INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

I F O R S International Federation of Operational Research Societies

INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

Intl. Trans. in Op. Res. 00 (2011) 1–17
DOI: 10.1111/j.1475-3995.2011.00831.x

Heuristic search for the stacking problem

Rui Jorge Rei[a,b] and João Pedro Pedroso[a,b]

[a] *Departamento de Ciência de Computadores, Universidade do Porto—Faculdade de Ciências, Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal*
[b] *INESC Porto, Rua Dr. Roberto Frias, 378, 4200-465 Porto, Portugal*
E-mail: rui.rei@dcc.fc.up.pt [Rei]; jpp@fc.up.pt [Pedroso]

Received 11 February 2011; received in revised form 22 June 2011; accepted 3 August 2011

Abstract

This paper presents the Stacking Problem, a hard combinatorial optimization problem concerning handling and storage of items in a warehouse, where they are handled by a crane and organized into stacks. We define the problem, study its complexity class, and present a mathematical programming model to solve it. In order to tackle medium- or large-scale instances, we propose a simulation-based algorithm using semi-greedy construction heuristics. This simple approach allows for multiple constructions, finding solutions within reasonable time even for large instances. Three semi-greedy heuristics are proposed and compared in an extensive computational experiment, where we study the relation between the number of constructions and the best solution obtained using each heuristic.

Keywords: Stacking Problem; optimization; simulation; heuristics

1. Introduction

The Stacking Problem (SP) has its origins in the steel industry, where the production sequence is very strict, in order to achieve the desired product quality. This sequence is usually very different from that of customer orders, implying that steel bars have to be kept in warehouses before being delivered. The bars are commonly kept in stacks, and handled by stacking cranes that can only access the top item of each stack. Hence, in order to deliver an item that is not at the top of a stack, all the items above it have to be moved to other places.

The problem we are dealing with is that of minimizing the number of movements that a stacking crane has to do in order to place each incoming item in a stack, and later deliver it, after moving all the items that were possibly placed above it in the mean time. SPs such as this one appear in a variety of application areas, such as port operations (Dekker et al., 2006; Hartmann, 2004) and ship stowage planning (Avriel et al., 2000, 1998).

The SP consists of a series of placement decisions for a set of items with known dates for entrance and exit from a warehouse, known as *release* and *due* dates, respectively. If two or more of these *events* occur at the same date, they can be processed in an arbitrary order. Inside the warehouse, the items are manipulated by a stacking crane and organized in stacks, i.e., in a last-in first-out order. An important operational constraint is that only one item—at the top of a stack or entering the warehouse—may be moved at a time. This may cause relocations when delivering items that are not located at the top of a stack.

This forced relocation of upper items is called *reshuffling*, and becomes necessary when due date inversions exist. A due date inversion, or simply an *inversion*, occurs when an item i with due date D_i is placed above any other item j with due date D_j, such that $D_i > D_j$. This means that at some point in time $t \leq D_j$, item i will have to be placed in another stack to allow the delivery of item j. Taking these rules into consideration, placement decisions can be made at the following points:

- when an incoming item arrives at the warehouse (a release move); in this case, it may go to any stack;
- when an item that is not at the top of its stack has to be delivered; in this case, the topmost item must be placed in a different stack (a reshuffling move).

A third case is when, at any time, an unforced relocation of items in the warehouse exists. This type of movements, called *remarshalling*, has the objective of reorganizing the warehouse in order to reduce the total number of relocations later on. For the sake of simplicity, remarshalling is not considered in the algorithm proposed in this paper.

As the objective of this problem is to find the shortest sequence of crane movements, placement choices should attempt to minimize the retrieval effort when items' due dates are reached. In order to do so, inversions should be avoided. Note that minimizing the number of movements is equivalent to minimizing the number of relocations, since the number of release and delivery movements is constant ($2N$).

Possible variations of the problem include imposing a maximum stack height limit (capacitated SP), noninstantaneous crane movements, allowing late deliveries with associated penalties, or a crane movement cost matrix. These additional constraints naturally result in more realistic scenarios; however, the variant studied in this paper has no limit on stack sizes (uncapacitated SP), crane movements are assumed instantaneous, and all deliveries must be done on time.

The method used to approximately solve the problem is based on a simulation model of the warehouse, which is operated with the help of three semi-greedy construction heuristics; for a detailed description of simulation concepts, see Law (2006). Each simulation corresponds to the construction of one solution for a particular instance using a given heuristic. At each point of decision, the heuristic is used to select the destination stack for the item being moved. Since all the heuristics contain a probabilistic component, making several constructions with the same heuristic usually leads to different solutions.

When compared to a complete search, running a simulation of the warehouse is computationally inexpensive. Thus, we can afford to run multiple simulations and exploit the probabilistic component of the heuristics, recording the best solution obtained in one of those simulations. We compare the heuristics using repeated construction with the simulation model, and study the expected number of simulations required to obtain solutions of good quality for the three methods.

The paper is structured as follows. In Section 2, we present a formal description of the SP. Section 3 gives a short summary of related research. In Section 4, we present a mathematical programming formulation for the SP, followed by a description of the construction heuristics in Section 5. The computational experiment and its results are discussed in Section 6. Section 7 offers some concluding remarks and possible directions for future research.

2. Problem description

Consider the problem of stacking N items in a warehouse with W stacks. Let $R \in \mathbb{R}^N$ be the list of release dates, where R_i denotes the release date of item i, i.e. the time at which i will enter the warehouse. In the same way, let $D \in \mathbb{R}^N$ be the list of due dates, i.e. the times at which items will leave the warehouse. We assume that due dates are greater than release dates for every item, i.e. $R_i < D_i$, for $i = 1, \ldots, N$.

The objective is to store and deliver all the items, satisfying release and due dates, using the shortest sequence of movements, or equivalently, the sequence containing the least relocations. A move consists in taking a single item, either incoming or from the top of a stack, and placing it on top of another stack, or delivering it. Hence, there are three types of moves:

- a *release move* occurs when an item enters the warehouse at its release date and is placed on the top of a stack;
- a *delivery move* occurs when an item, located at the top of its stack, is removed from the warehouse at its delivery date;
- a *relocation move* occurs when an item is shifted from the top of its stack to the top of another stack, either to reorganize the warehouse (remarshalling), or to allow access to another item which must be delivered immediately (reshuffling).

A solution can be represented as an ordered sequence of movements. In turn, a movement can be represented as a triplet (i, s_j, s_k), where i is the item being moved, s_j is the source stack, and s_k is the destination stack. For the special case of releases, the movement is represented as having source stack s_R, and for deliveries the destination stack is s_D.

A movement sequence is constructed by selecting a stack for items as they arrive at the warehouse. Also, when an item is delivered, if reshuffling is required, new stacks must be chosen for the upper items as well. The heuristics presented in this paper do not take remarshalling into consideration, therefore all relocation moves are due to reshuffling.

3. Background

No previous references to an SP as described in this paper could be found. However, the literature presents several problems with many similarities.

One such problem is the container ship stowage problem (CSSP), an SP with a similar objective—minimizing the number of shifts. In the CSSP, a ship that calls at many ports has to transport containers with known source and destination ports. The containers are stored in a stacking bay inside the ship, where they can be accessed only from above, one at a time. When a container

that has port i as destination has to be removed from the ship, any containers above it with destinations further away from port i are removed from the ship together with the target container, and afterwards are reloaded into the ship in an arbitrary order, possibly to different positions. This temporary removal and reloading operation is known as shifting.

In Avriel et al. (1998), an integer programming model to optimally solve the CSSP is presented. However, the model's usability is very limited because of the large number of binary variables and constraints. For this reason, the authors developed a heuristic procedure to solve the problem approximately.

As for the complexity of the CSSP, a relation between the uncapacitated (no limit on stack height) CSSP and the graph coloring problem for overlap graphs (equivalently, circle graphs) is presented in Avriel et al. (2000), in order to show the NP-completeness of the CSSP. The reduction to the coloring of overlap graphs is obtained by creating a unique overlap graph from a container transportation matrix. Then, the given stowage problem can be solved without shifts using C stacks if and only if the corresponding overlap graph is colorable with C colors.

In Unger (1988), the C-coloring problem of overlap graphs for fixed C was shown to be NP-complete for any $C \geq 4$. For $C = 3$, a polynomial time algorithm has been presented in Unger (1992), and two-coloring of graphs can be done in polynomial time using an algorithm for recognizing bipartite graphs. Hence, the C-coloring problem of overlap graphs is NP-complete for any fixed $C \geq 4$, and polynomial for $C < 4$. Consequently, the uncapacitated zero-shift problem is NP-complete for any fixed number of columns $C \geq 4$.

It should be noted that if an instance of the CSSP can be solved without shifts, an equivalent instance of the SP, obtained by a trivial transformation (relating the relative order of release and delivery events with port numbers), can be solved without relocations. Based on this observation, we conclude that the zero-relocation SP (decision) is NP-complete for any fixed $C \geq 4$, and polynomial for $C \in \{2, 3\}$. Also, the NP-completeness of the zero-relocation SP implies that the x-relocation SP is NP-hard for any $x \geq 1$. Furthermore, we conclude that the SP (optimization), i.e. finding a sequence of moves of minimum length for a given set of items and a fixed number of columns, is NP-hard.

Another problem similar to the SP is the Blocks Relocation Problem (BRP). In the BRP, given an initial configuration of a stacking bay with C columns and R rows, and a sequence of retrieval of the items, the objective is to find a relocation plan that minimizes the number of movements to retrieve the blocks in the given order.

It is important to note that, since in the BRP the initial placement of items in the stacking area is given, and it does not consider the arrival of new items amidst the retrieval of others, the BRP is a subproblem of the SP.

In Kim and Hong (2006), branch-and-bound algorithms are proposed for the case with precedence relationships among individual blocks and among groups of blocks with similar features. Additionally, a heuristic rule for the BRP is proposed, and its performance is compared to the branch-and-bound algorithms, with satisfactory results and a reduced computation time, suitable for real-time applications. The heuristic is based on the estimated number of additional relocations for each stack, and uses that information to decide the destination stack of relocated items.

In Caserta et al. (2009), an algorithm based on the corridor method is presented. The corridor method is a hybrid metaheuristic combining mathematical programming techniques with heuristics. Its main idea is applying an exact algorithm to smaller areas of the search space, and moving the

focus of the search to different areas using neighborhood rules similar to those in local search. A dynamic programming formulation was used in that paper to solve the small subproblems exactly.

4. Mathematical programming formulation

In this section, we present a linear mathematical programming formulation for the SP, using binary decision variables. The formulation is for the capacitated version of the problem, but it can be used for the uncapacitated SP by choosing a maximum stack height equal to the number of items. This formulation is valuable because it defines the problem in a formal and rigorous manner, and it can be used to obtain optimal solutions. However, due to the large number of binary variables and constraints, even using state-of-the-art solvers, only toy instances can be tackled within a reasonable amount of time.

4.1. Problem data

Before presenting the variables, constraints, and objective function, we introduce the parameters of the model.

$T \in \mathbb{N}$ – time horizon, i.e. the number of periods in the model.
$N \in \mathbb{N}$ – number of items
$W \in \mathbb{N}$ – the number of stacks in the warehouse (warehouse width).
$H \in \mathbb{N}$ – the maximum number of items that can be in a stack at any given instant (warehouse height).
$R \in \mathbb{R}^N$ – item release dates (R_i denotes the release date of item i).
$D \in \mathbb{R}^N$ – item due dates (D_i denotes the due date of item i).

The time horizon is an upper bound of the number of movements required, including the $2N$ movements for item releases and deliveries, and an upper bound on the number of relocations given by the worst-case scenario shown in Fig. 1. The expression for T is then

$$T = 2N + \sum_{n=1}^{N-1} n.$$

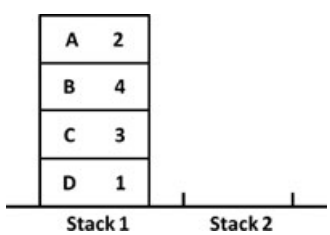

Fig. 1. Worst-case scenario for an instance of the SP having items A, B, C, and D, with due dates 2, 4, 3, and 1, respectively. From this position, delivering the items in the proper sequence ($D \to A \to C \to B$) requires $\sum_{n=1}^{N-1} n$ relocations, i.e. 6 relocations.

4.2. Variables

Our formulation is a multiperiod model that uses three sets of position variables to represent items that did not enter the warehouse yet (a_{nt}), the state of all positions in the warehouse (b_{ijnt}), and the items that already left the warehouse (c_{nt}), at any given period t.

- a_{nt} is equal to 1 if item n has not entered the warehouse yet at period t, and 0 otherwise.
- b_{ijnt} is equal to 1 if item n is in row j of stack i (henceforth referred to as position (i, j)) at period t, and 0 otherwise.
- c_{nt} is equal to 1 if item n has already left the warehouse at period t, and 0 otherwise.

We use another three sets of variables for all possible movements. As previously mentioned, there are three types of movements: releases (x_{ijnt}), relocations (y_{ijklnt}), and deliveries (z_{ijnt}).

- x_{ijnt} is equal to 1 if item n is released into position (i, j) at period t, and 0 otherwise.
- y_{ijklnt} is equal to 1 if item n is relocated from position (i, j) into position (k, l) at period t, and 0 otherwise.
- z_{ijnt} is equal to 1 if item n is delivered from position (i, j) at period t, and 0 otherwise.

4.3. Objective function

The goal of the SP is to find the shortest sequence of movements satisfying the constraints for storage space, as well as release and delivery dates. In other words, the objective is to minimize the number of crane movements. However, since this number includes the constant $2N$ for item release and delivery movements, we are only interested in minimizing the number of relocations, that is,

$$\textit{minimize} \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{W} \sum_{l=1}^{H} \sum_{n=1}^{N} \sum_{t=1}^{T} y_{ijklnt}.$$

4.4. Constraints

We now define the constraints that enforce the correct placement in the stacks, the order of events, and the correctness of the solutions according to the definition of the SP.

Constraint 1 ensures that every item is in exactly one location (unreleased, inside the warehouse, or delivered) at all times. Note that since a movement at any period is only reflected in the position variables at the next period, the position variables are also defined for a final period $T + 1$, to reflect the changes of a possible movement at period T (the final period for movement variables).

$$a_{nt} + \sum_{i=1}^{W} \sum_{j=1}^{H} b_{ijnt} + c_{nt} = 1, \quad \text{for } n = 1, \ldots, N, \quad t = 1, \ldots, T + 1. \tag{1}$$

Similarly, a position inside the warehouse can have at most one item at any given instant. This is ensured by constraint 2.

$$\sum_{n=1}^{N} b_{ijnt} \leq 1, \quad \text{for } i = 1, \ldots, W, \quad j = 1, \ldots, H, \quad t = 1, \ldots, T+1. \tag{2}$$

For $j > 1$, a position (i, j) can only be occupied by an item if position $(i, j-1)$ is also occupied. This is achieved in constraint 3, which prevents items from *floating* in the air.

$$\sum_{n=1}^{N} (b_{ijnt} - b_{i,j-1,nt}) \leq 0, \quad \text{for } i = 1, \ldots, W, \quad j = 2, \ldots, H, \quad t = 1, \ldots, T+1. \tag{3}$$

Before proceeding further, let us define

$$m_t = \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{n=1}^{N} \left(x_{ijnt} + \sum_{k=1}^{W} \sum_{l=1}^{H} y_{ijklnt} + z_{ijnt} \right)$$

as the number of movements made at period t.

Next, in constraint 4, we state that for each period we may have at most one movement, that is, m_t may not exceed the value one.

$$m_t \leq 1, \quad \text{for } t = 1, \ldots, T. \tag{4}$$

In constraint 5, we state that a move can only be made in period t if one has been made in period $t-1$, therefore forcing all moves to be made in the earliest periods.

$$m_t \leq m_{t-1}, \quad \text{for } t = 2, \ldots, T. \tag{5}$$

Constraint 6 asserts that an item is unreleased at instant t if it was unreleased and not moved at instant $t-1$.

$$a_{nt} = a_{n,t-1} - \sum_{i=1}^{W} \sum_{j=1}^{H} x_{ijn,t-1}, \quad \text{for } n = 1, \ldots, N, \quad t = 2, \ldots, T+1. \tag{6}$$

Constraint 7 is used to determine the state of any warehouse position (i, j) at instant t based on the state at instant $t-1$ and possible moves involving position (i, j) made at instant $t-1$.

$$b_{ijnt} = b_{ijn,t-1} + x_{ijn,t-1} + \sum_{k=1}^{W} \sum_{l=1}^{H} (y_{klijn,t-1} - y_{ijkln,t-1}) - z_{ijn,t-1} \tag{7}$$

for $i = 1, \ldots, W, \quad j = 1, \ldots, H, \quad n = 1, \ldots, N, \quad t = 2, \ldots, T+1$.

In the same way, constraint 8 is used to determine the state of delivered items at instant t based on the state at instant $t-1$ and a possible delivery move made at instant $t-1$.

$$c_{nt} = c_{n,t-1} + \sum_{i=1}^{W} \sum_{j=1}^{H} z_{ijn,t-1}, \quad \text{for } n = 1, \ldots, N, \quad t = 2, \ldots, T+1. \tag{8}$$

The next four constraints are related to the order of events (releases and deliveries). Constraint 9 enforces the correct relative order between release events: for an item n released before another item m, the number of periods that n takes to enter the warehouse ($\sum_{t=1}^{T+1} a_{nt}$) is smaller than that value for item m.

$$\sum_{t=1}^{T+1} a_{nt} + 1 \leq \sum_{t=1}^{T+1} a_{mt}, \quad \text{for } n = 1, \ldots, N, \quad m = 1, \ldots, N \text{ such that } R_n < R_m. \tag{9}$$

Likewise, in constraint 10 we enforce the relative order between delivery events, using the number of periods that items are already delivered.

$$\sum_{t=1}^{T+1} c_{nt} \geq \sum_{t=1}^{T+1} c_{mt} + 1, \quad \text{for } n = 1, \ldots, N, \quad m = 1, \ldots, N \text{ such that } D_n < D_m. \tag{10}$$

Constraint 11 imposes the correct order between release and delivery events, by comparing the number of periods that n takes to enter the warehouse to the number of periods before m is delivered.

$$\sum_{t=1}^{T+1} a_{nt} + 1 \leq \sum_{t=1}^{T+1} (1 - c_{mt}), \quad \text{for } n = 1, \ldots, N, \quad m = 1, \ldots, N \text{ such that } R_n < D_m. \tag{11}$$

Constraint 12 is the inverse of the previous constraint, also relating deliveries and releases.

$$\sum_{t=1}^{T+1} (1 - c_{nt}) + 1 \leq \sum_{t=1}^{T+1} a_{mt}, \quad \text{for } n = 1, \ldots, N, \quad m = 1, \ldots, N \text{ such that } D_n < R_m. \tag{12}$$

Finally, we conclude our model by specifying the starting and ending conditions. Constraint 13 sets all items as unreleased at period 1, and constraint 14 forces all items to be delivered at instant $T + 1$.

$$a_{n1} = 1, \quad \text{for } n = 1, \ldots, N. \tag{13}$$

$$\sum_{n=1}^{N} c_{n,T+1} = N. \tag{14}$$

4.5. Solution with a general-purpose MIP solver

The model presented above is valuable as a theoretical reference and formal definition of the SP. However, it is still impractical to obtain optimal solutions of interesting instances, even using state-of-the-art commercial solvers. This is due to the large number of variables and constraints that does not allow concluding the search within an acceptable computational time. We studied the limits of the model with the `Gurobi solver`, version 3.0.1, and present the results in Table 1.

Table 1
Results obtained using the `Gurobi solver` for toy instances within 3600 seconds of CPU time

Items	Width	Height	z^*
3	2	3	6
4	2	4	8
5	2	5	10
6	2	6	14
7	2	7	15
8	2	8	*
9	2	9	*
10	2	10	*

*The optimal solution could not be found within the allowed time.

It is evident that the model cannot be used for practical cases, since it has an extremely low threshold of seven items, for which the optimal solution is found in 3600 seconds. This is probably because the linear relaxation of the model provides a very loose lower bound (equal to zero in most cases) on the number of relocations, leading to a very large branch-and-bound tree to be explored by the MIP solver.

5. Heuristics

Since we are dealing with an NP-complete problem and the exact approach is rather limited, we propose a set of semi-greedy construction heuristics. The heuristics are used to construct solutions, in conjunction with a discrete-event simulation model that enforces the storage and event order constraints (e.g. does not allow moving items that are not at the top of a stack, and does not allow items to be released or delivered out of the corresponding dates). All of the proposed heuristics contain a probabilistic component, so that multiple constructions generally lead to different solutions.

The setting where the heuristics are used is rather complex, as we need to keep track of the state of all the stacks, including the delivery dates of its items, in addition to the queue of items to be released. We call the construction of a complete solution a *simulation*. The end result of a simulation is a solution to the SP, represented as a sequence of movements, as described in Section 2. In a simulation, item release and delivery events are executed in a chronological order. For each release event, the heuristic is used to select a stack for the incoming item. As for delivery events, the simulator first locates the stack where the target item currently is. Then, as long as there are items above the target, the heuristic is used to select a different stack for the topmost item in the stack. Finally, after the necessary reshuffles, the target item is removed from the warehouse.

Every placement decision mentioned above has the following steps: the heuristic evaluates a set of candidate target stacks T according to given criteria, and builds a restricted candidate list (RCL) of stacks that obtained at least a given minimum score; the item being moved is placed in a target stack that is randomly chosen from the RCL with a given probability distribution.

The set of candidate target stacks is computed as follows. For each item being moved, a set T of stacks exists, such that no two stacks in T are *equivalent*. Two stacks are equivalent if and only if

they have the same number of items and, at each height, the items in both stacks have the same due date. To obtain the set of nonequivalent stacks, we modify the set of all stacks in the warehouse, replacing a group of equivalent stacks with only the first stack in the group. This technique allows removing any symmetries in the problem, and can greatly reduce the size of the search space. It is easy to observe that the SP possesses a great amount of symmetries, e.g. when the first item enters the warehouse, it can be put in any of the W stacks, leading to W identical subproblems. Removing these symmetries is crucial in avoiding repetition during the search.

We will now define some notation that will be used for describing the heuristics. We extend the notation defined in Section 2 with information concerning the current solution.

I – set of all items, $I = \{1, \ldots, N\}$
S – set of all stacks, $S = \{1, \ldots, W\}$
o – item (or object) currently being moved
L_i – location of item i, i.e. the stack where it currently is
T – set of nonequivalent candidate target stacks, computed from S in release moves, or from $S \setminus \{L_i\}$ in relocation moves
I_s – set of items currently in stack s, $I_s = \{i \in I : L_i = s\}$
M_s – smallest date at which some item in the stack forcibly has to be moved, i.e. $\min\{D_i : i \in I_s\}$ if $I_s \neq \emptyset$, $+\infty$ otherwise.
RCL – restricted candidate list, from where the target stack for item o is randomly selected (for relocation moves, this list never includes the stack where the item is).

5.1. Conflict Minimization (CM)

This is a very simple heuristic that tries to avoid reshuffles by simply distinguishing between moves that cause inversions and those that do not.

Let X be the set of all noninversion causing stacks, i.e.

$$X = \{s \in T : D_o \leq M_s\}.$$

Then, the RCL in the Conflict Minimization heuristic is given by:

$$\text{RCL} = \begin{cases} X & \text{if } X \neq \emptyset \\ S & \text{otherwise.} \end{cases}$$

For each item o being placed in the warehouse, or relocated, its destination stack is picked from the RCL with uniform distribution.

5.2. Flexibility Optimization (FO)

Another way to interpret M_s is to view it as a measure of stack *flexibility*, that is, the range of item due dates that can be placed in a stack without causing an inversion. The higher M_s, the more flexible the stack is.

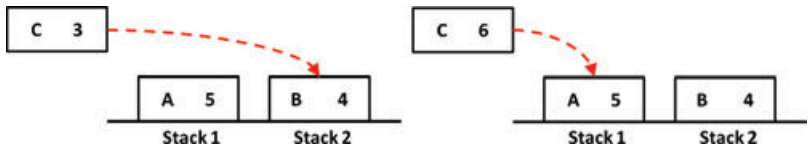

Fig. 2. Flexibility maximization (left) and relocation postponing (right) strategies. Items are placed in noninversion stacks when possible, maximizing flexibility by putting items with close due dates together. When placing an item that forcibly causes an inversion, postponing the forced relocation is beneficial in some cases. The letters A, B, and C represent item names, and the numbers represent their due dates D_A, D_B, and D_C.

Like in the CM heuristic, items are placed without creating new inversions whenever possible. However, the FO heuristic places items with close due dates together, in order to maximize stack flexibility (Fig. 2, left). When inversions are unavoidable, we propose the rule of thumb of postponing the relocation as much as possible (Fig. 2, right); in many cases, this leads to less relocations.

For formalizing these ideas, let us define the scoring function f, taking an item i and a stack s as arguments, as

$$f(i,s) = \begin{cases} K & \text{if } I_s = \emptyset \\ M_s & \text{if } M_s \geq D_i \\ 2K - M_s & \text{if } M_s < D_i \end{cases}$$

where K is defined as the constant $1 + \max\{D\}$. This function maps moves to values, such that noninversion-causing moves (first and second branches) are mapped to lower values, and inversion-causing moves (third branch) are mapped to higher values. In both cases, the function assigns lower values to moves that place item i above items with due dates closer to D_i.

We can now define the RCL as the set of stacks that have the minimum value of f for the item o being moved, i.e.

$$\text{RCL} = \{s \in T : f(o,s) \leq f(o,s'), \forall s' \in T\}.$$

As in CM, the destination stack for any item o being moved is picked from the RCL with uniform distribution.

5.3. *Parameterized Flexibility Optimization (PFO)*

As presented above, the FO heuristic builds an RCL containing stacks with the same value of $f(o,s)$. However, in many cases the RCL will have only one element; in these cases, repeated construction with FO can lead to very similar or identical solutions.

The PFO heuristic builds on FO by considering a second-best candidate when FO's RCL would contain only one element. The second-best stack is chosen with probability ρ, and the best stack is chosen with probability $1 - \rho$, i.e. following a Bernoulli distribution. The parameter ρ controls how likely the second-best candidate is chosen; in particular, if $\rho = 0$, this heuristic is equivalent to FO. The inclusion of second-best stacks in the RCL leads to increased solution diversity, which is exploited when making multiple constructions.

The RCL in PFO is created as follows (using function f defined in Section 5.2):

1. Let $Y = \{s \in T : f(o, s) \leq f(o, s'), \forall s' \in T\}$.
2. Pick a stack $s_1 \in Y$ (randomly chosen with uniform distribution).
3. Then, let $T' = T \setminus \{s_1\}$, and $Z = \{s \in T' : f(o, s) \leq f(o, s'), \forall s' \in T'\}$.
4. Pick a stack $s_2 \in Z$ (randomly chosen with uniform distribution).
5. Finally, let RCL $= \{s_1, s_2\}$, where s_1 is chosen with probability $1 - \rho$ and s_2 with probability ρ.

6. Computational experiment

In order to assess the quality of the heuristics, we performed a computational experiment where all the transactions in the warehouse, corresponding to the release and delivery of a series of items, were simulated using the heuristics for selecting a stack for each of the items being released or relocated.

The heuristics presented in this paper were extensively tested on a set of 12 challenging instances, two for each number of stacks $W \in \{2, 3, 4, 10, 20, 40\}$. All instances have 1000 items ($N = 1000$). In addition to the heuristics presented here, a completely random placement heuristic (RAND) was also part of the experiment, to serve as a basis of comparison for the other heuristics. For the PFO heuristic, the value used for the probability parameter ρ was 0.01 (determined empirically). On each of the test instances, 10,000 independent simulations were executed with each heuristic, making a total of 480,000 simulations; in each of them, the number of relocations was recorded. The system used for the experiment has a Intel Core2-Duo T7500 processor at 2.2 GHz, 3 GB of memory, and is running Windows Vista. All the algorithms mentioned in the paper were implemented in `Python` version 2.6.

The large set of results mentioned above is used to compute the statistics of interest. As mentioned in Section 1, the size of the search space forces the use of incomplete search methods, so we will base our statistics on the multiple simulation method. The main indicator that we want to estimate is the expected number of relocations, for each of the heuristics, if we execute a given number of simulations and select the best of them. The maximum, minimum, median, and quartiles to be expected, with respect to the number of simulations, were also computed and plotted. For computing these statistics, we used a bootstrap method (Moore and McCabe, 2005) where the main sample consists of the 10,000 values obtained from the simulations of one heuristic in one instance. This set was then resampled 10,000 times using a resampling scheme without replacement. To compute the best solution obtained after a given number of simulations $\theta \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$, the value μ_x^θ, representing the minimum of resample x (or subsample minimum) with size θ, was taken and recorded for each resample.

The bootstrap method was chosen for several reasons. It allows us to obtain many possible versions of the data from a single sample, therefore it is very useful when it is computationally costly to obtain enough samples to estimate the distribution of a given statistic. In our case, since we are studying the distribution of the minimum number of relocations after a certain number of simulations, we have a large pool of data for each instance and heuristic pair, and we take many resamples that represent possible cases when we make a number θ of simulations. From each of those resamples (in this case each resample has size θ) we take the minimum, and so estimate the sampling distribution of the subsample minimum. The method is straightforward and allows us

to estimate arbitrarily complex statistics. The number of resamples traditionally used is 1000, but increasing the number of resamples can reduce the effect of random sampling errors, therefore we decided to take 10,000 resamples.

Figures 3 and 4 show the observed bootstrap distributions of the subsample minimum μ_x^θ for all combinations of instance, heuristic, and subsample size.

The graphics show a number of interesting facts. First of all, we were probably expecting the RAND heuristic, which represents the naive brute-force approach, to be by far worse than any other heuristic, since it does not possess any form of "intelligence." However, the graphics surprisingly reveal that it was the best heuristic for both two-stack instances after 1024 simulations. Another surprising fact is that, despite being far from the best, the RAND heuristic performed better than CM in the remaining instances, with the only exception being the 40-stack instances.

If we consider all instances with more than two stacks, we see that RAND generally beats CM, but is still far from the performance of the flexibility heuristics (FO and PFO). This indicates that simply avoiding inversions without looking into stack flexibility is worse than a brute-force method, but if we also take flexibility into account the results are much better than brute-force, except for instances with two stacks.

There is also an interesting observation about the variability in the results of the two flexibility heuristics (FO and PFO). The inclusion of second-best stacks in PFO's RCL allows it to produce more solutions than FO, consequently having a higher variability in its results. In Fig. 3, this is especially visible in the two-stack instances, where the increased diversity allows PFO to reach better results. However, this advantage is apparently negligible and brings no benefit in the remaining instances, where the performance of the two heuristics is roughly equivalent, with only a small advantage for PFO on instances with 10 or more stacks.

Numerical results for subsample size $\theta = 1024$ are presented in Table 2. The table shows, for each heuristic and instance, the expected value of the minimum, average, and maximum of the subsample minimum, i.e. the best solution after 1024 simulations. These values estimate the best, most likely, and worst scenarios, respectively, giving us an idea of the variability of the best solution obtained with each heuristic.

We can observe very small amplitude in FO's interval for most instances, indicating that the heuristic's performance is very consistent after 1024 simulations. PFO shows a larger amplitude, but slightly better average performance in most cases, which means that, despite being less reliable, it can achieve better results as the number of constructions increases. In addition, this suggests that more solid and possibly better results can be achieved with PFO by increasing the subsample size.

The table also confirms that, at the end of 1024 simulations, the RAND heuristic is better than CM in all but the 40-stack instances. This exception may be related to having few necessary inversions on these instances, thus being generally better to avoid them.

The heuristics presented in this paper are greedy construction heuristics that use very simple rules. Due to their simplicity, the computational time required to solve even large-scale instances is reduced (less than 1 second in the system used). This allows us to quickly obtain solutions of reasonable quality, and can be exploited by e.g. repeating simulations to take advantage of their probabilistic component, or by using the heuristics as a guide for a depth—first tree search algorithm. Additionally, this simplicity eases understanding, verification, and predictability of the heuristics' behavior.

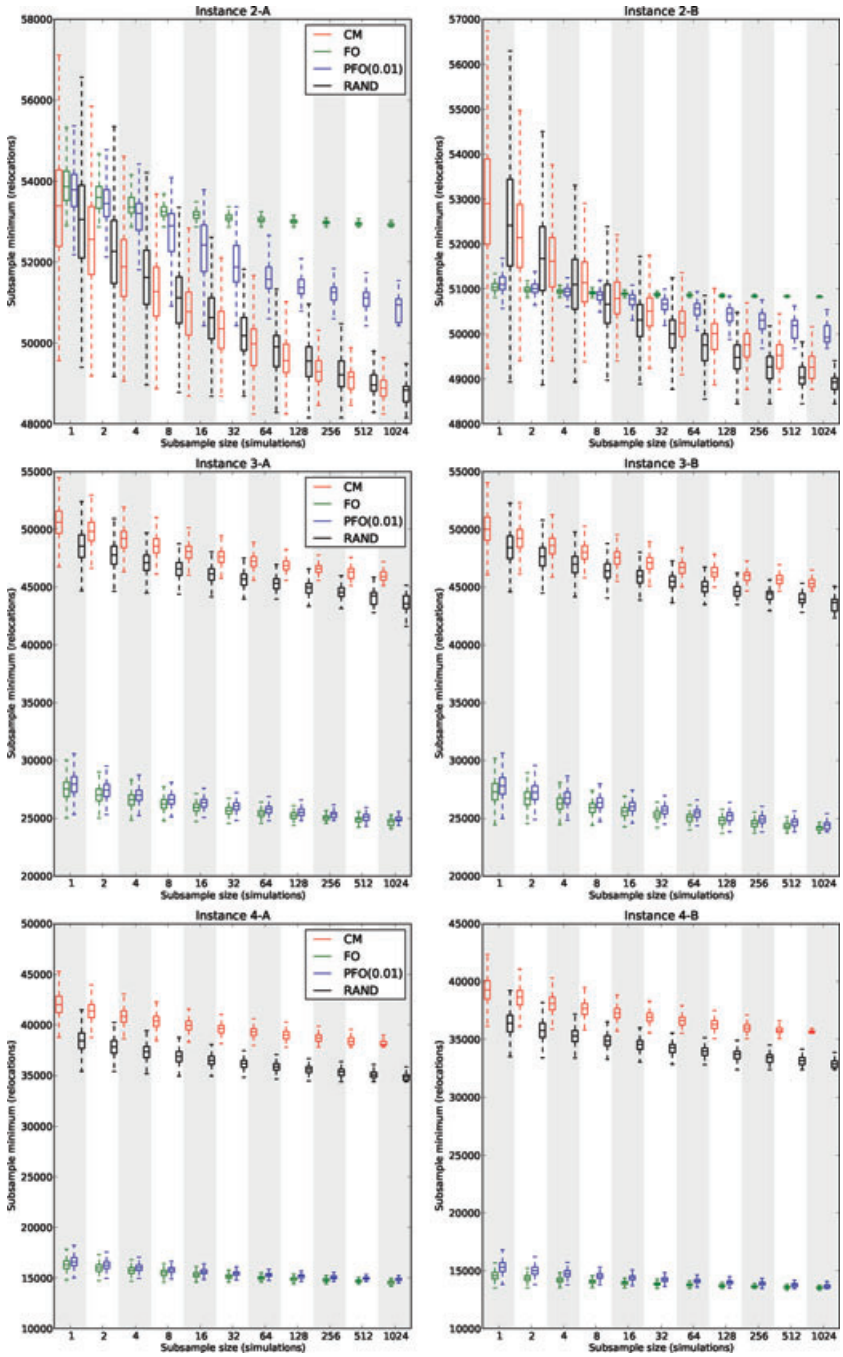

Fig. 3. Boxplots of the bootstrap distributions (10,000 resamples) of the best solution (resample minimum) after θ simulations (subsample size). Results for increasing number of stacks (2, 3, and 4) are shown from top to bottom.

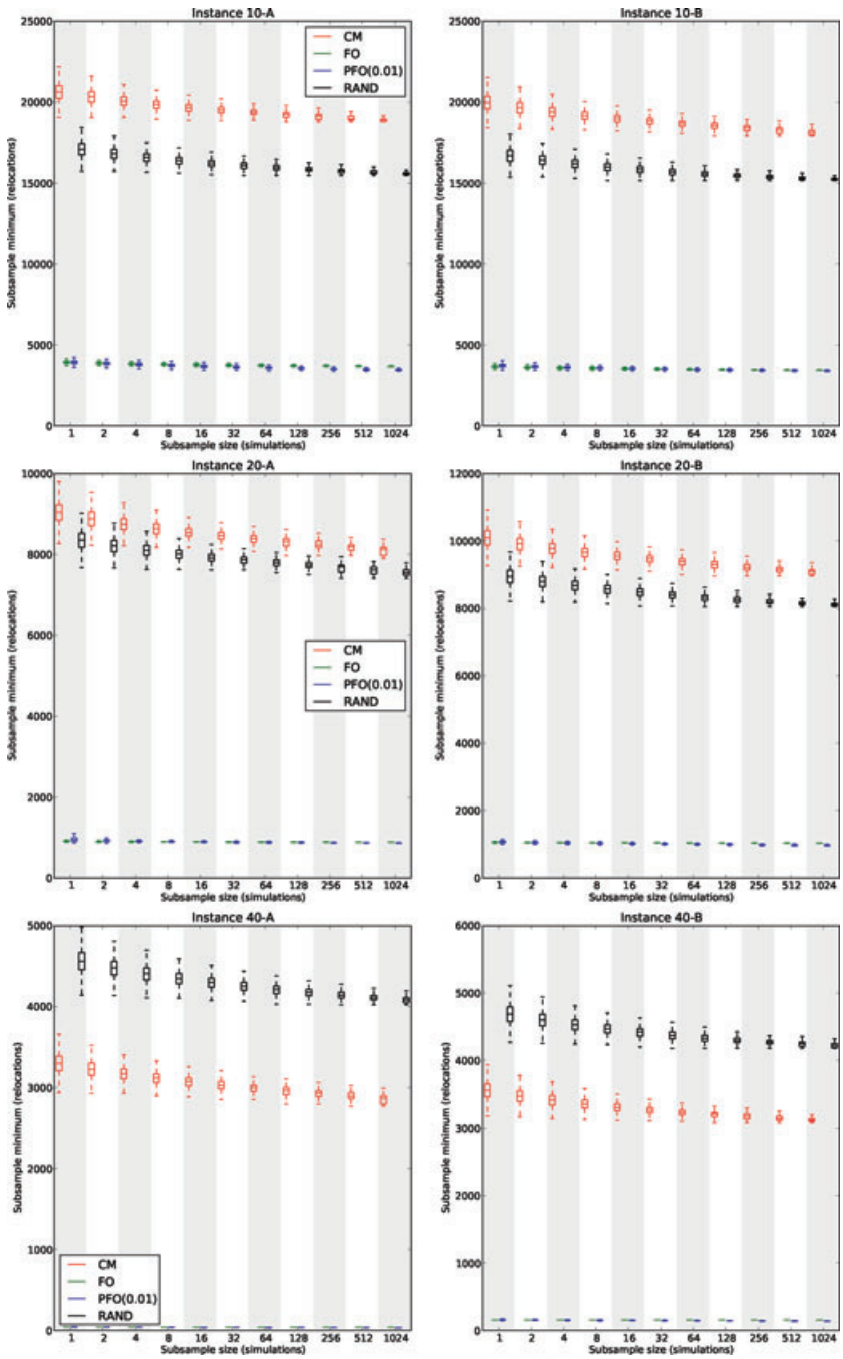

Fig. 4. Boxplots of the bootstrap distributions (10,000 resamples) of the best solution (resample minimum) after θ simulations (subsample size). Results for increasing number of stacks (10, 20, and 40) are shown from top to bottom.

Table 2
Minimum, average, and maximum of the best solution after 1024 simulations, obtained with bootstrap without replacement (10,000 resamples)

Instance	RAND			CM		
	min	*avg*	*max*	*min*	*avg*	*max*
2 stacks - A	48,153	48,764.12	49,896	48,240	48,854.17	49,850
2 stacks - B	48,447	48,882.95	49,691	48,767	49,251.49	50,157
3 stacks - A	41,610	43,567.45	45,178	45,126	45,931.16	47,184
3 stacks - B	42,330	43,515.76	45,080	44,672	45,358.99	46,468
4 stacks - A	34,402	34,846.84	35,843	37,798	38,193.11	39,164
4 stacks - B	32,391	32,858.65	33,867	35,082	35,640.65	36,589
10 stacks - A	15,231	15,564.27	15,921	18,121	18,834.99	19,315
10 stacks - B	15,159	15,239.04	15,547	17,899	18,126.28	18,636
20 stacks - A	7412	7547.02	7788	7898	8097.58	8375
20 stacks - B	7907	8101.57	8286	8964	9084.90	9384
40 stacks - A	4021	4084.53	4207	2773	2858.70	2993
40 stacks - B	4185	4225.26	4322	3081	3126.23	3234
Instance	FO			PFO(0.01)		
	min	*avg*	*max*	*min*	*avg*	*max*
2 stacks - A	52,867	52,922.14	53,028	50,420	50,856.81	51,541
2 stacks - B	50,807	50,824.97	50,858	48,974	49,901.30	50,537
3 stacks - A	24,084	24,660.63	25,319	24,300	24,876.96	25,719
3 stacks - B	23,724	24,151.24	24,948	23,832	24,380.79	25,411
4 stacks - A	14,156	14,556.20	14,959	14,500	14,837.15	15,278
4 stacks - B	13,086	13,483.66	13,753	13,453	13,655.12	14,086
10 stacks - A	3643	3667.77	3718	3378	3442.08	3574
10 stacks - B	3418	3435.40	3473	3358	3389.99	3473
20 stacks - A	883	886.90	891	861	866.69	884
20 stacks - B	1032	1036.63	1044	954	968.57	1003
40 stacks - A	41	41.00	41	34	35.40	37
40 stacks - B	156	156.00	156	133	137.89	145

7. Conclusion

We presented a hard combinatorial optimization problem with direct application in real life. We developed a mathematical formulation of the problem, which proved to be unusable for any real-life case. Therefore, to tackle the problem, we proposed three different heuristics for placement decisions, which can be used within a simulation model to construct complete solutions. The main contribution of this work is an in-depth study of the behavior of the heuristics, analyzed through the bootstrap distribution of the best solution found after a given number of simulations.

The first heuristic studied in this paper was named Conflict Minimization (CM), and simply places items in a random noninversion stack, if possible; otherwise it chooses a random target from all available stacks. The second heuristic is FO, which tries to preserve stack flexibility by putting items with close due dates together, while also avoiding inversion moves. On unavoidable inversions, the item is placed in the stack where it will be moved latest, postponing the relocation move for as long as possible. The last heuristic is PFO, a variant of FO that attempts to increase the diversity

of its solutions by including a second-best stack in the RCL when FO would consider only one possibility.

Our experimental results show that the PFO heuristic performs slightly better than FO in most instances. The difference is greatest in the two-stack instances, where PFO greatly outperforms FO, possibly due to the need of counterintuitive moves (not considered by FO) to obtain better results.

We also observed that a brute-force method can (generally) outperform the CM heuristic, but is worse than a flexibility approach, indicating that solely avoiding inversions without looking into flexibility does not bring any advantage over brute-force, but using both criteria we can achieve consistently better results.

Directions for future work include testing different methods of solving the SP, namely variants of branch-and-bound, and possibly other metaheuristics. Also, since the problem naturally lends itself to a recursive definition, developing a dynamic programming formulation for solving it is another interesting research possibility.

Acknowledgements

We would like to thank the contributions of Prof. Cláudia Neves, from the University of Aveiro, Portugal, and Prof. Mikio Kubo, from the Tokyo University of Marine Science and Technology, Japan. We also thank the three anonymous referees for their constructive comments on a previous version of this paper.

The work was partially supported by PhD grant SFRH/BD/66075/2009 from the Portuguese Foundation for Science and Technology (FCT).

References

Avriel, M., Penn, M., Shpirer, N., 2000. Container ship stowage problem: complexity and connection to the coloring of circle graphs. *Discrete Applied Mathematics* 103, 271–279.

Avriel, M., Penn, M., Shiper, N., Witteboon, S., 1998. Stowage planning for container ships to reduce the number of shifts. *Annals of Operations Research* 76, 55–71.

Caserta, M., Voß, S., Sniedovich, M., 2009. Applying the corridor method to a blocks relocation problem. *OR Spectrum* 33, 915–929.

Dekker, R., Voogd, P., van Asperen, E., 2006. Advanced methods for container stacking. *OR Spectrum* 28, 563–586.

Hartmann, S., 2004. A general framework for scheduling equipment and manpower at container terminals. *OR Spectrum* 26, 51–74.

Kim, K. H., Hong, G.-P., 2006. A heuristic rule for relocating blocks. *Computers and Operations Research* 33, 940–954.

Law, A. M., 2006. *Simulation Modeling and Analysis* (4th edn). McGraw-Hill Publishing, Tucson, AZ.

Moore, D. S., McCabe, G. P., 2005. *Introduction to the Practice of Statistics*, chapter 14 (5th edn). W. H. Freeman, New York.

Unger, W., 1988. On the k-colouring of circle-graphs. In *Proceedings of the 5th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '88, Springer-Verlag, London, pp. 61–72.

Unger, W., 1992. The complexity of colouring circle graphs. In Finkel, A., Jantzen, M. (Eds.), *STACS 92*, volume 577 of *Lecture Notes in Computer Science*. Springer, Berlin/Heidelberg, pp. 389–400.