 2011.
6. M. Caserta and S. Voß. A corridor method-based algorithm for the pre-marshalling problem. *Lecture Notes in Computer Science*, 5484:788–797, 2009.
7. R. Choe, T. Park, M.S Oh, J. Kang, and K.R. Ryu. Generating a rehandling-free intra-block remarshaling plan. *Journal of Intelligent Manufacturing*, 22(2):201–217, 2011.
8. J.C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
9. R. Dekker, P. Voogd, and E. van Asperen. Advanced methods for container stacking. *OR Spectrum*, 28(4):563–586, 2006.
10. E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
11. C. Expósito-Izquierdo, B. Melián-Batista, and M. Moreno-Vega. Pre-marshalling problem: Heuristic solution method and instances generator. *Expert Systems with Applications*, 39(9):8337–8349, 2012.
12. S. Felsner and M. Pergel. The complexity of sorting with networks of stacks and queues. *Lecture Notes in Computer Science*, 5193:417–429, 2008.
13. M.S. Gheith, A.B. El-Tawil, and N.A. Harraz. A proposed heuristic for solving the container pre-marshalling problem. In E. Qi, J. Shen, and R. Dou, editors, *The 19th International Conference on Industrial Engineering and Engineering Management*, pages 955–964. Springer, Berlin, 2013.
14. N. Gupta and D.S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2–3):223–254, 1992.
15. P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.
16. A.M. Hinz. The tower of Hanoi. *Enseign. Math.*, 35(2):289–321, 1989.
17. R.C. Holte and N. Burch. Automatic move pruning for single-agent search. *AI Communications*, 27(4):363–383, 2014.
18. S.-H. Huang and T.-H. Lin. Heuristic algorithms for container pre-marshalling problems. *Computers & Industrial Engineering*, 62(1):13–20, 2012.
19. ISO/IEC. Information technology – Programming languages – C++, Third Edition. ISO/IEC 14882:2011, International Organization for Standardization / International Electrotechnical Commission, Geneva, Switzerland, 2011.
20. R. Jovanovic, M. Tuba, and S. Voß. A multi-heuristic approach for solving the pre-marshalling problem, 2016. Accepted in the Central European Journal of Operations Research.
21. J. Kang, M.S. Oh, E.Y. Ahn, K.R. Ryu, and K.H. Kim. Planning for intra-block remarshalling in a container terminal. *Lecture Notes in Artificial Intelligence*, 4031:1211–1220, 2006.
22. J. Kang, K.R. Ryu, and K.H. Kim. Deriving stacking strategies for export containers with uncertain weight information. *Journal of Intelligent Manufacturing*, 17(4):399–410, 2006.
23. K.H. Kim. Evaluation of the number of rehandles in container yards. *Computers & Industrial Engineering*, 32:701–711, 1997.
24. K.H. Kim and J.W. Bae. Re-marshalling export containers in port container terminals. *Computers & Industrial Engineering*, 35(3–4):655–658, 1998.

25. K.H. Kim and G.-P. Hong. A heuristic rule for relocating blocks. *Computers & Operations Research*, 33(4):940–954, 2006.
26. J. Klaws, R. Stahlbock, and S. Voß. 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, 2011.
27. R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
28. Y. Lee and S.L. Chao. A neighborhood search heuristic for pre-marshalling export containers. *European Journal of Operational Research*, 196(2):468–475, 2009.
29. Y. Lee and N.Y. Hsu. An optimization model for the container pre-marshalling problem. *Computers & Operations Research*, 34(11):3295–3313, 2007.
30. Y. Lee and Y.-J. Lee. A heuristic for retrieving containers from a yard. *Computers & Operations Research*, 37(6):1139–1147, 2010.
31. J. Lehnfeld and S. Knust. Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. *European Journal of Operational Research*, 239(2):297 – 312, 2014.
32. T. Nishi and M. Konishi. An optimisation model and its effective beam search heuristics for floor-storage warehousing systems. *International Journal of Production Research*, 48:1947–1966, 2010.
33. K. Park, T. Park, and K.R. Ryu. Planning for remarshalling in an automated container terminal using cooperative coevolutionary algorithms. In *SAC 2009 – Honolulu, Hawaii*, pages 1098–1105, 2009.
34. M.E.H. Petering. Real-time container storage location assignment at an RTG-based seaport container transshipment terminal: problem description, control system, simulation model, and penalty scheme experimentation. *Flexible Services and Manufacturing Journal*, 27(2-3):351–381, 2015.
35. M. Prandtstetter. A dynamic programming based branch-and-bound algorithm for the container pre-marshalling problem. Technical report, Austrian Institute of Technology, 2013.
36. A. Rendl and M. Prandtstetter. Constraint models for the container pre-marshaling problem. In G. Kat-sirelos and C.-G. Quimper, editors, *ModRef 2013: 12th International Workshop on Constraint Modelling and Reformulation*, pages 44–56, 2013.
37. J.P. Rodrigue, C. Comtois, and B. Slack. *The Geography of Transport Systems*. Routledge, Milton Park, 2 edition, 2009.
38. M. Rodríguez-Molins, M. Á. Salido, and F. Barber. Domain-dependent planning heuristics for locating containers in maritime terminals. In N. García-Pedrajas, F. Herrera, C. Fyfe, J.M. Benítez, and M. Ali, editors, *Trends in Applied Intelligent Systems*, volume 6096 of *Lecture Notes in Computer Science*, pages 742–751. Springer Berlin Heidelberg, 2010.
39. A.G. Romero and R. Alquézar. To block or not to block? *Lecture Notes in Computer Science*, 3315:134–143, 2004.
40. S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.
41. M.A. Salido, M. Rodriguez-Molins, and F. Barber. A decision support system for managing combinatorial problems in container terminals. *Knowledge-Based Systems*, 29:63–74, 2012.
42. M.A. Salido, O. Sapena, M. Rodriguez, and F. Barber. A planning tool for minimizing reshuffles in container terminals. In *21st IEEE International Conference on Tools with Artificial Intelligence*, pages 567–571. IEEE, 2009.
43. C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29 – 83. Elsevier, 2006.
44. A. Shleyfman, M. Katz, M. Helmert, S. Sievers, and M. Wehrle. Heuristics and symmetries in classical planning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
45. R. Stahlbock and S. Voß. Operations research at container terminals: A literature update. *OR Spectrum*, 30(1):1–52, 2008.
46. D. Steenken, S. Voß, and R. Stahlbock. Container terminal operations and operations research – A classification and literature review. *OR Spectrum*, 26(1):3–49, 2004.
47. L. Tang, R. Zhao, and J. Liu. Models and algorithms for shuffling problems in steel plants. *Naval Research Logistics (NRL)*, 59(7):502–524, 2012.
48. K. Tierney, S. Voß, and R. Stahlbock. A mathematical model of inter-terminal transportation. *European Journal of Operational Research*, 235(2):448 – 460, 2014.

49. UNCTAD. *United Nations Conference on Trade and Development (UNCTAD), Review of maritime transport*. 2012.
50. E. van Asperen, B. Borgman, and R. Dekker. Evaluating impact of truck announcements on container stacking efficiency. *Flexible Services and Manufacturing Journal*, 25(4):543–556, 2013.
51. S. Voß. Extended mis-overlay calculation for pre-marshalling containers. In H. Hu, X. Shi, R. Stahlbock, and S. Voß, editors, *Computational Logistics*, volume 7555 of *Lecture Notes in Computer Science*, pages 86–91. Springer, Berlin, 2012.
52. G. Zäpfel and M. Wasner. Warehouse sequencing in the steel supply chain as a generalized job shop model. *International Journal of Production Economics*, 104(2):482–501, 2006.
53. E. Zehendner and D. Feillet. A branch and price approach for the container relocation problem. *International Journal of Production Research*, 52(24):7159–7176, 2014.
54. H. Zhang, S. Guo, W. Zhu, A. Lim, and B. Cheang. An investigation of IDA* algorithms for the container relocation problem. In N. Garca-Pedrajas, F. Herrera, C. Fyfe, J.M. Bentez, and M. Ali, editors, *Trends in Applied Intelligent Systems*, volume 6096 of *Lecture Notes in Computer Science*, pages 31–40. Springer Berlin Heidelberg, 2010.
55. R. Zhang, Z. Jiang, and W. Yun. Stack pre-marshalling problem: A heuristic-guided branch-and-bound algorithm. *International Journal of Industrial Engineering: Theory, Applications and Practice*, 22(5):509–523, 2015.
56. R. Zhang, S. Liu, and H. Kopfer. Tree search procedures for the blocks relocation problem with batch moves. *Flexible Services and Manufacturing Journal*, pages 1–28, 2015. Online available.
57. Y. Zhang, W. Mi, D. Chang, and W. Yan. An optimization model for intra-bay relocation of outbound containers on container yards. In *Proceedings of the IEEE International Conference on Automation and Logistics*, pages 776–781, 2007.
58. W. Zhu, H. Qin, A. Lim, and H. Zhang. Iterative deepening A* algorithms for the container relocation problem. *IEEE Transactions on Automation Science and Engineering*, 9(4):710–722, 2012.