A Dynamic Programming Based Branch-And-Bound Algorithm for the Container Pre-Marshalling Problem☆

Matthias Prandtstetter

AIT Austrian Institute of Technology, Mobility Department – Dynamic Transportation Systems
Giefinggasse 2, 1210 Vienna, Austria
matthias.prandtstetter@ait.ac.at
phone: +43-50550-6692; fax: +43-50550-6439

Abstract

Nowadays, container terminals play a major role in global transport networks since they act as primary transhipment nodes. To reduce idle strokes—especially during vessel stowage—it is necessary to re-arrange so-called container bays in the daytime when little workload occurs. Obviously, re-arrangement plans with a minimal number of container movements are sought. We tackle the resulting *container pre-marshalling problem* (PMP) by proposing a novel *dynamic programming* (DP) approach, which is then embedded in a *branch-and-bound* (B&B) framework. In addition, a heuristic version of the resulting DP-based B&B method is introduced. Comprehensive computational experiments demonstrate the strengths of the proposed methods and emphasize the applicability of them to (small and medium-sized) real-world instances.

Keywords: Combinatorial Optimization, Dynamic Programming, Branch-And-Bound, Logistics, Pre-Marshalling Problem

1. Introduction

In the last century, containers have emerged as the standard transportation unit for various types of goods, e.g. furniture, cartons of cigarettes, etc. Most of the employed containers comply with ISO 668, which mainly defines two types of containers: 20ft and 40ft long containers. Originally introduced to facilitate the stowing process of vessels, there are nowadays transport vehicles for road and railway as well. So-called container terminals providing transshipment equipment are used as intermediate and interchange points along an intermodal route. Additionally, it is often necessary that containers are temporarily stored in the terminals since perfect synchronization between different modes along an intermodal cargo route is in most cases not possible. Due to the standardization of the containers and to reduce the occupied space in the terminal, containers are usually stacked on each other. Multiple stacks, compiled next to each other, are referred to as bay. Multiple bays form the complete container stockyard, cf. Fig. 1.

Obviously, this arrangement of containers in terminals leads to unavoidable reorganizations since the sequence of in- and outbound containers are mostly not known a priori or at least only short before (direct) access to the containers will be necessary. Two different types of equipment can be used for this required reorganization processes: reach stackers, see Fig. 2a,

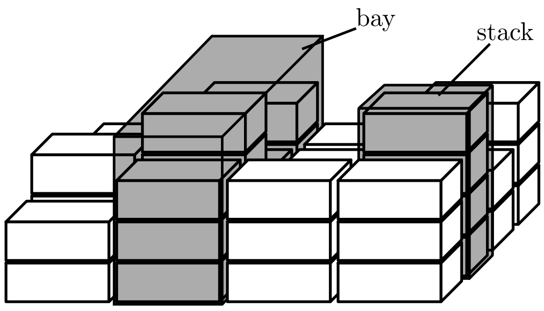

Figure 1: A schematic illustration of a container terminal. Containers are stacked on each other. Multiple container stacks form a so-called container bay.

which are independent wheeled vehicles able to access only the topmost container of the outermost stacks in a bay, and gantry cranes, see Fig. 2b, which have access to the topmost containers of stacks only. For a more detailed description of container terminals, tools and vehicles employed as well as related processes we refer to [1, 2, 3].

To be able to face competition, it is important for seaport container terminal operators to keep the loading times of ships as short as possible. Therefore, it is convenient that during unproductive times in terms of loading operations, reorganization operations are performed such that the containers can be directly loaded into the vessel according to the desired orders imposed by the shipping company owners. However, to keep the reorganization workload as low as possible, it is demanded to find a work plan which consists of a minimized number of container movements.

☆This work is partially funded by the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) within the strategic programme I2VSplus under grant 831736 (TRIUMPH).

Email address: `matthias.prandtstetter@ait.ac.at` (Matthias Prandtstetter)

(a) (b)

Figure 2: A gantry crane (b) and a reach stacker (a).

The main goal of the container pre-marshalling problem [4] (PMP) as examined within this paper is the forward-looking re-ordering of containers in a bay by using the gantry crane such that during vessel stowage no additional relocations are necessary. Although the restriction to one bay only as well as the sole employment of the gantry crane seems to be counter-intuitive at a first glance, it turns out that inter-bay movements for gantry cranes are rather time-consuming while intra-bay movements are assumed to be fast (and constant). The applicability for reach stackers, on the other hand, is limited to outermost stacks only. Therefore, it is not possible to efficiently re-shuffle containers in the interior of a bay via reach stackers only. Hence, terminal operators normally decide that re-shuffles between bays are not desired. Nevertheless, the number of relocations to reach a final bay layout should be kept as low as possible.

Within this paper, we focus on the pre-marshalling problem as it is classically defined, i.e. intra-bay operations performed by a gantry crane are solely looked at. Although a lot of papers have already been published focusing on the pre-marshalling problem, our paper is the first one presenting a dynamic programming approach embedded in a branch-and-bound algorithm such that optimal work plans for small and medium-sized real-world instances can be provided within short computation times. In addition, a short summary on the complexity of the problem is given.

The rest of the paper is organized as follows: First we give a (more) formal description of the problem (Sec. 2) and outline related work (Sec. 3). Afterward, the computational complexity is examined (Sec. 4) followed by a novel dynamic programming approach (Sec. 5) which is then extended for use within a branch-and-bound algorithm (Sec. 6) as well as a heuristic method (Sec. 7). Finally, computational results are provided (Sec. 8) and conclusions are drawn (Sec. 9).

2. Formal Description

We are given a set of containers with each container c having a priority $p_c \in \mathbb{N}$ assigned which indicates the aimed order during vessel stowage. The smaller p_c, for $c \in C$, the higher is the priority of the container, i.e. the earlier the container will be stowed into a vessel. Please note, that two containers may have the same priority, meaning that no actual ordering between these containers is necessary. This might, for example, apply for containers which are empty and therefore are interchangeable. Since only the priorities of the containers are of relevance, we define without loss of generality $C = \{1, \dots, C\}$ which is an ordered multiset of all container priorities. Therefore, we will write c instead of p_c in the further context. We denote by $m(c)$ the multiplicity of container priority c, i.e. the number of containers with priority c. The cardinality $|C|$ corresponds to the number of all considered containers.

In addition, we are given a container bay of w (container) stacks. We denote by a *stack state* s an h-tuple $s = (c_1, \dots, c_h) \in C^h$ where h is the maximal possible number of containers stacked on each other. With $m_s(c)$ we denote the number of containers of priority c in stack s. Further, we write $c_i = 0$ if no container is stored at location i, with $1 \leq i \leq h$.

A stack state $s = (c_1, \dots, c_h)$ is said to be *valid*, iff

$$m_s(c) \leq m(c), \qquad \text{for all } c \in C, \tag{1}$$

$$c_i = 0 \Rightarrow c_{i+1} = 0, \qquad \text{for } 1 \leq i < h \tag{2}$$

That is, we require that no priority occurs more often than allowed (1) and that each container may only be placed on another existing container (or the ground), cf. (2).

Analogously, we define a *bay state* $\mathcal{B}$ as the set of current stack states, for each stack in the bay. Therefore, we call a bay state $\mathcal{B} = \{s_1, \dots, s_w\}$ *valid*, iff

$$s \text{ is valid}, \qquad \text{for all } s \in \mathcal{B} \tag{3}$$

$$\sum_{s \in \mathcal{B}} m_s(c) = m(c), \qquad \text{for all } c \in C \tag{4}$$

So, a bay state is valid if (and only if) it contains only valid stack states (3), no container is stored twice and all containers are stored in the bay (4). By c_i^j we denote the i-th container in stack state $s_j = (c_1^j, \dots, c_h^j)$.

Let us further define a *perfect stack state* as a stack state which is valid and fulfills

$$c_i \geq c_{i+1}, \qquad \text{for } 1 \leq i < h \tag{5}$$

That is, containers with higher priority are not blocked by containers with lower priority. However, containers of same priority may be stacked on each other. Analogously, a *perfect bay state* is a bay state for which all stack states are perfect.

For the pre-marshalling problem (PMP), we are given the following situation: At the beginning, a bay state is given which is not perfect. Therefore, container relocations are performed (during idle times) such that a perfect bay state is obtained. However, we are not interested in finding one possible sequence of container movements leading to a perfect bay state but in finding a *minimal* sequence of container relocations.

Therefore, we define a container relocation $r = (i, j)$ as the movement of the topmost container on stack s_i to the top of stack s_j. A move is said to be *valid* iff $c_1^i \neq 0$ and $c_h^j = 0$, i.e., at least one container is stored at stack i and the number of containers stored in stack j is less than the maximum height.

A solution $\sigma = (r_1, \dots, r_m)$ to our problem is an m-tuple of valid moves such that after applying the m moves to the initial bay layout a perfect bay layout is reached. An optimal solution $\sigma^* = (r_1, \ldots, r_{m^*})$ to our problem is a solution such that $m^* \leq m$ holds for all possible solutions σ.

3. Related Work

Several (real-world) variants of the problem have been discussed in the literature. Common to all variants is that only gantry cranes are employed for the relocation operations of containers. However, the related work can be mainly divided in two areas: First, the *container (or blocks) relocation problem* (CRP), where the containers are relocated during the stowing process. And second, the *container pre-marshalling problem* (PMP), where all relocation operations are performed *before* the stowing process starts.

Caserta, Voß and Sniedovich [5] provide a dynamic programming (DP) model for solving the CRP. However, their goal is not to solve the CRP to optimality by means of DP but to incorporate the DP model into a corridor method which is done via the definition of constraints on the moves to be further examined by the DP approach. Computational experiments show that this approach outperforms previously published approaches in terms of necessary container relocations while the computation times are rather identical.

In [6], Forster and Bortfeldt introduce a method for computing slightly improved lower bounds on the number of container relocations which is then incorporated into a tree search procedure for the CRP. Comparisons with other existing approaches are performed showing that the average number of container relocations can be reduced while the computation times are kept short.

In their work, Lee and Hsu [7] formulate the PMP as multi-commodity flow problem with additional side-constraints. An integer linear programming (ILP) formulation is presented together with heuristic solution methods to tackle this hard ILP model. However, computational results show that this approach is not applicable for real-world instances. Although it could be shown that the definition of target layouts improves the solution process, it has to be emphasized that for real-world applications a multitude of valid end states exist. To decide which of those can be reached with the overall minimum number of container relocations is an optimization problem for itself.

In [8], Caserta and Voß present a corridor method-based algorithm for the PMP. Analogously to [5] the search space is pruned by defining constraints on the options to be further examined.

A neighborhood search based method for the PMP is presented by Lee and Chao in [4]. The method proposed can be divided into two sub-routines: The first one focuses on the reduction of blocking and misplaced containers using neighborhood search. The second sub-routing tries to reduce the number of container relocations incorporating a binary integer program to reach the targeted layouts computed during the first sub-routine. In contrast to all other approaches presented, the authors highlight that in some cases blocking containers might be acceptable, if reaching a perfect layout incorporates much more container relocations.

In [9], the authors present a greedy heuristic for the PMP which follows the idea to (optimally) place containers with low priorities first such that during later relocations they do not block containers with higher priorities. In addition, an A*-search is outlined. Computational results are presented for problem instances found in literature as well as instances generated with the instance generator presented within that paper. Although the results are promising, no detailed comparisons with other approaches are carried out.

Huang and Lin [10] present heuristic methods for the PMP as examined within this paper as well as an interesting variant where the containers need to be sorted such that they are finally located in predefined cells or regions of the container bay/terminal. The heuristics proposed (for both variants) are following the simple idea that perfect stacks of containers are created where all available slots are occupied by a container such that ample space is available for re-arranging not perfect stacks. Computational experiments are only performed on a very small set of selected instances such that the potential of this approach cannot be fully determined.

Finally, Bortfeldt and Forster [11] define for the PMP *bad-bad*, *bad-good*, *good-bad* and *good-good* moves, which indicate whether a blocking or non-blocking container, i.e. a container with lower priority is stacked on a container with higher priority or vice versa, is relocated such that the container becomes blocking or non-blocking. Building on this concept a methodology for computing (tight) lower bounds as well as a tree search procedure is developed. Although computational results provided in [11] highlight the dominance of this method compared to other leading methodologies the results presented have to be handled with care since detailed examinations revealed that the amount of available slots for re-shuffling differ between the methods presented in that paper and other so far published approaches. Therefore, it was also not possible to compare our approaches with those of Bortfeldt and Forster.

4. Statement on the Complexity

Although the container pre-marshalling problem as well as the blocks relocation problem have been extensively studied both from the practical and theoretical point of view no concrete statements on the complexity of the related problem variants can be found in the literature. Gupta and Nau [12] showed in their paper that the well-known *blocks-world planning problem* is $\mathcal{NP}$-hard (but no worse) in the general case (where—in terms of the PMP—an unlimited number of stacks is available) as well as in a version where the number of stacks is limited—but unlimited in height (referred to as LBW in the following). Now, let us assume the PMP would be polynomial. Then LBW could be solved in polynomial time as well by simply solving a PMP instance with the stack height being limited to n (with n being the number of containers/blocks). Thus, the PMP is at least $\mathcal{NP}$-hard. However, the PMP is not harder since a solution (which is a sequence of moves) can be checked on validity in polynomial time by first subsequently applying the moves and finally checking the perfectness of the reached bay layout.

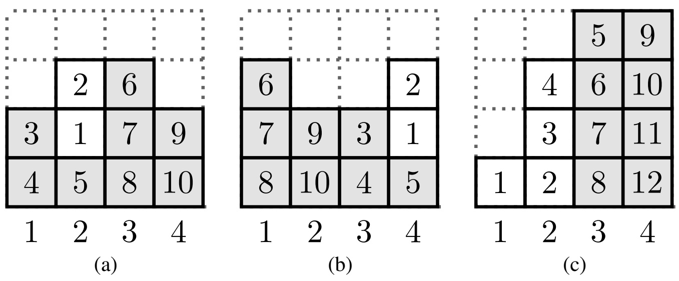

Figure 3: Two equivalent states are shown in (a) and (b). In (c), a bay layout is presented where at least one container in $\overline{C}$ (gray containers) has to be relocated for reaching a perfect layout.

However, keep in mind that in most container terminals a real-world variant of the PMP arises which requires containers to be stowed into vessels during marshalling (cf. *container relocation problem*). Simultaneously, additional containers may arrive which have to be stored in the bay. The resulting variant is equivalent to the (theoretical) problem described in [13] where an unsorted sequence of integers is sorted using complete networks of stacks. In [13], it is, however, shown that this problem is $\mathcal{NP}$-hard as well.

5. Dynamic Program

Based on the *principle of optimality* ("An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.") stated by Bellman [14] *dynamic programming* (DP) divides a given problem instance into interdependent subproblems whose solutions are merged such that a solution to the original problem statement can be achieved. For this purpose, it is convenient to define DP states such that each DP state is either trivial to solve or it has one or more derived DP states whose solutions are a decision basis for finding an optimal solution to the current DP state. In the resulting DP state tree it is often possible to identify sub-problems which are equivalent to each other (e.g. same (number) of decision variables included) but evaluate differently due to prior made decisions (e.g. actual values of those decision variables). In that case, only those subproblems are further regarded whose evaluation value is optimal. By this operation the DP state tree is pruned and therefore the search is accelerated. For our given problem DP states can be intuitively defined based on bay states, i.e., each DP state corresponds to a (unique) bay state.

5.1. Equivalence of DP States

As already outlined above, equivalent DP states need only be examined once, leading to the fact that whenever a state is reached for a second time, further search can be terminated in that branch of the DP tree. Therefore, it is important to define necessary and sufficient conditions for determining whether or not two DP states are equivalent.

For this purpose, let us define multiset $\overline{C}$, which contains all containers of the bay which are judged to be most likely not to be moved again by using a simple heuristic. I.e., it is assumed that the containers in $\overline{C}$ are already placed at their final location. Having a closer look at Figure 3c, it turns out that in some cases some containers in $\overline{C}$ (gray containers) have to be moved as well for reaching a perfect bay state. Nevertheless, $\overline{C}$ can be used as hash function for determining whether or not two DP states are equivalent, cf. Eq. (7).

$$c_i^j \in \overline{C} \text{ if } \begin{cases} m_{\overline{C}}(c_i^j + 1) = m(c_i^j + 1), \\ i = 1 \vee c_{i-1}^j \in \overline{C} \end{cases} \quad \text{for } \begin{cases} 2 \le i \le h, \\ 1 \le j \le w \end{cases} \tag{6}$$

That is, a container is element of $\overline{C}$ if all containers of lower priority are contained in $\overline{C}$ and the container itself is either placed directly on the ground or on a container in $\overline{C}$ of priority at least as low as its own priority. Now let us specify the datastructures used for representing a DP state:

- A stack is represented as an array of length h, where index 0 corresponds to the lowest container location in the stack.

- For each stack two integers are given. The first one ($\overline{s}$) indicating the number of containers placed in this stack being member of set $\overline{C}$ and the second one ($\widetilde{s}$) indicating the number of empty spaces.

- The bay is represented by an array of stacks, i.e. an array of length w. Since the ordering of stacks is irrelevant for determining whether two DP states are equivalent, the stacks in the array are ordered such that for two stacks s_i, s_j with s_i stored at a lower index than s_j, $\overline{s_i} \ge \overline{s_j}$ holds. If $\overline{s_i} = \overline{s_j}$, $\widetilde{s_i} \le \widetilde{s_j}$ has to be true. If furthermore $\widetilde{s_i} = \widetilde{s_j}$, then $c_1^i < c_1^j$, i.e., the priority of the lowest container in the left stack is higher than the priority of the lowest container in the right stack. However, if $c_1^i = c_1^j$, we iteratively continue with c_k^i and c_k^j, $2 \le k \le h - \widetilde{s_i}$ until a decision can be made. If, however, no decision can be made an arbitrary order can be chosen.

- Due to this ordering of stacks, a mapping is necessary which maps the actual stack index to the original stack index. Using this mapping, at each point in time the physical bay layout can be reproduced which is finally necessary for presenting the obtained solution to warehousemen.

- Furthermore, it is necessary, to keep track of how many moves have already been applied for reaching the current state. Therefore, the number of container relocations is stored in a counter variable.

- Finally, to be able to reproduce the complete sequence of moves, for each DP state its parent DP state and the move necessary to reach the current DP state from the parent DP state is stored.

Two DP states $\mathcal{B}$ and $\mathcal{B}'$ are then said to be equivalent if

$$\overline{s_i} = \overline{s'_i}, \qquad \text{for } 1 \le i \le w \tag{7}$$

$$\widetilde{s_i} = \widetilde{s'_i}, \qquad \text{for } 1 \le i \le w \tag{8}$$

$$c^i_k = c'^i_k, \qquad \text{for } 1 \le i \le w, 1 \le k \le h - \widetilde{s_i} \tag{9}$$

In other words, this means that two DP states are equivalent when for each stack (ordered according to the procedure described above) the number of containers in $\overline{C}$, the empty spaces and all containers including their actual position are equal, for an example see Fig. 3a and 3b. Among two (or more) equivalent DP states, the one that can be reached via fewer moves is further investigated. If two (or more) DP states need the same number of moves, either of them can be kept while all others are disregarded.

Now let us examine in more detail why two DP states $\mathcal{B}$ and $\mathcal{B}'$ fulfilling requirements (7)–(9) are actually equivalent. While Equations (8) and (9) are obvious, Equation (7) needs some clarification. Although this equation is included in Equation (9), it can be used for determining non-equivalence of two states more efficiently, since if $\overline{s_i} \neq \overline{s'_i}$ Equation (9) cannot hold either.

5.2. Generation of DP States

The DP states building the next level in the DP tree are derived from the current DP states in a straightforward way. In fact, all possible successors are created by applying all possible, i.e. valid, moves to the current state. Please note, that even those moves relocating a container contained in $\overline{C}$ are examined since one can find examples where a relocations of those containers is necessary as well, cf. Fig. 3c.

5.3. Strategy for Finding Equivalent DP States

Let us assume the following example: Given a bay state $\mathcal{B}$ and a move sequence $\Sigma = ((1,2),(2,1))$. Obviously, bay state $\mathcal{B}' \leftarrow^{\Sigma} \mathcal{B}$ derived by applying Σ is equivalent to $\mathcal{B}$ (since one container is moved from stack 1 to stack 2 and immediately back again). However, the number of moves is different. It is now the question, how it is possible to efficiently determine whether another equivalent state was already reached. Since the number of moves is the main objective of the PMP, it is necessary that at no time a state $\mathcal{B}'$ equivalent to $\mathcal{B}$ is reached after $\mathcal{B}$ where the number of moves for $\mathcal{B}'$ is lower than the number of moves for $\mathcal{B}$. For this purpose, we create DP states by increasing number of moves. That is, a list of DP states is necessary which is ordered by increasing number of moves. New states are then generated always by removing the first DP state from the list and inserting the newly created states according to their corresponding number of moves.

Since this approach is straightforward, it suffers from the fact that many states are created which do not contribute to the further search. Furthermore, all of these states have to be stored such that during the later search it can be determined that a state has already been examined leading to inefficient memory and computation time performance.

Algorithm 1: DPBnB

```
Data: priority queue $Q$ (empty in the beginning; ordered
      by the minimum number of moves necessary for
      reaching a perfect layout)
$s^*$ so far best solution
$UB$ upper bound
$moves(s)$ computes the number of moves necessary to
reach $s$ plus the lower bound on moves for reaching a
perfect state

begin
    add the initial state to $Q$;
    $s^* \leftarrow$ by a (greedy) heuristic;
    $UB \leftarrow$ number of moves to reach $s^*$;
    while $Q$ is not empty do
        $s \leftarrow$ pop($Q$);
        if $moves(s) < UB$ then
            foreach state $s'$ based on $s$ do
                if $s'$ is perfect and the number of moves to
                reach $s' < UB$ then
                    $s^* \leftarrow s'$;
                    $UB \leftarrow$ number of moves to reach $s^*$;
                else if $moves(s') < UB$ then
                    push($Q, s'$);    // if equivalent
                    state is existing, keep only
                    that state which is reachable
                    with fewer moves
    return $s^*$;
```

6. DP-based Branch-And-Bound

While dynamic programming is a method primarily based on the idea to divide a given problem into interdependent subproblems, *branch-and-bound* (B&B) is a general method based on the *divide-and-conquer* principle for cutting off unpromising branches in a decision tree. The main difference between DP and divide-and-conquer is that for the latter the subproblems created can be independent of each other. However, since both methods are limited enumeration methods, they can be intertwined such that for example the DP tree can be further pruned [15].

The main disadvantage of the DP presented in Section 5 is that the number of states to be examined is increasing extremely fast with the number of necessary moves. While the presented DP follows the idea to create all states reachable by $i+1$ moves based on a state reachable with i moves, it is necessary to explore all possible states reachable with i moves before states reachable with $i+1$ moves can be examined. However, it turns out, that most of the states reachable with i moves are not contributing to the final (optimal) solution. Therefore, the goal is to generate states which are most likely to be contributing to the final solution only. However, it is not easy to pre-estimate which states are promising and which are not.

This is where B&B comes in. B&B is based on the idea that for a minimization problem (as given in our case) it is possible to define for each node in the search tree a (global) upper and a (local) lower bound. These two bounds are then used for deciding whether subtrees of the current state are to be further investigated. While in our case the upper bound corresponds to the length of a valid solution (i.e. the so far best known solution for reaching a perfect bay state) the lower bound gives an estimate on how many moves are at least necessary to reach a perfect bay state starting from the current bay layout. As soon as the lower bound plus the number of moves already used for reaching the current bay layout is at least as high as the upper bound, no better solution than the incumbent can be found within the current subtree of the search tree. Therefore, the current subtree can be cut off, cf. Alg. 1. Applying this idea straightforwardly to the search tree defined by the DP in Section 5 only a few adaptions are necessary:

- While in the pure DP all states emerging from the current state are created, in our case only states are created (and added to the queue) for which the (theoretically) reachable number of moves is lower than the current best solution.
- Whenever a state is opened for detailed exploration (i.e. generation of substates) it is checked whether the (theoretically) best reachable solution is still better than the current best solution (since the state was originally created, a new incumbent could have been found).
- Finally, the order used for examining the states is based on the lower bound, i.e., states closer to a perfect bay layout are examined first.

6.1. Bound Computations and Initialization

To benefit from the lower and upper bounds, it is crucial to provide (good) methods for obtaining them. While as upper bound any valid solution can be used, the lower bound needs to be approximated. For this purpose, we implemented the approach presented in [11] for which preliminary tests revealed that bounds provided by this method are relatively tight—especially for small but yet real-world instances.

For initializing the upper bound (and the algorithm as well) we adapted the *lowest priority first heuristic* (LPFH) presented in [9]. For limiting the multistart heuristic, we kept the limit on the total number of iterations but changed the method how to handle the number of iterations without improvements. Since we noticed that a fixed number of iterations without improvement (as stopping criterion) is not promising for instances strongly varying in their size, we employ the following stopping criterion (in addition to the limit on the maximum number of iterations): Whenever the expression

$$\frac{\#solutions \cdot \#iterations}{iterationlimit} \geq 1 \tag{10}$$

becomes true, we stop the search for an initial solution. Here, *#solutions* corresponds to the number of solutions found so far, *#iterations* corresponds to the number of iterations already performed and *iterationlimit* indicates the limit on the number of total iterations. The reasoning behind this stopping criterion is, that in cases where a vast amount of solutions can be found by LPFH, the initialization process can be stopped early to save computation time. If, however, only a few (or no) solutions can be found, the search is prolonged such that at least one or two solutions can be provided.

Furthermore, we extended LPFH such that not only the best found solution is returned but all solutions which were identified during search. This way, it is possible to pre-initialize the DP-based B&B algorithm such that not only the initial state is added to the processing queue but all intermediate states (leading to a perfect layout) obtained during LPFH search. Since the B&B algorithm first examines DP states close to perfect layouts, this initialization procedure leads to exploring states close to solutions before continuing with states close to the initial state. Obviously, it can be expected, that this modus operandi leads faster to promising solutions than a sole uninitialized B&B approach.

7. DP-based Heuristic

Although the branch-and-bound algorithm presented in the previous section strictly limits the number of DP states examined, the overall number of states might still be very large. Obviously, this leads to long computation times. To reduce the number of states, a heuristic procedure can be provided for determining whether or not two DP states are equivalent, leading to a decreased number of DP states to look at. For this purpose, the ordering of stacks as presented in Section 5.1 has to be slightly modified such that for two stacks s_i, s_j with s_i stored at a lower index than s_j, $\overline{s_i} \geq \overline{s_j}$ holds. If $\overline{s_i} = \overline{s_j}$, $\widetilde{s_i} \leq \widetilde{s_j}$ has to be true. If furthermore $\widetilde{s_i} = \widetilde{s_j}$, then $c^i_{\overline{s_i}+1} < c^j_{\overline{s_j}+1}$, i.e. the priority of the lowest container in the left stack being not in set $\overline{C}$ is higher than the priority of the lowest container in the right stack being not in set $\overline{C}$. However, if $c^i_{\overline{s_i}+1} = c^j_{\overline{s_j}+1}$, we iteratively continue with $c^i_{\overline{s_i}+k}$ and $c^j_{\overline{s_j}+k}$, $2 \leq k < h - \overline{s_i} - \widetilde{s_i}$ until a decision can be made. If no decision can be made an arbitrary order can be chosen.

This leads to the observation that two states are said the be equivalent iff

$$\overline{s_i} = \overline{s'_i}, \qquad \text{for } 1 \leq i \leq w \tag{11}$$

$$\widetilde{s_i} = \widetilde{s'_i}, \qquad \text{for } 1 \leq i \leq w \tag{12}$$

$$c^i_{\overline{s_i}+k} = c'^i_{\overline{s_i}+k}, \qquad \text{for } 1 \leq i \leq w, 1 \leq k \leq h - \overline{s_i} - \widetilde{s_i} \tag{13}$$

These equivalence rules are based on the reasoning that elements being member of $\overline{C}$ are already perfectly placed and relocations of them will occur only in rare situations. However, please note that for crowded bays this method cannot find an (optimal) solution, cf. Fig. 3c.

8. Computational Results

To evaluate the performance of the proposed method, comprehensive computational experiments have been performed.

Table 1: Number of instances for which the algorithm reaches the optimum (opt), the best known value (best) and no valid solution at all (none) for the bays of dimensions 4x4, 4x7 and 4x10 (dims) and utilization rates 50%, 75% and 100% (util) within either 600 seconds or 3600 seconds available computation time.

dims		heur_{600}			heur_{3600}			iDPBnB_{600}			iDPBnB_{3600}			DPBnB_{600}			DPBnB_{3600}		
	util	opt	best	none	opt	best	none	opt	best	none	opt	best	none	opt	best	none	opt	best	none
4×4	050	400	400	0	400	400	0	400	400	0	400	400	0	400	400	0	400	400	0
	075	400	400	0	400	400	0	400	400	0	400	400	0	400	400	0	400	400	0
	100	272	317	14	297	368	12	270	316	14	297	364	12	229	257	34	265	316	30
4×7	050	400	400	0	400	400	0	400	400	0	400	400	0	400	400	0	400	400	0
	075	399	399	0	399	399	0	399	399	0	399	399	0	397	397	0	398	398	0
	100	153	234	75	177	347	8	147	220	73	187	349	7	129	187	86	183	346	9
4×10	050	400	400	0	400	400	0	400	400	0	400	400	0	400	400	0	400	400	0
	075	360	371	0	364	374	0	362	373	0	362	373	0	378	390	0	380	392	0
	100	66	232	102	70	283	91	65	227	102	74	285	92	65	237	102	71	288	92

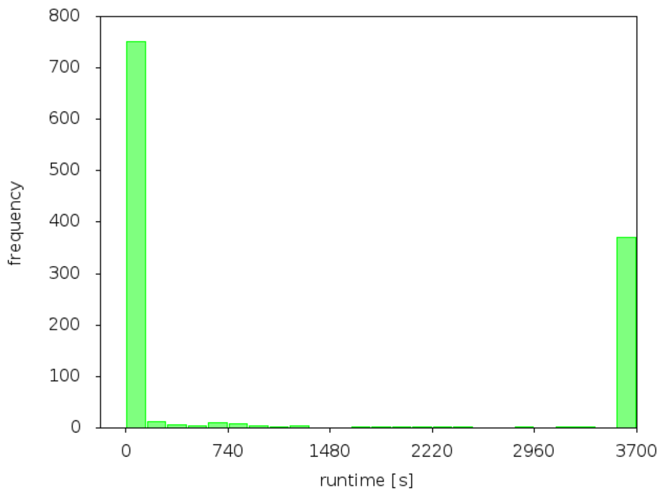

Figure 4: Histogram of runtimes with class width 120s for all instances of dimensions 4×10.

For this purpose, the algorithms were implemented in Java 7 and all tests were carried out on a single core of an Intel® Core™2 Quad CPU Q9300 with 2.50GHz. Although 3GB RAM were available for each core only a fraction of this was used by the proposed methods.

Since there is no set of standard benchmark instances yet established for the PMP we performed our experiments on instances available at [16], where the instance generator presented in [9] can be found as well. Unfortunately, the instances used in [9] are not available for download. Comparisons to other approaches were unfortunately not possible since either the instances were not available or the results reported in other works are not meaningful for detailed comparisons (e.g. varying stack heights for the same sets of instances, cf. Sec. 3, etc.).

The instances available at [16] can be clustered according to the following categories: In total there are 3600 instances, where 1200 are sized 4×4, 1200 4×7 and 1200 4×10. That is, they consists of bays with width 4, 7 and 10, respectively, and stack height 4. All of these instances can be further divided into 3 sets of 50%, 75% and 100% utilization rate—meaning that $x\%$ ($x \in \{50, 75, 100\}$) of all available slots are occupied by a container. The resulting 9 sets consist of 400 instances each. For all instances, the maximum stack height is expanded to 6 such that even for instances with utilization rate 100% free tiers are available for container re-arrangements.

Preliminary tests revealed that the pure DP approach was not able to find solutions (in acceptable computation times) for many instances given in the test set. Therefore, we limit the further discussion to results obtained by an uninitialized DP-based branch-and-bound method, an initialized DP-based branch-and-bound approach, and the DP-based heuristic. We refer to Table 1 for results obtained when applying these three different methods. For each algorithm—the DP-based branch-and-bound approach without (DPBnB) and with (iDPBnB) initialization as well as the DP-based heuristic (heur)—the number of instances for which the optimal solution (opt), the best known solution (best) and no solution at all (none) could be found is given. In addition, we report these values for applying the methods either with a time limit of 600 seconds or with 3600 seconds.

According to these numbers, it can be seen that with respect to solution quality no considerable difference between those 6 algorithmic setups can be determined for instances with utilization rates of 50% and 75%. For instances with an utilization rate of 100% the runs with additional computation time reach superior solutions (or at least have a reduced number of instances without any solution). It has to be highlighted that in fact the heuristic DP state equivalence determination method as well as the initialization phase of iDPBnB does not considerably contribute to the solution quality.

In addition, we provide the average computation times for each algorithmic setup (together with standard deviations) in Table 2 for providing the optimal solution (opt) and reaching a solution at all (tot), respectively. It can be concluded from the numbers shown in Table 2, that the heuristic setting does not even outperform the exact settings with respect to computa-

Table 2: Average computation times in seconds (standard deviations written with cursive numbers) for the results shown in Table 1. The mean value for optimal solution found (opt) and solution found (tot) are reported.

dims		heur$_{600}$		heur$_{3600}$		iDPBnB$_{600}$		iDPBnB$_{3600}$		DPBnB$_{600}$		DPBnB$_{3600}$	
	util	opt	tot	opt	tot	opt	tot	opt	tot	opt	tot	opt	tot
4×4	050	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.001	0.001	0.001	0.001
		0.006	*0.006*	*0.006*	*0.006*	*0.007*	*0.007*	*0.006*	*0.006*	*0.003*	*0.003*	*0.003*	*0.003*
	075	1.954	1.954	2.065	2.065	2.058	2.058	1.923	1.923	0.039	0.039	0.040	0.040
		13.426	*13.426*	*16.366*	*16.366*	*17.776*	*17.776*	*14.636*	*14.636*	*0.116*	*0.116*	*0.120*	*0.120*
	100	148.169	15.202	368.949	1121.629	13.411	15.238	375.753	1131.969	13.148	15.749	443.670	1339.405
		214.653	*273.924*	*677.269*	*1486.562*	*211.296*	*273.877*	*704.324*	*1500.218*	*204.847*	*276.062*	*798.077*	*1576.748*
4×7	050	0.007	0.007	0.006	0.006	0.007	0.007	0.006	0.006	0.007	0.007	0.005	0.005
		0.023	*0.023*	*0.022*	*0.022*	*0.031*	*0.031*	*0.018*	*0.018*	*0.012*	*0.012*	*0.008*	*0.008*
	075	8.331	9.811	26.768	35.701	6.137	7.622	16.747	25.705	13.948	18.344	9.587	27.539
		61.197	*67.904*	*267.255*	*321.197*	*48.663*	*56.958*	*189.931*	*260.928*	*68.602*	*85.053*	*68.286*	*262.553*
	100	139.698	383.343	523.453	2184.784	130.131	388.809	526.388	2137.570	168.132	422.641	10.541	2154.114
		198.355	*267.257*	*881.709*	*1648.132*	*185.634*	*265.042*	*892.579*	*1655.450*	*220.417*	*255.303*	*876.368*	*1655.643*
4×10	050	2.562	2.562	11.503	11.503	2.486	2.486	11.684	11.684	0.735	0.735	1.886	1.886
		32.558	*32.558*	*182.406*	*182.406*	*32.240*	*32.240*	*182.808*	*182.808*	*9.486*	*9.486*	*25.537*	*25.537*
	075	22.221	80.002	97.392	412.634	23.496	78.266	116.012	446.999	15.413	47.568	59.178	236.226
		87.561	*192.407*	*429.043*	*1083.885*	*87.399*	*188.572*	*464.061*	*1114.043*	*69.880*	*149.745*	*314.375*	*831.230*
	100	244.293	520.940	974.573	2997.914	248.662	523.426	1003.173	2976.320	246.201	522.892	976.486	2995.439
		218.139	*179.704*	*1064.459*	*1214.226*	*226.411*	*179.383*	*1107.856*	*1235.748*	*221.293*	*178.834*	*1010.507*	*1207.451*

tion times. Even more, it can happen that the uninitialized pure DP based branch-and-bound approach (DPBnB) performs better than the other variants (e.g. instances of dimensions 4×4 and utilization rate 75%) since on the one hand there exists an overhead for initializing the search via LPFH and on the other hand the initialization can be misleading if an unpromising initial solution is provided.

With respect to the numbers shown in Table 2 it has to be highlighted that standard deviations are relatively high compared to the average runtimes. This can be explained by the fact that for many instances the methods either find a solution in very short time or very long runtimes are obtained. "Medium-sized" runtimes are, however, almost not existing, cf. also Fig. 4 showing a histogram of runtimes for all tests performed on instances with dimensions 4×10. Comparing the results for 600 second and 3600 seconds runtime limit, it can be observed that the additional available computation time does not contribute to the solution quality, cf. Tab. 1 leading to the conclusion that even (much) longer runtimes would be necessary for achieving a considerable quality enhancement. Although this observation is not stunning with respect to the applicability of the proposed methods to (very) large real-world instances, it gives a promising outlook towards a hybridization of these methods with (meta-)heuristic approaches since small and medium-sized instances can be solved in very short computation times. Therefore, these methods can be employed as subordinates for exactly solving (small) subproblems identified by an enclosing (metaheuristic) approach.

A further observation on numbers not shown in the tables can be made with respect to the difficulty of the instances used for testing: In addition to the classification into instances of different dimensions and utilization rates, for each of the 9 above presented test sets four difficulty classes are defined. While for the small instances, they do not have an impact on the solution quality as well as the runtimes, it can be observed that with increasing difficulty, the solution quality (i.e. the number of runs reaching an (optimal) solution) decreases. However, since it is at this time not possible to determine in advance whether or not an instance is difficult, no algorithmic advantage can be taken.

9. Conclusions

Within this paper, we proposed a novel *dynamic programming* (DP) approach for solving the container pre-marshalling problem (PMP). Since preliminary tests revealed that the pure DP method is not able to find solutions in acceptable computation times for instances comparable to real-world settings, we extended the DP formulation by embedding it into a *branch-and-bound* (B&B) framework. An appropriate initialization procedure is adopted from [9] as well with the goal to speed up the search. Finally, a heuristic DP state equivalence determination procedure is introduced sacrificing optimality in favor of search speed.

Although extensive computation experiments showed that for large (real-world) instances exact methods are still inapplicable, these experiments also revealed that all approaches proposed for accelerating the originally introduced DP perform equally good, such that it is possible to obtain optimal container re-arrangement plans within short computation times for a large set of realistic instances.

Although it could be shown that the proposed methods are applicable to (small and medium-sized) real-world instances, many container terminals are facing a more complex variant of the PMP. While in the classical PMP, it is assumed that only one bay has to be re-arrangement at the same time, it is often necessary to do the same procedure for several bays simultaneously while still only one gantry crane is available. Therefore, so-called reach-stackers are employed. While they have the clear advantage that inter-bay movements, i.e., movements from one bay to another, can be performed much faster than with gantry cranes, they have only limited access to the container—namely, only the top-most container of stacks s_1 and s_w can be accessed for each bay. It is therefore necessary, in future works to find container re-arrangement plans which try to optimize all container transports concurrently taking different container lifting vehicles in consideration.

Acknowledgments

We want to thank our project partners Logistikum Steyr (FH OÖ Forschungs & Entwicklungs GmbH), Ennshafen OÖ GmbH and via donau – Österreichische Wasserstraßen-Gesellschaft mbH for providing us valuable insights into container transport and container terminal processes as well as problems and challenges related to them.

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

[2] D. Steenken, S. Voß, R. Stahlbock, Container terminal operation and operations research - a classification and literature review, OR Spectrum 26 (2004) 3–49.

[3] I. F. Vis, R. de Koster, Transshipment of containers at a container terminal: An overview, European Journal of Operational Research 147 (1) (2003) 1–16.

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

[5] M. Caserta, S. Voß, M. Sniedovich, Applying the corridor method to a blocks relocation problem, OR Spectrum 33 (2011) 915–929.

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

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

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

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

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

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

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

[13] F. König, M. Lübbecke, Sorting with complete networks of stacks, in: S.-H. Hong, H. Nagamochi, T. Fukunaga (Eds.), Algorithms and Computation, Vol. 5369 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 895–906.

[14] R. E. Bellman, Dynamic Programming, Dover Publications, 2003.

[15] J. Puchinger, P. J. Stuckey, Automating branch-and-bound for dynamic programs, in: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM '08, ACM, New York, NY, USA, 2008, pp. 81–89.

[16] `http://sites.google.com/site/gciports` (last visited Dec 2012).