Tree Search and Simulation

João Pedro Pedroso and Rui Rei

Abstract This chapter presents a general methodology for embodying simulation as part of a tree search procedure, as a technique for solving practical problems in combinatorial optimization. Target problems are either difficult to express as mixed integer optimization models, or have models which provide rather loose bounds; in both cases, traditional, exact methods typically fail. The idea then is to have tree search instantiating part of the variables in a systematic way, and for each particular instantiation—i.e., a node in the search tree—resort to a simulation for assigning values to the remaining variables; then, use the outcome of the simulation for evaluating that node in the tree. This method has been used with considerable success in gameplaying, but has received very limited attention as a tool for optimization. Nevertheless, it has great potential, either as a way for improving known heuristics or as an alternative to metaheuristics. We depart from repeated, randomized simulation based on problem-specific heuristics for applications in scheduling, logistics, and packing, and show how the systematic search in a tree improves the results that can be obtained.

1 Introduction

Tree search and branch-and-bound variants are among the most powerful search methods in combinatorial optimization. How to direct the search through the tree, in terms of the selection of a node, the selection of a variable within a node, and the selection of a value to assign to that variable, are key factors for performance. Whereas in some applications these choices are relatively straightforward, in other cases it is very difficult, mostly due to the absence of good bounds. In this work we will focus on applications for which the outcome of a decision is very difficult to

J.P. Pedroso · R. Rei (✉)
INESC TEC and DCC-FCUP, Porto, Portugal
e-mail: rui.rei@dcc.fc.up.pt

J.P. Pedroso
e-mail: jpp@fc.up.pt

M. Mujica Mota et al. (eds.), *Applied Simulation and Optimization*,
DOI 10.1007/978-3-319-15033-8_4

assess before having a complete solution, therefore requiring the simulation of the whole construction process to probe into the quality of the decision.

Designing effective methods under these circumstances involves decisions that overcome the main weaknesses of tree search: not reaching a leaf node in a limited amount of time; unbounded growth of the queue of unexplored nodes; and most importantly, getting trapped in a particular, limited part of the tree. We will develop on methods for this, based on good heuristics for constructing a solution; tree search may be seen as an enhancement of these heuristics which, in the limit case, may completely explore the search space.

The applications covered in this work are the following:

1. Number partitioning—an easy problem to formulate, which will allow us to clearly state the important aspects of tree search and simulation in this context (Sect. 3).
2. Stacking—a hard problem involving the choice of a stack to place an item in such a way that the number of item relocations is minimal (Sect. 4).
3. Recursive circle packing—a variant of circle packing where annular items can either be placed inside a rectangular container or inside other items (Sect. 5).

2 Background

2.1 Tree Search

Tree search is a method for systematically exploring the search space, with the aim of finding the best solution appearing therein. In its simplest form, all solutions are enumerated and verified, hence taking exponential time with respect to the number of variables. In optimization problems for which there is an appropriate mathematical programming formulation, it is usually easy to find bounds on the solutions that can be obtained from a node, which allows the search tree to be pruned without loss of optimality. For a minimization problem, if the lower bound in a given node is greater than the value of a known solution (i.e., greater than an upper bound on the optimum), then the tree can be safely pruned at this node. This is the standard procedure in branch-and-bound (BB) [1, 2]. Literature on BB is ample, as the method is fully general and can be applied in widely diverse areas. For a sample of recent applications of BB methods, see, e.g., [3–6].

For the problems we are dealing with here, formulations are very loose, rendering the provided bounds very ineffective and leading to very little or no pruning at all. This implies that, except for toy instances, exploration of the whole tree is unreasonable, either due to time limitations or because the size of the tree would grow unacceptably large. Consequently, the best solution found may not be optimal because the search space has not been fully explored. In this context, tree search may be used as an *approximative* algorithm.

There are several ways for exploring the nodes of a search tree. In *uninformed search* there is no use of information concerning the value of a node during tree exploration. For example, in *breadth-first search* all nodes in a level of the tree (i.e., nodes that are equally distant from the root) are explored before proceeding to the next level. In *depth-first search* (DFS) each node is expanded down to the deepest level of the tree, where a node with no expansion—i.e., a *leaf*—is found; then search backtracks until a node with unexplored children is found, which is again expanded until reaching a leaf, and so on, until the whole tree is explored. As only one path from the root to a leaf has to be stored at any given time, DFS has modest memory requirements. Problem-specific heuristics may be used in conjunction with DFS for deciding the order of exploration of each node's children; as this information is only considered locally at each node, this is usually called *partially informed search*. On *informed search*, a set of open nodes is; the most common variant is *best-first search*, which selects the next node to expand based on problem-specific knowledge. To this end, an *evaluation function* is used which conveys information about the worth of each open node, and the one with the highest rating from all open nodes is selected at each iteration.

When good guiding heuristics exist, DFS is usually very effective. When compared to greedy construction based on such heuristics, DFS allows for substantial improvements; furthermore, these improvements can be obtained very quickly due to the simplicity of DFS, which imposes an almost nonexistent overhead on the greedy construction algorithm. For situations where the exploration of the full tree is expected to be possible in a reasonable time, DFS is usually an appropriate choice. However, this is not the case for most practical problems. For sufficiently large trees, DFS suffers the problem of being unable to recover from poor decisions taken at the beginning of the search.

For trees with high branching factor, *iterative broadening*, proposed in [7], attempts to overcome the deficiencies of DFS by running a sequence of restricted depth-first searches. Each restricted DFS considers a larger subset of the tree than the previous: the first iteration examines the heuristically preferred node, the second iteration examines the two top-ranked children of each node, and so on; in a tree of depth d, at iteration k, iterative broadening visits k^{d-1} leaves.

Best-first search tries to overcome the deficiencies of DFS by considering, at each iteration, nodes from different levels in the tree. A seminal example is the $A*$ algorithm for the shortest path problem [8], where heuristic information that never overestimates the cost of the best solution reachable from a node is used to evaluate it. Best-first search suffers the problem of requiring exponential space, thus becoming impractical in many situations.

A related algorithm taking only linear space is *iterative deepening* $A*$ (IDA*): each iteration is a DFS modified to use heuristic evaluation as in A*, and a limit on the heuristic value to interrupt the search; the interruption threshold is increased at each iteration [9]. In practice, for many combinatorial optimization problems IDA* visits too many internal nodes before reaching its first leaf; this is mainly due to the underestimation provided by the heuristics being substantially different from the best value that can be reached from each node. Besides, IDA* does not cope well with the fact that in many optimization problems every node has a different heuristic value, only the best of which is expanded at each iteration. In combinatorial optimization problems the depth of the search tree is bounded, and reaching a leaf is usually inexpensive; this is not exploited in IDA*.

The most widely used tools for solving combinatorial problems that can be formulated as mathematical programming models are mixed integer linear programming (MIP or MILP) solvers. These usually incorporate state-of-the-art tools in terms of pruning the search tree and adding cuts to the model, in a black-box solver targeted at obtaining the best performance in the widest range of problems possible. Still, deficiencies similar to those of DFS are often observed in practice, as the solvers may be stuck in the search for very long periods. Methods for overcoming this are a current trend, following the observation of heavy-tailed phenomena in satisfiability and constraint satisfaction [10]. The basic idea is to execute the algorithm for a short time limit, possibly restarting it from with some changes, usually with an increased time limit until a feasible solution is found. This has been recently addressed in [11], where variability of the solutions on which the MIP solver is trapped, for different initial random start conditions, is exploited. The proposed method consists in making a number of short runs with randomized initial conditions, bet on the most promising run, and bring it to completion, in an approach named *bet-and-run*. Diversified runs are produced in several fronts: exchanging rows and columns in the input instance, perturbing parameters of the MIP solver, and changing coefficients in the objective function. The choice of the candidate to bring to completion is based on 11 criteria, the most important of which are the number of open nodes, the lower bound improvement with respect to the root node, and number of integer-infeasible variables among all open nodes. Results are reported for a set of benchmark instances, showing that bet-and-run can lead to significant speedups.

We propose to exploit variability in the search in a rather different manner, by making a *dive* from each open node until reaching a leaf, and using its outcome in the evaluation of the dive's starting node. Dive results are also backpropagated up to the root node. Each of these dives corresponds to a descent in the tree, possibly in a randomized way, which in the context of Monte Carlo Tree Search—the subject of the next section—is called a *simulation*. The two terms are used interchangeably in the remainder of this chapter. In the problems dealt with in this work, simulations always lead to feasible solutions; hence, our method provides both an *anytime algorithm* and, for small instances, complete search.

2.2 Monte Carlo Tree Search: State of the Art

Monte Carlo Tree Search (MCTS) is a method for exploring the search tree and exploiting its most promising regions. Although the idea of combining Monte Carlo evaluation with tree search had been studied before (see, e.g., [12, 13]), it was not until recently—with the appearance of MCTS—that it received greater scientific attention. Coulom [14] proposed the initial version of MCTS and applied it with considerable success to the game of Go (9×9 board). Gameplaying is still the area where the algorithm and its many variants are most commonly used [15]. In this context, MCTS has the aim of finding the most favorable decision at each step, by taking random samples from the decision space and valuating nodes of the tree according to the results of those samples. MCTS has had a particularly strong impact on games, where its application has led to the best gameplaying software, most notably for the game of Go [16, 17]; but it has also been applied on artificial intelligence approaches for other domains that can be represented as trees of sequential decisions and for planning problems (see [18] for a comprehensive survey).

There are, however, very few publications on combinatorial optimization; some results have been provided for general mixed integer optimization [19], but to the best of our knowledge, there are no reports of its application for solving specific optimization problems. This possibility will be illustrated, with a detailed implementation and results, in the following sections. In the remainder of this section we describe the basic algorithm that will be used as the foundation for the three applications presented.

MCTS is an iterative procedure in which a search tree is asymmetrically constructed, attempting to expand the tree toward its most promising parts while balancing exploitation of known good branches with exploration of seemingly unrewarding branches. The algorithm is based on the idea of Monte Carlo evaluation, i.e., the reward associated with a particular node can be estimated from the results of random simulations started from that node. Each node keeps track of the number of simulations started from its state as well as their outcomes, and these data are used to produce an estimate value for the node when deciding how to expand the tree. Each iteration of MCTS can be divided into the following four steps:

1. *Selection*: starting from the root node, select the child node which currently looks more "promising". This is done recursively until a node n which has not yet been fully expanded (i.e., some of its children have not yet been generated) is reached. The definition of promising is one of the key aspects determining the performance of MCTS. In plain MCTS, average win rate is used for node selection. The UCT algorithm [20] provides an enhancement to this simple rule, by posing the selection problem at each node as a multiarmed bandit problem, and then using the UCB1 policy [21] to achieve an optimal bound on regret. In UCT, the score or utility $U(n)$ of node n is defined as

$$U(n) = X(n) + E(n), \tag{1}$$

   where $X(n)$ is the exploitation utility associated with n, and $E(n)$ is the exploration utility of n. At each node, the child with maximum $U(n)$ is selected, until an unexpanded node is reached.
   In gameplaying, $X(n)$ is typically taken to be the average reward of simulations run from n. Later in this section, we propose an alternative expression for $X(n)$ that is more suitable for optimization.

The exploration component is normally defined as

$$E(n) = c\sqrt{\frac{\ln s_{p(n)}}{s_n}}, \tag{2}$$

where c is an exploration parameter (theoretically equal to $\sqrt{2}$), $s_{p(n)}$ is the number of simulations done under the parent node $p(n)$, and s_n is the number of simulations done under the child node n (i.e., simulations started from n or any node in the subtree under n). This expression is designed such that exploration progressively gives way to exploitation, although all siblings are eventually selected if enough iterations are allowed. This means that the search cannot become permanently trapped in any region of the search space.
To summarize, selection starts at the root node and proceeds down the tree, selecting at each node the child with highest utility. Upon reaching an unexpanded node, selection stops and the current node is chosen for expansion.

2. *Expansion*: one or more children of the selected node n are created by applying possible decisions in n to copies of n. We present two strategies for node expansion:

Single expansion:	a single child node is created using a randomly chosen unexplored decision in n. Other unexplored decisions are kept for a later time when node n is again selected for expansion.
Full expansion:	all children of n are immediately created by generating all possible decisions in the node.

3. *Simulation*: from each node created in the expansion step, perform a simulation until a terminal state (i.e., a solution) is reached, and record the value obtained. Various approaches can be taken in simulation, ranging from uniform random decisions—requiring nothing more than a generative model of the problem—to heuristic construction algorithms incorporating domain-specific knowledge. The latter approach typically allows for faster convergence at the expense of simplicity and generality. In the applications presented in this work, we will use problem-specific construction heuristics for simulation; for each problem, the heuristic used is detailed in the corresponding section.
4. *Backpropagation*: propagate the outcome of each simulation up the tree until the root node is reached; this updates statistics (simulation scores and counts) on all nodes between the selected node and the root.

Because one simulation is done per new child of the selected node n, the two expansion strategies described above will exhibit different behavior with respect to the variation of $U(n)$; namely, the number of children generated is proportional to the decrease in the exploration term $E(n)$ for the parent node n. This difference ultimately leads to different search paths in the tree, which may in turn have an impact on the performance of the algorithm. In trees with high branching factors, if single expansion is used, a node may be confirmed (to a certain degree) as a poor choice before generating all its children, thereby saving both time and space which can be used to explore other areas of the tree. This effect is dampened in trees with lower branching factors.

Applying Monte Carlo tree search to solve optimization problems has many similarities, but also significant differences to its application in gameplaying. First, the size of the search trees in both domains is commonly large enough to prevent complete search within reasonable computational time. Hence, MCTS should direct the search, leveraging all the information gathered up to the moment in the most suitable way.

A significant difference concerns the evaluation of nodes and their associated statistics. Whereas in gameplaying a branch with a high average win rate is suggestive of a strong line of play, in optimization—since we are interested in finding extrema—the average solution under a node is not a good estimator of the optimal solution to the node's underlying subproblem. Additionally, rewards in gameplaying often take 0–1 values for loss and win, respectively; objective functions, on the other hand, may take arbitrary values. Since UCB1/UCT were designed with rewards in the [0, 1] interval in mind, we must map objective function values into that interval, in order to maintain the proper balance between the two components of (1). To address these two issues, we propose the following expression for $X(n)$:

$$X(n) = \frac{e^a - 1}{e - 1}, \text{ with } a = \frac{\hat{w}^* - \hat{z}_n^*}{\hat{w}^* - \hat{z}^*}, \tag{3}$$

where $\hat{z}^*$ and $\hat{w}^*$ are, respectively, the best and the worst simulation results found so far in the whole tree, and $\hat{z}_n^*$ is the best simulation result under node n. A plot of the proposed reward function can be seen in Fig. 1.

Although our main guiding criterion should be based on the best simulation outcome $\hat{z}_n^*$, the average outcome of simulations under a node—denoted as $\bar{z}_n$—may

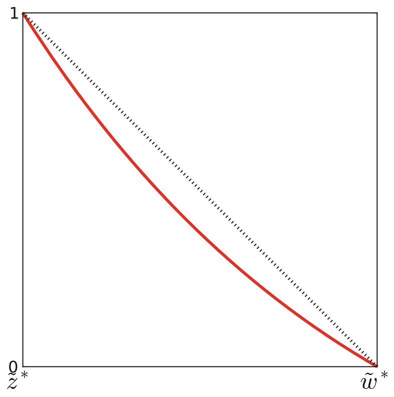

Fig. 1 The proposed reward function $X(n)$ (*solid line*), in terms of the best simulation outcome under node n, $\hat{z}_n^*$. The linear reward function (*dotted line*) is shown for reference

still provide a useful hint on the interest in exploring the node. We propose to use a function similar to $X(n)$ representing the average reward under node n, and define it as

$$\overline{X}(n) = \frac{e^b - 1}{e - 1}, \text{ with } b = \frac{\hat{w}^* - \overline{z}_n}{\hat{w}^* - \hat{z}^*}. \tag{4}$$

Instead of incorporating this information directly into the exploitation term, we integrate $\overline{X}(n)$ as a factor in the exploration term; we call this *average-weighted exploration*, and define it as

$$E'(n) = \overline{X}(n)E(n). \tag{5}$$

The above ideas, applied to the problem of combinatorial optimization, are summarized in the pseudocode in Algorithm 1.

Algorithm 1: MCTS for (minimization) combinatorial optimization.

```
   Data: problem instance $I$
   Result: upper bound on optimal value (minimization problems)
1  $r \leftarrow$ create root node with initial state from $I$
2  $z^* \leftarrow \infty$
3  repeat
4      $n \leftarrow$ starting from $r$, recursively select child node with maximum $U(n)$
5      $C \leftarrow$ set of child nodes obtained from expanding $n$
6      foreach $c \in C$ do
7          run a simulation from $c$
8          $z \leftarrow$ result of the simulation
9          propagate $z$ up until reaching $r$
10         if $z < z^*$ then
11             $z^* \leftarrow z$
12 until computational budget is depleted
13 return $z^*$
```

3 Number Partitioning

The number partitioning problem (NPP) is a classical combinatorial optimization problem, with applications in public key cryptography and task scheduling. Given a set (or possibly a multiset) of N positive integers $\mathscr{A} = \{a_1, a_2, \ldots, a_N\}$, find a partition $\mathscr{P} \subseteq \{1, \ldots, N\}$ that minimizes the discrepancy

$$E(\mathscr{P}) = \left| \sum_{i \in \mathscr{P}} a_i - \sum_{i \notin \mathscr{P}} a_i \right|.$$

Partitions such that $E = 0$ or $E = 1$ are called perfect partitions.

A pseudo-polynomial time algorithm for the NPP is presented in [22] for the case where all a_j are positive integers bounded by a constant A; for the general case of exponentially large input numbers (or exponentially high precision, if the a_j's are real numbers) the problem is NP-hard. If the numbers a_j are independently and identically distributed random numbers bounded by $A = 2^{\kappa N}$, the solution time abruptly raises at a particular value $\kappa = \kappa_c$; this is due to the high probability of having perfect partitions for $\kappa < \kappa_c$, and this probability is close to 0 for $\kappa > \kappa_c$ (see [23, 24] for more details).

A direct application of the NPP occurs in load balancing of two identical machines. The common two-way NPP, as well as a generalization to an arbitrary number of subsets (equivalently, machines) are tackled in [25] by recasting the problem as an unconstrained quadratic binary program (UQP); the UQP is then solved using a tabu search algorithm. Another application arises in high-performance multidisk database systems. To promote parallelization of I/O and minimize query response times in such systems, data that are likely to be accessed by same queries are distributed across K disks—a process called *declustering*. This problem, equivalent to a multiway NPP, is tackled in [26] using a two-phase approach: the first phase consists in recursive bipartitioning to obtain an initial K-way partition; in the second phase, the initial partition is improved through a refinement heuristic.

The best polynomial time heuristic known for the NPP is the differencing method of Karmarkar and Karp [27] (the KK heuristic). It consists of successively replacing the two largest numbers by the absolute value of their difference and placing those items in separate subsets, but without actually fixing the subset into which either number will go (see Algorithm 2).

Algorithm 2: The Karmarkar–Karp heuristic.

```
  Data: ordered set of positive integers $\mathscr{A}$
  Result: discrepancy obtained
1 $\forall a_i \in \mathscr{A}$, create a vertex $i$ with label $l_i \leftarrow a_i$
2 $\mathscr{E} \leftarrow \{\}$
3 while there is more than one labeled vertex do
4     $u, v \leftarrow$ vertices with the two largest labels
5     $\mathscr{E} \leftarrow \mathscr{E} \cup \{\{u, v\}\}$
6     set label $l_u \leftarrow l_u - l_v$
7     remove label $l_v$ from vertex $v$
8 return discrepancy (i.e., the last label)
```

Extensions of the KK heuristic for a complete search have been proposed in [28, 29]. In each step of the KK heuristic the two largest numbers are replaced by their difference; for a complete search, the alternative of replacing them by their sum must also be considered. For the previous example, the complete search tree is represented in Fig. 2.

When applied to the set $\mathscr{A} = \{8, 7, 6, 5, 4\}$, the KK heuristic leads to the partitions $\{8, 6\}$ and $\{7, 5, 4\}$ with discrepancy 2 (Fig. 3). Figure 4 illustrates the graph corresponding to the optimal solution, as obtained by complete search. Straight edges connect differencing vertices, that will be in different partitions; curly edges connect

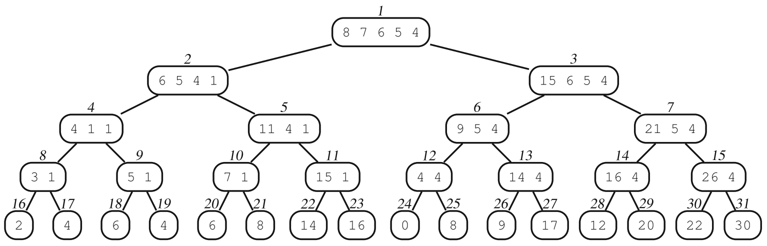

Fig. 2 Search tree for the complete differencing method with the set $\mathscr{A} = \{8, 7, 6, 5, 4\}$

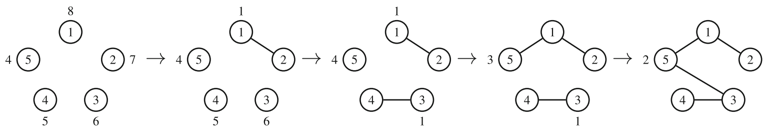

Fig. 3 Graph created with the KK heuristics, corresponding to the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16$ in Fig. 2

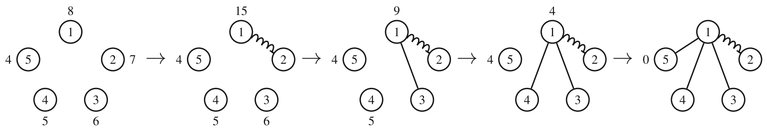

Fig. 4 Graph created while applying complete search: steps followed for creating the optimal partition, i.e., the path $1 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 24$ in Fig. 2

addition vertices, that will be in the same partition. The optimal solution is the partition {8, 7} and {6, 5, 4}, which is a perfect partition.

In the worst case, the complete differencing method has exponential complexity. Parts of the search tree may be pruned by observing that:

- the KK heuristic is exact for partitioning 4 or less numbers;
- the algorithm can be stopped when a perfect partition is found;
- when the difference between the largest remaining number and the sum of other remaining numbers is >1, the best possible solution is to place the largest number in one set and all the other numbers in the other set.

Algorithm 3 introduces depth-first search for the complete KK.

We propose the application of MCTS to this problem, using the KK heuristic as the construction method in the simulation step. Since this heuristic is deterministic, running it from the differencing child (which coincides with the heuristic's choice) would lead to the exact same solution as that obtained for the parent node; thus, we can safely reuse the parent's solution, and therefore only one new construction is required for the two children of each node. Another advantage of using this heuristic is that

Algorithm 3: dfs($\mathscr{A}$)—depth-first search.

```
Data: ordered set of positive integers $\mathscr{A}$
Result: optimum discrepancy
1  if $|\mathscr{A}| \leq 4$ then return $KK(\mathscr{A})$
2  $u, v \leftarrow$ largest and second-largest values in $\mathscr{A}$
3  $d \leftarrow u - \sum_{a \in \mathscr{A} \setminus \{u\}} a$
4  if $|d| \leq 1$ then return $|d|$
5  if $d > 0$ then return $d$
6  $\mathscr{A} \leftarrow \mathscr{A} \setminus \{u, v\}$
7  $\ell \leftarrow$ dfs ($\mathscr{A} \cup \{u - v\}$)
8  if $\ell \leq 1$ then return $\ell$
9  $r \leftarrow$ dfs ($\mathscr{A} \cup \{u + v\}$)
10 return $\min(\ell, r)$
```

repeated solutions are avoided altogether; such would not be the case with random or semi-greedy simulations. Finally, the KK heuristic allows the above pruning rules to be used, reducing the size of the search space without sacrificing completeness.

Due to the very low branching factor, and from empirical observation, we use a full expansion strategy in the expansion step of MCTS, meaning that both children of each node are immediately generated once the node is selected. Furthermore, for this problem, we choose to use the classical exploration term $E(n)$ (see (2)) in the evaluation of nodes, instead of the average-weighted exploration term $E'(n)$ (see (5)). This is motivated by the fact that average node performance can be deceptive in the NPP; varying just one or two decisions during construction has very deep and unpredictable consequences on the quality of the final solutions; this lack of correlation between particular decisions and solution quality makes average performance a poor guiding principle in this problem.

Table 1 presents results obtained with the KK heuristic and with time-limited DFS and MCTS, run on 10 hard instances from [29].

As expected, both tree search variants provide very considerable improvements over KK. It is well known that for difficult instances the optimum for NPP is very hard to find; DFS, being able to explore a much larger portion of the tree—hundreds to thousands of millions of nodes per hour, for these instances—is very difficult to beat. Also, as previously mentioned, the performance of DFS is highly dependent on the construction heuristic used, which in this case provides solutions of very good quality to begin with.

Although there is no clear indication of a winner from the results obtained, the observation of mixed results between DFS and MCTS is nonetheless impressive: MCTS was highly competitive despite exploring less than a thousandth of the nodes explored by DFS. This suggests that MCTS can effectively guide the search toward promising regions. In our view, these are very encouraging results.

We conclude with the hypothesis that MCTS's rate of convergence should improve over time, not only for this problem but also in general. While DFS gains practically nothing as the search progresses, MCTS constantly accumulates knowledge of the

Table 1 Results obtained for number partitioning with the KK heuristic, DFS, and MCTS on hard, large instances from [29]

Instance	KK	DFS (60 s)	MCTS (60 s)	DFS (600 s)	MCTS (600 s)
Hard0100	73.37	56.12	54.56	53.36	51.20
Hard0200	171.81	151.91	150.88	151.35	149.79
Hard0300	265.77	247.07	248.82	247.07	247.18
Hard0400	366.18	343.36	347.55	339.07	344.84
Hard0500	461.60	443.54	441.42	438.79	437.22
Hard0600	557.09	540.22	542.59	539.13	540.66
Hard0700	659.50	640.48	641.78	637.70	633.84
Hard0800	751.27	737.65	738.49	731.98	735.67
Hard0900	853.36	834.95	837.93	834.95	833.87
Hard1000	952.47	932.55	938.05	932.55	930.23

DFS and MCTS results are reported at 60 s (center columns) and 600 s (right-hand side columns). For MCTS, the average best solution of 10 independent runs is shown. Values reported are $\log_2(n+1)$, where n is the discrepancy obtained (which for these instances is a very large integer)

search landscape, which should theoretically help convergence toward the optimum. Having admittedly few results to support this hypothesis, we leave its confirmation as future work.

4 Stacking

The stacking problem (SP) consists of a series of placement decisions for a set of items with known dates for entrance and exit from a warehouse (see Fig. 5), denoted by *release* and *due* dates, respectively. Items are placed in vacant positions, or on top of other items forming stacks, i.e., last-in first-out queues. At any given time, only the top item of each stack can be taken, so in order to take an item that is not at the top of a stack, it is necessary to first relocate all items above it to other stacks. The objective is to store and then deliver all items, while respecting their release and due

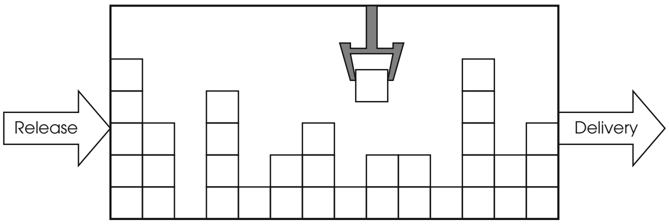

Fig. 5 A warehouse where items are stored in stacks, using a stacking crane that can only handle one item at a time

dates, with a minimum number of relocations. In the variant tackled in this work, we assume that there is no height limit on stacks (uncapacitated SP), movements occur instantaneously, and releases or deliveries cannot be anticipated or delayed.

Stacking problems have evident practical importance in container port operations [30, 31] and ship stowage planning [32, 33]. Stacking is also important in the steel industry as a means of storage for finished products [34].

As demonstrated in [34], the zero-relocation SP (decision problem) is NP-complete for any fixed number of stacks $W \geq 4$, by a polynomial reduction to the problem of coloring circle graphs. From this it follows that the general r-relocation SP is also NP-complete, and the optimization version of the SP is NP-hard.

Although the SP is first described in [34], the paper mentions the existence of similar problems and presents an overview of the literature. One such problem is the Block Relocation Problem (BRP), which is actually a subset of the SP. In the BRP, the initial state of the stacks is given as input data along with a (partial) order of retrieval of the items. The objective is to find the shortest sequence of movements such that items are retrieved in the given order. In [35], a branch-and-bound algorithm for the BRP, as well as a simple heuristic for real-time applications are proposed. The heuristic rule is based on an estimate of the number of additional relocations for each stack. In [36], an algorithm for the BRP based on the corridor method is presented. The corridor method combines mathematical programming techniques with heuristics. The main idea is to exactly solve subproblems where some variables are fixed, creating a "corridor-like" region where an item is allowed to go.

For tackling the SP itself, in [34], a discrete-event simulation model of the warehouse is used, and construction of solutions is based on a semi-greedy heuristic. The heuristic is invoked when deciding the placement of an item during a release or *reshuffle*. Reshuffling is defined as the relocation of items that is necessary to reach an item lower in the stack. For simplicity, the voluntary relocation of items in-between releases or deliveries—called *remarshalling*—is not considered during a simulation. Note, however, that this may potentially leave the optimal solution(s) out of the search space, therefore losing the guarantee of optimality even for a complete search.

A simulation consists in traversing a schedule of events (i.e., releases and deliveries) in chronological order and processing each event appropriately. When multiple events have the same date, deliveries are processed first, in increasing order of *item depth*, where the depth of an item is defined as the number of items above it in the same stack. Then, any releases are processed in inverse order of due date, that is, items with greater due date are released first. Ties are resolved randomly.

The probabilistic component of the construction heuristic is exploited by repetition of the process using different seeds for the pseudorandom number generator, in a simple method called Multiple Simulation (MS). This simple yet effective idea can also be exploited in Monte Carlo tree search, taking advantage of the tree structure to implicitly force different simulations to be executed.

In order to accelerate MS, a simulation is interrupted as soon as it is known that the number of relocations of the incumbent solution cannot be improved. This is actually a weak form of pruning, as seen in branch-and-bound algorithms. Whenever a better solution is found, the cutoff value is updated, tightening the upper bound for future simulations. This optimization is not used in MCTS because, even after a simulation is known to be poor, the algorithm can still benefit from knowing how poor the simulation is, and therefore it is run to completion anyway.

We now describe the construction heuristic used in multiple simulation, called Flexibility Optimization (FO). FO will be used in the MS method, as well as in the simulation step of MCTS. For details on the remaining steps of the MCTS algorithm, please refer to Sect. 2.2.

First, we need to define the concept of *stack movement date*: the earliest movement date of stack s is represented as

$$m_s = \min_{i \in I_s} D_i,$$

where I_s represents the set of items currently in stack s, and D_i is the due date of item i. When stack s is empty, then $m_s = \infty$ by definition. Another important concept is that of *due date inversion*: an inversion occurs when an item i is placed (directly of indirectly) above an item j with $D_i > D_j$. In order to deliver j, item i will have to be relocated.

One can view m_s as an indicator of the *flexibility* of stack s for receiving new items without creating inversions. To illustrate this idea, consider an empty stack s, with $m_s = \infty$; this stack is considered as having infinite flexibility, since any item can be added to it without creating a new inversion. On the other hand, if $I_s \neq \{\}$, then m_s is finite and only items with due date up to m_s can be added to s without creating an inversion.

When placing an item, FO will prefer stacks where loss of flexibility is minimized, whenever this is possible without creating new inversions. If a new inversion is unavoidable, the heuristic places the item in the stack with the largest movement date, in order to postpone the forced relocation as much as possible. The heuristic makes use of a function associating to each placement decision $i \to s$ a score $f(i, s)$; this function embeds the above rules, and is defined as

$$f(i, s) = \begin{cases} \frac{1}{1+D_i-m_s} - 1 & \text{if } D_i > m_s, \\ \frac{1}{1-D_i+m_s} & \text{otherwise.} \end{cases} \tag{6}$$

The graph for this function is shown in Fig. 6. Note that the top branch in (6) represents the creation of a new inversion (since $D_i > m_s$), therefore lower scores ($f(i, s) \in [-1, 0]$) are assigned to it. The bottom branch represents the placement of an item without creating an inversion, being given a higher score than any inversion-inducing decision. Given an item i and a set of possible destination stacks T, the FO heuristic constructs a *restricted candidate list* of stacks $\text{RCL} = \{s \in T : f(i, s) \geq f(i, s'), \forall s' \in T\}$, and randomly selects a stack from the RCL as the destination of item i.

A computational experiment was conducted on the 24 benchmark instances of [34], comparing MCTS with the MS method. We use the generic MCTS

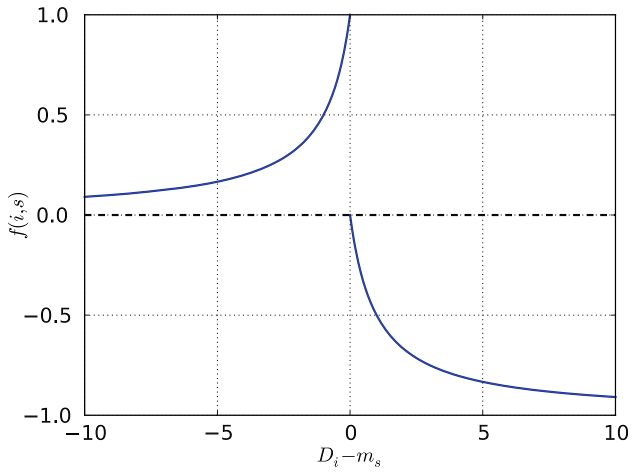

Fig. 6 Score function $f(i, s)$ used by the FO heuristic

algorithm presented in Algorithm 1, running simulations with the FO heuristic in the algorithm's simulation step. In this case, since the heuristic used is nondeterministic, a new simulation must be run for each node created in MCTS. As for the expansion strategy used, in this problem we choose single expansion due to the potentially high branching factor. Additionally, we use the average-weighted exploration term defined in (5).

The two methods were run ten times on each instance, for 600 s, using different seeds for the pseudorandom number generator. Table 2 presents the average number of relocations of the best solution found after 60 s (left-hand side columns), and at the end of the 600 s period (right-hand side columns).

The tree search is naturally expected to perform better, but the reduced computational budget (especially the 60 s limit) presents some difficulties for MCTS. As MCTS uses the results of simulations from each node to estimate their worth, it usually has a warmup period during which its decisions may be poor due to the lack of available information. As more time is allowed, node evaluation estimates improve and results are expected to be more consistent. The reduced time is an advantage for MS also because of its inexistent overhead, as opposed to MCTS which must traverse the tree from the root to a nonexpanded node at each iteration, must create and maintain the tree structure in memory, and must propagate simulation results upward.

The results indicate that even with 60 s, MCTS performs better than MS in most instances; this shows that the search is able to quickly focus on the more promising branches. As expected, performance is further improved with the increased time limit of 600 s.

Table 2 Results for the stacking problem: average number of relocations over 10 runs with the MS and MCTS algorithms, for large instances from [34], with a time limit of 60 s (left-hand side columns) and 600 s (right-hand side columns)

Instance	MS (60 s)	MCTS (60 s)	MS (600 s)	MCTS (600 s)
2-A	52872.6	52358.1	52794.3	51854.5
2-B	50820.6	50760.2	50799.1	50727.3
2-C	49450.7	48880.0	49249.2	47514.3
2-D	50227.2	50230.6	50058.9	49090.4
3-A	23148.2	23124.3	22568.4	22502.2
3-B	24608.7	24665.6	24137.5	24201.0
3-C	24627.9	24738.3	23915.3	24036.8
3-D	24130.7	24065.4	23349.3	23510.7
4-A	14439.4	14441.0	14159.9	14132.4
4-B	13355.9	13429.9	13204.6	13150.5
4-C	13981.8	14050.8	13720.9	13854.0
4-D	14796.0	14865.8	14501.4	13975.5
10-A	3524.9	3484.1	3494.1	3289.5
10-B	4031.3	3790.9	4013.4	3751.4
10-C	3644.3	3497.1	3606.1	3421.7
10-D	3416.3	3437.9	3390.3	3410.4
20-A	883.9	887.5	882.6	885.3
20-B	1029.4	950.6	1022.8	842.2
20-C	1098.0	977.1	1098.0	975.7
20-D	1178.0	1159.2	1178.0	1158.9
40-A	79.0	53.7	79.0	53.4
40-B	59.0	30.5	59.0	9.0
40-C	41.0	32.1	41.0	29.3
40-D	154.0	99.7	154.0	94.3

5 Recursive Circle Packing

The recursive circle packing problem (RCPP) originates from the tube industry, where shipping costs represent an important fraction of the total cost of product delivery [37]. Tubes are produced in a continuous extraction machine and cut to the length of the container inside of which they will be shipped. Before being placed in the container they may be inserted into other, wider tubes, so that usage of container space is maximized—a process called *telescoping*. As all the tubes occupy the full length of the container, maximizing container load is equivalent to maximizing the area filled with circles (or, more precisely, rings/annuli) in a section of the container.

This problem is evidently more general than circle packing, which is known to be NP-complete (see, e.g., [38]). We propose a heuristic method for tackling it, which has proven to be able to produce very good solutions for practical purposes.

A nontechnical, general overview of circle packing is presented in [39]; for a bibliographic review article see [40], which surveys the most relevant literature on efficient models and methods for packing circular objects/items into regions in the Euclidean plane; objects/items and regions considered are either two- or three-dimensional. A survey of industrial applications of circle packing and of methods for their solution, both exact and heuristic, is presented in [41].

In the base RCPP, a number A of tubes are available for packing in a container of width W and height H, in such a way that the value of the packing is maximum. Let $\mathscr{A} = \{1, \ldots, A\}$ be the index set of the tubes; each tube $i \in \mathscr{A}$ is characterized by an external radius r_i^{ext} and an internal radius r_i^{int}, and may be placed in the container or not. A formulation in mixed integer nonlinear programming, provided in [37], considers:

- binary variables w_i, for all $i \in \mathscr{A}$, where $w_i = 1$ if tube i is placed directly inside the container, $w_i = 0$ otherwise;
- binary variables u_{ki} for $k, i \in \mathscr{A}$ such that $r_k^{\mathsf{int}} \geq r_i^{\mathsf{ext}}$, where $u_{ki} = 1$ if tube i is placed directly inside tube k, $u_{ki} = 0$ otherwise (only required if $r_k^{\mathsf{int}} \geq r_i^{\mathsf{ext}}$; other pairs (k, i) are not excluded for facilitating notation);
- position variables (x_i, y_i) of the center of tube i, for all $i \in \mathscr{A}$ (only relevant if i is packed).

Each loaded tube is placed within the bounds of a container, which we assume to be a rectangle with vertices $(0, 0)$, $(W, 0)$, $(0, H)$, and (W, H). The constraints are the following:

- inserted tubes must be completely inside the container;
- loaded tubes may be placed either directly in the container or inside other tubes;
- for each pair of tubes (i, j) directly placed in the container, the distance between them must be greater than or equal to the sum of their external radii;
- the above constraint is likewise applied for each pair of tubes (i, j) directly placed inside the same tube k;
- if tube i is placed directly in tube k, their centers must be close enough for i to remain completely inside k.

The objective of this problem is to maximize the value of the packing, i.e., the sum of a user-defined value v_i for loaded tubes:

$$\text{maximize } V = \sum_{i \in \mathscr{A}} v_i \left(w_i + \sum_{k \in \mathscr{A}} u_{ki} \right). \tag{7}$$

We now describe a heuristic method for quickly constructing a solution to the RCPP. The method begins with an empty container and iteratively inserts new tubes either directly into the container or into a previously packed tube. An auxiliary set $\mathscr{O}$ of *open objects* is used, which initially has the container as its only element. An object (a tube or the container) is said to be open while it is possible to insert at least one of the remaining tubes into it. Whenever a new tube is packed, it is added to $\mathscr{O}$;

and when it is found that no tubes can be inserted into an object, it is removed from $\mathcal{O}$. The algorithm packs a new tube per iteration until either all available tubes have been packed or $\mathcal{O}$ becomes empty.

In order to prioritize telescoping—which seems intuitively advantageous because value is gained without further occupying the container—we choose, if possible, to insert the next tube into the open object $o \in \mathcal{O}$ with minimum free space/area. It is then checked if at least one tube can be inserted into o: if it is possible, we move on to the next step in the algorithm; otherwise, o is removed from $\mathcal{O}$ and the object having the next minimum free space is checked. If during this selection $\mathcal{O}$ becomes empty, the algorithm stops and the current solution is returned.

After selecting the object o into which a tube will be inserted, the actual tube to be inserted is selected. In this step, the algorithm greedily chooses the tube t which has the maximum estimated *value-to-area ratio*. This ratio is an estimate of the total value of a tube and all tubes that can potentially be telescoped into it, divided by its area. It is approximated in a manner similar to the greedy heuristic for the knapsack problem, which is based on the value-to-weight ratio of items.

Finally, from the set of positions of tube t in the open object o, a position p with minimum ordinate is chosen; for tiebreaking, the position with smallest abscissa is selected. An iteration ends by inserting tube t into object o at position p, and updating $\mathcal{O}$ to include t.

Since the position variables (x_i and y_i) in the mathematical model are real variables, the set of positions for a tube is often infinite. In the computation of candidate positions for tube t, we reduce this possibly infinite set to a finite set through a number of simple rules. When inserting a new tube t directly into the container (Fig. 7), the candidate positions considered are:

- the two positions placing t at the bottom corners of the container;
- for each tube u already packed directly into the container, include all positions where t is tangent to u and to any wall of the container;
- for each pair of tubes (v, w) already packed directly into the container, include all positions where t is tangent to both v and w.

Similarly, when t is being inserted into a wider, previously packed tube t' (Fig. 8), the set of candidate positions includes:

- the position placing t at the bottom center of t';
- for each tube u packed directly into t', include all positions that are tangent to both t' (from the inside) and u (from the outside);

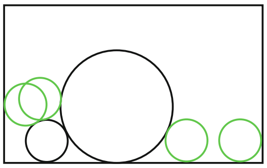
Fig. 7 Circle packing inside a rectangle: positioning possibilities given previously placed, fixed circles (in *black*)

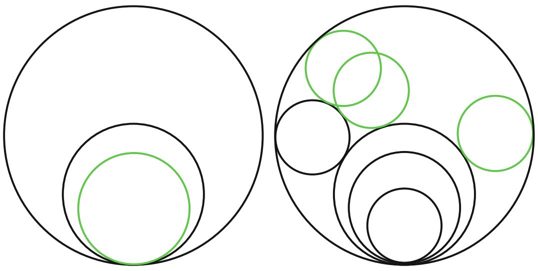
Fig. 8 Circle packing inside another circle (telescoping): positioning possibilities given previously placed, fixed circles (in *black*)

- for each pair of tubes (v, w) already packed directly into t', include all positions that are tangent to both v and w;

After the set of candidate positions is computed, positions violating any constraint (e.g., positions where t overlaps with a packed tube) are discarded. It must be noted that with this simplification we are excluding the majority of the original problem's search space, so the property of proven optimality is lost even when this restricted search space is fully explored. Figure 9 shows an example of an optimal solution which cannot be generated by the described method; whichever the first tube is, it is not at a corner of the container.

The above construction method is purely greedy, leading to a single solution if the same data is provided multiple times. The method is converted into a semi-greedy algorithm by introducing a probabilistic component into one of the choices; in our implementation, the position into which a new tube is inserted is randomly selected, giving higher probability to positions closer to the greedy decision. The full construction method is summarized in Algorithm 4.

This semi-greedy variant is used in both algorithms compared in the computational experiment. The semi-greedy (SG) algorithm consists in repeating constructions with different pseudorandom number generator seeds, whereas MCTS uses it for the simulation step. As in the stacking problem, MCTS uses a single-expansion strategy and scores nodes using the average-weighted exploration term.

The computational experiment included six instances of the RCPP, adapted from [37]. The two algorithms were run 10 times on each instance, with a time limit of 600 s. Table 3 reports the average total value of the best solution found after 60 s, and at the end of the full 600 s period.

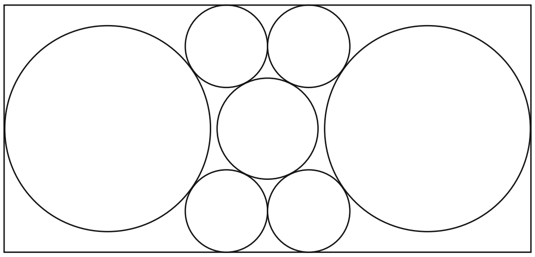
Fig. 9 An optimal solution which is not contained in the restricted search space

Table 3 Average value packed in a container, for 10 independent observations, for the RCPP with semi-greedy construction and with Monte Carlo tree search, with a time limit of 60 s (left-hand side columns) and 600 s (right-hand side columns)

Instance	SG (60 s)	MCTS (60 s)	SG (600 s)	MCTS (600 s)
Large03	3660023.3	3660021.8	3660024.2	3660023.5
Large05	4140042.1	4140044.0	4140043.0	4140047.1
Large16	30311008.0	30625928.6	30311016.6	31192827.3
Small03	938000.0	950000.0	940000.0	957000.0
Small05	1090000.0	1120000.0	1090000.0	1120000.0
Small16	10438035.5	10388039.2	10450036.6	10534032.5

Instances adapted from [37]

Although the number of instances is small, the results indicate a superior performance of MCTS after 60 s, further widening the gap when the full 600 s are allowed. This is an expected consequence of allowing extra time, as more information is gathered and MCTS's estimates of node values are refined.

Algorithm 4: Construction heuristic for recursive tube packing.

```
   Data: container $C$ and set of available tubes $\mathscr{A}$
   Result: set $\mathscr{S}$ of tubes packed and their respective positions
1  $\mathscr{S} \leftarrow \{\}$
2  $\mathscr{O} \leftarrow \{C\}$
3  while $\mathscr{O} \neq \{\}$ and $\mathscr{A} \neq \{\}$ do
4      $o \leftarrow$ element of $\mathscr{O}$ with minimum unused area
5      foreach $t \in \mathscr{A}$ (from largest to smallest value-to-area ratio) do
6          $\mathscr{P} \leftarrow$ positions for $t$ inside $o$
7          if $\mathscr{P} \neq \{\}$ then
8              choose position $p \in \mathscr{P}$ (semi-greedy choice)
9              $\mathscr{S} \leftarrow \mathscr{S} \cup \{(t, p)\}$
10             $\mathscr{O} \leftarrow \mathscr{O} \cup \{t\}$
11             $\mathscr{A} \leftarrow \mathscr{A} \setminus \{t\}$
12             break
13     if could not place any tube inside $o$ then
14         $\mathscr{O} \leftarrow \mathscr{O} \setminus \{o\}$
15 return $\mathscr{S}$
```

6 Conclusion

Tree search is at the heart of the solution of combinatorial optimization problems. The use of simulation to estimate node values is twofold advantageous: it provides quick solutions to the problem; and it prevents misleading evaluations given by poor bounds.

In this work, we propose the use of tree search in combination with problem-specific heuristics for three relevant problems in logistics, to provide better estimates of node values and speed up convergence toward good solutions. In the first application—the number partitioning problem, arising e.g., in load balancing—tree search exploits the quality of the well-known Karmarkar–Karp construction heuristic. The second application is the stacking problem, common in container port operations and in handling warehouse storage, for which full simulation is necessary to evaluate even the earliest placement decisions. Finally, tree search is applied to the recursive circle packing problem—a generalization of circle packing in rectangles—where, once more, early decisions during construction may not be accurately assessed before the solution is complete. In the two last applications, a random component is introduced in the construction heuristic for the sake of diversification. A simulation based on such construction heuristics builds a feasible solution, whose value is used in the evaluation of all nodes in the path between the root of the search tree and the node from which the simulation originated.

In logistics, many situations are studied using simulation models, due to the difficulty in fully characterizing them as formal mathematical models. Simulation-based optimization is a general framework for improving solutions for these models; tree search provides a systematic way for using the simulation models already available in an optimization context. One promising application area with relevance in logistics is resource-constrained scheduling; construction rules can be used for simulation, and decisions complementary to these rules can be implicitly explored under the tree search.

The integration of the paradigms of simulation and, potentially, exact search, is a promising step toward the solution of hard problems, whose formulation in mathematical optimization is too loose for mixed integer solvers to provide acceptable solutions. It also enlarges the scope in which Operations Research exact methods can be applied, in the sense that the simulation may deal with problems which are not linear or even well-defined. The only requirements in this regard would be to contain all relevant solutions in the search tree, and to have consistent evaluations provided by simulation. Although complete enumeration is usually a remote possibility, solutions found on the searched path may provide excellent, realistic approximations, as has been illustrated with the study cases of number partitioning, stacking, and recursive circle packing.

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

References

1. Land, A.H., Doig, A.G.: An automatic method of solving discrete programming problems. Econometrica **28** (1960) 497–520
2. Lawler, E.L., Wood, D.E.: Branch-and-bound methods: A survey. Operations Research **14** (1966) 699–719

3. Buchheim, C., Caprara, A., Lodi, A.: An effective branch-and-bound algorithm for convex quadratic integer programming. Mathematical Programming **135** (2012) 369–395
4. Ng, C., Wang, J.B., Cheng, T.E., Liu, L.: A branch-and-bound algorithm for solving a two-machine flow shop problem with deteriorating jobs. Computers & Operations Research **37** (2010) 83–90
5. Bazin, J., Li, H., Kweon, I.S., Demonceaux, C., Vasseur, P., Ikeuchi, K.: A branch-and-bound approach to correspondence and grouping problems. IEEE Transactions on Pattern Analysis and Machine Intelligence **35** (2013) 1565–1576
6. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Exact combinatorial branch-and-bound for graph bisection. In: Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12), Society for Industrial and Applied Mathematics (2012) 30–44
7. Ginsberg, M.L., Harvey, W.D.: Iterative broadening. Artificial Intelligence **55** (1992) 367–383
8. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **4** (1968) 100–107
9. Korf, R.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence **27** (1985) 97–109
10. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24** (2000) 67–100
11. Fischetti, M., Monaci, M.: Exploiting erraticism in search. Operations Research **62** (2014) 114–122
12. Bouzy, B.: Associating shallow and selective global tree search with Monte Carlo for 9×9 Go. In: Computers and Games. Springer (2006) 67–80
13. Juille, H.R.: Methods for Statistical Inference: Extending the Evolutionary Computation Paradigm. PhD thesis, Waltham, MA, USA (1999)
14. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: Proceedings of the 5th International Conference on Computers and Games. CG'06, Berlin, Heidelberg, Springer-Verlag (2007) 72–83
15. Winands, M.H., Björnsson, Y., Saito, J.T.: Monte-Carlo tree search solver. In: Computers and Games. Springer (2008) 25–36
16. Takeuchi, S., Kaneko, T., Yamaguchi, K.: Evaluation of Monte Carlo tree search and the application to Go. In: 2008 IEEE Symposium On Computational Intelligence and Games (CIG), IEEE (2008) 191–198
17. Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., Teytaud, O.: The grand challenge of computer Go: Monte Carlo tree search and extensions. Communications of the ACM **55** (2012) 106–113
18. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games **4** (2012) 1–43
19. Sabharwal, A., Samulowitz, H., Reddy, C.: Guiding combinatorial optimization with UCT. In: Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2012), Springer (2012) 356–361
20. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In Fürnkranz, J., Scheffer, T., Spiliopoulou, M., eds.: Machine Learning: ECML 2006. Volume 4212 of Lecture Notes in Computer Science. Springer, Berlin Heidelberg (2006) 282–293
21. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning **47** (2002) 235–256
22. Garey, M.R., Johnson, D.S.: Computers and Intractability. W. H. Freeman, New York (1979)
23. Mertens, S.: The easiest hard problem: Number partitioning. In Percus, A., Istrate, G., Moore, C., eds.: Computational Complexity and Statistical Physics, New York, Oxford University Press (2006) 125–139
24. Mertens, S.: Phase transition in the number partitioning problem. Physical Review Letters **81** (1998) 4281–4284

25. Alidaee, B., Glover, F., Kochenberger, G.A., Rego, C.: A new modeling and solution approach for the number partitioning problem. Journal of Applied Mathematics and Decision Sciences **9** (2005) 113–121
26. Koyutürk, M., Aykanat, C.: Iterative-improvement-based declustering heuristics for multi-disk databases. Information Systems **30** (2005) 47–70
27. Karmarkar, N., Karp, R.: The differencing method of set partitioning. Technical Report UCB/CSD 82/113, University of California - Berkeley, Computer Science Division (1982)
28. Korf, R.E.: A complete anytime algorithm for number partitioning. Artificial Intelligence **106** (1998) 181–203
29. Pedroso, J.P., Kubo, M.: Heuristics and exact methods for number partitioning. European Journal of Operational Research **202** (2010) 73–81
30. Dekker, R., Voogd, P., Asperen, E.: Advanced methods for container stacking. In Kim, K.H., Günther, H.O., eds.: Container Terminals and Cargo Systems. Springer, Berlin Heidelberg (2007) 131–154
31. Hartmann, S.: A general framework for scheduling equipment and manpower at container terminals. OR Spectrum **26** (2004) 51–74
32. Avriel, M., Penn, M., Shpirer, N.: Container ship stowage problem: Complexity and connection to the coloring of circle graphs. Discrete Applied Mathematics **103** (2000) 271–279
33. Avriel, M., Penn, M., Shpirer, N., Witteboon, S.: Stowage planning for container ships to reduce the number of shifts. Annals of Operations Research **76** (1998) 55–71
34. Rei, R.J., Pedroso, J.P.: Tree search for the stacking problem. Annals of Operations Research **203** (2013) 371–388
35. Kim, K.H., Hong, G.P.: A heuristic rule for relocating blocks. Computers and Operations Research **33** (2006) 940–954
36. Caserta, M., Voß, S., Sniedovich, M.: Applying the corridor method to a blocks relocation problem. OR Spectrum **33** (2011) 915–929
37. Pedroso, J.P., Cunha, S., Tavares, J.N.: Recursive circle packing problems. International Transactions in Operational Research (2014). doi:10.1111/itor.12107
38. Lenstra, J., Rinnooy Kan, A.: Complexity of packing, covering, and partitioning problems. In Schrijver, A., ed.: Packing and Covering in Combinatorics. Mathematisch Centrum, Amsterdam (1979) 275–291
39. Stephenson, K.: Circle packing: A mathematical tale. Notices of the American Mathematical Society **50** (2003) 1376–1388
40. Hifi, M., M'Hallah, R.: A literature review on circle and sphere packing problems: Models and methodologies. Advances in Operations Research **2009** (2009) 1–22
41. Castillo, I., Kampas, F.J., Pintér, J.D.: Solving circle packing problems by global optimization: Numerical results and industrial applications. European Journal of Operational Research **191** (2008) 786–802