Noname manuscript No.
(will be inserted by the editor)

A Multi-Heuristic Approach for Solving the Pre-Marshalling Problem

Raka Jovanovic · Milan Tuba · Stefan Voß

Received: date / Accepted: date

Abstract Minimizing the number of reshuffling operations at maritime container terminals incorporates the Pre-Marshalling Problem (PMP) as an important problem. Based on an analysis of existing solution approaches we develop new heuristics utilizing specific properties of problem instances of the PMP. We show that the heuristic performance is highly dependent on these properties. We introduce a new method that exploits a greedy heuristic of four stages, where for each of these stages several different heuristics may be applied. Instead of using randomization to improve the performance of the heuristic, we repetitively generate a number of solutions by using a combination of different heuristics for each stage. In doing so, only a small number of solutions is generated for which we intend that they do not have undesirable properties, contrary to the case when simple randomization is used. Our experiments show that such a deterministic algorithm significantly outperforms the original nondeterministic method. The improvement is twofold, both in the quality of found solutions, and in the computational effort.

Keywords Pre-marshalling · Logistics · Container terminal · Heuristics

Raka Jovanovic
Qatar Environment and Energy Research Institute (QEERI), PO Box 5825, Doha, Qatar
E-mail: rjovanovic@qf.or.qa

Milan Tuba
Megatrend University Belgrade, Faculty of Computer Science, Bulevar umetnosti 29, N. Belgrade, Serbia

Stefan Voß
Institute of Information Systems, University of Hamburg, Von-Melle-Park 5, 20146 Hamburg, Germany, and Escuela de Ingeniería Industrial, Pontificia Universidad Católica de Valparaíso, Chile
E-mail: stefan.voss@uni-hamburg.de

1 Introduction

Maritime shipping has gained considerable importance throughout the last decades. As a major part of maritime shipping we see the almost unprecedented increase of using containers as a transport means for freight movement with container terminals literally serving as open systems of material flow (Steenken et al, 2004; Tran and Haasis, 2014). Within container terminals the time that is needed for loading containers to transport vehicles and vessels is of utmost importance.

Terminals usually have to operate with a limited amount of storage space. Because of this, block stacking is used to increase the space utilization. More precisely, containers are simply stored over each other in several stacks. A certain number of stacks (vertical storage of containers one over the other) located side by side are referred to as bay. Furthermore, the notion of tiers refers to the horizontal storage of containers within one line or row within a bay. A problem may arise as only the top container can be retrieved from each stack. While containers usually need to be loaded to transport vehicles in a certain order, in general, not only will it be necessary to move containers from the stacks to the transport vehicles but they also have to be relocated within container bays (reshuffling) to make retrieval in the specified order possible. The order is reflected by priorities, where a small priority value means that a container must be retrieved earlier than one with a larger priority value. Regarding container terminal operations efficiency is related to several key performance indicators. For instance, the loading process of a vessel may be most efficient if the number of block movements is minimized (where the notion of a block is used interchangeably with that of a container). The practical problem of minimizing the number of block movements has been formalized in several forms like the Blocks Relocation Problem (BRP), the Re-Marshalling Problem (RMP), i.e. intra-block marshalling, and the Pre-Marshalling Problem (PMP) (Caserta et al, 2011a).

The overall goal of these problems may be described as minimizing the number of block movements which eventually refers to minimizing the number of necessary reshuffling operations. With that goal in mind it seems reasonable to distinguish the different types of problems just mentioned more comprehensively. That is, for the BRP containers are moved onto a vessel and reordering (or reshuffling) is done while this process is lasting. On the other hand, if we are concerned with "housekeeping" then containers are reordered within a single bay (called PMP) or among several bays (in case of RMP) without interfering with moves towards the vessel. The latter moves are only performed if the housekeeping has been finished.

In this paper we are concerned with the PMP. The goal of the PMP is to reorder the blocks within a container bay to have all the blocks well located while minimizing the number of moves (or reshuffling operations, respectively). We use the term "well located" for a block that satisfies the following two conditions. First, there are no blocks of smaller priority value located below it. Second, all blocks located below it are also well located. We should note in

passing that the notion of priority values is common understanding in most papers dealing with the PMP (see the references in the next section). A priority value can be thought of as the number within a sequence according to which containers need to be moved onto a vessel. That is, if all numbers are unique, the container with priority 1 has to go first, then the container with priority value 2 and so on. In case that priority values are not unique this characterizes the case that several containers are treated "equally," i.e., they have the same destination, are of the same weight class and share all other important characteristics used for distinction. For example, if a number of containers are of same weight class and need to go onto the same vessel with no important distinction regarding other characteristics, they gain the same priority values. Due to the possibly large time differences in containers leaving the yard the maritime shipping literature seems to prefer to use priority values rather than due dates as the latter might not reflect other characteristics (like weight classes). Nevertheless, one may interpret the priority values as due date values where a container with a smaller due date value needs to be moved onto a vessel earlier than one with a larger due date value. Different from the notion of due dates in, e.g., machine scheduling, due dates in our setting refer to priorities within some (handling) sequence. That is, if one does not understand due date values or due dates as specific times then this notion may perfectly reflect the situation envisaged in the PMP.

In this paper we focus on solving the PMP using a deterministic greedy algorithm. In our approach we adapt the algorithm described in Expósito-Izquierdo et al (2012). In the original work, a greedy algorithm is developed using a heuristic, which incorporates a certain level of randomization to improve its performance.

Usually greedy heuristics for the PMP give results that are of lower quality compared to more complex methods like those following the tree search or the corridor method's paradigm. The advantage of greedy algorithms is that they, in most cases, have a much lower computational cost than other more complex approaches. The main problem with such methods is that they usually create only one solution which is frequently just a "relatively" good one. One possible way to avoid this problem is adding some type of randomization or even some learning process like the ant colony optimization. In the case when simple randomization is used, although an improvement is achieved, it often has a significant increase in the computational cost. More precisely, the randomized algorithm needs to generate a large number of solutions, and the computation cost for each of them is equivalent to the calculation time of the basic greedy algorithm. This is due to the fact that, in general, the only difference between executions is the seeds for the random number generator. The problem is that by using simple randomization many of these solutions simply do not have good (desired) properties.

In our work, we attempt to avoid the use of randomization and try to generate only solutions that satisfy some desirable properties. More precisely, we exploit the fact that the original algorithm has several stages, and each one of them has a separate heuristic function. By running the greedy algorithm

with different combinations of such heuristics a large number of solutions is generated. A heuristic can, in a simplified fashion, loosely be defined as a function that gives us a measure for good properties of an object. The problem is that this function just corresponds to a specific "guess" about what good properties are. Because of this, for many problems several competing heuristics are developed, that are suitable for different types of problem instances; see, e.g., well known approaches for the simple assembly line balancing problem (Scholl and Voß, 1996). A common practice is to use several heuristics and choose the best found solution. This idea is well known, though, in the case of the PMP it can be exploited extensively as we can combine the heuristics for different stages of the algorithm.

In the implementation of this method special care is necessary to make sure feasible solutions are generated. In the case of the PMP the algorithm would frequently bring the container bay into a state from which it is not possible to generate adequate solutions by simply applying the heuristic. Note that no such issue exists for similar greedy algorithms implemented for the BRP. This type of deadlock would happen in cases when there is a high level of occupancy of the bay. The standard approach to resolving this is to use a certain level of backtracking. The addition of backtracking to the basic greedy algorithm results in a significant increase of implementation complexity. Because of this in many cases, it seems a better choice to directly implement some more advanced methods like the use of a tree search.

To resolve this, we add a basic look-ahead mechanism that would find feasible solutions in all cases when they are possible, from the initial state of the bay. The use of such a mechanism, and certain combinations of heuristics, have in some cases resulted in the addition of unnecessary relocations (or reshuffling operations). Because of this, a simple correction stage is added at the end of the algorithm to improve the results. The performed computational experiments show that the proposed method manages to outperform the original work by Expósito-Izquierdo et al (2012) in both computational time and the quality of found solutions for almost all the test data sets.

The article is organized as follows. In the next section we give the problem formulation and a brief overview of published work. The following section presents an outline of the original method by Expósito-Izquierdo et al (2012) and gives a detailed specification of the heuristics used at different stages of the algorithm. Section 4 gives details of the proposed multi-heuristic method. Section 5 is dedicated to the computational experiments.

2 The Pre-Marshalling Problem

The PMP is defined using some simplified assumptions which are consistently used in literature (Kim and Hong, 2006), in the following way:

- All blocks (containers) are of the same size.
- The container bay will be viewed as a two dimensional stacking area, with W stacks, for which a maximal height (number of tiers) H is given.

- The initial configuration of the container bay is known (and consists of a set C of containers).
- Only blocks from the top of a stack can be accessed.
- Blocks can only be placed either on top of another block, or on the ground (tier 0).
- Each container has a due date value greater or equal to 1 (which is not necessarily unique).
- A container is well located if no container with a larger due date value is on top of it.
- A well located container can only be above other well located containers, and has a smaller due date value than all of those below it (or the same due date value as the one immediately below it).
- The goal is to have all of the containers in the bay well located. (For the final bay layout this means that containers can be retrieved according to increasing due date values without any further relocations.)

The problem is to minimize the number of moves needed to create a container bay with only well located containers.

It has been shown that this problem is NP-hard (Caserta et al, 2011a). In an early paper (Kim, 1997) influencing this area of research, various stack configurations and their influence on the expected number of rehandles are investigated in a scenario of loading import containers onto outside trucks with a single transfer crane. For easy estimation regression equations are proposed.

There are quite a few papers proposing solution approaches for solving the PMP. This incorporates using a tree search algorithm (Bortfeldt and Forster, 2012), integer programming (Lee and Lee, 2010) or the corridor method paradigm (Caserta and Voß, 2009). Algorithms with direct heuristics have been developed by Huang and Lin (2012); a neighborhood search heuristic can be found in Lee and Chao (2009). Another heuristic is the one by Expósito-Izquierdo et al (2012). Moreover, this paper also incorporates a simple A*-algorithm which was later on improved and appended by some symmetry breaking rules by Tierney et al (2013). Some comments on logical observations leading to a lower bound are provided in Voß (2012). Some of these ideas are also incorporated in the tree search algorithm of Bortfeldt and Forster (2012). A constraint programming approach together with a more general problem description allowing for priority ranges rather than priority values has been proposed by Rendl and Prandtstetter (2013). Note that the PMP is also closely related to blocks world planning; see, e.g., Gupta and Nau (1992). A more comprehensive survey on the PMP and related problems is provided in Caserta et al (2011a) and more recently by Lehnfeld and Knust (2014).

Based on the analysis of the existing approaches there is still the necessity to have fast heuristics with good quality. The existing approaches are either time-sensitive or their quality seems not yet fully satisfying. This is where we attempt to provide new insights.

3 The Basic Heuristic Scheme

In this section we describe and extend the heuristic of Expósito-Izquierdo et al (2012). The general idea of this algorithm is to well locate containers one by one, starting with the containers with largest due date value, say p. Note that according to the problem definition a container with a largest due date value cannot be on top of a container with a smaller due date value as it would otherwise hinder this container from being well located. The following pseudo-code gives an outline of the method.

```
i = p
while (i ≠ 0) do
  A_i=Set of not well located containers with due date value i
  while (A_i ≠ ∅) do
    Select container c ∈ A_i
    Select a destination stack s* for c
    Well locate container c from its current position to stack s*
    A_i = A_i \ {c}
    Fill destination stack;
  end while
  i = i − 1
end while
```

This type of algorithm can be divided into four stages which will be detailed in the following subsections.

1. Select a container to be well located.
2. Select a destination stack.
3. Relocate the necessary containers to make the well locating possible.
4. Filling.

Compared to the original algorithm we introduce a new heuristic where we select which container will be well located. In this way the containers will not necessarily be well located in the descending order of due date values.

In the case of the original work (Expósito-Izquierdo et al, 2012), the selection of the next target container has been done using a random selection between the blocks with the maximal due date value. In the following subsections we shall analyze the heuristics used at each stage of the algorithm.

3.1 Selecting a Destination Stack

To ease exposition, we first provide some details of how a destination stack $\mathbf{s}^*$ is selected for block c that we wish to well locate. The goal in this stage is to well locate c in the smallest number of container relocations. The number of relocations is depending on two factors. First the number of containers necessary to be relocated to access block c, more precisely the blocks above c

have to be removed. We define $g(c, \mathbf{s})$ as the number of blocks above c in stack $\mathbf{s}$.

The second factor is how many blocks need to be removed from a potential destination stack $\mathbf{s}^*$ to make well locating of block c possible. Practically, this is the number of blocks that are above a well located block c_a with a larger or the same due date value than c. In this way we define $f(c, \mathbf{s}^*)$. By placing c above c_a, c will be retrieved from the final bay layout before c_a.

For an empty stack s^*, we will have $f(c, \mathbf{s}^*) = 0$ as every block can be well located once it is put onto the ground. We give a graphic representation of functions f and g in Figure 1. As presented in Expósito-Izquierdo et al (2012), a heuristic function w can be presented in the following form.

$$w(c, \mathbf{s}^*) = \begin{cases} f(c, \mathbf{s}^*) + g(c, \mathbf{s}) + 1 & \mathbf{s} \neq \mathbf{s}^* \\ f(c, \mathbf{s}^*) + 1 & \mathbf{s} = \mathbf{s}^* \end{cases} \quad (1)$$

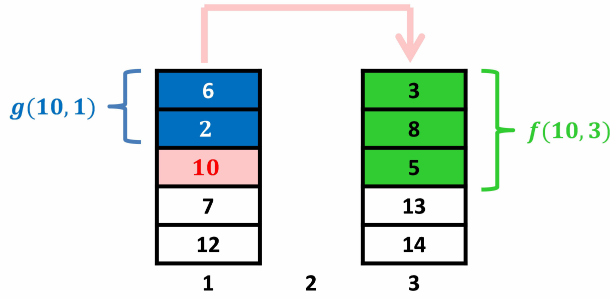

Fig. 1 Graphic presentation of the basic functions when the block with priority 10 is being well located. In this case $f(10, 3) = 3$ and $g(10, 1) = 2$. Red color is used for the block that is being well located (block 10), blue color for blocks that are being removed from the source stack (blocks 6 and 2), and green for the ones that are removed from the destination stack (blocks 3, 8 and 5).

We minimize the heuristic function $w(c, \mathbf{s}^*)$ to determine the stack $\mathbf{s}^*$ to which block c will be well located. This will be the stack $\mathbf{s}^*$ that has the minimal value of $w(c, \mathbf{s}^*)$. It has been shown that this approach gives results of good quality (Expósito-Izquierdo et al, 2012). In case of ties we simply select the stack with a lower index.

The problem with the heuristic function given in Eq. (1) is that it does not consider that this move can create some new, in many cases avoidable relocations. The most obvious source of this is the relocation of already well located containers. This is illustrated in Figure 2. When using the original heuristic to well locate block c with priority 12 all stacks are equal, since in all the cases $f(c, \mathbf{s}^*) = 3$. But it is evident that selecting stack 2 might be a very bad choice because three well located containers will be moved.

We introduce a new approach that takes this into account. We first define $nw(c, \mathbf{s}^*)$ as a number of well located containers that need to be relocated

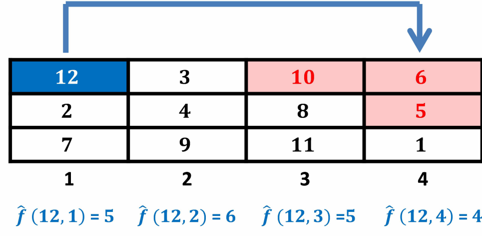

Fig. 2 Exemplifying the heuristic function. The values are shown for the case when block 12 is being well located. In the figure red color indicates blocks that are not well located (blocks 12, 10, 6 and 5).

when c is moved to stack $\mathbf{s}^*$. Using the $nw(c, \mathbf{s}^*)$ we define a new heuristic for the number of relocations related to well locating c in stack $\mathbf{s}^*$.

$$\hat{f}(c, \mathbf{s}^*) = f(c, \mathbf{s}^*) + nw(c, \mathbf{s}^*) \tag{2}$$

Note that it is not necessary to use a similar extension of function g as we know there are no well located blocks above block c. The improved heuristic function $\hat{w}(c, \mathbf{s}^*)$ is defined by substituting f by $\hat{f}$ in w. Using the improved heuristic function, we can differentiate between the stacks in Figure 2, and choose stack 4 as it has the smallest value of $\hat{w}(c, \mathbf{s}^*)$.

In this way two heuristic functions are defined for this stage, the first one corresponds to the functions w and the second one to $\hat{w}$.

3.2 Selecting the Block to be Well Located

In our approach we introduce the use of a heuristic function for selecting which block will be well located next. This is an adaptation of the original algorithm in the sense that we do not strictly select a block c that has the highest due date value. In the original algorithm, the blocks are well located in descending order of their due date values. The idea of this approach is that once the container with the highest due date value is well located it will no longer interfere with the well locating of succeeding blocks. Although this approach proves to be very efficient, this rule can be considered overly strict.

The reasoning for the new stage of the algorithm is as follows. In many cases well locating a specific block with the highest due date value can be difficult because many relocations need to be performed. It may be advantageous to first well locate some other block, as the relocations that will be performed might make it easier to subsequently well locating the one with highest due date value.

Before continuing this explanation, let us define the notion of a forced relocation. Given a current bay configuration, the number of forced relocations of a specific stack is given by the number of blocks in that stack currently on

top of a block with a strictly smaller due date value. Such blocks will necessarily be relocated, in order to retrieve the block with smaller due date value located below. With this definition, another aspect that should be considered when well locating a block from stack $\mathbf{s}$ to stack $\mathbf{s}^*$ is how many new forced relocations will be created. We shall consider that we have created a forced relocation if we move block c_a over a not well located block c_b, and the due date value of c_a is smaller than the one of c_b. This is due to the fact, that we attempt to well locate c_b before c_a. As a consequence, during the procedure of well locating c_b, it will be necessary to relocate block c_a. In this way an extra reshuffle operation will be performed. Note that in some specific situations this will have no negative effect. For example, the relocation of block c_a, during this process, may result in its well locating, but such "accidents" are not considered in the proposed heuristic. We give an example of a forced relocation in Figure 3.

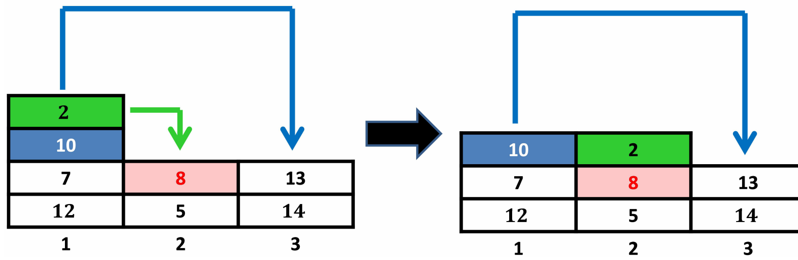

Fig. 3 Illustration of creating a forced relocation. When trying to well locate block 10 on stack 3, block 2 needs to be positioned over block 8 which is not well located. The new position of 2 creates a forced relocation since block 2 will be moved again when well locating block 8. Blue color is used to indicate which block is being well located (block 10), green for containers that created the forced relocation (block 2) and red for not well located containers (block 8).

Forced relocations have been defined in a relatively simple way in the attempt to make their calculation straightforward. A good approximation of the total number of forced relocations can be efficiently calculated only by tracking the due date value at the top of each stack. In this approximation, a forced relocation occurs only if a block c is being relocated and all of the top stack due date values are larger than $p(c)$. Note that in some cases more actual forced relocations may be created, e.g., if a block with a higher due date value exists below c_a even if $p(c_a) < p(c_b)$. If all the blocks in a stack are well located we will consider that stack having due date value zero. (Note that this is only for calculating an eventually approximated number of forced relocations. In that sense, well located stacks cannot create forced relocations.) In this way any block can be placed on this stack without creating a forced relocation using the simplified definition. Let us define $fr(c, \mathbf{s}^*)$ as the number of forced relocations that will be generated when block c is being well located at stack $\mathbf{s}^*$.

A heuristic function for the selection of the block that will be well located next needs to have the following properties:

- Prefer well locating containers with high due date values.
- Prefer a low number of relocations.
- Prefer a low number of forced relocations.

The proposed heuristic can be formalized by the following function:

$$d(c) = argmin_{s \in S} \hat{w}(c, \mathbf{s}) \tag{3}$$

$$\hat{h}(c) = -p(c) + \hat{w}(c, d(c)) + fr(c, d(c)) \tag{4}$$

In Eq. (3), $d(c)$ is the index of the stack that has the lowest value $\hat{w}(c, \mathbf{s})$ for a block c, or in other words the stack to which block c can be well located with a minimal number of relocations. In case of ties we will select the stack with a lower index value. Eq. (4) represents the heuristic function for selecting the block that will be well located next. In Eq. (4), $fr(c, d(c))$ gives the number of created forced relocations corresponding to the selection of stack $d(c)$. As illustrated in Figure 3, $fr(10, 3) = 1$. $p(c)$ is the due date value of block c, the negative prefix is used since we wish to minimize our heuristic function and high due date values are more desirable. Finally, the block that will be well located next is the one that has the minimal value of $\hat{h}(c)$, as given in the following equation.

$$next = argmin_{c \in C} \hat{h}(c) \tag{5}$$

Note that the minimization in Eq. (5) can in most cases be calculated by evaluating $\hat{h}(c)$ for a small number of blocks. More precisely, the highest number of blocks that are tested is

$$n = \hat{w}(c_m, d) + fr(c_m, d). \tag{6}$$

In Eq. (6), c_m is used for the block with the highest due date value of the ones that have not been well located. As it can be seen in Eq. (4) the heuristic function $\hat{h}$ depends on the priority $p(c)$ of the block being relocated. It is obvious that for block c_a which has a priority $p(c_a) < p(c_m) - n$, even if $\hat{w}(c_a, d) + fr(c_a, d) = 0$, we have $\hat{h}(c_a) > \hat{h}(c_m)$.

The addition of the new stage to the original algorithm is better illustrated using the following pseudo code.

while (Not All Containers Well Located) **do**
  Select next container c to be well located using Eq. (5)
  Destination stack $\mathbf{s}^* = d(c)$ was calculated in the previous step
  Well locate container c from its current position to stack $\mathbf{s}^*$
  Fill destination stack
end while

Note that by allowing a non fixed order of well locating containers it is necessary to add an additional structure that can track which containers are well located.

As previously stated, we have extended the original algorithm developed by Expósito-Izquierdo et al (2012) by including a new stage to the algorithm in which we select the block that will be well located next. Although such a stage is not explicitly defined in the original algorithm, in practice it is analogous to the selection of the block with the highest due date value. To summarize, in this way two heuristic functions are defined for this stage, the first one corresponds to highest due date values and the second one to the function $\hat{h}$.

3.3 Relocating the Necessary Blocks

The heuristic functions presented in the previous subsections tell us that block c should be well located in stack $\mathbf{s}^*$. To perform this action it is necessary to relocate several containers from the source $\mathbf{s}$, where block c is located, and destination stack $\mathbf{s}^*$ as explained in Subsection 3.1. The goal is to relocate all the required blocks without creating new avoidable relocations in the future. This process can be divided into two parts.

- Order in which blocks are relocated.
- Selection of the stack to which a block will be relocated.

In Expósito-Izquierdo et al (2012) a detailed description of the ordering in which the blocks are relocated is presented. The basic idea is that at each step we relocate one of the two top blocks of stacks $\mathbf{s}$ or $\mathbf{s}^*$ that has a higher due date value. Of course, only the blocks whose relocation is necessary for well locating c are considered. The process is continued until all the required blocks c_a are relocated. Using this approach the number of blocks to be moved in future iterations is minimized.

The second part is about deciding to which stack $\hat{\mathbf{s}}$ to relocate block c_a. The stack is selected by some heuristic function that measures their desirability, in the sense that we do not wish to create new unnecessary relocations. This problem is very similar to what appears in the BRP. Heuristic functions of this type have been widely researched and analyzed for this problem. As a consequence we can use these heuristics in the case of the PMP. Several different heuristics have been developed for which detailed descriptions can be found in literature. We give a short overview of the ones that seem most suitable for the PMP.

- *The Lowest Position (TLP) heuristic* (Zhang, 2000). In the TLP we relocate the block to a stack that has the lowest number of already used tiers. The goal is to keep the container bay as balanced as possible. In this way the average number of relocations should stay low. It is expected that extreme cases where a large number of blocks needs to be moved from a stack with many tiers are avoided.
- *Lowest Priority Index (LPI) heuristic* (Expósito-Izquierdo et al, 2012). In this approach, the blocking block will be moved to the stack in which it blocks the highest due date value of a not well located block. It is expected

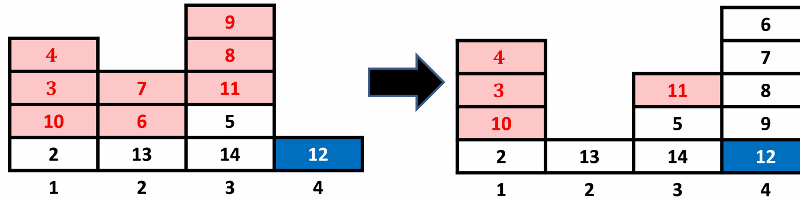

Fig. 4 Illustration of filling a stack. The filling is done after block 12 has been well located. Red color is used for not well located blocks (blocks 4, 3, 10, 7, 6, 9, 8, 11), blue for the block that has been well located last before the filling process (block 12).

that the overall number of reshuffles will be lowered since every time a block is put over another with a lower due date value, extra reshuffles need to be done (Wu and Ting, 2010).

– *The MinMax heuristic* presented in Caserta et al (2011b), and a very similar approach in Ünlüyurt and Aydin (2012), only takes into account the maximal due date value of a block in each stack. An extended version, including a look-ahead mechanism, of this algorithm has been presented by Jovanovic and Voß (2014). This heuristic has a different way of choosing the stack to which the block will be moved to, depending if the block will be well located there or not. The general idea is to avoid relocations of blocks in the near future while grouping blocks of similar priorities.

In practical applications of these heuristics to the PMP, certain improvement can be achieved by adding some fine tuning. First, in the case when the source and destination stack are the same, i.e. $\mathbf{s} = \mathbf{s}^*$, special care should be taken when temporarily relocating the block c that is being well located. In this case, the heuristic function should be used with inverse values compared to the previously defined heuristics. When using the LPI and MinMax heuristics, reaching the top tier of a stack should also be avoided.

3.4 Filling

In the original algorithm the idea of stack filling is introduced to exploit the fact that after a specific container is relocated to a well located position it is at the top of a stack and the whole stack is well located. More precisely, when a target container c is relocated in a destination stack $\mathbf{s}^*$, this container is at the top of stack $\mathbf{s}^*$.

The idea is that we can take advantage of the empty slots (empty tiers) in a well located stack, by maximizing the number of well located containers and maximizing the number of usable slots. In practice this is equivalent to filling the empty slots in the destination stack with not well located containers in the most adequately sorted sequence. So for each destination stack $\mathbf{s}$, all accessible not well located containers with a due date value equal to or lower

than the container at the top of $\mathbf{s}$ are candidates that can potentially be moved to this stack. The filling is done step by step, by relocating the container with the highest due date value that is possible to well locate to the stack $\mathbf{s}$. The process is finished either when the stack is full or there are no more containers that can be properly located in stack $\mathbf{s}$. An illustration of this process is given in Figure 4.

Tests conducted by Expósito-Izquierdo et al (2012) show that this approach is very effective, in a vast majority of problem instances, and manages to significantly reduce the number of necessary relocations needed to set the container bay in a suitable state. They have also pointed out that the use of filling degraded the quality of solutions in case of problems with a small number of stacks and tiers. There are two main reasons for the possible negative effect of stack filling. First, the filling can be responsible for significant changes in the container bay. By doing so the use of the heuristics presented previously can be neglected to a large extent. The other problem is that the minimal due date value of a well located stack can decrease too rapidly. In this way we can lose the possibility of well locating a block a with a high due date value to this stack $\mathbf{s}$. In many cases this results in having to add new relocations, for well locating a, in some case even by relocating blocks from stack $\mathbf{s}$ that have just been placed there.

We try to balance out the positive and negative properties of stack filling by using four different types of stack filling.

- **None.** In this case no filling is conducted
- **Standard.** We try to well locate as many not well located blocks to stack $\mathbf{s}$. This approach is used in the original work.
- **Safe.** In this case we consider the desirability of performing the filling of stack $\mathbf{s}$. We can define the heuristic function $F(\mathbf{s}) = H - Tier(\mathbf{s}')$, where $\mathbf{s}'$ presents stack $\mathbf{s}$ after filling and H is the maximal allowed number of tiers given in the definition of PMP. Filling of $\mathbf{s}$ is only done if $F(\mathbf{s}) \leq a$, where a is a predefined parameter. The logic behind this, i.e., having a high number of blocks well located at $\mathbf{s}$, is worth losing the possibility of well locating some block c with a high due date value at $\mathbf{s}$.
- **Stop.** In this case a simple look-ahead mechanism is used to discontinue the filling process if it can potentially have a negative effect. This can happen if during the filling we are relocating block c_a which is directly above block c_b and $p(c_b) > p(c_a)$. Let us assume that block $c_{s'}$ is at the top of $\mathbf{s}'$ and $p(c_{s'}) > p(c_b)$. It is obvious that by well locating c_a at the top of stack s' the possibility of well locating c_b to the same stack is lost. Since this can potentially add unnecessary relocations, in such cases the filling process is discontinued.

4 Multi-Heuristic Approach

In the previous section we have presented several heuristics for solving specific sub-problems in the PMP, which correspond to different stages of the

greedy algorithm. By combining them we can solve the problem of interest. A greedy algorithm that uses one specific heuristic function for each stage of the algorithm is presented in the original work by Expósito-Izquierdo et al (2012). The performance of the basic deterministic algorithm is improved by adding a certain level of randomization. This is done by randomly selecting one of the n stacks with the top value of the heuristic function at each step of the algorithm. In this way solutions of higher quality are found by searching a wider range of potentially good solutions. This method manages to significantly improve the performance by exploring a high number of solutions. The down side of this randomized approach is that not only high quality solutions are generated but also a number of lower quality ones. This is due to the fact, that we often select stacks which have undesirable properties, in the sense of having a lower value of the heuristic function.

In our new approach the goal is to avoid generating solutions for which we expect that they have undesirable properties. In other words, we wish only to generate solutions for which it is presumed that they will be of good quality, while not making the original greedy algorithm more complex. For most optimization problems our main focus is to find the best possible heuristic function. As it will be shown in the following section in the case of the PMP, none of the proposed heuristics is overwhelmingly superior to the competing ones. Another problem with using heuristic functions is that the use of filling greatly changes the state of the bay that we have used for evaluating the heuristic functions and as a consequence makes our choices less valid.

It is well known that if we have several competing heuristics for some problem, their performance will be highly dependent on the specific properties of the instances that are being solved. Because of this a common practice is to generate solutions using several heuristics and just choose the best one found. This simple logic can be very efficiently exploited in case of using the proposed algorithm and heuristics for the PMP. The idea is to test a relatively small group of good candidates for optimal or close to optimal solutions. In case of our problem this can be done by combining different heuristics at different stages of the algorithm. With this simple method we can generate $\alpha * \beta * \gamma * \delta$ different solutions that have desirable properties in different frames of reference. $\alpha, \beta, \gamma, \delta$ give us the number of different heuristics for each stage of the algorithm, and their product represents the total number of combinations. Using the proposed heuristics we have a total of $48 = 2 * 3 * 2 * 4$ generated solutions.

For such a method to work it is necessary for all runs of the algorithm to create feasible solutions. In the case of the BRP, for well defined problem instances, a feasible solution is always acquired by a greedy algorithm using some of the heuristics proposed in several articles (Zhang, 2000; Murty et al, 2005; Ünlüyurt and Aydin, 2012; Caserta et al, 2011b; Jovanovic and Voß, 2014). Contrary to this in case of the PMP, in many cases it is not possible to well locate all the blocks by directly applying the greedy algorithm. This is especially noticeable for bays with a high level of occupancy. The standard approach for avoiding this situation is to use some kind of backtracking. There

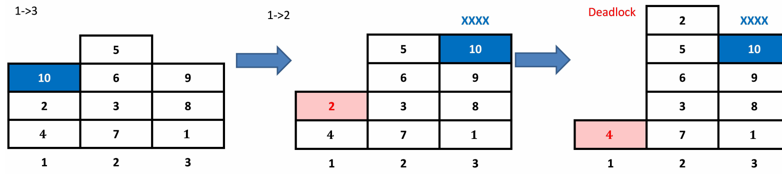

Fig. 5 Illustration of a deadlock. In the example we are trying to well locate block 10, in a bay with a maximal tier of 5. After block 10 is relocated to stack 3 a slot is lost (marked with xxxx). After relocating block 2, a deadlock has been reached since there are no valid relocations for block 4.

are two main drawbacks of this approach. First, the calculation time can in case of large problems be significantly increased and in general it becomes unpredictable. On the other hand, when backtracking is added, the original greedy algorithm becomes more complex in the sense of implementation.

From this we can see that by adding backtracking we have lost the two main advantages of the greedy algorithm, i.e. its speed and simplicity of implementation. We can avoid such deadlocks (states of the algorithm in which there are no feasible relocations) using a much simpler logic. It is possible to add a simple mechanism that can always make one more relocation possible from the source stack. In this way we can always bring the bay to a state where all the blocks are well located.

4.1 Avoiding Deadlocks

For a well defined problem instance, in a sense that a feasible solution exists, it is possible for the proposed algorithm to enter a deadlock. We use the term "deadlock" for a state of the bay from which it is not possible to add a relocation using the defined heuristic. In the proposed method this can only occur when trying to well locate a block c, whose source stack $\mathbf{s}$ is equal to the destination stack $\mathbf{s}^*$. The reason for this is that when performing the necessary relocations some free slots in the bay can be lost when block c is temporarily relocated. This happens since we are not allowed to place any new block c_a above c, and as a consequence $H - t(c)$ slots will be lost. Here $t(c)$ gives us the tier of block c. Because of this if there is a stack with only one free slot block c should always be relocated there. An illustration of a deadlock and the loss of slots is given in Figure 5.

If such a stack did not exist, when c was relocated it is possible to enter a deadlock, a state from which it is not possible to perform a desired move. This situation can be avoided by a few simple steps. First we can remove the last relocation from the solution. Let us say that the removed relocation was $(\mathbf{s}, \mathbf{s_r})$. After reverting the last step we know that stack $\mathbf{s_r}$ is not full. We can select a random full stack $\mathbf{s_f}$, and relocate a block from it to $\mathbf{s_r}$. Now we can relocate block c to this stack. In this way a new free slot has been created.

This process can be more formally presented in the following way.

$$Except = \{\mathbf{s}, \mathbf{s}(c))\} \tag{7}$$

$$\mathbf{s_f} = Random(Full(S \backslash Except)) \tag{8}$$

And we add the following reshuffle operations to the solution.

$$(\mathbf{s_f}, \mathbf{s}(\mathbf{c_r})), (\mathbf{s}(\mathbf{c}), \mathbf{s_f}), (\mathbf{s}, \mathbf{s}(\mathbf{c})) \tag{9}$$

4.2 Correction

Through the analysis of the solutions generated using various combinations of heuristics it has been observed that the acquired relocation sequences in some cases have unnecessary relocations. More precisely, the resulting solution can have consecutive relocations of a single block. This can be a consequence of the use of filling, not well locating blocks in the descending order of their due date values or the mechanism for avoiding deadlocks. Although the creation of such "chain moves" is relatively rare, they can be easily removed from generated solutions. This can be done by adding the following simple correction stage at the end of the greedy algorithm to improve results.

$$(\mathbf{s_1}, \mathbf{s_2}), (\mathbf{s_2}, \mathbf{s_3}) \rightarrow (\mathbf{s_1}, \mathbf{s_3}) \tag{10}$$

$$(\mathbf{s_1}, \mathbf{s_2}), (\mathbf{s_2}, \mathbf{s_1}) \rightarrow \varnothing \tag{11}$$

Equations (10) and (11) give us the substitution/removal rule for correcting the solutions.

4.3 Implementation

The proposed multi-heuristic method can be better understood if it is presented by the following pseudo-code.

for all $h_b \in H_b$ **do**
  for all $h_s \in H_s$ **do**
    for all $h_w \in H_w$ **do**
      for all $h_f \in H_f$ **do**
        while (Not All Containers Well Located) **do**
          Select next container c to be well located using h_b
          Select Destination stack $\mathbf{s}^* = h_s(c)$
          Well locate container c to stack s using h_w
          Perform filling of destination stack using h_f;
        end while
        Perform correction of acquired solution S
        Check if S is the best found solution
      end for
    end for

end for
end for

In the proposed method a number of solutions is generated using all possible combinations of heuristic functions. This is done by selecting one of the possible heuristics for each stage of the algorithm. In the pseudo-code H_b represents the set of heuristics for selecting the block to be well located, H_s for stack selection, H_w is the set of heuristics used when well locating a block and H_f is the set of different filling heuristics. For each combination of heuristics h_b,h_s,h_w and h_f a solution S is generated inside of the while loop which corresponds to the greedy algorithm. The correction procedure is applied on S, and the solution is saved if it is better than all the previously generated ones.

The method has been implemented using an object oriented approach. Although it is possible to use a matrix representation of the bay we have found it advantageous to use separate objects for individual containers and stacks. The main reason for this is that many of the same calculations are used in different stages, and in consecutive iterations of the proposed algorithm. One example is the values of function $\hat{h}(c)$ in Eq. (4). In many cases the stacks used for calculating the value of $\hat{h}(c)$ do not change for block c from one iteration to the next. By using separate classes for stacks and blocks it is possible to store the results of previous calculations, and information if recalculation is needed. One of the consequences of the auxiliary structures is the very similar execution time for different heuristics.

5 Experimental Results

All of the algorithms have been implemented in C# using Microsoft Visual Studio 2012. The calculations have been done on a machine with Intel(R) Core(TM) i7-2630 QM CPU 2.00 Ghz, 4GB of DDR3-1333 RAM, running on Microsoft Windows 7 Home Premium 64-bit.

We have used two types of benchmark data sets to evaluate the performance. The first one is the test data used in the original work (Expósito-Izquierdo et al, 2012), more precisely the data sets that had unique due date values.[1] They make it possible to have a clear comparison of the proposed method and an existing one. This data set consist of problems having from $3x3$ to $6x10$ containers in the initial configuration. To the best of our knowledge the initial positions of the containers are purely random. This has the consequence that these problem instances have a significant level of hardness (level of disorder in the initial bay). The second set of test problems have been generated by the same group and made available online. This data has been structured by different levels of hardness (C1 (hardest), C2, C3, C4(simplest)) and occupancy (100%, 75%, 50%) of the bay. Details of the configurations can

[1] Note that the data from that paper had been lost and was replaced by those on the webpage of the authors: https://sites.google.com/site/gciports/premarshalling-problem/bay-generator. These data sets are used in the second group of tests.

be found on the corresponding website. The test instances have been considered as having unique due date values. By using structured data it was possible to have a better evaluation of the various heuristics. From our observations the first group of benchmark data would have an occupancy rate of 100% and having a hardness level equivalent to C1. For both groups of benchmark data the problem instances have only been defined by a set of containers and their positions. As it is commonly used in published articles, a maximal allowed tier of the bay has been added. More precisely, containers can only be placed at a maximum of two tiers above the problem instance height. For example, for a problem instance having 3 stack with 3 tiers the maximal allowed tier of a container is 5. It is important to mention that in case of the second group of test problems the maximal allowed tier is acquired in the same way. This is due to the fact that, for this group of problems, in case of occupancy levels less than 100%, the containers are randomly distributed within the stacks. This has a consequence that stacks have different heights ranging from zero to the problem instance height.

Table 1 Comparison of heuristics for the relocating of necessary blocks when attempting to well locate a container for the first group of benchmark data. TLP, LPI and MinMax are alternative heuristics described above (see Section 3.3). BF stands for the number of unique best solutions found by each heuristic for different problem sets.

Problem	TLP		LPI		MinMax	
	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF
3-3	11.0(4.5)	1	10.8(4.4)	0	10.8(4.4)	0
3-4	12.2(4.1)	5	11.8(3.8)	0	11.6(3.9)	6
3-5	13.8(4.2)	2	12.8(3.7)	2	12.6(3.5)	8
3-6	15.7(4.8)	2	14.5(4.1)	0	14.1(4.0)	8
3-7	17.8(3.8)	0	16.4(3.5)	1	16.0(3.3)	13
3-8	19.6(3.8)	0	17.0(3.2)	2	16.7(3.1)	10
4-4	23.4(7.3)	13	23.3(7.4)	0	22.9(7.3)	9
4-5	29.2(6.6)	6	27.5(6.3)	0	26.3(6.3)	19
4-6	30.7(6.5)	2	27.9(6.2)	2	27.1(5.9)	17
4-7	35.5(6.3)	1	31.1(5.0)	9	30.3(4.9)	21
5-5	45.2(7.4)	6	42.9(8.2)	3	41.9(7.2)	7
5-6	56.4(10.3)	7	51.8(9.4)	5	50.3(9.6)	15
5-7	61.0(9.4)	0	51.9(7.9)	3	49.9(7.4)	25
5-8	69.2(10.6)	0	60.7(9.1)	8	58.3(8.7)	31
5-9	75.2(10.3)	1	64.4(10.1)	6	61.5(10.6)	30
5-10	81.8(10.9)	0	69.2(9.8)	6	65.7(9.0)	31
6-6	84.2(14.0)	8	77.0(14.6)	9	75.0(13.2)	17
6-10	123.1(15.1)	0	102.8(12.1)	4	96.3(12.1)	34

Tests have been conducted for a wide range of bay sizes with different proportions of maximal tier and number of stacks. In the case of the first/second benchmark data set there have been 40/100 different problem instances for each problem group. For each of the problem groups we compare several properties of the generated solutions. In our experiments we analyze the behavior of different heuristics for each stage of the algorithm. The calculation time for

all the implemented heuristics was fast, and in the performed computational experiments there has been no significant difference in execution time.

In the first group of computational experiments we compare the different heuristics for relocating the necessary blocks to perform the well locating of a block. In these tests, to give a better evaluation, no filling or heuristic selection of blocks is used. No improvement is used for the selection of the destination stack. The results can be seen in Tables 1 and 2, for the first and the second benchmark data sets, respectively.

In Tables 1 and 2 we show the average number of reshuffle operations in solutions generated by using each of the heuristics. The tables also include the corresponding standard deviations. In the goal of having a more extensive evaluation of the proposed heuristics we also include the number of best solutions that have been found by only one of the competing heuristics. We have used the notation BF, for the number of such unique best solutions found. First noticeable issue for the results given in Tables 1 and 2 is that the MinMax heuristic manages to outperform the TLP and LPI for all the bay sizes, levels of occupancy and hardness when the average number of reshuffle operations is observed. This advantage has been confirmed by a single tailed Wilcoxon signed-rank test. The advantage of using the MinMax heuristic is more significant for problems having a higher number of stacks and tiers with a higher level of occupancy and hardness. We wish to point out that when the results are observed for individual problem instances, the other two methods have achieved a number of BF solutions for different problem groups. This is especially noticeable for the TLP heuristic for problems with a higher occupancy level where the number of stacks was equivalent to the number of tiers.

In the second group of tests we compare the effect of the two improvements for MinMax heuristics. The first one is the use of a look-ahead for selecting which block will be well located next. The second one is the inclusion of the number of moved well located blocks when selecting the destination stack. The results are presented in Tables 3 and 4. These tables show the performance of each of the improvements, and their combination, when added to the MinMax heuristic. In it MinMax represents the basic algorithm, MinMax-W is used if we take into account the number of well located containers, MinMax-L if we select which block will be well located next and MinMax-LW if both improvements are applied. For all problem instances for a bay type we observe the normalized change in the number of reshuffle operations using the formula $100 * (MinMax - Improvement)/MinMax$. In Tables 3 and 4 we give the corresponding average values, standard deviation and BF.

Contrary to the case of the initially compared heuristics the performance of these improvements is highly dependent on the hardness and occupancy levels. The results show that a MinMax-LW gives the best results for hardness level C1 and high occupancy of 100%. As previously mentioned this classification also corresponds to the first group of benchmark data sets. In many cases the average improvement is significant and is over 10% when compared to the basic method. For problem instances with such properties, all the improvements manage to reduce the total number of relocation operations when compared to

the basic method. The single tailed Wilcoxon signed-rank test has shown that for such problems MinMax-L and MinMax-LW give a statistically significant improvement to the basic heuristic. For problems with a low level of hardness and occupancy MinMax-L and MinMax-LW prove to be a bad choice since they generally have a negative improvement. From this we can conclude, that for such problem instances, it is best to well locate blocks in the descending order of their due date values. On the other hand MinMax-W proves to be advantageous to MinMax for such problems when average improvement is considered, but a statistical test did not prove the significance.

In Tables 5 and 6 we give results of our experiments regarding different methods of filling and their combination with the look-ahead mechanism. The same properties are presented as in the case of Tables 3 and 4. In all of the tests we use MinMax-W as the base heuristic for the algorithm. This is due to the fact that the results in Tables 3 and 4 show that MinMax-W has a very robust behavior in the sense that it rarely degrades the solution of the basic method.

Results acquired by using different methods of filling are much less conclusive than the ones presented in the previous tables. First, we have confirmed the results of Expósito-Izquierdo et al (2012) that in case of problem instances with a small number of stacks and tiers it is often advantageous not to use filling. In the original paper the effect of using filling was only analyzed for the first benchmark data set, and no such test is given for the second group. Note again that for the second group we have used unique due date values, instead of groups with equal values.

Our computational experiments show that the use of filling is generally advantageous in case of problems with a high level of occupancy and hardness. But for problem instances having 3 stacks, for all proposed heuristics, filling frequently had a negative effect when the normalized change in the average number of reshuffle operations is considered. For problems with a lower level of hardness, the use of filling produced a negative effect in the vast majority of cases. From our observations of generated solutions, for individual problem instances with such configurations, we have noticed two occurrences that may give some explanation. First, during the process of well locating block c the necessary reshuffle operations frequently place blocks to stacks where they are well located. In this way the positive effect of filling is diminished. Secondly, the loss of a stack where a block with a high due date value can be well located by the filling operation, often adds extra reshuffle operations in the later iterations of the algorithm. Such negative effects can be significantly decreased by incorporating the look-ahead mechanism. This can be seen in Tables 5, 6 by comparing the pairs of columns Stop, Stop-L and Safe, Safe-L. The evaluation of the filling only by the use of the average value is somewhat misleading in the sense that it presents the performance worse than it actually is. If we observe the number of BF, it is noticeable that the use of different filling heuristics often manages to find unique best solutions. Because of this, their inclusion in the multi-heuristic approach significantly improves the overall performance of the method.

In Table 7, we give a comparison of the multi-heuristic approach and the original algorithm (Expósito-Izquierdo et al, 2012). In this table we use the notation "Exp. D" for our implementation of the deterministic version of the original algorithm. More precisely Exp. D corresponds to the greedy algorithm using the LPI heuristic, no look-ahead mechanism and the Standard filling heuristic. This column is included to be able to observe the effect of randomization included in the original algorithm.

It is important to point out that the results for our method are obtained generating only 48 different solutions compared to 150 in the original work. From our test we have observed that there is no significant difference in execution time for different heuristics. Because of this we can say that the number of generated solutions gives us a good estimate of the execution time for comparison of the two algorithms. Table 7 shows that the average results obtained using the multi-heuristic approach are noticeably better, in many cases close to 10% improvement, than the original. These results are also significantly better than the ones acquired by any of the individual heuristics.

In Table 7, we have also included the optimal results from (Expósito-Izquierdo et al, 2012) for smaller instances. Our results show that the positive effect of using randomization, although it always improved the average result compared to Exp. D, is much larger in case of smaller problem instances. For larger problem instances the use of a more suitable heuristic is of significantly higher importance. If we observe the results for deterministic algorithms given in Table 5 we can see that in many cases we outperform the randomized algorithm. Table 7 also shows the execution times (in seconds) for different problem sizes, where each of them contains 40 different instances. The approximate calculation time for solving one problem instance using only one heuristic would be more than 1900 (48*40) times shorter.

From the presented results we can see that the multi-heuristic approach is very suitable for instances of higher dimensions, due to the good scaling. The increase of calculation time from the smallest instance (3*3) to the largest one (6*10) is only 4.5 times. The second observation is that the execution time depends more on the number of stacks than the maximal number of tiers. One of the main reasons for both of these behaviors is that auxiliary structures have been used for storing/updating previously calculated properties of stacks (stack height, maximal due date value...). Because of this it was rarely necessary to include all the tiers of a stack in the calculation on the heuristic functions. As a consequence their calculation time was highly dependent on the number of stacks and significantly less on the height of the bay. This was especially evident for the heuristics used in the stage of relocating necessary blocks to make well locating possible. It is important to mention that the initialization of the auxiliary structures would be a considerable part of the computational cost for small problem instances due to the overall short execution time. Because of this we believe the execution times given in Table 7, show a somewhat better scaling than it actually is.

From the presented results we can see that the multi-heuristic approach is very suitable for instances of higher dimensions, due to the good scaling. The

increase of calculation time from the smallest instance (3*3) to the largest one (6*10) is only 4.5 times. The second observation is that the execution time depends more on the maximal number of tiers than the number of stacks.

Finally, in Table 8 we give the results of the multi-heuristic approach for the second set of benchmark data. These results are given as a reference for comparison with algorithms developed in the future on structured data. In this table we give the average number of reshuffle operations for each problem type. We also include the normalized improvement to the basic MinMax heuristic, and the corresponding standard deviation. From these results we can see that the multi-heuristic approach produces a notable improvement to the basic algorithm. The positive effect is the least significant in the case of low levels of occupancy and hardness, and grows with their increase.

6 Conclusion

In this paper we have presented a new method for solving the pre-marshalling problem. It can be seen as an improvement to a previously developed heuristic by Expósito-Izquierdo et al (2012). We have analyzed different stages of that algorithm, and for each of them we have developed several different heuristics. We have tested and compared the performance of the developed approach on a wide range of problem instances and shown that the newly developed approach outperforms the one used in the original algorithm in most cases. Our tests have also shown that for the PMP it is very hard to find a universal heuristic that will always give solutions of high quality. We have observed that the performance of the proposed heuristics is highly dependent on the properties of specific problem instances under consideration.

We have used this knowledge to develop a multi-heuristic approach for solving the PMP. The idea of the new method is to exploit the fact that the given greedy algorithm for solving the PMP consists of four stages, and that for each of them several different heuristics exist. We have generated a number of solutions by using a combination of different heuristics for each stage. In this way only a small group of solutions was generated for which it was expected that they would not have undesirable properties, contrary to the case when simple randomization is used. Our tests have shown that this deterministic algorithm significantly outperforms the original nondeterministic method when the quality of found solutions is observed. Another advantage of the proposed method manages to do so by producing a much lower number of generated solutions which directly corresponds to execution time.

In the future we plan to develop a more adaptive method for heuristic selection which will provide a higher variation of generating solutions while still avoiding the creation of solutions for which it is expected that they are of lower quality. Moreover, it would be interesting to extend our approach to the problem where each container does not have a specific due date value but some sort of range of due date values, eventually corresponding to prospective changes of due date values, e.g., due to modified ship or truck arrivals.

Acknowledgements

We greatly appreciate the constructive response from three anonymous reviewers regarding our paper which helped to improve its presentation.

References

Bortfeldt A, Forster F (2012) A tree search procedure for the container pre-marshalling problem. Eur J Oper Res 217(3):531–540

Caserta M, Voß S (2009) A corridor method-based algorithm for the pre-marshalling problem. Lecture Notes in Computer Science, vol 5484, Springer, Berlin, pp 788–797

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

Caserta M, Voß S, Sniedovich M (2011b) Applying the corridor method to a blocks relocation problem. OR Spectr 33:915–929

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

Gupta N, Nau D (1992) On the complexity of blocks-world planning. Artificial Intelligence 56(23):223 – 254

Huang SH, Lin TH (2012) Heuristic algorithms for container pre-marshalling problems. Comput Ind Eng 62(1):13 – 20

Jovanovic R, Voß S (2014) A chain heuristic for the blocks relocation problem. Comput Ind Eng 75(1):79 – 86

Kim K, Hong GP (2006) A heuristic rule for relocating blocks. Comput Oper Res 33(4):940–954

Kim KH (1997) Evaluation of the number of rehandles in container yards. Comput Ind Eng 32:701–711

Lee Y, Chao SL (2009) A neighborhood search heuristic for pre-marshalling export containers. Eur J Oper Res 196(2):468–475

Lee Y, Lee YJ (2010) A heuristic for retrieving containers from a yard. Comput Oper Res 37(6):1139–1147

Lehnfeld J, Knust S (2014) Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. Eur J Oper Res 239(2):297–312

Murty K, Wan YW, Liu J, Tseng M, Leung E, Lai KK, Chiu H (2005) Hongkong international terminals gains elastic capacity using a data-intensive decision support system. Interfaces 35(1):61–75

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

Scholl A, Voß S (1996) Simple assembly line balancing – heuristic approaches. J Heuristics 2:217–244

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

Tierney K, Pacino D, Voß S (2013) Solving the pre-marshalling problem to optimality with A* and IDA*. Tech. Rep. Technical University of Denmark

Tran, N.K., Haasis, H.-D. (2014) Empirical analysis of the container liner shipping network on the East-West corridor (1995-2011). Netnomics 15(3):121-153

Ünlüyurt T, Aydin C (2012) Improved rehandling strategies for the container retrieval process. J Adv Transport 46(4):378–393

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

Wu KC, Ting CJ (2010) A beam search algorithm for minimizing reshuffle operations at container yards. In: Proceedings of the 2010 International Conference on Logistics and Maritime Systems, Busan, Korea

Zhang C (2000) Resource planning in container storage yard. PhD thesis, Department of Industrial Engineering, The Hong Kong University of Science and Technology

Table 2 Comparison of heuristics for the relocating of necessary blocks when attempting to well locate a container. TLP, LPI and MinMax are alternative heuristics described above (see Section 3.3). The hardness of the problems is given in a decreasing order from C1 to C4.BF stands for the number of unique best solutions found by each heuristic for different problem sets.

Problem	TLP		LPI		MinMax	
	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF
C1						
4x4(50)	6.0(1.1)	0	5.8(1.0)	0	5.8(1.0)	0
4x7	9.0(1.2)	0	8.8(1.1)	0	8.8(1.0)	2
4x10	11.7(1.2)	0	11.5(1.1)	1	11.5(1.2)	5
4x4(75)	18.6(1.0)	12	18.1(1.4)	0	17.8(1.3)	8
4x7	27.3(1.3)	2	26.1(1.1)	9	25.6(0.9)	36
4x10	35.7(1.5)	0	33.5(1.4)	2	33.1(1.2)	34
4x4(100)	32.6(4.2)	32	31.9(4.0)	3	31.8(4.0)	3
4x7	47.7(3.3)	3	43.6(3.5)	7	42.9(3.4)	45
4x10	63.2(3.4)	0	56.5(3.5)	8	55.4(3.5)	59
C2						
4x4(50)	3.7(1.4)	0	3.6(1.4)	0	3.5(1.4)	2
4x7	5.8(1.7)	0	5.5(1.6)	0	5.5(1.5)	4
4x10	8.0(1.8)	0	7.6(1.8)	0	7.5(1.7)	9
4x4(75)	11.0(2.7)	3	10.4(2.5)	0	10.2(2.4)	18
4x7	16.2(3.3)	3	14.7(3.2)	3	14.2(3.0)	31
4x10	22.2(3.8)	0	20.1(3.2)	4	19.4(3.0)	45
4x4(100)	20.6(4.5)	21	19.9(4.9)	7	19.6(4.8)	14
4x7	33.5(4.8)	3	29.7(4.4)	7	28.6(4.2)	54
4x10	45.4(6.5)	0	39.1(5.0)	4	36.9(4.9)	77
C3						
4x4(50)	2.8(1.8)	0	2.8(1.7)	0	2.7(1.7)	5
4x7	4.5(1.8)	0	4.3(1.8)	0	4.3(1.7)	7
4x10	5.7(2.2)	0	5.4(2.0)	0	5.4(2.0)	2
4x4(75)	8.2(2.6)	5	7.9(2.4)	0	7.7(2.4)	14
4x7	10.8(3.1)	0	10.2(2.9)	0	9.8(2.7)	23
4x10	16.0(3.7)	1	14.7(3.3)	0	14.1(3.0)	44
4x4(100)	17.2(5.1)	15	16.3(5.0)	0	16.2(5.1)	13
4x7	24.2(4.8)	2	21.8(4.6)	3	20.9(4.0)	45
4x10	33.2(5.5)	1	29.1(4.7)	6	27.7(4.2)	72
C4						
4x4(50)	2.8(1.5)	0	2.8(1.4)	0	2.8(1.4)	3
4x7	3.9(1.9)	0	3.6(1.7)	0	3.6(1.7)	2
4x10	5.4(2.5)	1	5.1(2.2)	0	5.1(2.2)	1
4x4(75)	7.2(3.0)	3	7.0(2.9)	0	6.9(2.9)	9
4x7	10.7(3.0)	0	10.0(2.9)	0	9.6(2.8)	27
4x10	14.1(3.9)	0	13.2(3.6)	0	12.5(3.3)	41
4x4(100)	15.1(4.9)	9	14.3(4.7)	0	14.2(4.7)	8
4x7	23.3(5.7)	1	21.3(4.9)	2	20.6(4.9)	49
4x10	31.4(6.3)	2	27.9(4.8)	0	26.2(4.7)	77

Table 3 Comparison of the effect of improvements to the MinMax heuristic for the first group of benchmark data. In the notation letters L, W are used to specify if some improvement is added: L if a look-ahead is included; W is included if the number of relocated well located containers is considered. The improvement is given as a normalized value. BF stands for the number of unique best solutions found by each heuristic for different problem sets.

Problem	MinMax	MinMax-W		MinMax-L		MinMax-LW	
	BF	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF
3-3	0	0.0(0.0)	0	-1.0(17.3)	0	-1.0(17.3)	0
3-4	0	2.4(11.9)	1	0.9(9.9)	0	2.8(13.7)	3
3-5	1	2.8(10.2)	1	-0.3(7.5)	0	2.1(11.8)	1
3-6	0	1.3(5.8)	2	0.0(8.0)	0	2.1(9.2)	2
3-7	1	3.8(8.1)	1	0.9(6.5)	1	2.5(9.9)	1
3-8	0	2.2(7.7)	0	-0.6(5.6)	0	1.9(9.4)	0
4-4	1	0.8(7.8)	2	3.5(16.4)	2	5.2(16.4)	5
4-5	1	1.1(7.3)	1	3.9(10.9)	2	4.4(13.9)	5
4-6	2	2.2(7.5)	1	-0.2(9.8)	2	4.0(11.0)	4
4-7	1	-0.6(3.2)	1	1.9(10.5)	1	2.0(9.4)	1
5-5	0	0.7(6.1)	2	12.1(12.9)	4	13.0(12.4)	5
5-6	1	1.5(6.6)	2	12.8(10.9)	4	14.6(10.5)	6
5-7	0	4.0(7.8)	3	7.1(10.2)	4	8.1(10.0)	7
5-8	0	1.4(5.5)	2	8.7(9.6)	4	8.7(10.2)	2
5-9	3	-0.1(5.1)	1	5.8(9.7)	2	6.4(10.0)	5
5-10	2	1.2(5.0)	4	6.4(9.2)	7	7.0(9.3)	9
6-6	0	2.0(5.9)	2	17.5(12.2)	1	18.5(12.6)	7
6-10	0	0.8(6.2)	0	11.1(10.5)	5	12.5(10.0)	11

Table 4 Comparison of the effect of improvements to the MinMax heuristic for the second group of benchmark data. In the notation letters L, W are used to specify if some improvement is added: L if a look-ahead is included; W is included if the number of relocated well located containers is considered. The improvement is given as a normalized value. BF stands for the number of unique best solutions found by each heuristic for different problem sets. The hardness of the problems is given in a decreasing order from C1 to C4.

Problem	MinMax	MinMax-W		MinMax-L		MinMax-LW	
	BF	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF
C1							
4x4(50)	0	-0.3(2.0)	0	0.0(0.0)	0	0.0(2.0)	1
4x7	0	0.0(0.0)	0	<u>0.1</u>(0.9)	0	<u>0.1</u>(0.9)	0
4x10	0	0.0(0.0)	0	<u>0.1</u>(0.8)	0	<u>0.1</u>(0.8)	0
4x4(75)	0	-2.1(9.5)	1	<u>15.3</u>(11.5)	33	13.6(11.6)	26
4x7	0	-0.6(5.0)	0	<u>9.5</u>(5.2)	42	8.5(5.6)	29
4x10	1	-0.4(3.8)	0	<u>5.8</u>(4.6)	40	5.0(4.8)	22
4x4(100)	4	-5.4(13.5)	6	<u>6.8</u>(13.2)	29	4.8(16.1)	17
4x7	0	1.0(5.6)	1	9.9(7.3)	26	<u>11.0</u>(7.3)	37
4x10	2	0.6(4.0)	0	7.1(5.8)	32	<u>7.7</u>(5.1)	39
C2							
4x4(50)	0	0.0(0.0)	0	-0.1(9.7)	0	-0.1(9.7)	0
4x7	0	0.0(0.0)	0	0.0(0.0)	0	0.0(0.0)	0
4x10	0	0.0(0.0)	0	-0.4(3.8)	0	-0.4(3.8)	0
4x4(75)	2	-1.0(6.1)	1	-0.5(11.1)	2	-1.4(11.6)	1
4x7	4	<u>0.7</u>(7.2)	4	-1.1(6.8)	0	-0.1(8.6)	2
4x10	0	1.9(5.3)	2	0.8(6.1)	0	<u>2.7</u>(8.0)	6
4x4(100)	1	3.2(12.1)	3	1.7(16.7)	2	<u>3.4</u>(18.4)	6
4x7	1	2.2(6.2)	11	1.5(9.5)	4	<u>3.4</u>(9.9)	10
4x10	1	<u>2.3</u>(5.4)	8	-0.2(8.0)	5	1.8(9.4)	15
C3							
4x4(50)	0	0.0(0.0)	0	-1.8(9.0)	0	-1.8(9.0)	0
4x7	0	0.0(0.0)	0	-0.8(6.1)	0	-0.8(6.1)	0
4x10	0	0.0(0.0)	0	-0.4(3.1)	0	-0.4(3.1)	0
4x4(75)	3	-0.3(3.6)	2	-2.4(11.6)	0	-2.4(12.2)	1
4x7	0	<u>0.6</u>(3.7)	0	-1.0(8.9)	0	-0.4(9.7)	0
4x10	0	<u>1.0</u>(4.0)	2	-0.2(6.1)	1	0.8(7.5)	2
4x4(100)	0	<u>0.3</u>(2.9)	3	-2.6(13.4)	0	-2.3(13.1)	1
4x7	1	<u>0.7</u>(5.0)	4	-3.2(8.4)	1	-2.9(9.2)	1
4x10	2	<u>1.2</u>(4.6)	5	-2.1(6.8)	1	-1.3(7.6)	0
C4							
4x4(50)	0	0.0(0.0)	0	-1.8(12.5)	0	-1.8(12.5)	0
4x7	0	0.0(0.0)	0	-1.6(10.1)	0	-1.6(10.1)	0
4x10	0	0.0(0.0)	0	-0.1(2.4)	0	-0.1(2.4)	0
4x4(75)	1	<u>0.3</u>(3.6)	2	-1.6(12.3)	0	-1.6(12.5)	0
4x7	0	<u>0.4</u>(3.7)	2	-2.3(10.8)	0	-2.3(11.0)	0
4x10	0	<u>0.7</u>(3.1)	4	-1.4(5.5)	0	-1.1(5.5)	0
4x4(100)	1	<u>0.4</u>(4.1)	1	-3.4(11.0)	1	-3.0(12.5)	1
4x7	1	0.0(0.6)	0	-5.3(8.9)	0	-5.3(9.2)	1
4x10	0	<u>0.2</u>(0.9)	3	-3.1(7.2)	0	-3.1(7.1)	0

Table 5 Evaluation of the effect of different filling algorithms and their combination with a look-ahead mechanism for the first group of benchmark data. In the notation an added letter L means a look-ahead is included. BF stands for the number of unique best solutions found by each heuristic for different problem sets.

Problem	MinMax	Standard		Stop		Safe		L-Stop		L-Safe	
	BF	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF
3-3	2	-3.9(10.3)	0	-3.1(15.6)	2	-1.0(12.0)	0	-2.4(18.2)	1	-0.7(16.9)	0
3-4	4	-2.2(8.9)	0	-5.7(13.5)	2	-1.5(7.5)	0	-0.1(11.8)	0	-0.5(13.3)	1
3-5	9	-1.3(9.8)	0	-3.8(11.8)	0	-2.4(9.9)	1	-2.6(11.0)	0	-4.9(12.2)	0
3-6	7	-1.2(10.5)	0	-1.6(10.9)	0	-0.8(8.5)	0	-1.4(12.9)	2	-2.1(11.4)	0
3-7	18	-5.5(9.4)	0	-6.3(9.7)	0	-5.1(8.8)	0	-6.2(9.5)	0	-5.7(8.9)	0
3-8	10	-1.7(7.7)	0	-1.7(7.2)	0	-1.8(8.0)	1	-3.3(9.5)	0	-2.6(8.9)	1
4-4	5	5.0(12.9)	2	5.4(15.1)	2	6.1(15.0)	2	2.6(17.3)	1	4.3(17.2)	3
4-5	4	4.8(11.1)	3	3.0(12.5)	2	3.6(13.3)	1	7.5(11.5)	7	4.7(12.3)	2
4-6	5	4.6(10.2)	4	1.9(10.8)	1	1.2(9.6)	0	5.4(10.4)	2	2.4(10.5)	2
4-7	8	3.7(9.0)	1	1.2(9.6)	0	1.0(10.0)	2	4.3(9.5)	3	2.3(10.7)	5
5-5	0	14.3(10.2)	1	16.0(9.4)	5	13.9(10.0)	2	17.5(10.4)	7	16.2(10.8)	5
5-6	3	14.7(13.4)	5	15.5(14.5)	7	12.4(12.6)	1	17.2(13.1)	8	14.9(13.6)	3
5-7	3	7.0(10.5)	4	8.2(9.5)	4	5.3(9.4)	4	9.4(10.0)	9	5.9(9.1)	4
5-8	2	13.4(9.4)	5	13.1(9.0)	4	9.1(11.4)	1	13.8(9.2)	10	10.1(9.7)	2
5-9	2	12.6(10.6)	3	11.9(10.7)	3	9.1(8.5)	4	13.0(10.8)	7	9.6(9.9)	1
5-10	1	11.6(9.0)	6	10.7(9.3)	0	8.2(9.9)	3	12.0(9.4)	11	9.4(9.8)	4
6-6	0	20.0(11.9)	3	26.2(10.6)	14	17.2(13.3)	2	25.5(10.6)	9	22.4(12.3)	5
6-10	0	20.0(8.0)	7	19.3(8.3)	1	14.7(9.1)	3	20.7(7.6)	15	16.7(8.3)	1

Table 6 Evaluation of the effect of different filling algorithms and their combination with a look-ahead mechanism for the second group of bench mark data. In the notation an added letter L means a look-ahead is included. BF stands for the number of unique best solutions found by each heuristic for different problem sets. The hardness of the problems is given in a decreasing order from C1 to C4.

Problem	MinMax	Standard		Stop		Safe		L-Stop		L-Safe	
	BF	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF	Avg(Std)	BF
C1											
4x4(50)	0	0.0(0.0)	0	-2.8(8.0)	0	-1.5(6.3)	0	0.3(2.5)	0	-1.3(6.8)	0
4x7	1	-0.1(2.5)	0	-0.1(2.5)	0	0.1(1.7)	0	0.0(2.7)	0	0.2(1.9)	0
4x10	1	0.4(2.2)	0	0.4(2.2)	0	0.1(1.6)	0	0.5(2.3)	0	0.2(1.7)	0
4x4(75)	0	5.0(8.3)	0	13.0(8.5)	0	14.6(9.2)	0	13.7(10.6)	7	17.8(7.7)	1
4x7	0	2.4(4.5)	0	1.8(4.8)	0	2.1(4.7)	0	8.6(4.5)	23	7.8(4.7)	9
4x10	1	4.6(3.4)	0	4.3(3.7)	0	4.3(3.7)	0	6.0(3.6)	10	5.8(3.6)	4
4x4(100)	4	7.0(15.2)	1	23.1(14.2)	30	12.5(19.1)	0	14.8(18.0)	7	19.5(18.1)	24
4x7	0	15.0(6.1)	4	14.7(5.8)	5	11.5(6.9)	5	16.2(6.1)	19	13.7(7.1)	9
4x10	0	11.2(6.3)	3	11.1(6.4)	5	7.9(6.1)	5	13.7(5.7)	28	10.9(5.5)	10
C2											
4x4(50)	0	0.0(0.0)	0	-2.2(11.2)	0	0.0(0.0)	0	-0.1(9.7)	1	-1.0(11.7)	0
4x7	1	-0.1(1.4)	0	-3.1(7.9)	0	-2.4(7.4)	0	-0.1(1.4)	0	-2.3(7.1)	0
4x10	12	-1.6(4.7)	0	-3.7(7.7)	1	-3.1(6.5)	0	-1.7(5.7)	0	-3.2(7.1)	0
4x4(75)	6	-1.5(6.1)	0	-5.3(10.9)	3	-2.5(7.5)	0	-1.4(12.1)	3	-3.1(14.0)	2
4x7	22	-1.4(9.0)	0	-2.9(9.5)	2	-2.5(9.1)	0	-0.5(9.4)	6	-1.4(8.9)	1
4x10	27	-1.0(7.8)	1	-2.9(7.6)	1	-1.9(6.9)	0	0.1(9.0)	5	-1.0(8.2)	1
4x4(100)	22	-0.1(13.1)	3	-2.8(16.9)	5	-0.1(13.9)	2	-1.0(16.4)	1	-1.1(17.4)	5
4x7	22	1.7(10.4)	4	0.9(10.1)	2	0.8(10.8)	6	1.3(10.9)	5	0.2(11.4)	2
4x10	32	-1.0(9.1)	0	-1.6(8.9)	0	-2.0(9.6)	4	-0.6(10.2)	12	-1.3(9.7)	6
C3											
4x4(50)	0	0.0(0.0)	0	-0.3(3.3)	0	0.0(0.0)	0	-1.8(9.0)	0	-1.2(9.2)	1
4x7	0	0.0(0.0)	0	-1.5(5.2)	0	-1.0(3.8)	0	-0.8(6.1)	0	-1.7(7.1)	0
4x10	5	-0.6(2.8)	0	-2.5(6.7)	0	-1.9(6.1)	0	-0.6(3.9)	0	-1.7(6.3)	0
4x4(75)	0	-0.5(3.3)	0	-5.6(10.4)	2	-1.9(6.0)	0	-2.8(11.6)	0	-1.7(11.7)	4
4x7	23	-4.7(8.3)	0	-8.6(13.0)	0	-7.0(10.8)	0	-3.8(10.8)	0	-5.6(12.7)	0
4x10	42	-5.9(9.3)	0	-8.4(10.9)	0	-7.1(9.8)	0	-6.1(10.5)	1	-7.3(10.5)	2
4x4(100)	24	-1.7(13.0)	0	-3.5(14.5)	3	-1.8(12.5)	1	-2.7(15.6)	2	-3.4(15.7)	1
4x7	53	-10.5(9.9)	0	-11.2(10.0)	0	-9.3(11.1)	6	-11.2(11.3)	1	-11.0(11.7)	2
4x10	70	-10.0(9.8)	1	-11.4(10.4)	1	-9.5(10.4)	3	-11.2(10.4)	2	-10.3(11.3)	4
C4											
4x4(50)	0	0.0(0.0)	0	0.0(0.0)	0	0.0(0.0)	0	-1.8(12.5)	0	-1.8(12.8)	1
4x7	0	0.0(0.0)	0	-1.6(6.7)	0	-1.0(5.6)	0	-1.6(10.1)	0	-2.7(11.7)	0
4x10	6	-0.8(3.4)	0	-1.3(4.3)	0	-1.4(4.6)	0	-0.6(3.8)	0	-1.2(4.9)	0
4x4(75)	0	0.1(2.0)	0	-2.6(8.6)	0	-0.9(5.1)	0	-1.8(11.9)	5	-3.3(12.6)	2
4x7	19	-4.8(8.5)	0	-8.7(11.4)	0	-6.5(9.6)	0	-4.8(10.8)	1	-6.4(13.0)	1
4x10	49	-7.4(7.7)	0	-10.8(9.9)	0	-8.7(8.8)	0	-7.9(9.1)	1	-8.5(9.8)	0
4x4(100)	20	-3.5(9.9)	2	-4.5(11.6)	5	-3.3(9.9)	0	-4.7(13.9)	2	-5.3(13.6)	0
4x7	59	-9.8(9.3)	1	-12.8(10.9)	2	-10.7(12.0)	5	-10.6(10.3)	1	-11.1(12.4)	1
4x10	80	-13.3(10.1)	2	-15.4(11.3)	1	-14.2(10.9)	0	-14.4(10.2)	1	-14.7(11.0)	0

Table 7 Comparison of the proposed multi-heuristic approach with previously published results on the first group of benchmark data. Original corresponds to the algorithm presented in Expósito-Izquierdo et al (2012). Exp. D is a deterministic version of the same algorithm. Opt represents the known optimal solutions.

Problem	Exp. D	Original	Multi-heuristic		Opt
			Avg	Time(s)	
3*3	11.85	10.95	**9.98**	0.821	8.78
3*4	12.20	11.03	**10.33**	0.832	9.03
3*5	13.18	11.98	**11.60**	0.812	10.15
3*6	14.52	13.40	**13.05**	0.853	11.28
3*7	16.77	15.40	**14.80**	1.140	12.80
3*8	17.50	16.38	**15.70**	1.181	13.68
4*4	21.08	20.10	**18.63**	1.543	15.83
4*5	25.55	22.13	**21.88**	1.625	21.05
4*6	26.15	24.20	**23.50**	1.865	-
4*7	30.03	27.88	**27.18**	1.900	-
5*5	35.03	31.78	**31.48**	2.074	-
5*6	41.30	38.40	**37.30**	2.596	-
5*7	44.50	41.43	**40.73**	2.472	-
5*8	50.35	47.80	**47.25**	2.473	-
5*9	54.42	53.73	**50.28**	2.788	-
5*10	59.05	58.08	**54.38**	3.171	-
6*6	54.80	51.55	**50.23**	3.047	-
6*10	78.45	77.90	**72.40**	3.641	-

Table 8 Performance of the multi-heuristic approach of the second group of benchmark data. The hardness of the problems is given in a decreasing order from C1 to C4.

Problem	Avg	Imp(Std)
C1		
4x4(50)	5.8	0.1(1.4)
4x7	8.8	0.4(1.8)
4x10	11.4	0.7(2.6)
4x4(75)	13.9	21.4(6.2)
4x7	21.9	14.3(3.6)
4x10	30.0	9.1(3.5)
4x4(100)	22.3	28.9(10.4)
4x7	33.5	21.5(6.6)
4x10	45.5	17.6(5.0)
C2		
4x4(50)	3.5	1.3(6.7)
4x7	5.5	0.0(0.0)
4x10	7.5	0.5(2.6)
4x4(75)	9.6	4.3(8.2)
4x7	13.3	5.8(7.3)
4x10	18.1	6.3(6.8)
4x4(100)	16.7	12.8(13.3)
4x7	25.5	10.1(8.3)
4x10	33.8	8.1(7.2)
C3		
4x4(50)	2.7	0.5(3.8)
4x7	4.2	0.3(3.3)
4x10	5.3	0.3(2.0)
4x4(75)	7.3	3.7(7.2)
4x7	9.5	2.8(6.5)
4x10	13.6	3.0(5.9)
4x4(100)	14.7	7.3(9.2)
4x7	20.2	3.1(5.5)
4x10	26.6	3.7(5.4)
C4		
4x4(50)	2.7	1.2(5.4)
4x7	3.5	0.7(3.7)
4x10	5.0	0.4(2.6)
4x4(75)	6.5	3.8(8.2)
4x7	9.4	2.6(6.0)
4x10	12.3	1.4(4.0)
4x4(100)	13.4	4.5(6.9)
4x7	20.0	2.4(4.3)
4x10	25.7	1.8(3.5)