European Journal of Operational Research 217 (2012) 531–540

Contents lists available at SciVerse ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Production, Manufacturing and Logistics

A tree search procedure for the container pre-marshalling problem

Andreas Bortfeldt, Florian Forster *

Department of Information Systems, University of Hagen, Profilstr. 8, D-58084 Hagen, Germany

ARTICLE INFO

Article history:
Received 19 August 2010
Accepted 6 October 2011
Available online 15 October 2011

Keywords:
Transportation
Logistics
Seaport
Container pre-marshalling
Tree search

ABSTRACT

In the container pre-marshalling problem (CPMP) n items are given that belong to G different item groups ($g = 1,\ldots,G$) and that are piled up in up to S stacks with a maximum stack height H. A move can shift one item from one stack to another one. A sequence of moves of minimum length has to be determined that transforms the initial item distribution so that in each of the stacks the items are sorted by their group index g in descending order. The CPMP occurs frequently in container terminals of seaports. It has to be solved when export containers, piled up in stacks, are sorted in a pre-marshalling process so that they can be loaded afterwards onto a ship faster and more efficiently. This article presents a heuristic tree search procedure for the CPMP. The procedure is compared to solution approaches for the CPMP that were published so far and turns out to be very competitive. Moreover, computational results for new and difficult CPMP instances are presented.

1. Introduction

The subject of this paper is the container pre-marshalling problem (CPMP), a combinatorial optimization problem that can be formulated as follows: we are given S stacks and each stack can accommodate up to H items. There are n items ($n \leqslant SH$) that belong to G different item groups ($g = 1,\ldots,G$; $G > 1$). Any distribution of the n items to the S stacks constitutes a layout while in a *final* layout the following condition is satisfied for each of the S stacks: if item i_2 lies above item i_1 (in the same stack) then the group index of i_2 is not bigger than that of i_1, i.e. $g(i_2) \leqslant g(i_1)$. A move can shift the topmost item of a stack to the top of another stack. Given an arbitrarily chosen layout, called initial layout, we have to determine a move sequence of *minimum* length, i.e. with the smallest number of moves, by which the initial layout is transformed to any final layout.

We propose a heuristic tree search procedure for the CPMP that mainly consists of three elements. First, the procedure is based on a natural classification of possible moves. Second, an effective lower bound for the number of moves, necessary to reach a final layout from any given layout, is derived. Finally, a branching schema is developed that is based on move sequences instead of single moves. In a comparison to former solution approaches the tree search procedure turns out to be very effective and efficient.

In the sequel, we show that the CPMP occurs in the pre-marshalling process of containers in seaports (Section 2) and give a short review of the relevant literature (Section 3). Some definitions and notation are introduced (Section 4) before a lower bound for the number of moves (as mentioned above) is proposed (Section 5). Afterwards, the tree search procedure is described (Section 6) and evaluated by extended computational experiments (Section 7). Finally, some conclusions are drawn (Section 8).

2. The container pre-marshalling problem in seaports

In order to show that the CPMP occurs in container terminals of seaports we describe how export containers are usually operated after their arrival in a container terminal. Export containers delivered on shore are first transported to the yard of the container terminal. There they are usually stored for some days before being transported to the berth of the planned container ship for loading. The yard is divided into several blocks, and each block consists of several bays. Each bay in a block has the same number of stacks (see Fig. 1).

Each stack has the same number of slots, i.e. can store the same number of containers. While a container yard can be operated in different ways, the dominant form is the indirect transfer system (ITS). In the ITS, the containers are transported to and from the yard by yard trucks. A gantry crane, e.g. a rail-mounted gantry crane, is available for moving containers in and out of a stack. This crane can only transport one container at a time and only access the top container in a stack. If a container lower down in the stack is to be taken out of store, all the containers above it have first to be transferred to other stacks. A movement of a container from stack to stack is referred to as re-handling. Fig. 2 shows a bay with a gantry crane.

A ship is loaded with the help of a stowage plan. In order to draw up a stowage plan, the containers to be loaded are usually divided into groups in accordance with the container type

* Corresponding author.
E-mail addresses: andreas.bortfeldt@fernuni-hagen.de (A. Bortfeldt), florianforster@mac.com (F. Forster).

0377-2217/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.ejor.2011.10.005

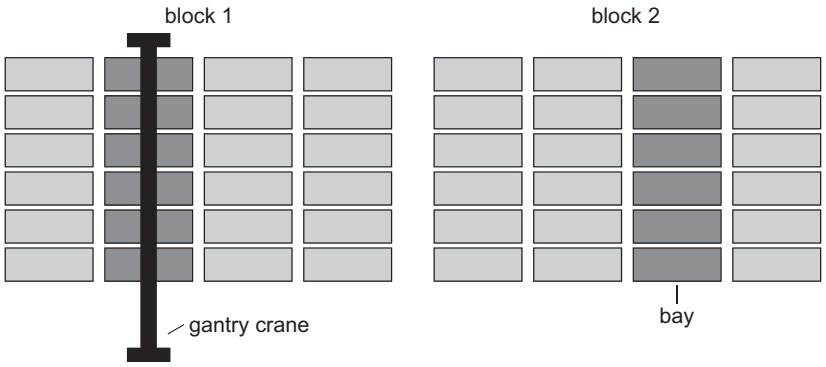

Fig. 1. Container terminal store with blocks and a gantry crane (view from above).

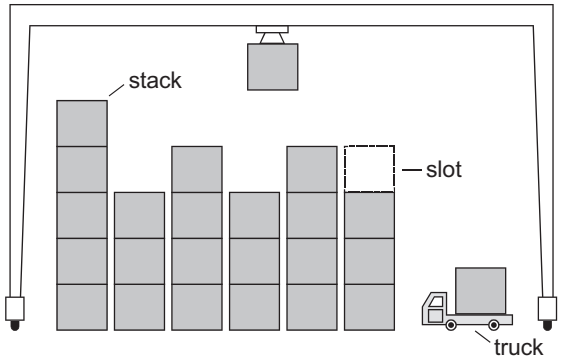

Fig. 2. Bay with gantry crane.

(20',40',...), the destination port and the weight class (light, medium, heavy). All containers with the same attributes belong to one container group. In the stowage plan, a container group, e.g. the group (20', Hamburg, light), is assigned to each slot that is to be loaded. The slot may then be loaded with any container of the specified group.

To some degree, the stowage plan determines the loading sequence of the containers. For stability reasons, heavier containers are loaded in lower and lighter containers in higher positions in the ship. Hence, in general heavier containers are loaded before lighter ones. The ports of destination also affect the loading sequence, because, in order to reduce the trans-shipping costs, containers for closer ports are placed above containers for more distant ports. Altogether we can assume that according to a given stowage plan the containers are loaded in groups. With a suitable numbering of the groups from 1 to G, all containers of a group g must be loaded first, before containers of the group $g+1$ can be loaded ($g = 1,\ldots,G-1$).

Loading a ship can only be handled efficiently if only few re-handlings have to be carried out in the yard during loading. Otherwise, considerable delays occur. Hence, the following general stacking condition (GSC) should already be fulfilled at the start of loading: none of the stacks that are reserved for a ship that is currently to be loaded holds a container in group g underneath a container in a group $g' > g (1 \leqslant g, g' \leqslant G)$. In other words, a container that is to be loaded earlier must not be positioned underneath containers that are to be loaded later. Note that the GSC can easily be extended to the case that containers of different ships are piled up in the same stacks of a yard. In this case all the containers not destined for the first ship to be loaded are temporarily put into one group with highest index. Of course, these containers have to be differentiated regarding their groups later.

When they arrive in the terminal yard the containers designated for a particular ship are usually not stacked in accordance with the GSC. The information that is required for attributing a group (port of destination, weight class) is often not available or is faulty, or the port of destination is subsequently changed.

Any allocation of a bay with containers for a ship is said to represent a layout. To ensure a container layout consistent with the GSC when loading the ship starts, the necessary re-handlings are often performed in the time remaining before loading (usually a few hours). This process is referred to as pre-marshalling the containers. The following two assumptions are made with regard to the pre-marshalling:

(1) The time that is required for moving a container from one stack to another stack in the same bay is not dependent on the distance of the stacks. This assumption is often justified because the time to position the gantry crane over a stack is negligible compared to the time it takes to pick up or deposit the container.
(2) Only re-handlings within a bay are permitted. As for example Lee and Hsu (2007) point out, this is a reasonable restriction since the re-handling over the borders of a bay via a gantry crane is very time-consuming. Therefore, if the containers for a ship are located in several bays, pre-marshalling is often carried out separately for each bay.

Pre-marshalling starts with the initial layout that is found after the containers have been brought into the yard. A final layout is any layout that fulfils the GSC. Since the pre-marshalling should take place with the least possible effort, a sequence of the fewest possible re-handlings has to be found that transforms an initial layout into a final layout.

Obviously, the real-world optimization problem just described corresponds to the (abstract) CPMP as formulated in Section 1. However, one should bear in mind that this correspondence bases on several assumptions that were formulated and justified above.

3. Related work

OR problems in container terminals are heavily discussed in the literature. Stahlbock and Voß (2007) and Steenken et al. (2004) describe the essential logistical processes in container terminals and provide an overview of the literature on this subject. There is a comprehensive representation of the decision problems in container terminals from Vis and de Koster (2003).

In particular, there are a lot of publications that deal with the various problems that arise in container yards. Chen (1999) examines the storage operations in container terminals and the strategies that are used here. Dekker et al. (2007) investigate different

stacking policies that can increase the throughput of a container yard. Kim (1997) develops a method for estimating the number of re-handlings for import containers. Kim and Bae (1998) deal with the rearrangement of containers in the yard to accelerate the subsequent loading process. However, while the movement of transfer cranes between bays is taken into account, there is no detailed examination of the rearrangement of containers within a bay. This also applies to other papers, e.g. for Kim and Kim (1998), Kim and Kim (1999) and Kim et al. (2000), which consider the transport of containers within the yard.

There are only few publications regarding the CPMP so far. Bortfeldt (2004) described a former version of the tree search heuristic that is presented and discussed in this article. The article at hand can be regarded as a revised and extended version of the former publication that takes into account the recent publications on the CPMP.

Lee and Hsu (2007) developed a mathematical model for the CPMP based on a multi-commodity flow network representation. By using integer programming software, they are able to solve small instances of the CPMP to optimality. However, due to the exponential growth in computation time, this approach seems to be not applicable to most real-world scenarios.

Lee and Chao (2009) propose a neighborhood search heuristic that is based on six different subroutines. Each subroutine either tries to add moves to the current sequence with the goal of getting closer to a final layout or attempts to reduce the number of moves in the current sequence by both finding shorter alternative paths for the containers and by eliminating obviously unnecessary moves. In contrast to the approach proposed by Lee and Hsu, the heuristic is able to find solutions for more difficult instances of the CPMP. The authors provide some computational results for random bays.

Caserta and Voß (2009a) describe an iterative four-step algorithm for solving the CPMP. In the first step, the container that will be moved next is randomly selected, with a probability proportional to the number of blocking containers in the stack. To constrain the computation time of the algorithm, the set of potential target stacks for the next container move is restricted to a preset number. For each target stack of this so-called corridor, the resulting layout is computed in step 2. Then, the actual target stack is chosen randomly based on the attractiveness of the resulting bay. As last step, two simple heuristics are applied to improve the layout. The authors computed a solution for a bay presented in the paper of Lee and Chao that is able to pre-marshal the given bay with fewer moves. Furthermore, they provide computational results for multiple random instances of the CPMP.

Up to the present time it is not known whether the CPMP belongs to the NP-hard optimization problems. However, the experience gained with recent CPMP solution approaches (as sketched above) makes it an obvious option to develop efficient heuristic procedures that do not guarantee the (global) optimality of generated solutions. This holds particularly if we want to solve large CPMP instances as in the paper at hand.

Moreover, there are several additional stacking problems that have a close relationship to the CPMP. The container relocation problem (CRP) is very similar to the CPMP in the general setting, but differs in one important aspect: the CRP allows for removing containers with the currently lowest container group from the bay. In consequence, the objective is to find a sequence of crane moves that empties a given bay and has a minimum length. Kim and Hong (2006) proposed a (complete) B& B algorithm and, noting that the approach is not feasible for larger problem instances, a greedy heuristic that yields solutions in very short computation times. Caserta et al. (2009e) developed a dynamic programming algorithm for the CRP, lowering the computational effort by processing only a subset of all potential moves in each step. Caserta and Voß (2009b,c) developed very competitive random-based heuristics for the CRP. Moreover, Caserta et al. (2009d) proposed a solution method that benefits from representing the bay as a binary matrix data structure. Forster and Bortfeld (2012) developed a tree search algorithm for the CRP that adapts the basic framework of an incomplete tree search as it is outlined in the article at hand, but uses a finer move classification scheme, different rules for branching and bounding, and requires an additional greedy heuristic. Lee and Lee (2010) define the container retrieval problem as a multi-bay generalisation of the container relocation problem and describe a solution approach that combines heuristics with integer programs.

Finally, there are also optimization problems occurring in other parts of the container terminal or even in other domains that have structural similarities to stacking problems in container yards, e.g., the container sequencing problem with internal reshuffles described by Meisel and Wichmann (2010) or the tram scheduling problem outlined by Blasum et al. (1999).

4. Definitions and notation

The following definitions and notation are needed for deriving a bound for the number of moves and describing the tree search procedure for the CPMP.

A layout (cf. Section 1) may be represented by a matrix L with H rows and S columns. The matrix L assigns a group index g ($0 \leqslant g \leqslant G$) to each slot, given by a pair (h,s) ($h = 1,\ldots,H$, $s = 1,\ldots,S$). An entry $L(h,s) = 0$ means that the slot (h,s) is empty, while an entry $L(h,s) = g > 0$ indicates that the slot (h,s) is filled by an item of group g. Of course, if $L(h,s) = 0$, then $L(h',s) = 0$ must also hold for all $h' = h+1,\ldots,H$ ($h = 1,\ldots,H-1$, $s = 1,\ldots,S$).

A move can be written as pair (d,r) of different stacks ($1 \leqslant d, r \leqslant S$, $d \neq r$). The first stack (d) is called the donator, the second stack (r) the receiver. A move shifts the highest container of the donator to the lowest free slot in the receiver that is called the receiver slot. Clearly, a move (d,r) is applicable to a layout L if and only if $L(1,d) > 0$, i.e. the donator is not empty, and $L(H,r) = 0$, i.e. the receiver is not full.

A move sequence $s = (m_1, m_2, \ldots, m_p)$ ($p \geqslant 1$) is permitted with reference to a layout L, if the move m_1 is applicable to L and each move m_{i+1} is applicable to the layout resulting after m_i ($i = 1,\ldots,p-1$). The CPMP can now be formulated more formally as follows: determine a move sequence s with p moves for a given initial layout L_{Init} that meets the following requirements: (i) s can be applied to L_{Init}; (ii) the layout resulting from the pth move is a final layout; (iii) if s' is any move sequence with p' moves that fulfils (i) and (ii), then $p' \geqslant p$.

In the following definitions L stands for any layout ($h = 1,\ldots,H$, $s = 1,\ldots,S$, $g = 1,\ldots,G$):

(1) Let $L(h,s) = g > 0$. The item in slot (h,s) of group g is called *badly placed*, if $h > 1$ and $L(h-1,s) < g$, or if there is a badly placed item in a slot (h',s), $1 < h' < h$. Otherwise the item of group g in slot (h,s) is *well placed*. A stack s is called *clean* if it has no badly placed items, otherwise it is *dirty*.
(2) A slot (h,s) is a *potential supply slot* of group g if and only if (i) the stack s is not empty and the highest well placed item in s belongs to group g *and* (ii) the slot (h,s) lies above the highest well placed item in s.
(3) A *clean supply slot* of group g is a potential supply slot of g in a clean stack. In addition, an empty stack provides H clean supply slots of group G.

Further terms and variables, also relating to any layout L, are defined in Table 1.

Note that the value of a clean supply slot in terms of possible moves is the greater the higher the group g. Therefore, the (total) clean supply of a layout L is defined as a weighted sum.

Finally, a classification schema for the possible moves regarding a layout L is introduced. Let the move m be applicable to layout L. The type of a move m is Bad–Good (in short BG), if the moved item was badly placed before the move and is well placed after the move. The move types Bad–Bad (BB), Good–Good (GG) and Good–Bad (GB) are defined analogously. A BX move is either a BG move or a BB move. GX moves are specified analogously.

5. A lower bound for the number of moves

A lower bound for the number of moves that are necessary for transforming any layout L into a final layout is presented in Proposition 1.

Proposition 1

(i) *Let* $n'_{BX} = n_b + \min\{n_b(s) | s = 1, \ldots, S\}$. n'_{BX} *is a lower bound for the number of BX moves that are necessary to reach a final layout from layout L.*

(ii) *Let* g^* *be the item group with maximum cumulative demand surplus, i.e.* $D_s(g^*) \geqslant D_s(g)$ *for* $g = 1, \ldots, G$. *The stacks in which the highest well placed item belongs to a group* $g < g^*$ *are called potential GX stacks. Let the number of potential GX stacks be* $n^p_{s,GX}$, $n_{s,GX} = \max(0, \lceil D_s(g^*)/H \rceil)$ *and assume* $n^p_{s,GX} \geqslant n_{s,GX}$. *The S stacks are renumbered so that the potential GX stacks precede all other ones and the inequality* $n^{g*}(s) \leqslant n^{g*}(s+1)$ *holds for* $s = 1, \ldots, n_{s,GX} - 1$. *Let* $n'_{GX} = \sum_{s=1}^{n(S,G,x)} n^{g*}(S)$. n'_{GX} *is a lower bound for the number of GX moves that are needed to reach a final layout from L.*

(iii) *Let* $n'_m = n'_{BX} + n'_{GX}$. n'_m *is a lower bound for the number of all moves necessary to reach a final layout from L.*

Proof

(i) Since at least one Bad–Good move must be carried out for every badly placed item in order to reach a final layout, a total of at least $n_b BG$ moves must be performed. If all stacks in layout L are dirty, BG moves can only be carried out after all badly placed items have been removed in at least one stack. This can only be done through BB moves, since there are still no slots available for badly placed items in which they would be well placed. Consequently, at least $\min\{n_b(s) \mid s = 1, \ldots, S\}$ BB moves must be carried out before the first BG move can be performed. Clearly, if layout L contains at least one clean stack, the required minimum number of BB moves is zero.

(ii) It suffices to consider the nontrivial case $D_s(g^*) > 0$. If all well placed items remain in their slots, still $D_s(g^*)$ slots are missing for repacking the badly placed items of groups $g \geqslant g^*$ so that they are well placed. These slots can only be found in potential GX stacks since all other slots were already taken into account. Moreover, additional slots for items of groups $g \geqslant g^*$ can be established in a potential GX stack only if all well placed items are removed. This must be done in at least $n_{s,\ GX} = \max(0, \lceil D_s(g^*)/H \rceil)$ potential GX stacks since a stack cannot offer more than H slots. If the $n^p_{s,GX}$ potential GX stacks are sorted by the number of well placed items in ascending order, then it results that at least $n'_{GX} = \sum_{S=1}^{n(S,G,X)} n^{g*}(S) GX$ moves must be carried out. The above assumption $n^p_{s,GX} \geqslant n_{s,GX}$ turns out to be a necessary condition for reaching a final layout if $D_s(g^*) > 0$.

(iii) This result follows immediately from (i) and (ii). □

Computing the lower bound $n'_m(L)$ is illustrated by means of the layout L in Fig. 3 with $S = 6$, $H = 4$, $G = 4$, in which the well placed containers are shown darker.

While the lower bound of BX moves is calculated as $n'_{BX} = 10 + 1 = 11$, the lower bound of GX moves is determined as follows:

(1) First, the necessary demand and supply quantities are calculated and listed in Table 2.

(2) The maximum cumulative demand surplus is 5 and occurs in item group 3. Hence, if all well placed items remain in their slots, there are still 5 slots missing for placing the badly placed containers in groups 3 and 4 in good slots. Because the stack height $H = 4$, well placed items must be removed in at least $n_{s,\ GX} = \lceil 5/4 \rceil = 2$ stacks, i.e. GX moves must be carried out from 2 stacks to provide the missing 5 slots.

(3) In stack 2 no further slots can be created by GX moves for items of groups 3 and 4. For the slots above item 3 were already considered in the potential supply and removing this item does not create any additional space, since it has to be put in a good slot again itself.

(4) In the potential GX stacks 3–6 at least one well placed item must always be removed to create additional slots for good placing of items of groups 3 and 4. Hence, at least 2 GX moves are needed, or $n'_{GX} = 2$.

6. A tree search procedure for the CPMP

In the tree search procedure the nodes of the tree represent layouts. The root node corresponds to the initial layout and each leaf node corresponds to a final layout. A direct successor L' of a layout or node L is generated by a so-called compound move. A compound move is a permitted move sequence with regard to L (cf. Section 4). A complete or incomplete solution of a CPMP instance is represented by a sequence s of compound moves. The number of all moves of s, designated by $n_m(s)$, results as the sum of all move numbers of all compound moves of s. The generation of compound moves will be described later.

Table 1
Terms and variables ($s = 1, \ldots, S$; $g = 1, \ldots, G$).

Term	Variable	Definition
–	n_b	Number of all badly placed items in L
–	$n_b(s)$	Number of all badly placed items in stack s
–	$n^g(s)$	Number of all well placed items of all groups $g' < g$ in stack s
Demand of group g	$d(g)$	Number of all badly placed items of group g
Cumulative demand of group g	$D(g)$	$d(g) + d(g+1) + \cdots + d(G)$.
Potential supply of group g	$s_p(g)$	Number of all potential supply slots of group g
Cumulative potential supply of group g	$S_p(g)$	$s_p(g) + s_p(g+1) + \cdots + s_p(G) + H \times n_{s,empty}$; $n_{s,\ empty}$ – number of empty stacks in layout L
Cumulative demand surplus of group g	$D_s(g)$	$D(g) - S_p(g)$
Clean supply of group g	$s_c(g)$	Number of all clean supply slots of group g
Clean supply of layout L	$S_c(L)$	Calculated as weighted sum $10^0 s_c(1) + 10^1 s_c(2) + \cdots + 10^{G-1} s_c(G)$

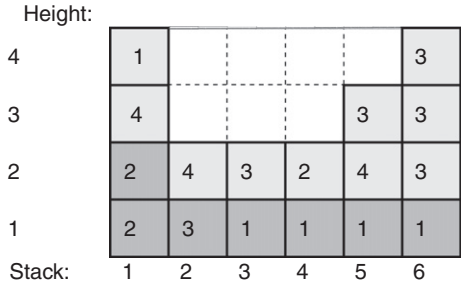

Fig. 3. Example layout *L*.

Table 2
Demand and supply values for item groups of example layout *L*.

Item group g	Demand $d(g)$	Cumulative demand $D(g)$	Potential supply $s_p(g)$	Cumulative potential supply $S_p(g)$	Cumulative demand surplus $D_s(g)$
4	3	3	0	0	3
3	5	8	3	3	**5**
2	1	9	2	5	4
1	1	10	12	17	−7

Primarily the following measures serve to keep the search effort within acceptable limits. The number of successors L' of a layout L is restricted. At most *nSucc* (*nSucc* > 1, fixed) different compound moves are applied alternatively to layout L. If L was achieved through the partial solution s, only *nSucc* solutions that extend s are taken into account. Moreover, the search is aborted if a time limit is exceeded. Hence, an incomplete search is carried out in general and the tree search procedure does not guarantee an optimal solution.

The search is carried out by means of the recursive procedure *perform_compound_moves* shown in Fig. 4. The following initializations are made before the procedure is called for the first time: current solution $s := \varnothing$, best solution $s^* := \emptyset$, current layout $L :=$ initial layout L_{Init}. At the end of the search, the best found solution s^* is returned.

In the procedure *perform_compound_moves* it is checked first whether the search may be aborted, i.e. whether an optimal solution has been reached or the time limit has been exceeded. Furthermore, the current instance of the procedure is aborted if a leaf node has been found and in this case the best solution so far is updated if necessary. Otherwise at most *nSucc* compound moves are generated and then carried out alternatively. To carry out a compound move means to concatenate the current input solution s and the compound move and to apply the compound move to the input layout L in order to get the layout L'. For each compound move the procedure *perform_compound_moves* is called again and the new partial solution s' and the successor layout L' are transferred.

Moves of the Bad–Good type reduce the number of badly placed containers and therefore the distance to a final layout, whereas this does not apply for the other move types. In order to reach a final layout from a layout L with the lowest number of moves, in particular the fewest moves of types *BB*, *GG* and *GB* must be carried out. This makes the following approach for the generation of compound moves obvious:

- Only two types of compound moves are considered. Compound moves of type 'normal' contain only *BG* moves, while compound moves of type 'extra' contain only moves of the types *BB*, *GG* and *GB*.
- Normal compound moves are preferred. If at least one *BG* move can be applied to a given layout L, only normal compound moves are generated; otherwise (only) extra compound moves are generated (cf. Fig. 4).

In the next sections we describe in detail how compound moves of both types are generated.

6.1. Generating compound moves of the normal type

The generation of normal compound moves is shown in Fig. 5. For each stack s at most one compound move is generated in which all moves have the same donator s. Since only BG moves are admitted only dirty stacks can serve as donator s. A normal compound

```
perform_compound_moves (in: s {current solution}, in: L {respective layout})

{abort the search where appropriate}
if s* ≠ ∅ and n_m(s*) = n_m' (L_Init) or time limit exceeded then return; endif;

{abort search path if leaf node was reached; update best solution if necessary}
if L is a final layout then
        if  s* = ∅  or n_m(s) < n_m(s*) then s* := s; endif;
        return;
endif;

{determine additional compound moves}
determine number n_BG of possible BG moves regarding layout L;
if n_BG > 0 then
        determine at most nSucc compound moves Cm(i), i = 1,…, nCm, of type normal;
else    determine at most nSucc compound moves Cm(i), i = 1,…, nCm, of type extra;
endif;

{perform compound moves alternatively}
for i := 1 to nCm do
        s' := s ∪ Cm(i);
        determine layout L' by applying Cm(i) to L;
        perform_compound_moves (s', L');
endfor;

end.
```

Fig. 4. Recursive procedure perform_compound_moves.

move only ends when no further BG move (from stack s) can be performed. That is, only normal compound moves of maximum length are generated according to the fact that each BG move reduces the distance to a final layout. Moreover, normal compound moves of maximum length serve the aim of converting the donator into a clean stack and thus to enable further BG moves.

A compound move with fixed donator s is generated move by move. To determine the next move, at first all possible *BG* moves (with donator s) regarding the current layout L_c (reached by the former move) are provided in a temporary vector M. However, no move is admitted that is inverse to a former move of compound move cm or imported solution s_c, respectively (not shown in Fig. 5 for brevity). The best move m_{best} is then selected from M according to the following *filtering* rules:

(1) For each move in M consider the nonnegative difference δg of the groups of the topmost item in the receiver and of the item to be moved, respectively (for an empty receiver the group of the topmost item is set to G). Discard all moves in M for which δg is not minimum.
(2) Determine the highest receiver slot *prmax* among all (not yet discarded) moves in M. Discard all (not yet discarded) moves in M with a receiver slot smaller than *prmax*.
(3) Choose the first move in M that is not discarded as best move m_{best}.

By means of the filtering rules (1) and (2) it is ensured that after a move as many other *BG* moves as possible can be carried out.

For a (nonempty) compound move it is checked whether it can be part of a new best solution. Only in this case the new compound move is accepted for list cm and the clean supply of the related layout L_c (cf. Table 1) is calculated and taken as clean supply of the compound move itself.

A compound move cm can only lead to a new best solution if the inequality $n_m(s) + n_m(cm) + n_m'(L_c) < n_m(s^*)$ holds. Herein $n_m(s)$ and $n_m(cm)$ stand for the number of moves in the transferred partial solution s_c and in the compound move cm, respectively, while $n_m'(L_c)$ denotes the lower bound of the number of moves that are needed to reach a final layout from L_c and $n_m(s^*)$ is the number of all moves of the current best solution s^*. In the beginning of the search and as long as no best solution s^* is available the number $n_m(s^*)$ is set to the (rounded) product $p_{ub*}n_m'(L_{init})$ wherein the factor $p_{ub}(p_{ub} > 1.0)$ is a percentage and $n_m'(L_{init})$ denotes the lower bound of the number of moves related to the initial layout. In so doing the search is focused on solutions with smaller move numbers and the search effort is reduced.

Finally, the generated normal compound moves are sorted in descending order in accordance with the number of moves and, with lower priority, in descending order in accordance with the clean supply. The second sorting criterion helps to enable further Bad–Good moves. Finally, only the first *nSucc* compound moves are returned.

6.2. Generating compound moves of the extra type

The generation of extra compound moves is shown in Fig. 6. Both the procedures for generating compound moves have a similar structure. However, besides the fact that an extra compound move contains only moves of the types *GG*, *BB* and *GB* (in short non-*BG*) there are three main differences compared to the generation of normal compound moves that are explained below.

First, a different set of three filtering rules is applied to select a next move for a compound move (M denotes here a temporary vector of all possible non-*BG* moves):

(1) If there is at least one *GG* move, then discard all *BB* and *GB* moves. Otherwise, if there is at least one move with a dirty receiver then discard all moves with a clean receiver.
(2) For each (not yet discarded) move in M consider the difference δg of the groups of the topmost item in the receiver and of the item to be moved, respectively (for an empty receiver the group of the topmost item is set to 0 here). If $\delta g \leqslant 0$ for at least one move, then discard all (not yet discarded) moves in M with positive δg.

```
determine_normal_compound_moves (in: s_c {current solution}, in: L {respective layout},
                                  out: Cm(i),i = 1,…,nCm {list of compound moves})
{initialise list of compound moves}
Cm := ∅; nCm := 0;

{check all stacks}
for s := 1 to S do
    current compound move  cm := ∅;
    current layout L_c := L;
    do
        generate all possible BG moves regarding layout L_c and stack s;
        if at least one BG move was found then
            select best BG move m_best;
            extend compound move cm by m_best and update layout L_c;
        endif;
    while move m_best was found;
    if cm ≠ ∅ and cm can lead to new best solution then
        determine clean supply S_c(L_c) for cm;
        Cm := Cm ∪ {cm}; nCm := nCm + 1;
    endif;
endfor;

sort list Cm
- by no. of moves in descending order and
- by clean supply in descending order;
{reduce list length if necessary}
nCm := min(nCm, nSucc);

end.
```

Fig. 5. Procedure determine_normal_compound_moves.

```
determine_extra_compound_moves (in: s_c {current solution}, in: L {respective layout},
                                 out: Cm(i),i = 1,…,nCm {list of compound moves})
{initialise list of compound moves}
Cm := ∅; nCm := 0;

{check all stacks}
for s := 1 to S do
        current compound move  cm := ∅;
        current layout L_c := L;
        do
                generate all possible non-BG moves regarding layout L_c and stack s;
                if at least one non-BG move was found then
                        select best non-BG move m_best;
                        extend compound move cm by m_best and update layout L_c;
                        if cm can lead to new best solution then
                                determine clean supply S_c(L_c) for cm;
                                if S_c(L_c) ≥ 1 then Cm := Cm ∪ {cm}; nCm := nCm + 1; endif;
                        endif;
                endif;
        while move m was found and compound move cm can lead to best new solution;
endfor;

sort list Cm
- by no. of moves in ascending order and
- by clean supply in descending order;
if compound move with maximum clean supply is not among the first nSucc compound moves then
        shift compound move with maximum clean supply at place nSucc;
endif;
{reduce list length if necessary}
nCm := min(nCm, nSucc);

end.
```

Fig. 6. Procedure determine_extra_compound_moves.

The third filtering rule corresponds to the second rule for normal compound moves. Again, the best move is selected as the first not discarded move at the end. On the one hand, by filtering rule (1) additional *BG* moves are saved by preferring *GG* moves. On the other hand, the rules (1)–(3) are aimed at facilitating and not obstructing further necessary *BG* moves.

Second, opposite to normal compound moves each permitted sequence of non-*BG* moves (with same donator *s*) is considered a complete extra compound move. This conforms to the objective of managing with as few moves of types *BB*, *GG* and *GB* as possible. On the other hand, only compound moves with a positive clean supply are accepted, since the only productive function of extra compound moves consists of enabling further *BG* moves.

Third, the compound moves are sorted in *ascending* order in accordance with the number of moves and with lower priority in descending order in accordance with the clean supply. Moreover, it is ensured that the compound move with maximum clean supply is among the first *nSucc* compound moves of list *cm*.

7. Evaluation

To evaluate the performance of our tree search procedure, we compared our algorithm with the algorithms proposed in the literature so far and performed additional computations with new random based instances. Moreover, some important components of the procedure are evaluated individually. All CPMP instances used here are available for download at www.fernuni-hagen.de/WINF. The procedure was implemented in C and compiled with XCode under Mac OS X. All tests were conducted on an Intel Core 2 Duo processor (P7350, 16 GFLOPS) with 2 GHz and 2 GB RAM under Mac OS X. Throughout the computational experiments the following parameter values were used: $nSucc = 5$, $p_{ub} = 1.75$. The time limit per instance was set to 20 s for the CPMP instances introduced by Lee and Hsu (2007), Lee and Chao (2009) and Caserta and Voß (2009a) (see also Section 7.4). Computation times are indicated in seconds throughout. Since the tree search procedure is a deterministic method each instance is calculated once only.

7.1. Benchmark against algorithm of Lee and Hsu (2007)

In their article, Lee and Hsu describe a bay with 16 stacks of maximum height 4. The bay is filled with 35 containers (we will use the problem terminology of Section 2 from now on). There are 4 different container groups and 37% of the containers are initially badly placed. Lee and Hsu find an optimal result with 15 moves on a Pentium PC (1 GHz) after 195 s. Our tree search procedure yields the same result after less than 1 s.

7.2. Benchmark against algorithm of Lee and Chao (2009)

Lee and Chao (2009) test their algorithm with 12 different CPMP instances (see Table 3). The first instance (LC1) has 10 stacks with a maximum height of 5. There are 35 containers belonging to 10 container groups. 13 containers are initially badly placed. The algorithm of Lee and Chao finds a solution with 31 container moves after 5 s. Caserta and Voß (2009a) present a solution with 19 container moves. The solution computed by our tree search procedure contains only 17 moves and is found after 1 s.

A second, slightly larger instance (LC2) contains 12 stacks with a maximum height of 6, 50 containers and 10 container groups. The algorithm of Lee and Hsu finds a solution with 61 moves after a computation time of 675 s. Since the exact initial bay layout is not provided and no information is given by the authors about the initially badly placed containers, we had to make some assumptions on this issue for our benchmark. First, we assumed 19 badly placed containers, so that the share of badly placed

Table 3
Test cases of Lee and Chao (2009).

Test case	No. of instances	No. of stacks	Stack height	No. of container groups	No. of containers	No. of badly placed containers
LC1	1	10	5	10	35	13
LC2	1	12	6	10	50	?
LC2a	10	12	6	10	50	19
LC2b	10	12	6	10	50	35
LC3	10	12	6	10	54	?
LC3a	10	12	6	10	54	21
LC3b	10	12	6	10	54	36

containers in the total number of containers is the same as in LC1. Second, we assumed a considerably higher value of 35 badly placed containers. In both cases, we generated 10 problem instances. The first instance set is referred to as LC2a, the second set as LC2b.

Finally, Lee and Chao present computational results for 10 CPMP instances (called LC3) that are quite similar to instance LC2 but have four more containers in the bay. Again, no information about the numbers of badly placed containers is given, so we estimated these values as for LC2. We generated 10 problem instances with a lower value of 21 initially badly placed containers (LC3a) and another 10 instances with a higher value of 36 badly placed containers (LC3b).

The results for the test cases of Lee and Chao are listed in Table 4. Mean values are presented for test cases with more than one instance. Compared to the algorithm of Lee and Chao the new tree search procedure was able to find significantly better solutions in less than 5 s of calculation time. The lower bound values calculated according to Proposition 1 are only slightly missed.

7.3. Benchmark against algorithm of Caserta and Voß (2009a)

Caserta and Voß (2009a) provide computational results for four different test cases CV1 to CV4 (Table 5). Each test case consists of 10 instances with a constant number of stacks (S) and containers. The initial layouts have square dimensions, i.e. in each stack the number of initially filled stack places equals the number of stacks. Each container has its own container group. For pre-marshalling to be possible, it is assumed that the stacks are not limited in height during the pre-marshalling process; in fact it is sufficient to set $H = S^2$.

The results for the test cases CV1 to CV4 are shown in Table 6. Caserta and Voß ran their algorithm on a Pentium IV Linux Workstation with 512 MB of RAM and observed a time limit of 20 s per instance. Each instance was run 10 times and the reported results were averaged over the 10 runs per instance.

For the test cases CV1, CV2 and CV3, the new tree search procedure was able to find substantially better solutions in less than 10 s of computation time. For test case CV4, the improvement of the solution quality is lesser but still significant. The lower bound gaps

Table 4
Computational results for the test cases of Lee and Chao (2009).

Test case	Lee and Chao		Authors' calculations		
	No. of moves	Time (in s)	No. of moves	Time (in s)	Lower bound gap (No. of moves)
LC1	31	5	17	< 1	1
LC2	61	675			
LC2a			22.6	1	1.5
LC2b			38.4	2.1	1.1
LC3	75.8	2106			
LC3a			23.7	1.4	1.9
LC3b			42.7	4.2	2.8

Table 5
Test cases of Caserta and Voß (2009a).

Test case	No. of instances	No. of stacks	Stack height	No. of container groups	No. of containers	No.of badly placed containers
CV1	10	3	9	9	9	5.4
CV2	10	4	16	16	16	10.6
CV3	10	5	25	25	25	17.5
CV4	10	6	36	36	36	26.5

are relatively small for the first two test cases and remain below one third of the move numbers for the other ones.

7.4. Computational experiments with new difficult CPMP test cases

In this section, we present results from computational experiments with new difficult random-based CPMP test cases. Generally, the difficulty of a CPMP instance depends on different factors and five important factors are: the number of stacks in the bay, the maximum height of the stacks, the number of container groups, the percentage of occupied slots in the bay and the percentage of badly placed containers in the initial layout in relation to all containers.

In most cases the difficulty of the CPMP should increase if one factor increases while the others remain unchanged. For example, a CPMP instance should be more difficult to solve if the dimensions of the bay are enlarged, but the occupation percentage of the bay and the other factors remain the same. This example also shows that fixing the other factors is necessary: with larger bay dimensions, but sinking occupancy rates (with the number of containers remaining the same, for example), the difficulty of the CPMP will be reduced.

In the generation of new test cases the following parameters were varied:

- Number of stacks: The bays consist of either 16 or 20 stacks. Referring to Lee and Hsu (2007), bays with 16 stacks are already viewed as large in practice.
- Stack height: The stacks either have a maximum height of 5 or 8 containers.
- Number of container groups: The number of different container groups in the bay was determined using a quota, defining how many containers in the bay have different container group values. In a bay with 100 containers, a quota of 50% means that there are 50 different container groups. For the new problem instances, container group quotas of 20% and 40% were used.
- Number of containers: The number of containers in the bay is determined relative to the dimensions of a bay. Either 60% or 80% of a bay's slots are initially filled with containers.
- Number of badly placed containers: Either 60% or 75% of the containers are initially badly placed.

Table 6
Computational results for the test cases of Caserta and Voß (2009a).

Test case	Caserta and Voß		Authors' calculations		
	No. of moves	Time (s)	No. of moves	Time (s)	Lower bound gap (no. of moves)
CV1	21.30	20	10.50	< 1	2.6
CV2	28.27	20	19.10	< 1	5.5
CV3	49.61	20	30.40	9.1	9.1
CV4	50.14	20	44.40	20	16.3

In total, 32 different test cases were defined as indicated in the first six columns of Table 7. For every test case 20 different instances were generated at random.

The resulting 640 CPMP instances were solved using our tree search procedure with a time limit of 60 s (while the other parameter values were chosen as stated above). The results are shown in Table 7, i.e. for each test case the mean number of moves per solution, the mean total computation time and the mean absolute lower bound gap are presented in columns 7–9. The results show that near-optimal solutions have been achieved for bays in which 60% of the slots are occupied by containers, even for large bay dimensions. If 80% of all slots are filled, the solutions for bays with a maximum stack height of 5 are still very close to the calculated lower bounds. Only for bays with a maximum stack height of 8, the numbers of moves in the solutions differ considerably from the lower bounds.

7.5. Evaluation of individual components of the tree search procedure

In the following we report on additional calculations that serve to evaluate individual components of the tree search procedure. In Table 8 the results of a comparison between two lower bounds of the number of moves that are needed to reach a final layout from given layout are listed. The lower bound n'_m according to Proposition 1 is compared to the simple lower bound n'_{BG} that only estimates the number of required Bad–Good moves by the number of badly placed items (cf. Proposition 1 and proof).

The lower bound improvement for a given CPMP instance set including p instances is then calculated as $(n'_m(p) - n'_{BG}(p))/n'_{BG}$ (in %), where $n'_m(p)$ and $n'_{BG}(p)$ denote the sums of the respective lower bound values over all p instances. In Table 8 three instance sets are considered. For each of the sets the number of instances, the belonging test cases (defined above) and an appropriate reference table are listed in columns 2–4 before the lower bound improvement, calculated for the entire instance sets, is presented in column 5.

A very high mean lower bound improvement was reached for the instances of Caserta and Voß (2009a) since in the initial layouts the stacks are evenly filled and no empty stacks are available so that many *BB* moves and *GX* moves are necessary. While the lower bound improvement averaged over all 640 instances of the new (BF) test cases is rather moderate, slightly larger improvements of more than 4% on average are observed for test cases with lower numbers of badly placed containers as well as for test cases with higher container numbers. Altogether the new lower bound represents a significant improvement compared to the simple bound that is solely based on Bad–Good moves.

In further calculations we studied the impact of different components and features of the tree sarch procedure on the solution quality. Besides the original variant, called V0, four additional variants were introduced. In each variant one component of the procedure is varied:

(V1) In this variant only the simple lower bound n'_{BG} (see above) is used instead of the lower bound n'_m as defined in Proposition 1.
(V2) In this variant only compound moves consisting of one simple move are applied, i.e. each compound move is cut after the first move (cf. Figs. 5 and 6).
(V3) In this variant the list *cm* of compound moves of a tree search instance is not sorted after completion (cf. Figs. 5 and 6).
(V4) In this variant no filtering rules are used to select the next move for a compound move, i.e. the first feasible move is chosen to continue a compound move (cf. Figs. 5 and 6).

The original procedure and the derived simplified variants V1 to V4 were applied to a special instance set composed by the test

Table 7
Definition of and results for 32 difficult random based CPMP test cases.

Test case	No. of stacks	Stack height	No. of containers	No. of container groups	No. of badly placed containers	Mean no. of moves	Mean time (s)	Mean lower bound gap
BF1	16	5	48	10	29	29.1	< 1.0	0.0
BF2	16	5	48	10	36	36.0	< 1.0	0.0
BF3	16	5	48	20	29	29.1	< 1.0	0.0
BF4	16	5	48	20	36	36.0	< 1.0	0.0
BF5	16	5	64	13	39	41.6	8.0	0.9
BF6	16	5	64	13	48	49.4	4.9	0.8
BF7	16	5	64	26	39	42.9	26.3	1.6
BF8	16	5	64	26	48	50.6	24.7	1.6
BF9	16	8	77	16	47	51.8	30.8	2.4
BF10	16	8	77	16	58	59.8	21.9	1.4
BF11	16	8	77	31	47	52.2	44.6	3.0
BF12	16	8	77	31	58	60.2	28.2	2.0
BF13	16	8	103	21	62	84.6	60.0	18.7
BF14	16	8	103	21	78	105.6	60.0	24.3
BF15	16	8	103	42	62	95.5	60.0	29.8
BF16	16	8	103	42	78	109.8	60.0	28.1
BF17	20	5	60	12	36	36.3	3.5	0.0
BF18	20	5	60	12	45	45.0	< 1.0	0.0
BF19	20	5	60	24	36	36.5	< 1.0	0.0
BF20	20	5	60	24	45	45.0	< 1.0	0.0
BF21	20	5	80	16	48	51.7	31.5	1.0
BF22	20	5	80	16	60	60.9	7.6	0.3
BF23	20	5	80	32	48	51.5	25.5	1.1
BF24	20	5	80	32	60	61.3	18.2	0.6
BF25	20	8	96	20	58	62.8	45.0	2.4
BF26	20	8	96	20	72	74.0	30.8	1.9
BF27	20	8	96	39	58	64.0	55.9	3.7
BF28	20	8	96	39	72	74.9	44.7	2.6
BF29	20	8	128	26	77	106.6	60.0	25.1
BF30	20	8	128	26	96	128.5	60.0	28.9
BF31	20	8	128	52	77	115.2	60.0	34.1
BF32	20	8	128	52	96	132.3	60.0	33.1

Table 8
Lower bound improvements over 721 CPMP instances.

Instance set	No. of instances	Includedtest cases	Referencetable	Lower bound improvement (in %)
LC	41	LC1, LC2a, LC2b, LC3a, LC3b	3	8.78
CV	40	CV1 – CV4	5	27.40
BF	640	BF1 – BF32	7	2.76

Table 9
Results for five variants of the tree search procedure for 113 CPMP instances.

Procedure variant	Rate of solved instances (%)	Average relative additional no. of moves for solved instances (%)	Average relative total computation time for solved instances (%)
V0	100.00	0.00	100.00
V1 (p_{ub} = 1.75)	205.86	81.42	2.01
V1 (p_{ub} = 5.0)	188.78	100.00	2.23
V2	68.14	0.88	151.81
V3	76.11	6.30	163.20
V4	77.88	13.43	214.13

cases LC1, LC2a, LC2b, LC3a, LC3b and CV1 to CV4 (cf. Tables 3 and 5). Additionally, the first instances of all 32 BF test cases were included resulting in a test set of 113 instances. The procedure variants were parameterized as stated above, i.e. the LC- and CV-instances were run with a time limit of 20 s while the 32 new instances were calculated using a time limit of 60 s. Moreover, the variant V1 was run with two different values of the parameter p_{ub}, namely p_{ub} = 1.75 and p_{ub} = 5.0, since the first value could lead to many unsolved instances as the lower bound is reduced to n'_{BG}.

Table 9 presents the results of the calculations using the procedure variants V0 to V4. For each variant the rate of solved instances is indicated as a percentage of all 113 instances. The percentages given in column 3 and 4 relate only to the *solved* instances (of the respective variant). In column 3 the mean additional number of moves – relative to the needed number of moves in variant V0 – is shown. In column 4 the mean total computation time – relative to the time needed for an instance by variant V0 – is indicated.

The results of Table 9 may be summarized as follows:

- The total of the given instances has only been solved by the variants V0 and V1 (with parameter p_{ub} = 5.0) while a high rate of unsolved instances, ranging between about 22% and about 32%, can be observed for the variants V2 to V4.
- Considering only those instances that were solved by the different procedure variants a loss of solution quality, i.e. an increase of the mean move number compared to the original procedure, can be observed in any case. A significant increase of the mean move number by more than 5% can be noted for the variants V3 (without sorting of compound moves) and V4 (without filtering rules for simple moves).
- Taking only the solved instances into account a drastical increase of the total computing time, ranging between about 50% and about 114% of the instance time needed by the original procedure, can be observed for all derived procedure variants V1 to V4. In particular it turns out that the new lower bound is very time-saving.

All in all we can state that all above investigated components and features have an essential share in the effectiveness and efficiency of the tree search procedure.

8. Conclusions

Since the container yard is often the bottleneck of a container terminal, effective solution approaches for the container pre-marshalling problem are of high economic relevance. In this article, we propose a heuristic tree seach procedure for the CPMP that is based on a natural classification of possible moves, makes use of a sophisticated lower bound and applies a branching schema with move sequences instead of single moves. The procedure was tested against all former solution approaches for the CPMP and it proved to be a very effective and efficient solution method that can be applied to large real-world CPMP instances. The performance of individual procedure components, e.g. the proposed lower bound, was verified in special tests. Moreover, 640 new and in part very challenging CPMP instances were introduced and first results were provided. The basic principle of deriving different types of gantry crane moves and using these move types for bounding and branching in a tree search might be a fruitful approach for effectively solving related problems like the container relocation problem or the CPMP with multiple bays. This remains a topic of future research.

References

Bortfeldt, A., 2004. A Heuristic for the container pre-marshalling problem. In: Proceedings of the Third International Conference on Computer and IT Applications in the Maritime Industries, pp. 419–429.

Blasum, U., Bussiek, M.R., Hochstättler, W., Moll, C., Scheel, H., Winter, T., 1999. Scheduling trams in the morning. Math. Meth. Oper. Res. 76, 55–71, 1997.

Caserta, M., Voß, S., 2009a. A corridor method-based algorithm for the pre-marshalling problem. In: Proceedings of the EvoWorkshops 2009, pp. 788–797.

Caserta, M., Voß, S., 2009b. A cooperative strategy for guiding the corridor method. Stud. Comput. Intell. 236, 273–286.

Caserta, M., Voß S (2009b). Corridor selection and fine tuning for the corridor method. In: Proceedings of the Third International Conference on Learning and Intelligent Optimization 2009; pp. 163–175.

Caserta, M., Schwarze, S., Voß, S. 2009c. A new binary description of the blocks relocation problem and benefits in a look ahead heuristic. In: Proceedings of the EvoCOP 2009, pp. 37–48.

Caserta, M., Voß, S., Sniedovich, M., 2009e. Applying the corridor method to a blocks relocation problem. Oper. Res. Spectrum (online), 1–15.

Chen, T., 1999. Yard operations in the container terminal – a study in the 'unproductive moves'. Maritime Policy Manage. 26, 27–38.

Dekker, R., Voogd, P., van Asperen, E., 2007. Ad vanced Methods for Container Stacking. In: Kim, K., Günther, H.-O. (Eds.), Container Terminals and Automated Transport Systems. Springer, pp. 131–154.

Forster, F., Bortfeld, A., 2012. A tree search procedure for the container relocation problem. Comput. Oper. Res. 39, 299–309.

Kim, K., 1997. Evaluation of the number of rehandles in container yards. Comput. Ind. Eng. 32, 701–711.

Kim, K., Bae, J., 1998. Re-marshaling export containers in port container terminals. Comput. Ind. Eng. 35, 655–658.

Kim, K., Hong, G., 2006. A heuristic rule for relocating blocks. Comput. Oper. Res. 33, 940–954.

Kim, K., Kim, H., 1998. The optimal determination of the space requirements and the number of transfer cranes for import containers. Comput. Ind. Eng. 35, 427–430.

Kim, K., Kim, H., 1999. Segregating space allocation models for container inventories in port container terminals. Int. J. Prod. Econ. 59, 415–423.

Kim, K., Young, M., Kwang-Ryul, R., 2000. Deriving decision rules to locate export containers in container yards. Eur. J. Oper. Res. 124, 89–101.

Lee, Y., Chao, S., 2009. A neightborhood search heuristic for pre-marshalling export containers. Eur. J. Oper. Res. 196, 468–574.

Lee, Y., Hsu, N., 2007. An optimization model for the container pre-marshalling problem. Comput. Oper. Res. 34, 3295–3313.

Lee, Y., Lee, Y., 2010. A heuristic for retrieving containers from a yard. Comput. Oper. Res. 37, 1139–1147.

Meisel, F., Wichmann, M., 2010. Container sequencing for quay cranes with internal reshuffles. OR Spectrum 32, 569–591.

Stahlbock, R., Voß, S., 2007. Operations research at container terminals: a literature update. Oper. Res. Spectrum 30, 1–52.

Steenken, D., Voß, S., Stahlbock, R., 2004. Container terminal operation and operations research – a classification and literature overview. Oper. Res. Spectrum 26, 3–49.

Vis, I., de Koster, R., 2003. Transshipment of containers at a container terminal. Eur. J. Oper. Res. 147, 1–16.