Ann Oper Res (2013) 203:371–388
DOI 10.1007/s10479-012-1186-2

Tree search for the stacking problem

Rui Rei · João Pedro Pedroso

Published online: 20 July 2012

Abstract The stacking problem is a hard combinatorial optimization problem with high practical interest in, for example, steel storage or container port operations. In this problem, a set of items is stored in a warehouse for a period of time, and a crane is used to place them in a limited number of stacks. Since the entrance and exit of items occurs in an arbitrary order, items may have to be relocated in order to reach and deliver other items below them. The objective of the problem is to find a feasible sequence of movements that delivers all items, while minimizing the total number of movements.

We study the scalability of an exact approach to this problem, and propose two heuristic methods to solve it approximately. The two heuristic approaches are a multiple simulation algorithm using semi-greedy construction heuristics, and a stochastic best-first tree search algorithm. The two methods are compared in a set of challenging instances, revealing a superior performance of the tree search approach in most cases.

Keywords Stacking problem · Optimization · Tree search

1 Introduction

The Stacking Problem (SP) originated in the context of a Japanese steel making company. In the steel industry, the production sequence is tightly bound to the quality of the final product, and thus respects very strict rules. This sequence is usually very different from that of customer orders, thus 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 which can only access the top item of each stack. Therefore, in order to deliver an item that is not at the top of a stack, all of the items above it have to be moved to other stacks.

The objective is to minimize the number of movements that a stacking crane has to make in order to place each incoming item in a stack and later deliver it, after first relocating all of the items that were potentially placed above it in the mean time. The SP consists of a series

R. Rei · J.P. Pedroso (✉)
INESC Porto and Faculdade de Ciências, Universidade do Porto, Rua do Campo Alegre,
4169-007 Porto, Portugal
e-mail: rui.rei@dcc.fc.up.pt

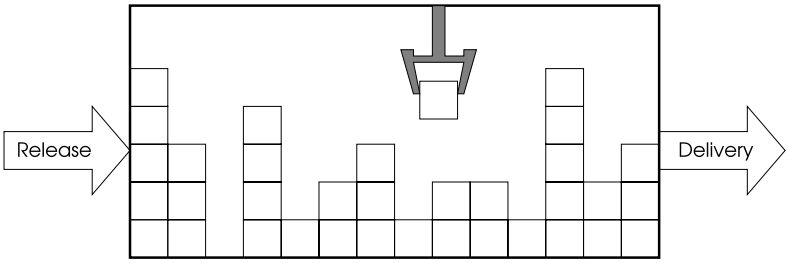

Fig. 1 A warehouse where items are stored in stacks, using a stacking crane that can only handle one item at a time

of placement decisions for a set of items with known warehouse entrance and exit dates (see Fig. 1), denoted by *release* and *due* dates, respectively. If two or more of these *events* occur on the same date, they can be processed in an arbitrary order.

Inside the warehouse, the items are manipulated using a stacking crane and they can either be placed in a vacant position, or on top of another item (forming a stack, *i.e.*, a last-in first-out queue). 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, since items may have to be delivered when they are not at the top of a stack. The relocation of upper items is called *reshuffling*, and is necessary when *due date inversions* exist. A due date 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 before D_j, item i will have to be placed in another stack to allow for item j to be delivered.

It is thus in our interest to avoid moves that cause inversions. However, since this is not always possible, the aim is to find strategies that keep relocations to a minimum. This can be achieved not only through careful placement decisions for incoming or reshuffled items, but also through another technique called *remarshalling*. Remarshalling consists of moving items in between release or delivery events, making unforced relocations (as opposed to reshuffling) with the objective of reorganizing the warehouse and decreasing the total number of relocations required afterwards.

The SP can have several additional constraints, such as imposing a maximum limit for the stack height (capacitated SP), non-instantaneous crane movements, allowing late deliveries with penalties, or a crane movement cost matrix. These additional constraints naturally result in more realistic scenarios; however, the variant studied in this paper assumes that there is no limit on stack sizes (uncapacitated SP), crane movements are instantaneous, and all deliveries must be completed on time.

Stacking problems arise in a variety of practical situations, making their study important for many areas of business or industry. Perhaps the most important and well-known of these areas are container port operations (Dekker et al. 2007; Hartmann 2004) and ship stowage planning (Avriel et al. 1998, 2000). Stacking is also important in the steel industry as a means of storage for finished products. The solution of such stacking problems is economically important, since there may be operating costs, either directly due to the operation of the cranes, or due to operation delays.

A problem related to the SP is the container ship stowage problem (CSSP), where a ship that calls at many ports must transport containers with known source and destination ports. The containers are stored in a stacking bay inside the ship, where they can only be accessed from above, one at a time. When a container that has port i as its destination has to be removed from the ship, any containers above it with destinations further away from

port i must be removed from the ship together with the target container. They then have to be reloaded onto the ship, in potentially different positions. This temporary removal and reloading operation is known as *shifting*.

The authors in Avriel et al. (1998) present an integer programming model to optimally solve the CSSP. However, the applicability of this model is very limited because of the large number of binary variables and constraints. For this reason, these authors developed a heuristic procedure to obtain approximate solutions.

Another similar problem 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. It is important to observe 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 in fact a subproblem of the SP.

The BRP may be applied to ship loading operations at container terminals (each ship and its respective containers are then an instance of the problem), where both the ship and the terminal are expensive resources. In such cases, minimizing the time required for loading operations can lead to significant savings.

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

In Caserta et al. (2011), an algorithm for the BRP based on the corridor method is presented. The corridor method is a hybrid metaheuristic combining mathematical programming techniques with heuristics. Its main idea is to apply an exact algorithm to smaller areas of the search space, and move the focus of the search to different areas using neighborhood rules similar to those in a local search. A dynamic programming formulation is used in that paper to solve small subproblems exactly.

This paper presents and compares two heuristic methods for tackling the SP. For the sake of simplicity, and because the number of remarshalling movements available at any given time is usually very large, these movements are not considered in either approach. The two methods are:

- The *Multiple Simulation* (MS) method, initially described in Rei et al. (2008), Rei and Pedroso (2009), is based on a simulation model of the warehouse, which is operated with the help of a semi-greedy construction heuristic to choose the destination stack when releasing or reshuffling items. Since the heuristic contains a probabilistic component, repeated construction will usually lead to different solutions. In order to exploit this variability and thus explore the search space, multiple simulations are executed and the best solution found is tracked and recorded.
- The *Stochastic Tree Search* (STS) algorithm borrows ideas from branch-and-bound and the artificial intelligence literature. Tree search is a general algorithmic framework that can be applied to a wide range of problems (see, for instance, Pedroso and Kubo 2010; Ruml 2001, 2002). It can be used in a complete search (leading to optimal solutions), or as a heuristic if the search is incomplete (leading to approximate solutions). Techniques will be described that allow the tree search to obtain high quality solutions after a short period of time, hence making it suitable for incomplete search.

The paper is structured as follows. Section 2 presents a formal description of the stacking problem followed by a study of its complexity class. The methods proposed in this paper

are presented in Sect. 3, which also describes a symmetry breaking technique used therein. Multiple simulation is described in Sect. 3.1. Section 3.2 explains the tree search approach, including a study on the use of this algorithm in complete search. Section 4 presents the computational experiment and a discussion of its results. Finally, Sect. 5 provides some concluding remarks and directions for future research.

2 Problem description and complexity

Consider the problem of stacking N items in a warehouse with W stacks, $W < N$, and no height limit. Let $R \in \mathbb{R}^N$ be the list of item 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 item due dates (*i.e.*, the times at which items will leave the warehouse). It is assumed that due dates are greater than release dates for every item, *i.e.*, $R_i < D_i, \forall i = 1 \ldots N$.

A movement consists of taking a single item, either coming into the warehouse or from the top of a stack, and placing it on top of another stack, or delivering it. 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 represented as s_D.

A description of the different types of movements can be found below:

- a *release movement* occurs when an item enters the warehouse on its release date and is placed at the top of a stack;
- a *delivery move* occurs when an item, located at the top of a stack, is removed from the warehouse on its delivery date;
- a *relocation move* occurs when an item is shifted from the top of a 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 feasible solution is a sequence of movements that respects item release and delivery date constraints, and the last-in-first-out policy of stacks. Therefore, with the exception of releases, only the items at the top of the stacks can be moved. The objective of the SP is to find a solution M^* with minimum number of movements (equivalently, relocations), *i.e.*,

$$M^* = \operatorname*{argmin}_{M \in \Gamma} |M|,$$

where Γ represents the set of all feasible movement sequences and $|\cdot|$ represents the length of a movement sequence.

Let us now focus on the complexity of this problem, starting with the definition of a decision version of the SP. Given an SP instance with N items and W stacks, the *r-relocation SP* consists of finding whether there is a movement sequence with r relocations that allows delivering the N items using the W stacks.

It is first shown that the r-relocation SP belongs to the complexity class NP, by presenting a polynomial verifier (Algorithm 1) for this problem. Given a movement list M, an instance I, and an integer r, the loop in Algorithm 1 iterates at most $2N + r + 1$ times, corresponding to $2N$ releases and deliveries (fixed), and at most r relocations. Therefore, the r-relocation SP has a verifier of order $O(N)$.

Next is the introduction of overlap graphs. Two intervals on the real line *overlap* if their intersection is nonempty but neither one of them properly contains the other. An *interval overlap graph* is a graph $G = (V, E)$, such that each vertex corresponds to an interval, and two vertices are adjacent if and only if the corresponding intervals overlap (Golumbic 2004).

Algorithm 1: Polynomial verifier for the r-relocation SP

Data: movement list M, SP instance I, integer r
Result: *True* if M is a valid solution of I containing $\leq r$ relocations

```
1  S ← set of W stacks in the warehouse, initially empty
2  z ← 0          /* relocation counter /*
3  for j = 1, ..., |M| do
4      (i, s, s') ← M_j (j-th movement in M)
5      if movement (i, s, s') is invalid then
6          return False
7      move item i from stack s to stack s'
8      if s ≠ s_R ∧ s' ≠ s_D then                    /* relocation */
9          z ← z + 1
10         if z > r then
11             return False
12 return True;
```

In Gavril (1973), overlap graphs are proven to be equivalent to *circle graphs*; a circle graph is a graph whose vertices can be represented as chords in a circle, such that two vertices are connected by an edge if and only if their corresponding chords intersect. Given a graph $G = (V, E)$, the k-coloring problem consists of finding whether a partition of V into k subsets exists, such that for all $(v_1, v_2) \in E$, v_1 and v_2 belong to different subsets in the partition.

It is now demonstrated that the W-coloring problem of overlap/circle graphs is equivalent to the zero-relocation SP with W stacks. Given an overlap graph $G = (V, E)$, the corresponding set of items in the SP is obtained by assigning an item i to each interval $(a, b) \in V$, with release date $R_i = a$ and due date $D_i = b$. It should be noted that the reverse transformation can be achieved by creating an interval (R_i, D_i) from each item in the SP.

If two intervals overlap, the corresponding vertices in G are connected, and they cannot have the same color in a valid coloring of G. Likewise, the corresponding items in the SP cannot be placed in the same stack in a zero-relocation sequence, because the first item to enter the warehouse is delivered while the second item is still inside the warehouse (*i.e.*, if the two items were placed in the same stack, a relocation would be necessary). If the intervals do not overlap, then the corresponding vertices in G are not connected by an edge and may have the same color in a valid coloring of G. In the SP, the corresponding items can be placed in the same stack, because either they are never simultaneously inside the warehouse, or the first item to enter the warehouse is delivered after the second item has been delivered.

Therefore, an overlap/circle graph G is W-colorable if and only if there is a movement sequence with zero relocations in the corresponding SP, using W stacks. In Unger (1988), the k-coloring problem of overlap graphs was shown to be NP-complete for any fixed $k \geq 4$. For $k = 3$, an $O(|V| \log |V|)$ algorithm has been presented in Unger (1992), and 2-coloring of graphs can be done in polynomial time using an algorithm for recognizing bipartite graphs. Consequently, the zero-relocation SP is NP-complete for any fixed number of stacks $W \geq 4$, implying that the general r-relocation SP is also NP-complete (it contains the zero-relocation SP and belongs to NP), and the optimization version of the SP (*i.e.*, finding a sequence of minimum length) is NP-hard.

Note that the decision problem can be solved in polynomial time for $W = 2, 3$ only if the optimal number of relocations is zero. If a solution with zero relocations does not exist, then the relation to the coloring of circle graphs no longer holds. The polynomial algorithms for $W = 2, 3$ are applicable to the coloring of circle graphs (equivalently, 0-relocation SP), but there is no evidence of the existence of polynomial algorithms for even the 1-relocation SP. For these reasons, solving large-scale instances with $W = 2, 3$ in polynomial time remains an open problem when the optimal number of relocations is greater than zero.

When solving the SP with $W = 2, 3$, existing algorithms for the coloring problem could be executed before going into more complex approaches. This combination would result in polynomial time when the optimal solution has zero relocations, and the more complex algorithms would only be executed otherwise.

3 Methodology

This section describes the two proposed approaches for solving the SP approximately. Section 3.1 presents the Multiple Simulation method, followed by the Stochastic Tree Search algorithm in Sect. 3.2.

Before describing the proposed algorithms, a technique is presented that can greatly reduce the size of the search space, and thus help reduce redundancy in the search. It is easy to observe that the SP possesses a large amount of symmetries. The simplest example of this is, when the first item enters the warehouse, it can be put on any of the W stacks, leading to W identical subproblems. Removing these symmetries is crucial to avoid repetition during the search. Whenever a choice for a movement is to be made, instead of considering all stacks as possible destinations, only the set of *nonequivalent* stacks is considered. Two stacks are *equivalent* 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, the set of all stacks in the warehouse is modified by replacing a group of equivalent stacks with only the first stack in the group.

Note that applying this technique for the first item being stored reduces the size of the search space by a factor of W, since at the beginning all of the stacks in the warehouse are equivalent. This symmetry breaking method was used in both algorithms described below.

3.1 Multiple simulation approach

The system composed of the warehouse and the item release and delivery events involved in the problem can be classified as a discrete event system, *i.e.*, a system where state transitions occur instantaneously and only at specific points in time. Discrete event simulation is a well-known tool used to model and study these types of systems. It can be used in a wide range of different applications, from academic studies of queuing networks to industrial applications of job shop modeling; see, *e.g.*, Law (2006).

Given the combinatorial and sequence-dependent nature of the Stacking Problem, using discrete-event simulation seems appropriate, as it is computationally far less expensive than a complete search method such as branch-and-bound or dynamic programming. Although complete methods deliver exact solutions, they quickly become unwieldy due to the exponential number of system states (Bellman's curse of dimensionality). Local search (LS) procedures have the useful property of finding local optima under a given neighborhood structure. However, intuition tells us that the effectiveness of LS-based methods on this problem will be greatly reduced. This is due to the difficulty in finding a neighborhood structure that keeps neighbor solutions relatively "alike": changing a single movement

causes the sequence to become infeasible from that point onward in most cases, requiring the remainder of the solution to be reconstructed from scratch. Nevertheless, LS methods cannot be discarded without the support of empirical evidence (this is suggested as future research in Sect. 5).

The *Multiple Simulation* (MS) method is based on a discrete-event simulation model that enforces the storage and event order constraints (*e.g.*, it does not allow items to be moved that are not at the top of a stack, and it does not allow items to be released or delivered outside of the corresponding dates). The simulation tool is used in conjunction with a construction heuristic to produce a sequence of movements specifying a solution. The heuristic is used to select a stack for each item being released or relocated. The simulator processes the instance's event schedule in chronological order; for release events, the heuristic is called with all stacks in the warehouse as possible destinations; for delivery events, any items above the target are reshuffled first (with all stacks except the source as possible destinations) and the target item is finally removed from the warehouse.

Note that if the construction heuristic is not deterministic, making multiple constructions usually leads to different solutions. Exploiting this variability is the primary idea of MS; running several simulations on the same instance and picking the best solution. In order to accelerate the search, the number of relocations λ of the best found solution (initially ∞) is used to interrupt simulations when they can no longer lead to better solutions. To do this, the cut value λ is checked by the simulation procedure after processing each event; if the current number of relocations plus the number of inversions in the warehouse is not smaller than λ, the simulation is interrupted. Pseudo-code for a simulation run is presented in Algorithm 2. Whenever the best-found solution is improved, the cut value is updated to the number of relocations of the new solution, thus tightening the upper bound for future simulations. Pseudo-code for an MS algorithm is shown in Algorithm 3.

The heuristic function for choosing a stack for each item being moved plays a central role in the algorithm. The construction heuristic used in this paper is called Flexibility Optimization (FO) and will now be explained. At any given time, m_s can be defined as the movement date of stack s, that is, the minimum due date of items located in stack s. If a stack s is empty, then $m_s = \infty$ by definition.

The movement date of a stack can also be seen as a measure of its *flexibility*, in the sense that a larger m_s means that more items with different due dates can be placed in s without causing inversions. The definition of $m_s = \infty$ for empty s is also coherent with our notion of flexibility, *i.e.*, stack s can be said to have infinite flexibility. The idea of the FO heuristic is to make choices that preserve stack flexibility at each placement decision, intuitively leaving more or better options for future decisions.

The cost function f will now be defined. This function takes an item i and a stack s as arguments and maps the movement to a real number $f(i,s)$ as

$$f(i,s) = \begin{cases} L & \text{if } m_s = \infty, \\ m_s & \text{if } m_s \geq D_i, \\ 2L - m_s & \text{if } m_s < D_i, \end{cases}$$

where L is defined as the constant $L = 1 + \max_j\{D_j\}$. We can interpret $f(i,s)$ as a cost for the hypothetical movement of item i onto stack s. The function maps moves to values, such that non-inversion-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 higher values to moves that place item i above items with due dates closer to D_i. It is important to note that, after a movement has been executed, the movement dates (m_s) of the source and destination stacks are updated to reflect their new contents.

Algorithm 2: Single simulation run

Data: SP instance I, placement heuristic h, cut value λ
Result: A sequence of movements

$M \leftarrow$ empty sequence of movements
$S \leftarrow$ set of stacks in the warehouse, initially empty
foreach event in event schedule of instance I **do**
  $i \leftarrow$ item being moved
  if event is a release **then**
    $S' \leftarrow$ set of nonequivalent stacks in S
    $s \leftarrow$ a stack for item i chosen from S' using heuristic h
    move item i to stack s
    append move (i, s_R, s) to M
  else /* delivery event */
    $s \leftarrow$ stack where item i is located
    while item i is not delivered **do**
      $t \leftarrow$ item at the top of stack s
      if $D_t = D_i$ **then** /* same due date, deliver top */
        $s' \leftarrow s_D$
      else /* reshuffle necessary */
        $S' \leftarrow$ set of nonequivalent stacks in $S \setminus \{s\}$
        $s' \leftarrow$ a stack for item t chosen from S' using heuristic h
      move item t to stack s'
      append move (t, s, s') to M
  if number of relocations in M + number of inversions in $S \geq \lambda$ **then**
    return no solution
return M

Algorithm 3: Multiple simulation algorithm

Data: SP instance I, placement heuristic h, maximum CPU time t
Result: A sequence of movements

$M^* \leftarrow$ empty move sequence /* best solution */
$\lambda \leftarrow \infty$ /* upper bound */
while CPU time $< t$ **do**
  $M \leftarrow$ run a simulation on I using heuristic h and cut value λ
  if number of relocations in $M < \lambda$ **then**
    $M^* \leftarrow M$
    $\lambda \leftarrow$ number of relocations in M
return M^*

Given an item i and a set of possible destination stacks T, the FO heuristic constructs a *restricted candidate list* (RCL) of stacks that have the minimum value of f, *i.e.*, RCL =

$\{s \in T\colon\ f(i,s) \leq f(i,s'),\ \forall s' \in T\}$, and randomly selects a stack from the RCL as the destination of item i.

The cost function f is dependent on the relation between the due date D_i and the movement date m_s of a stack, and (considering the same item) it may take the same value for different stacks in a warehouse. Therefore, for a given decision, the RCL may contain more than one stack. Consequently, making multiple constructions using FO can lead to different solutions, and such variety can then by exploited by the MS method.

3.2 Tree search approach

In the multiple simulation setup, improvement is achieved solely through brute force exploitation of the heuristic's variability, *i.e.*, running many simulations and extracting the best result. Moreover, nothing prevents the same solutions from being generated in different simulations, as these are independent. One possibility for improving the quality of the search consists of using a tree search method, which implicitly prevents the same solutions from being produced.

Construction with a placement heuristic makes it possible to obtain a solution in a very short time. For each item, the decision of putting it in a particular stack is made only once, and the alternative decisions are not analyzed. This method can be extended in order to conduct a complete search of the solution space by dividing the problem into several subproblems, where the same item is placed on each of the possible stacks. This branching operation can be repeated for each variable, creating a tree with all the possible alternatives. The process of complete enumeration without further refinements only works for very small instances, as the size of the tree increases exponentially with the size of the instance. Therefore, even using state-of-the-art software and hardware, this method cannot be applied to medium- or large-scale instances.

Considering that instances with practical interest have hundreds of items and thus at least as many points of decision, exploration of the whole tree is hopeless. One way of alleviating this problem and reducing the size of the tree is to apply the idea used in MS to cut simulations: if a partial solution has a relocations and b inversions, the branch containing that partial solution can be discarded if $a + b > \lambda$, where λ is the current upper bound. Note that, since the objective function is non-decreasing with time, this will only discard branches which are surely sub-optimal. Even though this bounding scheme is rather simplistic, it leads to a dramatic reduction in the effort required to (implicitly) explore the tree. However, it does not eliminate the tree's exponential growth, which makes complete search unusable in anything but toy instances.

This section will analyze how a basic tree exploration scheme can be enhanced, in order to first explore the parts of the tree where the most promising solutions are expected to be found. Later it will be shown that these enhancements allow finding good solutions even at very early stages of the search, while leaving the convergence properties of tree search untouched. This means that the search will slowly but surely progress towards global optima as more time is available to solve the problem.

3.2.1 Exploration of the tree

There are several ways of exploring the nodes of a search tree; some of these will be analyzed next. A generic algorithm for exploring a tree, making use of a priority queue, is presented in Algorithm 4. It should be noted that what is stored in each tree node is the data defining a subproblem and the (possibly partial) solution that led to it. In the case of the SP, this consists

Algorithm 4: A generic tree search method

```
1  create empty queue Q for open nodes
2  push root node into Q
3  λ ← ∞                                              /* upper bound */
4  while Q is not empty do
5      μ ← pop a node from Q
6      i ← item to be placed next in node μ
7      foreach s in the set of nonequivalent stacks where i can be placed do
8          μ′ ← a copy of node μ
9          move item i to stack s in node μ′
10         a ← number of relocations in μ′
11         b ← number of inversions in μ′
12         if a + b < λ then
13             execute all possible deliveries from the top of stacks in μ′
14             if μ′ is a leaf node then
15                 check if μ′ contains a better solution and update λ
16             else
17                 push node μ′ into Q
```

of the construction history up to the current time (*i.e.*, a movement list), the current state of the warehouse, and the remainder of the event schedule (*i.e.*, which items are awaiting release or delivery). For generality, the *push* and *pop* operations on the priority queue were purposely left undefined in the pseudo-code. The actual functioning of these operations is dependent on the search scheme used, as will be explained later.

With regard to the order of exploration of nodes, the classical method is depth-first order (where the *push* and *pop* operations work on the same end of the queue); the search keeps descending through child nodes until a leaf (containing a complete solution) is found. Using the FO heuristic function to drive the search, the first child to be explored at each node is the same as in a simple construction. After hitting the bottom of the tree, the search backtracks, resuming from the most recent node it has not completely explored yet. If the search is interrupted, it may remain at the bottom of the tree; this can be unsatisfactory, as alternatives to early decisions are not considered.

We propose the use of a *best-first* scheme, employing a criterion to score open tree nodes, providing an evaluation of how promising each node is in an attempt to improve the search path. If the criterion used to score nodes is adequate, the search can improve considerably since good solutions can, in many cases, be found earlier than in depth-first search.

Using this scheme, complete solutions are not guaranteed in the case of interruption. Since the search path is controlled by the node scoring function, the search may not be able to reach any leaf node within the allowed CPU time. To solve this, *diving* is used to quickly reach complete solutions. This has the additional advantage of providing upper bounds which enhance pruning, therefore accelerating the search. Diving is commonly used, for example, in mixed-integer programming, where the solution of a linear programming relaxation is used to select a path in the branch-and-bound search tree (see *e.g.*, Pochet and Wolsey (2006), for a description of several variants of diving in the context of mixed-integer programming).

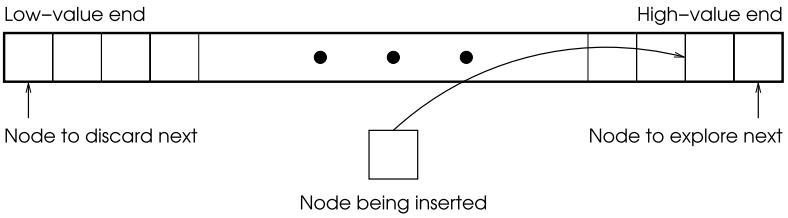

Fig. 2 The queue used to hold open nodes in the STS algorithm

Best-first search requires the use of an auxiliary priority queue to hold open nodes. However, in some cases, the memory required for the queue grows beyond control, possibly becoming problematic. To circumvent this problem, a queue size limit may be imposed, thereby effectively making the search incomplete, since nodes may have to be discarded. The loss of quality in the incomplete search can be minimized by making careful decisions concerning which branches should be explored.

3.2.2 Stochastic tree search

Since large-scale instances are going to be tackled, our goal is to develop a tree search algorithm which can quickly find high-quality solutions; therefore, the algorithm must be suitable for interruption of the search, based on computational time or queue size limitations.

The proposed method is *Stochastic Tree Search* (STS), a probabilistic tree search algorithm, using best-first exploration order. To solve the potential problem of not reaching a single leaf node when interrupted, a variant of diving is used to quickly find complete solutions. This consists of completing the partial solution contained in the node in the same way a simulation is executed (Algorithm 2), also using the FO heuristic. As in a simulation, the outcome of a dive is a complete solution, which serves multiple purposes; it quickly provides upper bounds on the number of relocations, and it is used to assess how promising the node which led to that solution is.

We refer to the *value of a node* as the negative of the number of relocations on the solution produced during the dive step on the given node. When inserting newly generated child nodes into the queue, the value is used to determine the position where the node will be placed. Looking at Fig. 2, it can be seen that high-value nodes (*i.e.*, low number of relocations on their dive solution) are placed near the right end of the queue, where the next node to explore is taken from. Low-value nodes are placed at the left end of the queue, where the next node to discard is taken from.

Clearly, the diving step is critical, as the objective function value obtained guides the order of exploration of nodes, and it is also used to select which branches are discarded when the queue is full. When two or more nodes have the same value, their exploration order is randomly chosen. Additionally, since the semi-greedy FO heuristic is being used, the value of a node may be different depending on the random choices made during the dive. Therefore, the order of exploration of nodes includes two sources of randomness, which grant the algorithm its stochastic character.

This tree search arrangement is suitable for incomplete search since the diving step allows finding complete solutions quickly. Furthermore, the search is not redundant, since the symmetry breaking technique and the branching operation promote the exploration of different partial solutions. The best-first scheme is not likely to become trapped in a specific

Table 1 Results obtained using the complete STS algorithm for toy instances within 3600 s of CPU time

Items	Mean relocations	Mean CPU time
10	0.0	0.00
11	0.0	0.00
12	0.0	0.00
13	2.0	0.13
14	1.4	0.07
15	1.8	0.24
16	1.2	0.14
17	2.2	0.51
18	4.6	252.99
19	5.2	857.86
20	???	> 3600.00

part of the tree (as would happen with depth-first search), and focuses the search on the most promising parts of the solution space.

Node values are not obtained from the partial solutions themselves, but from complete solutions derived from them during dives. It should be noted that the accuracy of a node value increases as the node's partial solution is closer to completion, since the number of decisions remaining in the evaluation dive is smaller. In case a less-complete solution has a higher value than a nearly-complete one, the former will be explored first since it is more promising (though its evaluation is possibly less accurate). Other than guiding the search to promising regions of the search space, node values are also useful if a queue size limit is imposed. In this case, when the queue is full, they allow informed discarding decisions to be made.

3.2.3 Complete search

In this section, the application of the STS algorithm is studied in the context of complete search. An experiment was performed using the STS algorithm with no limit on queue size, in order to estimate the size of the largest instances that can be solved exactly within reasonable computational time.

For each number of items between 10 and 20, five instances were generated with 2 stacks and no height limit (uncapacitated SP). For each of the instances, the algorithm was run with a maximum CPU time of one hour. Table 1 presents the results of this test, averaging the CPU time and optimal solution of all instances of the same size. The table shows that STS can be used exactly for instances with a maximum size of approximately 19 items under the specified conditions. The table also reveals an exponential growth in the time used by the tree search, indicating that it is not possible to use this method to solve large-scale instances, such as the ones in our computational experiment, within an acceptable time.

4 Computational experiment

The number of stacks in real-life instances can vary between a hundred and several thousands, and the number of items directly depends on the intended planning horizon. For instance, when planning for 24 hours in a container yard, the number of items can easily reach several thousands.

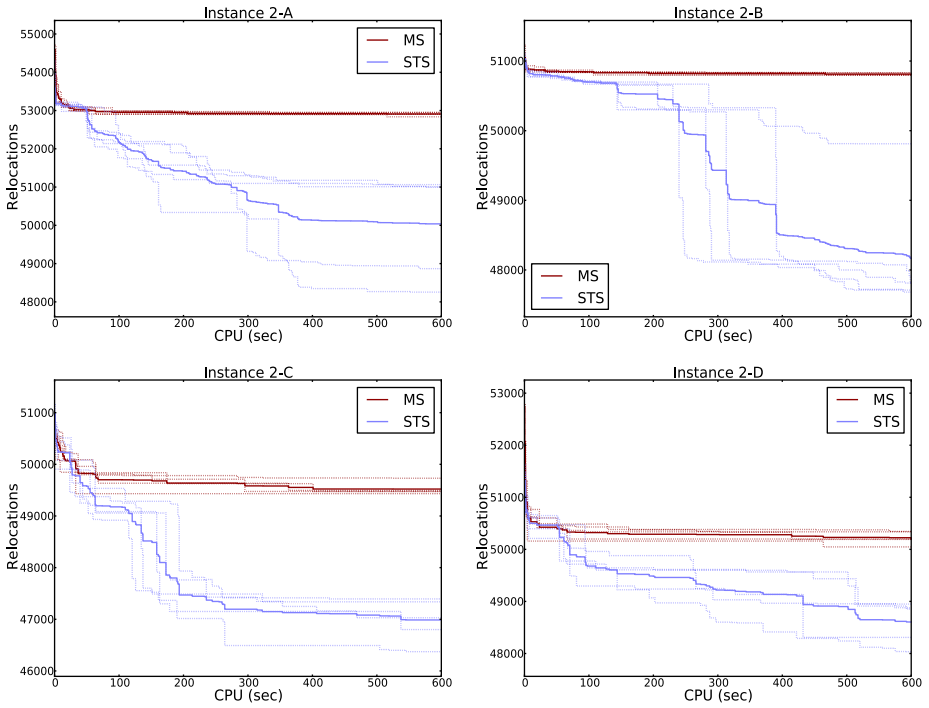

Fig. 3 Results for the four 2-stack instances showing the value of the upper bound with respect to elapsed CPU time

Since there are no standard benchmark instances for this problem, and we have no access to real data, we generated a set of instances with several sizes and with a varying degree of overlap for the periods that items remain in the warehouse. As the difficulty of an instance increases when the number of stacks is smaller, the testbed concentrates on a low number of stacks. It consists of a set of 24 instances, divided into 6 groups of 4 instances, each group having a number of stacks $W \in \{2, 3, 4, 10, 20, 40\}$. All instances have $N = 1000$ items, and there is no height limit in the warehouse. These instances, and the ones used for the complete search experiment in Sect. 3.2.3, are available in Rei and Pedroso (2012).

For each problem instance, a series of 5 independent runs of the MS and STS algorithms was executed, each of which running the respective algorithm for 600 CPU seconds. The platform used to run the experiment was an Intel Core2-duo at 2.2 GHz with 3 GB of memory, running Windows Vista. Both algorithms were implemented in the Python programming language, version 2.6.5.

The experiment was set up to study the evolution of the best solution found with respect to CPU time. The relatively short time allowed is motivated by the need for responsiveness of the algorithms in practical situations, where they are typically embedded in real-time systems that must be able to quickly react to changes in data.

The results of the experiment are shown in Figs. 3–8. Each figure shows two sets of dashed lines, that represent the behavior of the algorithms in each of the 5 runs, and two solid lines that represent the average behavior of the 5 runs for the given instance.

There are many observations that can be made from these results. Taking a quick look into the figures, it can be seen that STS is better than MS in most cases, although the two

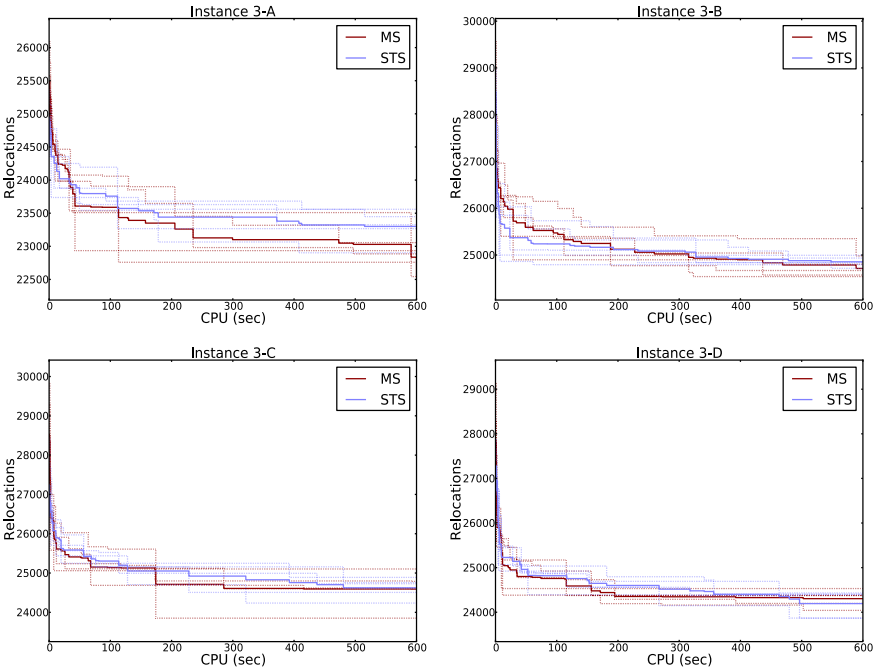

Fig. 4 Results for the four 3-stack instances showing the value of the upper bound with respect to elapsed CPU time

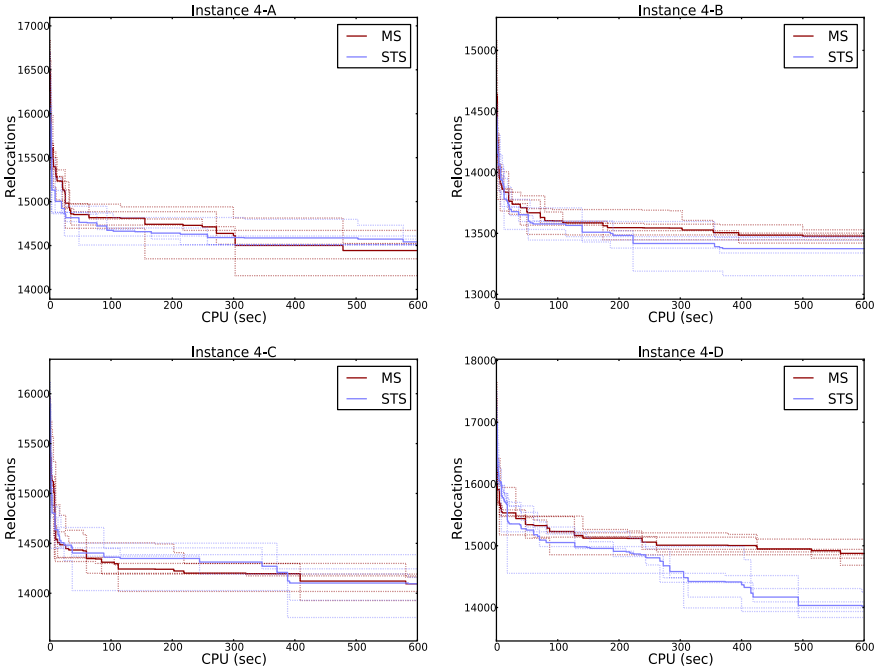

Fig. 5 Results for the four 4-stack instances showing the value of the upper bound with respect to elapsed CPU time

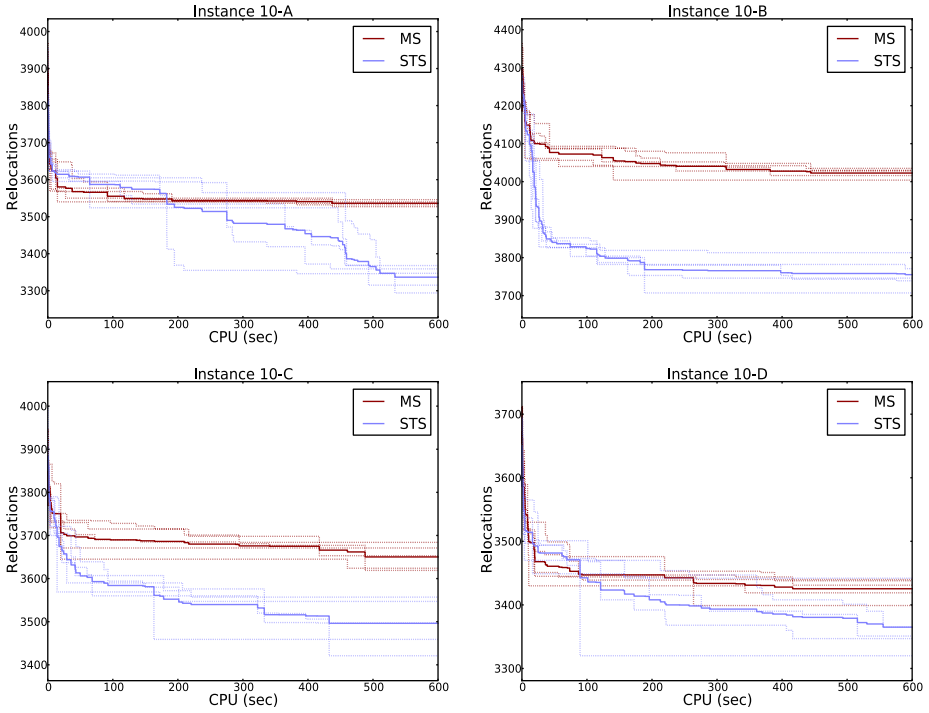

Fig. 6 Results for the four 10-stack instances showing the value of the upper bound with respect to elapsed CPU time

algorithms have a comparable performance in the 3-stack and 4-stack instances, and an exception occurs in instance 20-A. Taking a closer look, it can also be noted that the variability achieved by MS is much lower than that of STS, with the exception of the 3-stack and 4-stack instances. This is most noticeable in the 40-stack instances, where it can be seen that the initial solution of MS is not improved in any of the five runs, possibly because the construction heuristic is always making the same choices and therefore it generates the same, or very similar solutions. Note that, although the placement heuristic has a random component, it is only used to choose among stacks having the same (maximum) score according to its criterion. So, it is possible that the random tie-breaking is never called upon if the RCL has exactly one element in every choice.

The performance of the two methods is in fact related to the variety of solutions they can produce. Depending on the instance, a probabilistic placement heuristic such as FO may produce more or less ties, and thus a corresponding increase or decrease in variability for the given instance. While both methods inherit the construction heuristic's variability, STS may produce additional (different) solutions due to the implicit mutual exclusion of branches in the tree. Therefore, STS has at least as much variability as MS on any instance. However, it achieves this at the expense of construction speed: it reaches complete solutions at a slower pace than MS. Additionally, for instances where the construction heuristic produces many different solutions by itself, STS is not allowed to exploit its additional branching-related variability due to the limited run time and its slower construction speed, leading to a similar or even worse performance than MS in such cases. Extending the run time will reduce the

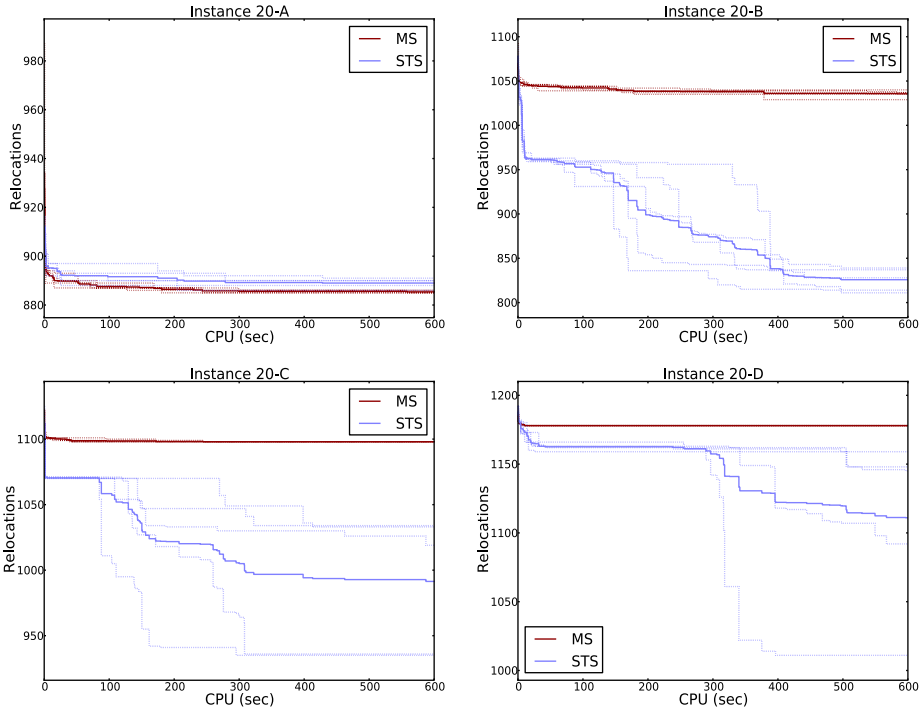

Fig. 7 Results for the four 20-stack instances showing the value of the upper bound with respect to elapsed CPU time

impact of the tree management overhead and allow STS to take advantage of its greater exploration capabilities, generally leading to better results.

The observations in the previous paragraph explain why the two methods performed similarly in the 3-stack and 4-stack instances. In these instances, the variety created by FO was sufficient to allow MS to keep exploring different solutions throughout the given period. This, together with MS's faster construction speed, resulted in a performance similar to STS's, and even allowed it to slightly outperform STS in instances 3-A, 3-D, and 4-A. We believe the same reasons are behind the results for instance 20-A.

With these observations, it can be concluded that the STS algorithm is superior, even within limited time, although not in all cases. Depending on the instance and the construction heuristic, the brute-force method of repeating simulations can achieve better results in short run times because it can construct more solutions per second (no tree management overhead). However, in the cases where the heuristic cannot generate enough different solutions to keep improving over the whole 600 seconds, the additional variety introduced by the branching operation is crucial to achieve better results. The fact that any intermediate state is only searched once is also very important, especially in the 40-stack instances, where MS is clearly constructing repeated solutions. While STS can fully exploit any extra time given, MS does not benefit as much from longer running periods, and is highly dependent on the quality and variability of the construction heuristic used.

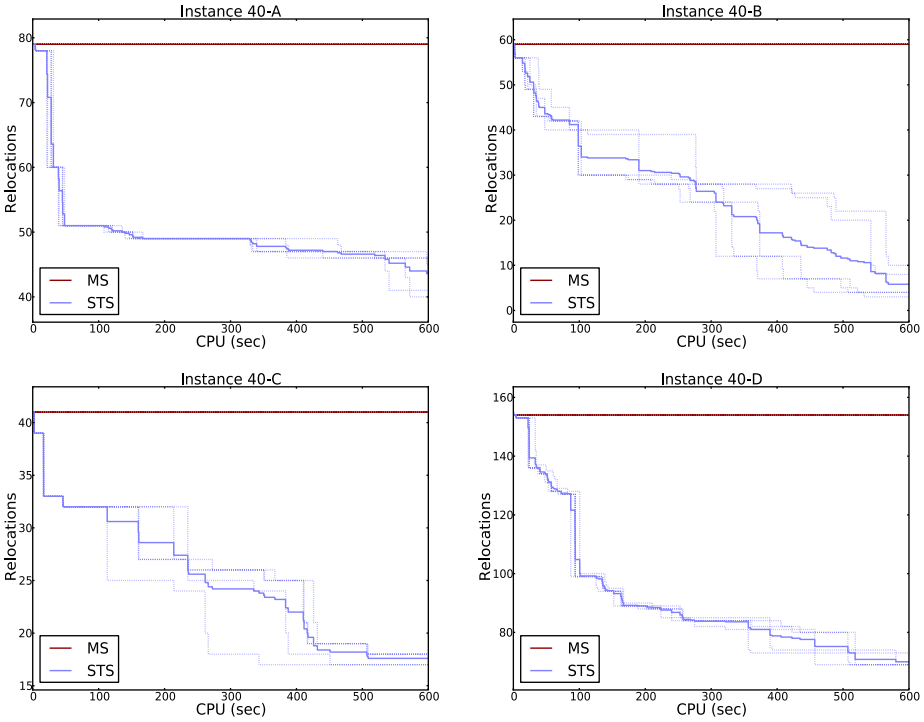

Fig. 8 Results for the four 40-stack instances showing the value of the upper bound with respect to elapsed CPU time

5 Conclusion

We presented a hard combinatorial optimization problem—the Stacking Problem—with practical relevance. The contributions of this paper include a formal description of the SP, a study of its complexity class, and the application of an exact method to solve it. Due to the scalability limitations of exact approaches, we propose two heuristic methods for tackling large-scale instances. The first method, MS, is a fast, repeated semi-greedy construction procedure that relies heavily on the quality of a construction heuristic. The second method, STS, is a probabilistic best-first tree search procedure that uses a simple bounding scheme which allows pruning of some branches of the tree.

To solve some deficiencies in the classical best-first search scheme, STS uses a *diving* step to obtain an upper bound on each node, based on greedy solution completion. This upper bound provides an estimate of how good a node is, and determines the order by which nodes are explored. An additional benefit of diving is that it ensures that complete solutions are found quickly, thus making STS suitable for incomplete search.

Our efforts to adapt the tree search algorithm to partial search have yielded satisfactory results as we can, in general, improve on the quality of the MS method. Using the same heuristic as the MS algorithm, STS achieves better solutions in most cases, even within a relatively short time. However, the full assessment of the methods' performance under real-life conditions is still required.

Directions for future research include introducing ideas from different areas into the STS algorithm, such as local search methods, or artificial intelligence techniques to create new

heuristics that change their behavior according to knowledge gained during the tree search. The study of the capacitated SP and other more realistic variants is also an interesting research topic.

Acknowledgements We would like to thank Prof. Mikio Kubo, from the Tokyo University of Marine Science and Technology, Japan, for his important contributions. We would also like to thank three anonymous reviewers for their constructive comments on previous versions of this paper. The work was partially funded by PhD grant SFRH/BD/66075/2009 from the Portuguese Foundation for Science and Technology (FCT).

References

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

Avriel, M., Penn, M., & Shpirer, N. (2000). Container ship stowage problem: complexity and connection to the coloring of circle graphs. *Discrete Applied Mathematics*, *103*(1–3), 271–279.

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

Dekker, R., Voogd, P., & Asperen, E. (2007). Advanced methods for container stacking. In K. H. Kim & H.-O. Günther (Eds.), *Container terminals and cargo systems* (pp. 131–154). Berlin: Springer.

Gavril, F. (1973). Algorithms for a maximum clique and a maximum independent set of a circle graph. *Networks*, *3*, 261–273.

Golumbic, M. C. (2004). *Annals of discrete mathematics*: *Algorithmic graph theory and perfect graphs*. Amsterdam: Elsevier.

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 ed.). New York: McGraw-Hill.

Pedroso, J. P., & Kubo, M. (2010). Heuristics and exact methods for number partitioning. *European Journal of Operational Research*, *202*, 73–81.

Pochet, Y., & Wolsey, L. A. (2006). *Production planning by mixed integer programming*. Berlin: Springer.

Rei, R. J., Kubo, M., & Pedroso, J. P. (2008). Simulation-based optimization for steel stacking. In H. A. Le Thi, P. Bouvry, & T. Pham Dinh (Eds.), *Communications in computer and information science: Vol. 14. Modelling, computation and optimization in information systems and management sciences* (pp. 254–263). Berlin: Springer.

Rei, R. J., & Pedroso, J. P. (2009). Heuristic search for the stacking problem. In A. Viana et al. (Eds.), *EUMe2009, The European chapter on metaheuristics' workshop on debating the future: new areas of application and innovative approaches*, Porto, Portugal (pp. 109–114).

Rei, R. J., & Pedroso, J. P. (2012). Stacking problem instances and instance generator. Internet repository, version 1.0. http://www.dcc.fc.up.pt/~jpp/code/stacking.

Ruml, W. (2001). Stochastic tree search: where to put the randomness? In H. H. Hoos & T. G. Stützle (Eds.), *Proceedings of the IJCAI-01 workshop on stochastic search* (pp. 43–47).

Ruml, W. (2002). *Adaptive tree search*. PhD thesis. Cambridge: Harvard University.

Unger, W. (1988). On the k-colouring of circle-graphs. In R. Cori & M. Wirsing (Eds.), *Lecture notes in computer science: Vol. 294. STACS 88* (pp. 61–72). Berlin: Springer.

Unger, W. (1992). The complexity of colouring circle graphs. In A. Finkel & M. Jantzen (Eds.), *Lecture notes in computer science: Vol. 577. STACS 92* (pp. 389–400). Berlin: Springer.