A tree search procedure for the container relocation problem

Florian Forster, Andreas Bortfeldt

Diskussionsbeitrag Nr. 461
Dezember 2010
A tree search procedure for the container relocation problem
Florian Forster, Andreas Bortfeldt

Abstract:

In the container relocation problem (CRP) \( n \) items are given that belong to \( G \) different item groups \((g = 1, \ldots, G)\). The items are piled up in up to \( S \) stacks with a maximum stack height \( H \). A move can either shift one item from the top of a stack to the top of another one (relocation) or pick an item from the top of a stack and entirely remove it (remove). A move of the latter type is only feasible if the group index of the item is minimum compared to all remaining items in all stacks. A move sequence of minimum length has to be determined that removes all items from the stacks. The CRP occurs frequently in container terminals of seaports. It has to be solved when containers, piled up in stacks, need to be transported to a ship or to trucks in a predefined sequence. This article presents a heuristic tree search procedure for the CRP. The procedure is compared to all known solution approaches for the CRP and turns out to be very competitive.

Key words:
Transportation, logistics, seaport, container relocation problem, tree search.
A tree search procedure for the container relocation problem

Florian Forster, Andreas Bortfeldt
University of Hagen, Department of Information Systems, 58084 Hagen, Germany

1. Introduction

The subject of this paper is the container relocation problem (CRP), also referred to as block relocation problem, 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 ($n \leq SH$) that belong to $G$ different item groups ($g = 1, \ldots, G$; $G > 1$). An arbitrary distribution of the $n$ items to the $S$ stacks is called an initial layout. A move can either shift one item from the top of a stack to the top of another one (relocation) or pick an item from the top of a stack and entirely remove it (remove). A move of the latter type is only feasible if the group index of the item is minimum compared to all remaining items in all stacks. A move sequence of minimum length, i.e. with the smallest number of moves, has to be determined that removes all items from the initial layout, leaving only empty stacks.

We propose a heuristic tree search procedure for the CRP that mainly consists of four elements. First, the procedure is based on a natural classification of feasible moves regarding any layout. Second, a greedy approach is used to quickly find a solution that acts as an initial upper bound for the tree search procedure. Third, a lower bound for the number of moves, necessary to empty the stacks from any given initial layout, is derived. Finally, a branching scheme 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 effective and efficient.

In the sequel, we show that the CRP occurs in the logistic process in seaports (Section 2) and review the relevant literature (Section 3). Some definitions and notations are introduced (Section 4) before a lower bound for the number of moves (see above) is derived (Section 5). Then, the tree search procedure is described (Section 6) and evaluated with computational experiments (Section 7). Finally, we draw some conclusions from the results (Section 8).

2. The container relocation problem in seaports

To show that the CRP occurs in container terminals of seaports we describe how containers are usually operated at a container terminal. A container terminal enables an efficient container flow between ships and landside means of transport, i.e. trucks or trains. Containers transported from trucks or trains to ships are called export containers while containers taking the reverse route are called import containers. The CRP occurs in both situations.

Export containers delivered on shore are first transported to the yard of the 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 topmost 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 relocation. A movement of a container from a stack to a waiting truck is referred to as remove. Fig. 2 shows a bay with a gantry crane and a truck.

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 (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 relocations have to be carried out in the yard during loading. Otherwise, considerable delays occur. Since the stowage plan for a ship is known in advance, terminal operators try to place the containers in stacks in a way that containers that are to be loaded later do not block containers that are to be loaded earlier. However, 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. Hence, at the time the ship arrives, the bay mostly cannot be cleared without relocations.

Import containers arrive in a high quantity at a well-known point in time when a ship is unloaded. But when the containers are to be stored in the container yard, the arrival time of the trucks (or trains) that will transport these containers to its next destination is only vaguely known. Moreover, unpredictable events like congestions or truck breakdowns can lead to changes of the timetables. It is hardly possible for the terminal operators to place all import containers at non-blocking positions in the stacks, since the order of truck arrivals and therefore the appropriate container group indices are undetermined when the decision has to be made. Hence, when a group of trucks arrive at the yard, it is likely that the requested import containers cannot be moved to the trucks without relocations of blocking containers. To minimize the waiting time for the trucks, the containers need to be removed from the stacks with a minimum of time-consuming relocations.

To summarize, for both export and import containers in container yards, sequences of crane moves are sought that clear a bay in minimum time while observing a given order when removing containers. Obviously, this situation corresponds to the (abstract) CRP formulated in Section 1, given that the following two assumptions are valid:

1) The time that is required for moving a container from one stack to another stack in the same bay does not depend 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 relocations within a bay are permitted. This is a reasonable restriction since relocations 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, relocations are often carried out separately for each bay.

Since the unloading of a bay should take place with the least possible effort, a sequence of minimum length is sought. Note that this problem definition is also applicable to the case when a certain number of containers should remain in the bay: We assume that only \( m \) \((m<n)\) containers should be removed and then assign the group index \( G+1 \) to the \((n-m)\) containers that should not leave the bay. By solving the CRP, we get a sequence of crane moves that removes all \( n \) containers. Then, the first moves concerning only containers of the groups 1 to \( G \) form a sequence of crane moves that removes the \( m \) requested containers and leaves the residual \((n-m)\) containers in the bay.

3. Related work

Logistical problems in container terminals are widely 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. A comprehensive survey of decision problems in container terminals was given by 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
a few 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.

The container pre-marshalling problem (CPMP) is fairly similar to the CRP. It can be regarded as a restrictive variant of the CRP that allows only relocations as moves. Consequently, the goal is not to empty a given bay (which is impossible without removes), but to pre-marshal the containers using only relocations, resulting in a layout that would allow then the containers to be removed without any further relocation. Algorithms for the CPMP were developed by Bortfeldt (2004), Lee and Hsu (2007), Lee and Chao (2009), Caserta and Vöß (2009e) and recently by Bortfeldt and Forster (2010).

So far, there are only few publications that address the CRP. Kim and Hong (2006) present a branch and bound (B&B) algorithm that uses the number of blocking containers in the initial layout as lower bound for the number of required relocations. In the branching procedure, new nodes are expanded using a depth-first strategy. Among the nodes of same depth, the nodes with smaller lower bound values are processed first. To reduce complexity, relocations may only be performed from a stack that holds a container with currently lowest group index. Noting that the B&B approach is only practical for relatively small problem instances, the authors also present a greedy heuristic. For this, they first define a function that estimates the number of relocations necessary to empty the bay and takes into account the number of blocking containers as well as the number of free slots in each stack. Starting with an initial layout, the heuristic determines all possible moves and generates the corresponding layouts. Then, the value of the estimation function is calculated for all layouts. Finally, the move that leads to a layout with minimum number of additional relocations is chosen. The procedure is repeated until the bay is empty. The authors evaluate their algorithms with random based problem instances. The evaluation shows that the B&B approach becomes infeasible for bays with a product of stack number and stack height higher than 20, while the greedy heuristic is able to solve even these larger instances in about one second of computation time.

Caserta et al. (2009a) propose a solution scheme for the CRP that is based on dynamic programming (DP). To make the DP scheme applicable to larger problem instances, the authors propose to limit the feasible moves regarding a given layout to a two-dimensional corridor defined by a tuple \((\delta, \lambda)\): Given that the next container that has to be removed resides in stack \(s\) at slot \(h\), all containers in \(s\) that are placed in slots above \(h\) need to be relocated. Instead of considering all (non-full) stacks in the bay as potential target stacks for these containers, only stacks that are \(\delta\) stacks away from \(s\) and currently contain less than \(\lambda\) containers are considered. This way, the solution space is restricted. The authors present computational results for random based test cases, comparing their algorithm with the greedy heuristic proposed by Kim and Hong (2006). The corridor-based approach yields considerable better results for small, medium and large problem instances.

Caserta et al. (2009b) show how a layout for a CRP can be represented as a matrix of binary values. Then, they formulate a heuristic that makes use of the following “min-max-greedy” rule: Assuming that each item has its own item group (no item group is assigned to more than one item), there is always a unique item \(i_0\) with lowest item group. If \(i_0\) is on the top of a stack, it is removed. Otherwise, there are \(m\) items \(i_j (1 \leq j \leq m)\) above \(i_0\) that need to be relocated, starting with the topmost item of the stack. For a given item \(i_j\) with item group \(r\), the target stack is chosen according to the following decision rule, in which \(\text{min}(s)\) denotes the minimum item group in a stack \(s\): If there is at least one non-full stack with \(r < \text{min}(s)\), the target stack with minimum \(\text{min}(s)\) value among those stacks is chosen. Otherwise an empty stack is chosen as target stack. If there is no empty stack in the layout, the non-full stack that maximizes \(\text{min}(s)\) is selected. This is repeated until all items above \(i_0\) are removed. In their
algorithm, the authors construct a solution \( s \) for a given layout move by move. If a remove is possible, it is appended to \( s \) and applied to the layout. Otherwise, the stack that contains the item with currently lowest item group is determined, and all possible relocations for the top item of this stack are constructed. For all resulting layouts of these relocations a complete greedy solution is computed using the “min-max-greedy” rule explained above. Then, the relocation that is actually appended to \( s \) is chosen at random with a probability inverse to the number of moves in the belonging greedy solution. This is repeated until the bay is empty. The whole random based procedure is carried out iteratively until a time limit is reached. Finally, the best solution is returned. The authors argue that their approach strongly benefits from the layout representation as binary matrix, since it allows for a very efficient calculation of greedy solutions, which in turn has to be done numerous times in each iteration.

In two further articles, Caserta and Voß (2009c, 2009d) describe a variant of this approach. Again, the solution is built move by move in a random based algorithm. If a relocation is necessary, a fixed-size subset of all possible relocations in the current layout is chosen with a roulette-type mechanism using an attractiveness score for the target stacks. Afterwards, the actual move is selected from the subset with a probability inverse to the numbers of blocked items after a respective move is applied. In a performance evaluation, the authors tried different subset sizes for each test case under consideration. Then, they listed the best solutions and the corresponding parameter values for each test case.

Up to the present time it is not known whether the CRP belongs to the NP-hard optimization problems. However, the experience gained with recent CRP solution approaches (as outlined above) makes it an obvious option to develop efficient heuristic procedures that do not guarantee the (global) optimality of generated solutions.

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 CRP.

4.1. Formal definitions
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 \( g \) \((0 \leq g \leq G)\) 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 \( h' = h + 1,\ldots,H \) \((h = 1,\ldots,H–1, s = 1,\ldots,S)\). Conversely, \( L(h,s) > 0 \) for \( h > 1 \) implies that \( L(h',s) > 0 \) for all \( h' = h–1,\ldots,1 \), i.e. a slot \((h,s)\) with \( h > 1 \) can only be filled if all lower slots in the stack are already filled. A layout is called empty if \( L(1,s) = 0 \) for all stacks \( s = 1,\ldots,S \).

A move can either be a remove or a relocation. A remove removes the topmost container of a stack from the layout and thus can simply be specified as \((s)\) \((1 \leq s \leq S)\). Clearly, a remove is only possible if \( L(1,s) > 0 \), i.e. if stack \( s \) is not empty. Moreover, remove \((s)\) is only allowed if the topmost item of \( s \) belongs to the currently lowest item group, i.e. if group \( g \) of the item is not greater than group \( g' \) of any other item in the current layout.

A relocation can be written as pair \((d,r)\) of different stacks \((1 \leq d,r \leq S, d \neq r)\). The first stack \( d \) is called the donator or source stack, the second stack \( r \) is said to be the receiver or target stack. A relocation shifts the highest item of the donator to the lowest free slot in the receiver that is called the receiver slot. Clearly, a relocation \((d,r)\) 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 \( s = (m_1, m_2, \ldots, m_p) \) \((p \geq 1)\) can be applied to a layout \( L \) (or is permitted for \( L \)), 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 CRP can now be formulated more formally as follows: determine a move sequence \( s \) with \( p \) moves for a given initial layout \( L_{\text{init}} \) that meets the following requirements: (i) \( s \) can be applied to \( L_{\text{init}} \); (ii) The layout resulting from the \( p \)-th move is empty; (iii) If \( s' \) is any move sequence with \( p' \) moves that fulfils (i) and (ii), then \( p' \geq p \).

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

1. Let \( L(h,s) > 0 \). The item in slot \((h,s)\) of group \( g \) is called **badly placed**, if \( h > 1 \) and there is at least one \( h' \) with \( 0 < h' < h \) and \( L(h',s) < g \). We call the item in slot \((h,s)\) a blocking item and the item in slot \((h',s)\) a blocked item. Otherwise the item is **well placed**. A stack \( s \) is called **clean** if it has only well placed items, otherwise it is **dirty**.

2. A slot \((h,s)\) is a **clean supply slot** of group \( g \) if (i) the stack \( s \) is not empty, (ii) the stack is clean, (iii) the highest item in \( s \) belongs to group \( g \) and (iv) the slot \((h,s)\) lies above the highest item in \( s \). 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.

<table>
<thead>
<tr>
<th>Term</th>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>( n(L) )</td>
<td>number of all items in ( L )</td>
</tr>
<tr>
<td>-</td>
<td>( n_b )</td>
<td>number of all badly placed items in ( L )</td>
</tr>
<tr>
<td>clean supply of group ( g )</td>
<td>( s_c(g) )</td>
<td>number of clean supply slots in ( L ) for item group ( g )</td>
</tr>
<tr>
<td>clean supply of layout ( L )</td>
<td>( S_c(L) )</td>
<td>calculated as weighted sum ( 1 \cdot s_c(1) + 2 \cdot s_c(2) + \ldots + G \cdot s_c(G) )</td>
</tr>
<tr>
<td>-</td>
<td>( g_{\text{top}}(s) )</td>
<td>group of topmost item in stack ( s ), if ( s ) is not empty and topmost item is badly placed, ( n(L) + 1 ) otherwise</td>
</tr>
<tr>
<td>-</td>
<td>( g_{\text{min}}(s) )</td>
<td>minimum group of all items in stack ( s ), if ( s ) is not empty, 0 otherwise</td>
</tr>
</tbody>
</table>

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.

### 4.2. A classification scheme for relocations

We suggest classifying the relocations in the following way:

- First, relocations can be differentiated according to the placement status of the moved item before and after the relocation. If a relocation \( m \) is applicable to layout \( L \) its type is Bad-Good (in short \( BG \)), if the moved item was badly placed before the relocation and is well placed after the relocation. The relocation types Bad-Bad (\( BB \)), Good-Good (\( GG \)) and Good-Bad (\( GB \)) are defined analogously.

- Second, \( BG \) and \( BB \) relocations can be classified more specifically regarding whether the moved item is blocking an item of the currently lowest item group or not. The former is called a \( FLG \) (free lowest group) relocation, the latter a \( FOG \) (free other group) relocation.

Thus, every relocation is exactly of one of the following six types: \( FLG \_BG, FOG \_BG, FLG \_BB, FOG \_BB, GG \) or \( GB \). Furthermore, the move types \( FLG \_BG, FOG \_BG, FLG \_BB, GG \) and the removes are called **productive** move types.

### 5. A lower bound for the number of moves

Proposition 1 presents a lower bound for the number of moves that are necessary for transforming any layout \( L \) into an empty layout.
Proposition 1:

(i) Let $n_{REM}' = n(L)$. Each move sequence that empties $L$ contains exactly $n_{REM}'$ removes.

(ii) Let $n_{BG}' = n_b$. $n_{BG}'$ is a lower bound for the number of required $BG$ relocations to empty $L$.

(iii) Let be $n_{non-BG}'$ a binary variable ($n_{non-BG}' \in \{0,1\}$) and $n_{non-BG}' = 1$ if and only if the following conditions regarding layout $L$ hold: (a) no item can be removed, (b) there is no empty stack, (c) $\min\{g_{top}(s) \mid 1 \leq s \leq S\} > \max\{g_{min}(s) \mid 1 \leq s \leq S\}$. Then $n_{non-BG}'$ is a lower bound for the number of relocations not of type $BG$ to reach an empty layout.

(iv) Let $n_m' = n_{REM}' + n_{BG}' + n_{non-BG}'$. $n_m'$ is a lower bound of all required moves to empty $L$.

Proof: (i) Obvious. (ii) Every badly placed item $c$ is blocking at least one item $d$ that must be removed before $c$ can be removed. Hence, $c$ has to be relocated at least once. Moreover, there must be at least one $BG$ relocation concerning item $c$ since after $BB$ relocations $c$ would continue blocking an item. (iii) Since a remove is ruled out a relocation has to be done first. No $BG$ relocation is possible as there is no empty stack and because of assumption (c) no badly placed item in topmost position can be placed well on another non-empty stack. Hence, a relocation not of type $BG$ has to be done. (iv) Follows from (i) to (iii).

The lower bound of proposition 1 represents a slight improvement compared to the bound introduced by Kim and Hong (2006) which does not consider non-$BG$ relocations.

In the following, the lower bound calculation is explained using the example layout $L$ in Fig. 3 with $S = 4$, $H = 6$, $n(L) = G = 16$, in which the well placed containers are shown darker.

![Fig. 3: Example layout for lower bound calculation.](image)

We have $n_{REM}' = n(L) = 16$ and $n_{BG}' = 8$ as there are eight badly placed items. No remove can be performed and there is no empty stack. Moreover, $\min\{g_{top}(s) \mid 1 \leq s \leq S\} = \min\{17,14,15,6\} = 6 > 4 = \max\{2,3,1,4\} = \max\{g_{min}(s) \mid 1 \leq s \leq S\}$. Hence, $n_{non-BG}' = 1$, i.e. at least one non-$BG$ relocation has to be made. Altogether the lower bound $n_m'$ for the number of moves to reach an empty layout from the layout in Fig. 3 amounts to $16+8+1 = 25$.

6. A tree search procedure for the CRP

The tree search procedure for the CRP is based on the following four elements:

1. Determining an initial solution $s_{GREEDY}$ with a greedy approach.
2. Performing a tree search to find better solutions than $s_{GREEDY}$.
3. Using compound moves, i.e. sequences of moves, instead of single moves for generating nodes of the search tree.
4. Using only a subset of promising single moves for composing compound moves.

The four elements are explained in detail in the following subsections.
6.1. Determining an initial solution with a greedy approach

An initial solution is calculated using the greedy approach listed in Fig. 4. Starting with the initial layout and an empty solution, in each step it is tried to find 1) a remove, 2) a BG relocation to a non-empty stack 3) a BG relocation to an empty stack or 4) a BB_FLG relocation. The next move is always determined according to the specified order. If for instance a remove is found, then moves of groups 2) to 4) with lower priority are not sought after for the current layout. As long as the layout is not empty, there is always a move available for at least one of the four priority groups: If no remove is possible, the bay must either be empty or all items with currently lowest item group are blocked. So if the bay is not empty, there is at least one blocking item that is placed at the top of a stack. Assuming that there are no BG relocations (group 2 and 3), it is known for certain that any relocation that can be performed must be of type BB, GB or GG. All blocking items are badly placed by definition, so a relocation of such an item can only be of type BB. Since there is at least one blocking item that can be relocated because it is at the top of a stack, there must be at least one possible BB_FLG relocation.

\[
\text{find_initial_solution} \quad (\text{in: } L \{\text{initial layout}\}, \text{out: } s \{\text{greedy solution}\})
\]

\[
\begin{align*}
s & := \emptyset; \quad \text{(initialise solution as empty list of moves)} \\
\text{while } L \text{ is not empty do} \\
\quad \text{Rm} & := \text{set of all removes applicable to } L; \\
\quad \text{if } \text{Rm} \neq \emptyset \text{ then} \\
\quad \quad \text{for each } rm \in \text{Rm} \text{ do} \\
\quad \quad \quad s & := s + rm; \quad \text{(append move rm to solution s)} \\
\quad \quad \quad L & := L \oplus rm; \quad \text{(apply move rm to layout L)} \\
\quad \quad \text{endfor;}
\quad \text{else} \\
\quad \quad \text{Bg} & := \text{set of all BG relocations to a non-empty stack in } L; \\
\quad \quad \text{if } \text{Bg} \neq \emptyset \text{ then} \\
\quad \quad \quad \text{determine } bg_{\text{Best}} \in \text{Bg} \text{ so that the item group difference between the moved item and the topmost item of the target stack is minimum;}
\quad \quad \quad s := s + bg_{\text{Best}}; \quad L := L \oplus bg_{\text{Best}};
\quad \text{else} \\
\quad \quad \text{Bge} & := \text{set of all BG relocations to an empty stack in } L; \\
\quad \quad \text{if } \text{Bge} \neq \emptyset \text{ then} \\
\quad \quad \quad \text{determine } bge_{\text{Best}} \in \text{Bge} \text{ so that the item group of the moved item is minimum;}
\quad \quad \quad s := s + bge_{\text{Best}}; \quad L := L \oplus bge_{\text{Best}};
\quad \text{else} \\
\quad \quad \text{Bb_flg} & := \text{set of all BB relocations that help to free an item with lowest item group in } L; \\
\quad \quad \text{determine } bb\_flg_{\text{Best}} \in \text{Bb_flg} \text{ so that the item group difference between the moved item and the item with the lowest item group in the target stack is maximum;}
\quad \quad \quad s := s + bb\_flg_{\text{Best}}; \quad L := L \oplus bb\_flg_{\text{Best}};
\quad \text{endif;}
\quad \text{endif;}
\text{endwhile;}
\text{end.}
\]

Fig. 4: Procedure find_initial_solution.

If multiple removes are possible then all of them are applied. As to relocations the (locally) best move from the set of possible relocations of the respective type and priority is specified by means of heuristic rules as shown in Fig. 4. If more than one BG relocation to a non-empty stack is possible, the relocation with minimum item group difference between the moved item and the topmost item of the target stack is chosen. The rational behind this rule is to build
“packed” stacks, where adjacent items in a stack have low item group differences. This generally allows for more 
BG relocations in the future, e.g., if items with item group 10 and 15 can be relocated to a stack with a lowest priority of 20, it is obviously better to first move the item with item group 15, since in the next step, the item with item group 10 can be placed on top of it. Similar to the rule for 
BG relocations to non-empty stacks, if more than one BB relocation to an empty stack is possible, the relocation for the item with lowest item group is chosen. If more than one 
BB relocation is possible, the relocation with maximum item group difference between the moved item and the topmost item of the target stack is chosen. This way, it is tried to avoid an early re-relocation of the moved item that would likely be again of type BB. As soon as a move is selected, it is appended to the solution and applied to the layout. This operation is repeated until the layout is empty.

This greedy approach can be considered a generalisation of the min-max-heuristic described by Caserta et al. (2009b) that is not limited to 
BG moves from a stack that contains an item with lowest item group. The result of the greedy approach is used as initial solution for the tree search, acting as the first upper bound and therefore constraining the solution space.

6.2. Performing the tree search

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 an empty 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 CRP instance is represented by a sequence 
s of compound moves. The number of all moves of 
s, designated by 
nm(s), results as the sum of all move numbers of all compound moves of 
s. The generation of compound moves will be described later.

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. 5. The following initializations are made before the procedure is called for the first time: current solution 
s := \emptyset, best solution \s* := \s_{\text{GREEDY}}, current layout \L := \text{initial layout \L_{\text{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 provided in a sorted list 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.
perform_compound_moves (in: s {current solution}, in: L {respective layout})

{abort the search where appropriate}
if \( n_m(s^*) = n_m'(L_{init}) \) or time limit exceeded then return; endif;

{abort search path if leaf node was reached; update best solution if necessary}
if L is empty then
  if \( n_m(s) < n_m(s^*) \) then \( s^* := s \); endif;
  return;
endif;

{determine compound moves for layout L}
\( Cm := \emptyset \); cmwork := \emptyset; {initialise list of compound moves and working compound move}
determine_compound_moves(L, cmwork, \( n_m(s^*) - n_m(s) - 1 \), \( 1, Cm \));

sort list \( Cm \)
- ascending by sum (number of moves in compound move + lower bound of moves for resulting layout when compound move is applied to L)
- descending by number of moves in compound move
- descending by clean supply for layout that results when compound move is applied to L

{reduce list length if necessary}
nCm := min(|Cm|, nSucc);

{perform compound moves alternatively}
for \( i := 1 \) to nCm do
  \( s' := s + Cm(i) \); {append \( Cm(i) \) to solution \( s \)}
  \( L' := L \oplus Cm(i) \); {apply \( Cm(i) \) to layout \( L \)}
  perform_compound_moves (\( s' \), \( L' \));
endfor;
end.

Fig. 5: Procedure perform_compound_moves.

Three steps are made to generate compound moves for a given layout \( L \). First an unsorted list of compound moves is generated by the recursive procedure determine_compound_moves that is explained below. Afterwards this list is sorted using three sorting criteria shown in Fig. 5 to shift the most promising compound moves at the head of the list. The first criterion is the expected number of moves necessary to reach a final layout, computed by adding the number of moves in the compound move and the lower bound of additional moves for the layout after the compound move is applied. The second criterion is the number of moves in the compound move. As third criterion, the clean supply of a layout is used. If a layout has a high clean supply value, it is likely that many \( BG \) relocations can be applied in the future, leading generally to a shorter solution than in the opposite case. After the list is sorted, it is reduced if necessary so that at most \( nSucc \) compound moves remain for being tested.

6.3. Generating compound moves

The procedure determine_compound_moves, shown in Fig. 6, generates compound moves of productive (single) moves (cf. Section 4). Moreover, only compound moves are provided that can be part of a new best solution.

The procedure takes the current layout \( L \) and the current (working) compound move \( cmwork \) as input data. The input argument \( cmStopValue \) is used to dynamically constrain the compound move length. In each recursive call, the \( cmStopValue \) for the recursive call is computed by multiplying the current value with the number of productive moves in the current layout. This leads to longer compound moves in cases when there are only few
productive moves and to shorter compound moves when there are many productive moves. The input argument \( \textit{maxMoves} \) gives the maximum number of (single) moves to an empty layout starting with current layout \( L \) that still ensures an improvement of the incumbent best solution. If \( \textit{determine\_compound\_moves} \) is called by procedure \( \textit{perform\_compound\_moves} \) then \( \textit{cmwork} \) is initialised as empty while \( \textit{cmStopValue} \) is set to 1 and \( \textit{maxMoves} \) is set to \( n_m(s^*) - n_m(s) - 1 \) (see Fig. 5). As transient (inout) parameter the list \( \textit{Cm} \) serves to collect compound moves that are generated by multiple calls of the procedure \( \textit{determine\_compound\_moves} \).

\[
\begin{align*}
determine\_compound\_moves & \quad \text{(current layout)} \\
& \quad \text{current compound move)} \\
& \quad \text{maximum number of moves to empty layout)} \\
& \quad \text{used to constrain compound moves in length)} \\
& \quad \text{list of generated compound moves}) \\

\end{align*}
\]

determine\_productive\_moves(L, Pm); \\
cmwork\_save := cmwork; {save cmwork} \\
for each productive move pm \in Pm do \\
L' := L \oplus pm; {apply move pm to layout L} \\
\{check whether complete solution was reached\} \\
if L' is empty then update best solution s* if necessary; continue; endif; \\
\{check whether extended compound move is useless\} \\
if \( n_m(L') > \textit{maxMoves} \) then continue; endif; \\
\{append move pm to passed current compound move\} \\
cmwork := cmwork\_save + pm; if \( \textit{cmStopValue} < \textit{cmStopThreshold} \) then \\
\{continue extending compound move\} \\
determine\_compound\_moves(L', cmwork, \textit{maxMoves} - 1, \textit{cmStopValue} \cdot |Pm|, Cm); \\
else \\
\{stop extending compound move, add it to list of compound moves\} \\
Cm := Cm \cup \{cmwork\}; \\
endif; \\
endfor; \\
end.
\]

Fig. 6: Procedure determine\_compound\_moves.

First, a set of productive moves is generated by the procedure \( \textit{determine\_productive\_moves} \) that will be explained below. Then, the passed working compound move \( \textit{cmwork} \) is extended by these productive moves alternatively. If the limit \( \textit{cmStopThreshold} \) is not yet reached, the compound move is further extended by a recursive call of the \( \textit{determine\_compound\_moves} \) procedure. Otherwise, the compound move is not longer extended and it is added to the list of compound moves. There are two special cases in the processing of productive moves. If the application of a productive move \( pm \) to the current layout \( L \) leads to an empty layout, then the partial solution \( s \) plus the current compound move \( \textit{cmwork} \) plus the productive move \( pm \) represent a new complete solution. In this case the best solution so far \( (s^*) \) is updated if necessary and the productive move is then ignored. A productive move \( pm \) is also skipped if the extension of the current compound move \( \textit{cmwork} \) by \( pm \) would result in a compound move that can no longer lead to a new best solution. This is the case if the lower bound \( n_m(L') \) of the number of moves necessary to empty the layout \( L' \) resulting by \( pm \) is greater than \( \textit{maxMoves} \).

6.4. Determining productive moves for compound moves

Limiting the set of moves offered to extend compound moves is vital since otherwise the generation of compound moves would quickly become impractical due to the exponentially
growing number of move sequences. The procedure \textit{determine_productive_moves}, shown in Fig. 7, generates a set of productive moves \( P_m \) that are applicable to the passed current layout \( L \) (cf. Section 4.2). The set \( P_m \) is determined in at most three steps:

1. If there are removes applicable to \( L \) the set of productive moves \( P_m \) is determined as the set of all feasible removes. Since remove operations have to be made to reach an empty layout and help to simplify the layout, they are always preferred.

2. Otherwise and if there are \( BG \) relocations that are applicable to \( L \) the set \( P_m \) is calculated as the set of all feasible \( BG \) relocations. Since \( BG \) relocations have to be made in order to enable removes in the future, they are favoured compared to other relocations.

3. If there are neither feasible removes nor \( BG \) relocations the set \( P_m \) is filled with \( FLG\_BB \) and \( GG \) relocations. \( FLG\_BB \) relocations are often indispensable to free a blocked item that needs to be removed next. The sorting of \( FLG\_BB \) moves helps to avoid multiple \( BB \) relocations of the same items. \( GG \) relocations can enable future \( BG \) moves and the sorting of \( GG \) moves strengthens this ability, since it prefers layouts with higher clean supply. To keep the search effort low the numbers of considered \( FLG\_BB \) and \( GG \) moves is limited by parameter \( maxFlg\_bb \) and \( maxGg \), respectively.

\begin{verbatim}
\textbf{determine_productive_moves} (in: L \{current layout\}, out: P_m \{set of productive moves\})

P_m := set of all removes applicable to L;
if P_m = ∅ then
    P_m := set of all moves of type BG applicable to L;
    if P_m = ∅ then
        Flg_bb := set of all moves of type FLG\_BB applicable to L;
        sort Flg_bb ascending by the difference between
group of the moved item and lowest group over all items in target stack
reduce Flg_bb to the first min(|Flg_bb|, maxFlg_bb) moves;
        Gg := set of all moves of type GG applicable to L;
        sort Gg ascending by the value of \((\text{diffBefore} – \text{diffAfter})\), where \text{diffBefore} is the item group
difference between the moved item \( m \) and the item in the slot directly below \( m \), and \text{diffAfter}
is the item group difference between \( m \) and the topmost item of the target stack.
reduce Gg to the first min(|Gg|, maxGg) moves;
        P_m := Flg_bb ∪ Gg;
    endif;
endif;
end.
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{procedure.png}
\caption{Procedure \textit{determine_productive_moves}.}
\end{figure}

7. Evaluation

To evaluate the performance of our tree search procedure, we compared it in numerical experiments 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 CRP instances used here are available for download at www.fernuni-hagen.de/WINF. The procedure was implemented in Java and tested on the JVM 1.6 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. Throughout the computational experiments, we have used the parameter setting \( nSucc = 10 \), \( cm\text{StopThreshold} = 150 \), \( maxFlg\_bb = 3 \), \( maxGg = 2 \) and a time limit of 60 seconds. Computation times are indicated in seconds throughout. As all valid solutions for a CRP instance have the same number of removes, it is sufficient to compare the number of relocations \( n_{rel} \) to evaluate the solution quality (cf. Section 5).
7.1. Benchmark against other CRP algorithms

Caserta et al. (2009a) provide various test cases for the CRP (see Table 2). Each test case consists of 40 instances with a constant number of stacks and items. Each item has its own group and the stacks are not constrained in height. Initially, all stacks are equally filled with items, i.e. the initial layouts have rectangular shape. The test cases are labelled CVS number of stacks – items per stack. For example, CVS 3-4 refers to 40 test instances with layouts that consist of 3 stacks, each containing 4 items.

Table 2: Test cases of Caserta et al. (2009a).

<table>
<thead>
<tr>
<th>Test case</th>
<th>No. of instances</th>
<th>No. of stacks</th>
<th>Max. stack height</th>
<th>No. of item groups</th>
<th>No. of items</th>
<th>No. of badly placed items</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVS 3-3</td>
<td>40</td>
<td>3</td>
<td>-</td>
<td>9</td>
<td>9</td>
<td>3.70</td>
</tr>
<tr>
<td>CVS 3-4</td>
<td>40</td>
<td>4</td>
<td>-</td>
<td>16</td>
<td>16</td>
<td>4.78</td>
</tr>
<tr>
<td>CVS 3-5</td>
<td>40</td>
<td>5</td>
<td>-</td>
<td>25</td>
<td>25</td>
<td>5.78</td>
</tr>
<tr>
<td>CVS 3-6</td>
<td>40</td>
<td>6</td>
<td>-</td>
<td>36</td>
<td>36</td>
<td>7.20</td>
</tr>
<tr>
<td>CVS 3-7</td>
<td>40</td>
<td>7</td>
<td>-</td>
<td>21</td>
<td>21</td>
<td>8.18</td>
</tr>
<tr>
<td>CVS 3-8</td>
<td>40</td>
<td>8</td>
<td>-</td>
<td>24</td>
<td>24</td>
<td>9.30</td>
</tr>
<tr>
<td>CVS 4-4</td>
<td>40</td>
<td>4</td>
<td>-</td>
<td>16</td>
<td>16</td>
<td>7.30</td>
</tr>
<tr>
<td>CVS 4-5</td>
<td>40</td>
<td>5</td>
<td>-</td>
<td>20</td>
<td>20</td>
<td>10.18</td>
</tr>
<tr>
<td>CVS 4-6</td>
<td>40</td>
<td>6</td>
<td>-</td>
<td>24</td>
<td>24</td>
<td>11.08</td>
</tr>
<tr>
<td>CVS 4-7</td>
<td>40</td>
<td>7</td>
<td>-</td>
<td>28</td>
<td>28</td>
<td>13.40</td>
</tr>
<tr>
<td>CVS 5-4</td>
<td>40</td>
<td>4</td>
<td>-</td>
<td>20</td>
<td>20</td>
<td>10.60</td>
</tr>
<tr>
<td>CVS 5-5</td>
<td>40</td>
<td>5</td>
<td>-</td>
<td>25</td>
<td>25</td>
<td>13.15</td>
</tr>
<tr>
<td>CVS 5-6</td>
<td>40</td>
<td>6</td>
<td>-</td>
<td>30</td>
<td>30</td>
<td>16.80</td>
</tr>
<tr>
<td>CVS 5-7</td>
<td>40</td>
<td>7</td>
<td>-</td>
<td>35</td>
<td>35</td>
<td>19.13</td>
</tr>
<tr>
<td>CVS 5-8</td>
<td>40</td>
<td>8</td>
<td>-</td>
<td>40</td>
<td>40</td>
<td>22.20</td>
</tr>
<tr>
<td>CVS 5-9</td>
<td>40</td>
<td>9</td>
<td>-</td>
<td>45</td>
<td>45</td>
<td>25.10</td>
</tr>
<tr>
<td>CVS 5-10</td>
<td>40</td>
<td>10</td>
<td>-</td>
<td>50</td>
<td>50</td>
<td>27.73</td>
</tr>
<tr>
<td>CVS 6-6</td>
<td>40</td>
<td>6</td>
<td>-</td>
<td>36</td>
<td>36</td>
<td>21.45</td>
</tr>
<tr>
<td>CVS 6-10</td>
<td>40</td>
<td>10</td>
<td>-</td>
<td>60</td>
<td>60</td>
<td>36.08</td>
</tr>
<tr>
<td>CVS 10-6</td>
<td>40</td>
<td>6</td>
<td>-</td>
<td>60</td>
<td>60</td>
<td>42.25</td>
</tr>
<tr>
<td>CVS 10-10</td>
<td>40</td>
<td>10</td>
<td>-</td>
<td>100</td>
<td>100</td>
<td>70.53</td>
</tr>
</tbody>
</table>

Caserta et al. (2009a) present computational results for these test cases calculated with their corridor-based DP algorithm and with the heuristic of Kim and Hong (2006). For the algorithms proposed by Caserta et al. (2009b) and by Caserta and Voß (2009c/d), results were only published for the largest test cases CVS 6-6 to CVS 10-10. Since the authors were so kind and provided us the source code of their algorithm described in Caserta et al. (2009b), we were able to compute the results for the remaining test cases. For all algorithms, a time limit of 60 seconds was defined. Table 3 shows the results for the four before mentioned algorithms and for our new tree search heuristic. The second row characterizes the PC that was used for testing the respective algorithm; unfortunately, no CPU frequency is given in the article by Caserta et. al (2009a). In the “LB gap” column, the average absolute deviation (in relocations) between the solution of the tree search heuristic and the lower bound from Section 5 over all instances of a test case is listed.

On average over all relevant test cases, the tree search heuristic found solutions with 47.3%, 9.6%, 5.4% and 0.1% less relocations than the algorithms proposed by Kim and Hong (2006), Caserta et al. (2009a), Caserta et al. (2009b) and Caserta and Voß (2009c/d), respectively. The test cases with a stack height up to 5 items, which is a common stack height in container yards in practice, are solved with a reasonable low absolute lower bound gap: the average difference between the number of relocations in the solution of the tree search heuristic and in the optimal solution, respectively, is always less than 6. For most of the test cases, the
solutions are computed in less than 1 second. Only the very large test cases CVS 6-10, CVS 10-6 and CVS 10-10 need considerably longer calculation times. In total, 69 of the 840 problem instances could be solved to optimality.

Table 3: Computational results for the test cases of Caserta et al. (2009a).

<table>
<thead>
<tr>
<th>Authors’ calculations</th>
<th>Authors’ calculations</th>
<th>Authors’ calculations</th>
<th>Authors’ calculations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium III, 128 MB RAM</td>
<td>Pentium IV, 512 MB RAM</td>
<td>Core 2 Duo, 2 GHz, 2 GB RAM</td>
<td>Pentium IV, 512 MB RAM</td>
</tr>
<tr>
<td>Test case nrel t (s) nrel t (s) nrel t (s) nrel t (s)</td>
<td>Test case nrel t (s) nrel t (s) nrel t (s) nrel t (s)</td>
<td>Test case nrel t (s) nrel t (s) nrel t (s) nrel t (s)</td>
<td>Test case nrel t (s) nrel t (s) nrel t (s) nrel t (s)</td>
</tr>
<tr>
<td>CVS 3-3</td>
<td>7.10 0.1</td>
<td>5.40 0.1</td>
<td>5.05 60.0</td>
</tr>
<tr>
<td>CVS 3-4</td>
<td>10.70 0.1</td>
<td>6.50 0.1</td>
<td>6.20 60.0</td>
</tr>
<tr>
<td>CVS 3-5</td>
<td>14.50 0.1</td>
<td>7.30 0.1</td>
<td>7.13 60.0</td>
</tr>
<tr>
<td>CVS 3-6</td>
<td>18.10 0.1</td>
<td>7.90 0.2</td>
<td>8.58 60.0</td>
</tr>
<tr>
<td>CVS 3-7</td>
<td>20.10 0.1</td>
<td>8.60 0.1</td>
<td>9.35 60.0</td>
</tr>
<tr>
<td>CVS 3-8</td>
<td>26.00 0.1</td>
<td>10.50 0.2</td>
<td>10.83 60.0</td>
</tr>
<tr>
<td>CVS 3-9</td>
<td>16.00 0.1</td>
<td>9.90 0.2</td>
<td>10.30 60.0</td>
</tr>
<tr>
<td>CVS 3-10</td>
<td>24.00 0.1</td>
<td>16.50 0.5</td>
<td>13.43 60.0</td>
</tr>
<tr>
<td>CVS 4-4</td>
<td>23.40 0.1</td>
<td>19.80 0.5</td>
<td>14.35 60.0</td>
</tr>
<tr>
<td>CVS 4-5</td>
<td>32.20 0.1</td>
<td>21.50 0.5</td>
<td>16.63 60.0</td>
</tr>
<tr>
<td>CVS 4-6</td>
<td>23.70 0.1</td>
<td>16.60 0.5</td>
<td>15.73 60.0</td>
</tr>
<tr>
<td>CVS 4-7</td>
<td>37.50 0.1</td>
<td>18.80 0.5</td>
<td>19.75 60.0</td>
</tr>
<tr>
<td>CVS 4-8</td>
<td>45.50 0.1</td>
<td>22.10 0.8</td>
<td>23.05 60.0</td>
</tr>
<tr>
<td>CVS 4-9</td>
<td>52.30 0.1</td>
<td>25.80 0.8</td>
<td>24.90 60.0</td>
</tr>
<tr>
<td>CVS 4-10</td>
<td>61.80 0.1</td>
<td>30.10 0.8</td>
<td>29.05 60.0</td>
</tr>
<tr>
<td>CVS 5-4</td>
<td>72.40 0.1</td>
<td>33.10 1.4</td>
<td>32.00 60.0</td>
</tr>
<tr>
<td>CVS 5-5</td>
<td>80.90 0.1</td>
<td>36.40 1.9</td>
<td>34.78 60.0</td>
</tr>
<tr>
<td>CVS 5-6</td>
<td>37.30 0.1</td>
<td>32.40 1.7</td>
<td>32.60 60.0</td>
</tr>
<tr>
<td>CVS 5-7</td>
<td>75.10 0.1</td>
<td>49.50 2.0</td>
<td>47.75 60.0</td>
</tr>
<tr>
<td>CVS 5-8</td>
<td>141.60 0.1</td>
<td>102.00 4.7</td>
<td>83.58 60.0</td>
</tr>
<tr>
<td>CVS 5-9</td>
<td>178.60 0.2</td>
<td>128.30 6.3</td>
<td>121.33 60.0</td>
</tr>
<tr>
<td>Mean</td>
<td>47.67 0.1</td>
<td>29.00 1.1</td>
<td>26.97 60.0</td>
</tr>
</tbody>
</table>

With the use case of the container yard in mind it can be criticized that the CVS test cases do not impose a reasonable maximum stack height. Since the stack height in a container yard is constrained by the crane height, it can be feared that many solutions generated by the algorithms are infeasible. To underline the practicality of our algorithm, we recalculated the CVS test cases with a maximum stack height that is assumed to be 2 slots higher than the height of the stacks in the initial layout. Taking the CVS 3-4 test case for example, the maximum stack height is set to 6. On average, the number of relocations was only 3.7% higher than for the test cases without stack height restriction while the computation times remained constant. For future comparisons the mean number of relocations per test case is listed in the rightmost column “nrel-msh” of Table 3.

7.2. Computational experiments with new CRP test cases

Bortfeldt and Forster (2010) introduced new test cases for the CPMP, which can be used just like that as CRP test cases and are characterised by the following data:

- Number of stacks: The layouts consist of either 16 or 20 stacks.
- Maximum stack height: The stacks either have a maximum height of 5 or 8 items.
- Number of item groups: The number of different item groups in the layout was determined using a quota, defining how many items in the layout have different item group values. In
a layout with 100 items, a quota of 50% means that there are 50 different item groups. For the new problem instances, item group quotas of 20% and 40% were used.

- Number of items: The number of items in the layout is determined relative to the layout dimensions. Either 60% or 80% of a layout’s slots are initially filled with items.

- Number of badly placed items: Either 60% or 75% of the items are initially badly placed.

In total, 32 different test cases were defined as indicated in the first six columns of Table 4. For every test case 20 different instances were generated at random. The resulting 640 CRP instances were solved using the proposed tree search procedure with a parameter setting as defined above and a time limit of 60 seconds. The results are also shown in Table 4.

Table 4: Definition of and tree search heuristic results for 32 new random-based CRP test cases.

<table>
<thead>
<tr>
<th>Test case</th>
<th>No. of stacks</th>
<th>Max. stack height</th>
<th>No. of items</th>
<th>No. of item groups</th>
<th>No. of badly placed items</th>
<th>Mean n_{rel}</th>
<th>Mean time (s)</th>
<th>Mean LB gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF1</td>
<td>16</td>
<td>5</td>
<td>48</td>
<td>10</td>
<td>29</td>
<td>21.80</td>
<td>&lt; 0.1</td>
<td>0.00</td>
</tr>
<tr>
<td>BF2</td>
<td>16</td>
<td>5</td>
<td>48</td>
<td>10</td>
<td>36</td>
<td>25.80</td>
<td>&lt; 0.1</td>
<td>0.00</td>
</tr>
<tr>
<td>BF3</td>
<td>16</td>
<td>5</td>
<td>48</td>
<td>20</td>
<td>29</td>
<td>22.55</td>
<td>&lt; 0.1</td>
<td>0.00</td>
</tr>
<tr>
<td>BF4</td>
<td>16</td>
<td>5</td>
<td>48</td>
<td>20</td>
<td>36</td>
<td>26.80</td>
<td>&lt; 0.1</td>
<td>0.00</td>
</tr>
<tr>
<td>BF5</td>
<td>16</td>
<td>5</td>
<td>64</td>
<td>13</td>
<td>39</td>
<td>29.90</td>
<td>&lt; 0.1</td>
<td>0.25</td>
</tr>
<tr>
<td>BF6</td>
<td>16</td>
<td>5</td>
<td>64</td>
<td>13</td>
<td>48</td>
<td>35.80</td>
<td>&lt; 0.1</td>
<td>0.00</td>
</tr>
<tr>
<td>BF7</td>
<td>16</td>
<td>5</td>
<td>64</td>
<td>26</td>
<td>39</td>
<td>30.90</td>
<td>&lt; 0.1</td>
<td>0.80</td>
</tr>
<tr>
<td>BF8</td>
<td>16</td>
<td>5</td>
<td>64</td>
<td>26</td>
<td>48</td>
<td>36.80</td>
<td>&lt; 0.1</td>
<td>0.00</td>
</tr>
<tr>
<td>BF9</td>
<td>16</td>
<td>8</td>
<td>77</td>
<td>16</td>
<td>47</td>
<td>39.45</td>
<td>4.8</td>
<td>1.45</td>
</tr>
<tr>
<td>BF10</td>
<td>16</td>
<td>8</td>
<td>77</td>
<td>16</td>
<td>58</td>
<td>45.55</td>
<td>7.4</td>
<td>0.50</td>
</tr>
<tr>
<td>BF11</td>
<td>16</td>
<td>8</td>
<td>77</td>
<td>31</td>
<td>47</td>
<td>40.60</td>
<td>12.3</td>
<td>2.45</td>
</tr>
<tr>
<td>BF12</td>
<td>16</td>
<td>8</td>
<td>103</td>
<td>31</td>
<td>58</td>
<td>46.00</td>
<td>4.7</td>
<td>0.10</td>
</tr>
<tr>
<td>BF13</td>
<td>16</td>
<td>8</td>
<td>103</td>
<td>21</td>
<td>62</td>
<td>58.30</td>
<td>23.0</td>
<td>7.90</td>
</tr>
<tr>
<td>BF14</td>
<td>16</td>
<td>8</td>
<td>103</td>
<td>21</td>
<td>78</td>
<td>68.30</td>
<td>40.1</td>
<td>6.15</td>
</tr>
<tr>
<td>BF15</td>
<td>16</td>
<td>8</td>
<td>103</td>
<td>42</td>
<td>62</td>
<td>60.70</td>
<td>43.9</td>
<td>9.55</td>
</tr>
<tr>
<td>BF16</td>
<td>16</td>
<td>8</td>
<td>103</td>
<td>42</td>
<td>78</td>
<td>72.25</td>
<td>53.8</td>
<td>9.20</td>
</tr>
<tr>
<td>BF17</td>
<td>20</td>
<td>5</td>
<td>60</td>
<td>12</td>
<td>36</td>
<td>27.25</td>
<td>&lt; 1.0</td>
<td>0.00</td>
</tr>
<tr>
<td>BF18</td>
<td>20</td>
<td>5</td>
<td>60</td>
<td>12</td>
<td>45</td>
<td>31.70</td>
<td>&lt; 1.0</td>
<td>0.00</td>
</tr>
<tr>
<td>BF19</td>
<td>20</td>
<td>5</td>
<td>60</td>
<td>24</td>
<td>36</td>
<td>27.60</td>
<td>&lt; 1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>BF20</td>
<td>20</td>
<td>5</td>
<td>60</td>
<td>24</td>
<td>45</td>
<td>34.35</td>
<td>&lt; 1.0</td>
<td>0.00</td>
</tr>
<tr>
<td>BF21</td>
<td>20</td>
<td>5</td>
<td>80</td>
<td>16</td>
<td>48</td>
<td>37.70</td>
<td>5.5</td>
<td>0.45</td>
</tr>
<tr>
<td>BF22</td>
<td>20</td>
<td>5</td>
<td>80</td>
<td>16</td>
<td>60</td>
<td>44.55</td>
<td>&lt; 1.0</td>
<td>0.00</td>
</tr>
<tr>
<td>BF23</td>
<td>20</td>
<td>5</td>
<td>80</td>
<td>32</td>
<td>48</td>
<td>38.05</td>
<td>6.3</td>
<td>0.75</td>
</tr>
<tr>
<td>BF24</td>
<td>20</td>
<td>5</td>
<td>80</td>
<td>32</td>
<td>60</td>
<td>43.95</td>
<td>&lt; 1.0</td>
<td>0.00</td>
</tr>
<tr>
<td>BF25</td>
<td>20</td>
<td>8</td>
<td>96</td>
<td>20</td>
<td>58</td>
<td>47.45</td>
<td>11.6</td>
<td>1.30</td>
</tr>
<tr>
<td>BF26</td>
<td>20</td>
<td>8</td>
<td>96</td>
<td>20</td>
<td>72</td>
<td>55.85</td>
<td>3.2</td>
<td>0.20</td>
</tr>
<tr>
<td>BF27</td>
<td>20</td>
<td>8</td>
<td>96</td>
<td>39</td>
<td>58</td>
<td>48.75</td>
<td>26.5</td>
<td>2.05</td>
</tr>
<tr>
<td>BF28</td>
<td>20</td>
<td>8</td>
<td>96</td>
<td>39</td>
<td>72</td>
<td>57.40</td>
<td>18.0</td>
<td>0.40</td>
</tr>
<tr>
<td>BF29</td>
<td>20</td>
<td>8</td>
<td>128</td>
<td>26</td>
<td>77</td>
<td>73.05</td>
<td>60.0</td>
<td>9.95</td>
</tr>
<tr>
<td>BF30</td>
<td>20</td>
<td>8</td>
<td>128</td>
<td>26</td>
<td>96</td>
<td>83.90</td>
<td>57.0</td>
<td>7.75</td>
</tr>
<tr>
<td>BF31</td>
<td>20</td>
<td>8</td>
<td>128</td>
<td>52</td>
<td>77</td>
<td>74.95</td>
<td>56.9</td>
<td>10.07</td>
</tr>
<tr>
<td>BF32</td>
<td>20</td>
<td>8</td>
<td>128</td>
<td>52</td>
<td>96</td>
<td>86.25</td>
<td>58.9</td>
<td>8.85</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test case</th>
<th>No. of stacks</th>
<th>Max. stack height</th>
<th>No. of items</th>
<th>No. of item groups</th>
<th>No. of badly placed items</th>
<th>Mean n_{rel}</th>
<th>Mean time (s)</th>
<th>Mean LB gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>46.07</td>
<td>15.9</td>
<td>2.52</td>
</tr>
</tbody>
</table>

For each test case the mean number of relocations per solution, the mean total computation time and the mean absolute lower bound gap are presented in columns 7 to 9. The results show that near-optimal solutions have been achieved for layouts in which 60% of the slots are occupied by items, even for large layout dimensions. If 80% of all slots are filled, the solutions for layouts with a maximum stack height of 5 are still very close to the calculated lower bounds. Only for layouts with a maximum stack height of 8, the numbers of relocations in the solutions differ considerably from the lower bounds. In total, 366 of the 640 problem instances could be solved to optimality.
7.3 Evaluation of individual components of the tree search procedure

In further calculations we studied the impact of different components and features of the tree search 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) The term $n_{non-BG}'$ of the lower bound proposed in Section 5 is omitted.

(V2) Only compound moves consisting of one single move are applied, i.e. each compound move is cut after the first move (cf. Section 6.3).

(V3) The list $C_m$ of compound moves of a tree search instance is not sorted after completion (cf. Fig. 5).

(V4) No special selection scheme (as introduced in Section 6.4) is used to select the next move for a compound move. Moreover, the set of moves from which an additional move for a compound move is selected is not limited to the productive moves. Instead, a move is chosen randomly from all possible moves.

The original procedure and the derived simplified variants V1 to V4 were applied to the CVS test cases with stack height limit (cf. Table 2). All instances were calculated using a time limit of 60 seconds. Table 5 presents the results of the calculations using the procedure variants V0 to V4. In column 2, the mean additional number of relocations – relative to the relocations in the solutions of variant V0 – is shown. In column 3, the mean total computation time – relative to the time needed to solve an instance with variant V0 – is indicated.

The results of Table 5 may be summarized as follows:

- V1: The additional term $n_{non-BG}'$ of the lower bound as described in section 5 has a positive though very limited effect on the solution quality and the calculation time.
- V2: Using compound moves for branching drastically improves calculation time.
- V3: Sorting compound moves has a positive impact on both solution quality and calculation time.
- V4: A random-based selection of moves instead of the move selection scheme proposed in Section 6.4 leads to a very time consuming search for suitable compound moves. Taking into account the time limit, the random approach was not able to improve the greedy solution once.

All in all we can state that all examined components and features contribute to the effectiveness and efficiency of the tree search procedure.

<table>
<thead>
<tr>
<th>Procedure variant</th>
<th>Average relative additional no. of relocations (%)</th>
<th>Average relative total computation time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>V1</td>
<td>0.1</td>
<td>100.3</td>
</tr>
<tr>
<td>V2</td>
<td>2.0</td>
<td>259.6</td>
</tr>
<tr>
<td>V3</td>
<td>7.4</td>
<td>107.3</td>
</tr>
<tr>
<td>V4</td>
<td>13.8</td>
<td>&gt; 10000.0</td>
</tr>
</tbody>
</table>

8. Conclusions

Effective solution approaches for the container relocation problem are of high economic relevance since the container yard is often the bottleneck of a container terminal. In this article, we propose a heuristic tree search procedure for the CRP that is based on a natural classification of possible moves and applies a suitable branching scheme using move
sequences of promising single moves. The procedure was tested against all former solution approaches for the CRP and it proved to be a competitive solution method that can be applied to large real-world CRP instances. The performance of individual procedure components was examined separately. 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 also a fruitful approach for solving related problems like the CRP with multiple bays recently described by Lee and Lee (2010). This remains a subject of our future research.

References


