Exploring search space trees using an adapted version of Monte Carlo tree search for combinatorial optimization problems

Jorik Jooken[a,*], Pieter Leyman[a,b,c], Tony Wauters[d], Patrick De Causmaecker[a]

[a]*Department of Computer Science, KU Leuven Kulak, Etienne Sabbelaan 53, 8500 Kortrijk, Belgium*
[b]*Department of Industrial Systems Engineering and Production Design, Ghent University, Technologiepark Zwijnaarde 46, 9052 Zwijnaarde, Belgium*
[c]*Industrial Systems Engineering, Flanders Make@UGent, Sint-Martens-Latemlaan 2B, 8500 Kortrijk, Belgium*
[d]*Department of Computer Science, CODeS, KU Leuven, Gebroeders De Smetstraat 1, 9000 Ghent, Belgium*

Abstract

In this article we propose a heuristic algorithm to explore search space trees associated with instances of combinatorial optimization problems. The algorithm is based on Monte Carlo tree search, a popular algorithm in game playing that is used to explore game trees and represents the state-of-the-art algorithm for a number of games. Several enhancements to Monte Carlo tree search are proposed that make the algorithm more suitable in a combinatorial optimization context. These enhancements exploit the combinatorial structure of the problem and aim to efficiently explore the search space tree by pruning subtrees, using a heuristic simulation policy, reducing the domains of variables by eliminating dominated value assignments and using a beam width. The algorithm was implemented with its components specifically tailored to two combinatorial optimization problems: the quay crane scheduling problem with non-crossing constraints and the 0-1 knapsack problem. For the first problem our algorithm surpasses the state-of-the-art results and several new best solutions are found for a benchmark set of instances. For the second problem our algorithm typically produces near-optimal solutions that are slightly worse than the state-of-the-art results, but it needs only a small fraction of the time to do so. These results indicate that the algorithm is competitive with the state-of-the-art for two entirely different combinatorial optimization problems.

Keywords: Combinatorial optimization, Monte Carlo tree search, Quay crane scheduling problem with non-crossing constraints, 0-1 Knapsack problem

1. Introduction

The operations research community has shown an increasing interest in approaches that use machine learning to solve combinatorial optimization problems. One of the main advantages of these approaches is that they can help to improve certain human-made algorithmic decisions. Some of these approaches rely almost exclusively on machine learning (e.g. [1, 2] and [3]), whereas other approaches combine machine learning with ideas found in more traditional approaches to solve combinatorial optimization problems (e.g. [4] and [5]). In this article, we follow the latter setting and we propose several adaptations for the Monte Carlo tree search algorithm [6], originally used in game playing, in order to efficiently explore the search space associated with a problem instance of a combinatorial optimization problem.

*Corresponding author
Email addresses: jorik.jooken@kuleuven.be (Jorik Jooken), pieter.leyman@ugent.be;pieter.leyman@kuleuven.be (Pieter Leyman), tony.wauters@kuleuven.be (Tony Wauters), patrick.decausmaecker@kuleuven.be (Patrick De Causmaecker)

DOI: https://doi.org/10.1016/j.cor.2022.106070

Solving such a problem instance boils down to finding an assignment of values to the decision variables such that all constraints are met and the objective function is minimized (without loss of generality). In case the domains of all decision variables are finite sets, the set of all feasible solutions can be represented by means of a rooted tree that represents the complete search space. The search space tree, and thus the set of all solutions, is often too big to exhaustively enumerate in a reasonable time. Nevertheless, there are many well-known techniques that still make use of this tree to solve the optimization problem. Each technique copes with this issue of a large search space in its own way. An example of a technique that can be used to explore a search space tree is branch-and-bound, which can skip exploring entire subtrees by proving that the set of solutions in a subtree are all worse than another solution. Other techniques cope with this issue by sacrificing the optimality guarantee of the algorithm in return for a time gain. These heuristic techniques only explore a subset of all feasible solutions and choose the best one in this subset (e.g. beam search). Finally, there are techniques that also only explore a subset of all solutions, but they do not explicitly use the search space tree to do so (e.g. several metaheuristics and algorithms based on local search).

The main contributions of this article are twofold. First, the article presents a heuristic algorithm to explore search space trees that is based on Monte Carlo tree search, a popular reinforcement learning algorithm for game playing [7, 6]. We show that this algorithm can be modified in several ways to combine machine learning with ideas found in more traditional approaches for solving combinatorial optimization problems. These modifications all exploit the combinatorial structure of the problem to efficiently explore the search space. The proposed algorithm is able to prune large parts of the search space tree by using bounds on the objective function value. Furthermore, it is possible to integrate heuristic, problem specific information into the algorithm by means of a heuristic simulation policy and the domains of the decision variables can be reduced by eliminating dominated value assignments (more details will follow later in Section 4). The algorithm has the ability to automatically learn how to navigate through the search space tree, just like regular Monte Carlo tree search for game playing can learn this for game trees. To speed up this learning process, and hence to keep learning in a very big search space manageable, the algorithm employs the idea of using a beam width (see for example [8]). By doing this, the search space is explicitly shrunk by removing possible solutions that do not look promising. The current article is the first study that uses all of these modifications at the same time. We will show that these modifications greatly improve the results in comparison with the non-modified version and that omitting any of the modifications yields (sometimes drastically) worse results. Second, the algorithm was empirically validated on two case studies using a set of extensive experiments requiring around 1,050 CPU-hours. For each case study, we instantiated the proposed algorithm by tailoring the problem specific components of the algorithm (i.e. we used a custom heuristic simulation policy, custom bounds on the objective function values and custom dominance rules for each case study). The first case study concentrates on the quay crane scheduling problem with non-crossing constraints [9, 10], which is classified in literature as $[1D||C_{Max}]$ according to the classification scheme proposed in [11]. For this case study, our algorithm surpasses the state-of-the-art results and several new best solutions are found. The second case study concentrates on one of the most studied combinatorial optimization problems: the 0-1 knapsack problem [12]. For this case study our algorithm typically produces near-optimal solutions that are slightly worse than the state-of-the-art results, but it needs only a small fraction of the time to do so.

The rest of this article is organized as follows: in Section 2 the Monte Carlo tree search algorithm for game playing is explained. An overview of the relevant literature for this article is given in Section 3, followed by the adaptations that we propose for Monte Carlo tree search in the context of combinatorial optimization (Section 4). Next, in Section 5 the quay crane scheduling problem with non-crossing constraints is introduced and the different problem specific components of the proposed algorithm are concretely demonstrated for this problem. Similarly, the same structure is followed in Section 6, but this time for the 0-1 knapsack problem. The computational results for both case studies are discussed in Section 7. Finally, a conclusion and possible ideas for future work are given in Section 8.

2. Monte Carlo tree search for game playing

Monte Carlo tree search is a heuristic search algorithm that is popular in game playing. Variants of this algorithm have been successfully applied to a variety of games (e.g. Havannah [13], Amazons [14], Lines of Action [15], Hex [16], Go [17], chess and Shogi [18]) and represent the state-of-the-art approach for many of them. The algorithm is capable of learning to play promising moves in a turn-based game. It does this by iteratively improving an estimate of how good a move is by using Monte Carlo sampling. Initially, these estimates are highly uncertain, but as more samples are collected, the estimates become more accurate. Thus, the algorithm's performance improves over time and it effectively learns how to play the game. The algorithm performs Monte Carlo sampling on a game tree to estimate the quality of the moves. A game tree is a tree in which each node represents a possible state of the game and the edges between the nodes represent the possibility to go from one state to another state by playing a move. The root node of the game tree represents the initial state of the game and leaf nodes correspond to final states (i.e. for two-player games a leaf node represents a state in which the game was won by one of the players). A detailed explanation of how the algorithm operates on the game tree follows next.

The algorithm keeps track of a part of the game tree, which we will denote as the *partial game tree* further on. The partial game tree is a subtree of the whole game tree with the same root node. Initially, the partial game tree only consists of a single node: the root node of the game tree. The partial game tree is iteratively grown until a certain computational budget is exhausted (e.g. a time limit or a number of iterations). In every iteration, one game is played until the end. Playing one game until the end corresponds to walking along a path from the root of the game tree to a leaf node. Every iteration consists of four phases:

Phase 1: Selection. The algorithm starts in the root of the game tree and will walk further down until it reaches a node of the game tree that does not yet belong to the partial game tree. When the algorithm walks down the game tree, it has to consecutively choose which child it will descend to. To make this decision, the algorithm tries to find a good balance between exploitation (playing moves that seem very good) and exploration (playing moves that have not been played very often in previous iterations in order to get a better estimate of how good the move is). To do this, it treats the problem of selecting which child to descend to for every node as a multi-armed bandit problem [19]. In a multi-armed bandit problem there is a player and there are k levers $l_1, l_2, ..., l_k$. With every lever l_i we associate a probability distribution P_i such that if lever l_i is pulled, the player receives a random reward sampled from P_i. These probability distributions are unknown to the player. The player will play multiple rounds and in each round the player will pull exactly one lever. The objective of the multi-armed bandit problem is to find a policy that determines which lever the player should pull in every round in order to maximize the expected sum of the received rewards. In the context of Monte Carlo tree search, this problem has to be solved every time a child node has to be selected from a given parent node. Hence, the levers in the multi-armed bandit problem correspond to the children of a node of the game tree in Monte Carlo tree search. The rewards correspond to 0 or 1 for a loss and win, respectively. Kocsis and Szepesvári solve this problem in their UCT algorithm [20] for Monte Carlo tree search by selecting the child for which the expression:

$$\frac{numberWins(parent, child)}{numberVisits(child)} + \sqrt{\frac{2 \cdot \ln(numberVisits(parent))}{numberVisits(child)}} \tag{1}$$

is maximized[1]. Here, $numberVisits(v)$ denotes the number of iterations in which node v was visited (belonged to a generated path) and $numberWins(v, w)$ denotes the number of iterations in which the edge from

[1] In [20], it was mentioned that the algorithm has to be implemented such that division by zero is avoided. Our approach to this corner case will be explained later in Section 4.

parent node v to child node w was followed and the game was won (by the player whose turn it is in the game state associated with node v). By maximizing this expression, the algorithm attempts to find a good balance between exploitation and exploration. The first term of this expression is the average win ratio and will be high for good moves, whereas the second term of this expression will be high for moves that have not been explored very often. It has been shown in [21] that this policy is optimal in the sense that the expected sum of rewards is asymptotically (when the number of rounds goes to infinity) as high as possible over all possible policies.

Phase 2: Expansion. The algorithm has arrived in a node which it has never visited before. For this node there is no information available from previous iterations to assess how good the possible moves are. The partial game tree is extended by adding this node to the tree.

Phase 3: Simulation. The algorithm will further walk down the game tree until a leaf node (a terminal game state) is reached. There are many possible policies to walk down the game tree. However, in the most basic version of Monte Carlo tree search, the policy recursively selects uniformly at random one of the available children of the current node until a leaf node is reached.

Phase 4: Backpropagation. The outcome of the game is evaluated by inspecting the game state associated with the leaf node. This information is backpropagated to all nodes v of the constructed path that belong to the partial game tree by updating $numberVisits(v)$ and $numberWins(v,w)$ (i.e. $numberVisits(v)$ is incremented and $numberWins(v,w)$ is incremented if and only if the player whose turn it is in the game state associated with v has won the game). The effect of this is that moves that were played during this iteration by the winner of the game will be more likely to be played again in future iterations.

These four phases are depicted in Figure 1. A partial game tree is shown for a turn based two-player game with a green player and a blue player (we refer readers of the physical journal to the online version of the paper, where colors are available). The colours of the nodes represent whose turn it is. For every node and edge, respectively $numberVisits(v)$ and $numberWins(v,w)$ are indicated. The moves that are played in Phase 1 are indicated by the arrows. In Phase 2, the partial game tree is extended and consists of ten nodes after the extension. Next, the algorithm walks further down the game tree in Phase 3 and ends in a final state in which the green player has won the game. Finally, the outcome of this game is backpropagated in Phase 4 by updating $numberVisits(v)$ and $numberWins(v,w)$ for all nodes and edges of the generated path in the partial game tree.

3. Literature overview

The algorithm that we propose in this article is a heuristic search algorithm based on Monte Carlo tree search that is enhanced in several ways in the context of solving combinatorial optimization problems. This algorithm was validated on two different case studies: the quay crane scheduling problem with non-crossing constraints (more specifically, the version that is known in literature as $[1D||C_{Max}]$) and the 0-1 knapsack problem. Hence, the relevant literature for this article comes from several different contexts. In this section, we will give an overview of related work concerning Monte Carlo tree search and we will also give an overview of the literature concerning the two problems from the case studies. The overviews are not meant to be exhaustive, but rather discuss the most important results relevant for this article. We refer the interested reader to [22], [23] and [12] for a more complete overview of respectively Monte Carlo tree search, the quay crane scheduling problem and the 0-1 knapsack problem.

3.1. Monte Carlo tree search

In this subsection we will discuss Monte Carlo tree search from the perspective of adaptations that have been proposed for the different phases of the algorithm as well as other contexts than game playing in which adaptations

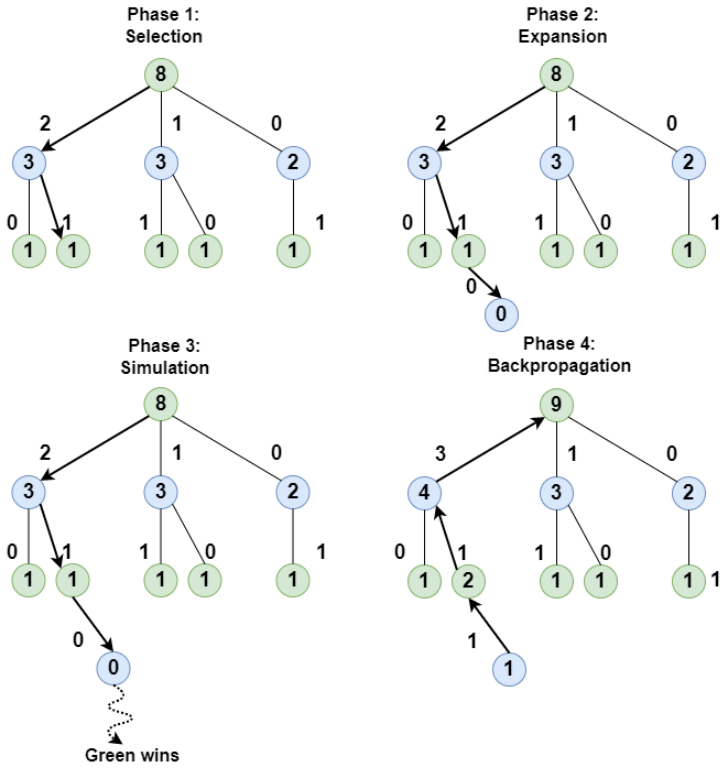

Figure 1: The four phases of an iteration of Monte Carlo tree search. Here, $numberVisits(v)$ and $numberWins(v,w)$ are indicated for every node and edge, respectively.

of Monte Carlo tree search have been used. Chaslot et al. propose *progressive bias* [24] to improve Monte Carlo tree search by changing expression (1) that is used to select children in Phase 2 of the algorithm. They add a weighted term to the expression that enables them to embed domain specific knowledge into the expression. In this term, a heuristic value is calculated for every possible child (moves that seem to be better according to the heuristic, get a higher value). The weight of this term (and hence the bias) becomes smaller after every iteration and goes to zero as the number of iterations goes to infinity. Winands et al. [25] further build upon this idea. They use this adapted expression only after the nodes have been visited a fixed number of times (and use the same policy as in the simulation phase otherwise) in order to avoid spending too much time on computing the heuristic values. Gelly et al. [26] embed domain specific knowledge into the selection phase of Monte Carlo tree search by integrating heuristic values for the moves that were calculated offline. Instead of initialising the values $numberVisits(v)$ and $numberWins(v,w)$ by 0, the heuristic values are used to appropriately initialise them, leading to a hot-start of the algorithm. Li et al. [27] propose a different static selection policy that optimally allocates a limited computing budget to maximize a lower bound on the probability of correctly selecting the best action at each node (whereas many papers focus on maximizing the cumulative rewards). Zhang et al. [28] recently proposed a closely related selection policy, which is dynamic instead of static. Baier and Winands [8] proposed Beam Monte Carlo tree search in which the number of traversed nodes at a given depth is restricted as soon as a certain number of iterations has been performed at that depth by only keeping the w most visited nodes. This idea is very similar to one of the modifications that we proposed in the context of Monte Carlo tree search for combinatorial optimization in this article (subsection 4.5), except for two differences. We keep the w most promising nodes according to the average objective function value instead of number of visits and the beam width restriction is applied for every level after a certain time has passed depending on the depth of the tree instead of a fixed number of iterations.

Winands et al. [29] focus on improving the simulation in Phase 3 of the algorithm. Instead of only evaluating the final game state as a winning state or losing state, they also attempt to prove for intermediate states the game-theoretic value (i.e. deciding whether a state is a proven loss or win). In case the game-theoretic value of a node can indeed be proven, the iteration is stopped and this value is backpropagated such that its outcome can be used again for future iterations. Drake et al. [30] attempt to improve the simulation by using domain specific knowledge. They use *heavy playouts*, in which a heuristic is used that is specifically tailored to the game under consideration as opposed to the default random move policy. Another approach is followed by AlphaGo [17], in which an artificial neural network is trained to predict the outcome of a game. Here, the game is not simulated in the simulation phase, but instead the outcome of the game is predicted and this value is used for backpropagation.

Gelly et al. [31] focus on improving Phase 4 of Monte Carlo tree search. In some games, all permutations of a set of moves lead to the same game state. To be able to share information amongst different states, they use the outcome of games in which an action a was played not only to update the node in which action a was played, but also all nodes in the subtree that has this node as root node. This provides a biased, but rapid value estimate. In their MC-RAVE algorithm [31], this bias decreases after many iterations by introducing a weight for the biased term that goes to zero as the number of iterations increases. Another attempt to improve the backpropagation phase can be found in [32]. Because the estimates of quality of the moves get better after every iteration, the estimates get more and more reliable. To account for this, a weight factor is introduced that gives a higher weight to iterations that are deemed more reliable.

Adaptations of Monte Carlo tree search have also been used in other contexts than game playing. It has been used in the context of constraint satisfaction problems by Previti et al. to obtain good results for structured instances of the SAT problem [33]. Satomi et al. [34] adapt Monte Carlo tree search to solve large-scale quantified constraint satisfaction problems in real-time. Perez et al. [35] use Monte Carlo tree search to build a controller for the physical travelling salesman problem (a real-time game). Two other surprising application domains for Monte Carlo tree search can be found in the work of Tanabe et al. where Monte Carlo tree search was used in a security context to detect *wolf attacks* [36] and the work of Dieb et al. [37] in which the task of automatic complex material design is studied.

It is also worth mentioning other uses of Monte Carlo tree search in a combinatorial optimization context and highlighting the most important differences with respect to the present article. In the work of Sabar et al. [38] Monte Carlo tree search is used in a hyper-heuristic [39] as a high level selection strategy to select a low-level heuristic. The most important difference with the present article is that the tree on which the algorithm operates is completely different. In the tree on which the algorithm operates in the work of Sabar et al., every node represents a low-level heuristic (e.g. a specific perturbation operator) and every path in the tree corresponds to a sequence of low-level heuristics. The goal is to find a sequence of low-level heuristics that can be applied to an initial starting solution in order to obtain a good solution. In the present article on the contrary, the tree on which the algorithm operates represents the search space of all possible assignments of values to decision variables and the goal is to find a leaf node (representing a solution in which all variables have been instantiated) for which the objective function value is as small as possible (without loss of generality). Another use of Monte Carlo tree search was reported in the work of Sabharwal et al. [40], in which Monte Carlo tree search was used to guide the node selection heuristic for the Mixed Integer Programming (MIP) solver CPLEX to select which node to expand. Again, the tree on which the algorithm operates is very different from the search space tree in the present article. Every node of the tree represents the solution of a linear programming relaxation of a mixed integer program and every edge corresponds to a branching decision for a non-integer variable. Furthermore, in the work of Sabharwal et al., no solution is constructed in the simulation phase of the algorithm, but instead a bound on the objective function value is used for the backpropagation phase.

We now turn our attention to literature in which the tree on which Monte Carlo tree search operates is the search space tree associated with a problem instance (as is also the case in the current article). Chaslot et al. [41] use Monte Carlo tree search to solve production management problems and obtain better solutions than an evolutionary planning heuristic. Liu et al. [42] propose a smart directed acyclic graph scheduling algorithm based on Monte Carlo tree search, in which suboptimal parts of the search space are pruned using a custom dual bound. Rosin [43] proposes the Nested Rollout Policy Adaptation algorithm (a variant of Monte Carlo tree search in which the rollout policy is

gradually adapted). They use this algorithm to produce good solutions for instances of Crossword Puzzle Construction and Morpion Solitaire. A more classical problem is tackled by Grelier et al. [44]. They combine several dedicated heuristics for the weighted vertex covering problem with Monte Carlo tree search and provide empirical evidence for the advantages and disadvantages of the different algorithmic variants. Similarly, Cazenave et al. [45] use the Nested Rollout Policy Adaptation algorithm and Nested Monte Carlo Search [46] (another Monte Carlo tree search variant) for solving the graph colouring problem. Although the results obtained by [45] do not surpass the state-of-the-art, they are almost as good (or equally good) for several problem instances. Runarsson et al. [47] compare the pilot method with Monte Carlo tree search to solve the Job Shop Scheduling problem [48] and conclude that the latter algorithm often outperforms the former one. Edelkamp et al. [49] use the Nested Rollout Policy Adaptation algorithm to solve several variations of classical problems in logistics: a vehicle routing problem, a motion planning problem and a container packing problem. For each problem, they report a promising solution for one problem instance.

As we can see from this subsection, several variants of Monte Carlo tree search have been proposed throughout the years. Given that the literature on Monte Carlo tree search is very large, it is not surprising to see that some of the enhancements from the current article (see Section 4) also appear in a limited number of other articles. However, they typically appear in isolation, whereas the current article is the first study that uses all enhancements at the same time. As we will later show in Section 7, the effect of omitting one or more of these enhancements can be quite big and the best results are obtained when all the enhancements are used. Using all enhancements at the same time is especially important for the most challenging problem instances.

3.2. The quay crane scheduling problem with non-crossing constraints

Quay cranes are used in ports for the loading and unloading of the bays of a container vessel. In order to guarantee an efficient execution, an important problem that arises in this context consists of finding a schedule for the quay cranes such that some quality criteria (e.g. the makespan or crane utilization rate) are optimized. The quay crane scheduling problem was first formulated by Daganzo in 1,989 [50]. They proposed an MIP to solve the quay crane scheduling problem, which was later improved in 1,990 by Peterkofsky et al. [51]. These two seminal papers attracted a lot of attention to this problem and led to a vast amount of literature that is concerned with modeling and solving it. The models in literature all describe the same concept of quay crane scheduling, but differ significantly in several aspects (e.g. the level of detail of the tasks, the objective function to be optimized, the given problem parameters and constraints). From the point of view of combinatorial optimization, this makes it difficult to compare the different models, because a problem instance for one formulation of the problem is not necessarily a valid problem instance for another formulation. Recently, a classification scheme was proposed by Boysen et al. [11] to group the different formulations in different classes according to three criteria: the terminal layout, the characteristics of the container moves and the objective function. The problem that we study in this article is classified with the name $[1D||C_{Max}]$ according to the classification scheme of Boysen et al. [11]. In this problem the terminal layout can be considered to be a one-dimensional line along which both the quay cranes and bays are located. There are no special characteristics of the container moves and the objective function that has to be minimized is the makespan (the last completion time of a bay). For this reason, we will restrict the rest of this subsection to an overview of related literature for the problem $[1D||C_{Max}]$ instead of general quay crane scheduling literature.

Zhu and Lim [52] model the quay crane scheduling problem as an integer program, which they solve by a branch-and-bound algorithm and also propose a simulated annealing algorithm for larger instances. This approach is significantly improved by Lim et al. [53] by making the crucial observation that instead of directly focusing on the schedule itself, it suffices to focus on the allocation of quay cranes to bays. More specifically, they prove that there exists an optimal schedule for their problem such that the cranes move in only one direction and this schedule can be derived from the allocation of quay cranes to bays. They also prove that the quay crane scheduling problem with non-crossing constraints is NP-hard and propose an approximation algorithm which guarantees an approximation factor of 2. Lee et al. [54] also propose an MIP for small problem instances and a genetic algorithm for larger problem instances. Lee and Chen [9] later realised that the models from [52], [53] and [54] can lead to solutions in which two quay cranes

would have to occupy the same position at the same time and they propose an MIP that mitigates this deficiency (the specific quay crane scheduling problem that is studied in the present article follows the model proposed in [9]). They also propose a heuristic two-stage algorithm, which they call enhanced best partition (EBP). In the first stage of the algorithm, the bays are partitioned into consecutive areas such that the resulting solution guarantees an approximation factor of 2. In the second stage, this solution is improved by merging adjacent areas into one area until no further improvements can be found. Lee and Wang [55] study an extension of $[1D||C_{Max}]$: the integrated discrete berth allocation and quay crane scheduling problem and propose an MIP for small problem instances and a genetic algorithm for larger instances. Another extension is studied in [56], where the quay crane scheduling problem is integrated into the truck scheduling problem. They propose both an MIP and an algorithm based on Particle Swarm Optimization to solve the problem. Santini et al. [10] improve the MIP model of Lee and Chen [9] by introducing a family of valid inequalities such that the set of feasible solutions does not change, but the efficiency of the MIP solver CPLEX is greatly affected. Finally, Zhang et al. [57] propose an approximation algorithm for $[1D||C_{Max}]$ and prove that their approximation factor is smaller than 2 (the best known approximation ratio until then). The achieved approximation ratio is $2-\frac{2}{m+1}$, where m denotes the number of quay cranes.

3.3. The 0-1 knapsack problem

The 0-1 knapsack problem is a classical problem which has already received attention for several decades. In this problem, the goal is to select several items from a given set of n items to include in a knapsack with limited capacity c such that the sum of the weights of the selected items does not exceed c and the sum of their profits is maximized. The problem is NP-hard [58], but it can be solved in pseudo-polynomial time (see e.g. [59] for a solution with a worst-case time complexity of $O(n \cdot c)$) and several fully polynomial time approximation schemes exist (see e.g. [60], [61], [62] and [63]). This is mainly interesting from a theoretical point of view, but it does not immediately give rise to algorithms with a good practical performance [12]. However, such practical algorithms do exist. Many of these algorithms are based on ordering the items according to the ratio between the profit and the weight of an item. This ordering can be used to efficiently solve the linear programming relaxation of the 0-1 knapsack problem and this gives rise to a well known upper bound by Dantzig [59]. This ordering is also used by a simple greedy heuristic that selects items in this order as long as the knapsack capacity is not exceeded. The resulting solution often differs from the optimal solution by only a few items whose profit-weight ratio is close to a certain distinguished item called the split item [12]. Martello et al. [64] proposed an algorithm that first determines a small subset of items, which is called the core, such that for items in the core it is usually difficult to decide whether they will belong to an optimal solution, whereas for items outside of the core this decision is usually easy. Next, the algorithm optimally selects items from the core and combines these with items outside of the core. A drawback of this algorithm is that the core size is fixed, but it is very difficult to know beforehand which core size will be appropriate. This drawback was tackled by Pisinger [65], who proposed a branch-and-bound algorithm where the core is adaptively extended each time the algorithm reaches the border of the core. This was later improved even further by Pisinger [66] by handling the core enumeration process more efficiently. Finally, Martello et al. [67] proposed a combination of all the previous techniques, which resulted in their famous Combo algorithm. Although this algorithm was invented more than twenty years ago, it still represents the current state-of-the-art (see e.g. [68], [69], [70] and [71] for relatively recent articles that support this claim). It is able to exactly solve many problem instances containing several thousands of items in a matter of (milli)seconds. In an attempt to find the most challenging knapsack problem instances, Pisinger [72] has proposed 15 different classes of instances. Recently, another class of very hard problem instances was proposed in [73]. Later in Section 7, we will focus in more detail on the most difficult class of problem instances proposed in [72] (strongly correlated spanner instances) and the class proposed in [73] (noisy multi-group exponential problem instances).

4. Monte Carlo tree search for combinatorial optimization

We propose an adapted version of Monte Carlo tree search for game playing (see Section 2) that is more suitable for solving a combinatorial optimization problem. We assume that all decision variables are discrete, although in some cases the algorithm remains applicable even when the decision variables are continuous, as will be demonstrated in Section 5. By leveraging the combinatorial structure of a problem, the algorithm can be enhanced in several ways, which are discussed in this section. Later in Section 5 and 6, we will illustrate these enhancements by tailoring them to the specific combinatorial optimization problems of the two case studies. In this section we will employ the standard terminology used for combinatorial optimization problems and we refer readers who are unfamiliar with this terminology to [74].

The algorithm will operate on the search space tree associated with a problem instance. Every node in this search space tree represents a partial solution (i.e. a solution for which at least one decision variable has not yet been assigned a value) except for the leaf nodes, which represent solutions in which all decision variables have been instantiated. The root of the tree corresponds to a partial solution in which no decision variable has been instantiated yet. Every edge represents the assignment of a value to a decision variable and all the edges on the same level of the tree correspond to value assignments to the same decision variable. Here, we assume that a natural ordering of the decision variables exists, which can often be obtained by modeling the optimization problem as making a natural sequence of decisions (note that the ordering of the decision variables often follows directly from the other components of the algorithm, as we will show later). We refer the interested reader to [75] for an overview of heuristics that can be employed in case such a natural ordering does not exist. Every path from the root of the tree to a leaf node contains every decision variable exactly once. With this search space representation, solving the optimization problem now corresponds to finding a path from the root to a leaf node such that the associated solution is feasible and its objective function value is as small as possible. With this in mind, the algorithm will keep track of the best found feasible solution and the final result of the algorithm is the best solution found after all iterations have been executed.

An example of a search space tree is given in Figure 2. For this problem instance, there are two decision variables X and Y that both have the same domain $\{1,2,3\}$. The search space tree is associated with a toy problem instance with objective function $(X-Y)^3-|X-Y|$ (to be minimized) and a single constraint $X \neq Y$. The search space tree begins by enumerating the domain of X on the first level, followed by an enumeration of the domain of Y on the second level. The root node corresponds to a partial solution in which no decision variable has been instantiated yet. This node has three children, each of which corresponds to a partial solution where the decision variable X has been instantiated. The leaf nodes of the tree correspond to solutions (either feasible or infeasible) in which all decision variables have been instantiated. These solutions are marked in green and red, respectively, with the optimal feasible solution having $X=1, Y=3$ and an objective function value of -10.

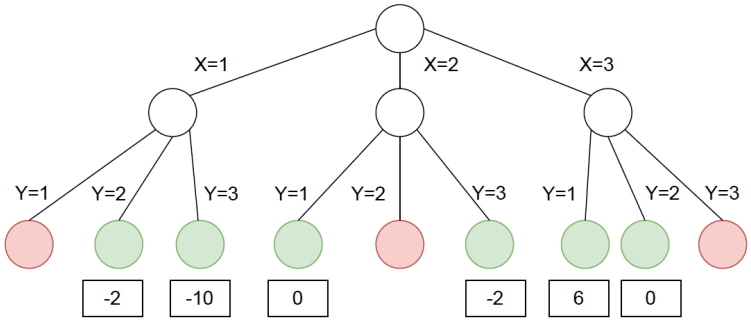

Figure 2: An example of a search space tree for a toy problem instance

The algorithm from the previous section was designed for game playing, but in this article the scope is combinatorial optimization. The similarity between these two is that there is the concept of a game tree in game playing

and the concept of a search space tree in combinatorial optimization. Finding a solution for a problem instance of a combinatorial optimization problem can also be thought of as a kind of game in which the moves correspond to assigning a value to a decision variable. However, there are certain important differences between game playing and combinatorial optimization and these differences are the reason for the changes that we propose to the Monte Carlo tree search algorithm from the previous section. These differences are as follows:

- First, there are typically only at most three possible outcomes for a game (win, loss and sometimes tie), whereas the objective function value of a problem instance of a combinatorial optimization problem can be any real number.
- Second, the possible moves for a game are determined by the rules of the game and thus cannot be freely chosen, but the possible values that can be assigned to the decision variables for a problem instance of a combinatorial optimization problem depend on what these decision variables are (i.e. how the problem is modeled). Note that the same combinatorial optimization problem can often be modeled in several ways such that each one has different decision variables, leading to very important consequences regarding for example the strength of bounds that can be derived from the model or the size of the search space.
- Third, there are several games that start from a fixed game state (e.g. chess and Go) and hence there is precisely one game tree associated with every game. For combinatorial optimization, however, there is one search space tree associated with every problem instance (as opposed to one search space tree for every problem). Since we are typically interested in solving several problem instances, this puts an additional burden on the available time.
- Fourth, the search spaces in the context of combinatorial optimization tend to be several orders of magnitude larger than the number of game states for game playing. For instance, the number of game states for Go and chess are at most equal to 10^{171} [76] and 10^{47} [77], respectively, whereas the number of solutions for a problem instance with 1,000 binary decision variables is equal to $2^{1,000} > 10^{300}$. However, the search spaces in the context of combinatorial optimization tend to have a certain structure that can be exploited to efficiently search them (e.g. using dominance and bounding rules). These differences motivate the enhancements that we propose, which are discussed next.

4.1. Domain reduction

When the algorithm arrives in a node, several decision variables have already been instantiated. The edges to the children of the current node represent all the possible assignments of values to the current decision variable (i.e. the domain is enumerated). By using problem specific information, it is sometimes possible to prove that a certain value assignment to the current decision variable is *dominated* by another value assignment, given the current partial solution. We make the concept of dominating more precise in the following definition:

Definition 1. *Assume (without loss of generality) that we are dealing with a minimization problem and that there are n decision variables $a_1, a_2, \ldots, a_n$, which can each be instantiated by a value of their finite domains $d_1, d_2, \ldots, d_n$. Let $[a_1 \leftarrow v_1, a_2 \leftarrow v_2, \ldots, a_n \leftarrow v_n]$ denote a solution where decision variable a_i has been instantiated with value $v_i \in d_i$ ($\forall 1 \leq i \leq n$) and let $f([v_1, v_2, \ldots, v_n])$ denote the objective function value associated with the solution $[a_1 \leftarrow v_1, a_2 \leftarrow v_2, \ldots, a_n \leftarrow v_n]$ (equal to ∞ if the solution is infeasible). Consider a partial solution $[a_1 \leftarrow v_1, a_2 \leftarrow v_2, \ldots, a_k \leftarrow v_k]$ where the first $k < n$ values have already been instantiated. We say that value assignment $a_{k+1} \leftarrow v_{x_{k+1}}$ is dominated by value assignment $a_{k+1} \leftarrow v_{y_{k+1}}$ (with $v_{x_{k+1}}, v_{y_{k+1}} \in d_{k+1}$), given the partial solution $[a_1 \leftarrow v_1, a_2 \leftarrow v_2, \ldots, a_k \leftarrow v_k]$ if and only if*
$\forall v_{x_{k+2}} \in d_{k+2}, v_{x_{k+3}} \in d_{k+3}, \ldots, v_{x_n} \in d_n : \exists v_{y_{k+2}} \in d_{k+2}, v_{y_{k+3}} \in d_{k+3}, \ldots, v_{y_n} \in d_n :$
$f([v_1, v_2, \ldots, v_k, v_{y_{k+1}}, v_{y_{k+2}}, \ldots, v_{y_n}]) \leq f([v_1, v_2, \ldots, v_k, v_{x_{k+1}}, v_{x_{k+2}}, \ldots, v_{x_n}])$.

Hence, in the context of combinatorial optimization such children that lead to dominated solutions can be removed, despite the fact that they may represent feasible solutions.

4.2. Pruning subtrees by calculating bounds

When the algorithm reaches a node it has never visited before, a dual bound on the objective function is calculated to determine whether it is worthwhile to continue building a solution in the current iteration or not. The bound that is calculated is a lower bound for a minimization problem and an upper bound for a maximization problem, respectively. This bound is compared with the best solution value found so far and, in case it is impossible to find a solution that is better than this, the iteration is stopped. This bound is stored such that it can be used in further iterations without having to recalculate the bound. If the bound is worse than the current best objective function value, the current node is deleted as a child node of its parent node to avoid revisiting this node in further iterations (i.e. the subtree rooted at the current node is pruned). After deleting a child node, it is possible that a parent does not have any children left, which means in turn that there is no leaf node in the subtree rooted at the parent node that is better than the best solution found so far. In this case, we can also delete this parent node as a child node of its own parent node. This process can continue and hence deleting a node should be implemented in a recursive fashion such that the deletion of a node can give rise to the subsequent deletion of ancestors of this node.

4.3. Using a heuristic simulation policy

The default simulation policy for the most basic version of Monte Carlo tree search, as described in Section 2, chooses children uniformly at random until a leaf node is reached. In the context of combinatorial optimization, however, often problem specific constructive heuristics are available to construct a solution. If one uses a heuristic for the solution completion policy of Monte Carlo tree search, the algorithm can be seen as an algorithm that attempts to learn to correct incorrect choices of the heuristic by selecting a different path in every iteration during the selection phase, corresponding with different value assignments to the decision variables. Furthermore, since the heuristic is executed starting from the root node of the tree in the first iteration of the algorithm, it is also guaranteed that the solution quality of the algorithm is at least as good as the solution quality of the heuristic.

4.4. Selection Policy

To decide which child to descend to in the selection phase, the algorithm keeps track of two values (for all nodes and edges, respectively): $numberVisits(v)$ and $averageObjectiveFunctionValue(v,w)$. Here, $numberVisits(v)$ denotes the number of times that node v was visited and $averageObjectiveFunctionValue(v,w)$ denotes the average objective function value over all iterations in which the edge between parent node v and child node w belonged to the generated path. We constructed a similar expression as expression (1) that is used in Monte Carlo tree search for game playing to select a child node. In this expression the first term is the average win ratio and is always between 0 and 1. We replace this term by a similar term (applicable to combinatorial optimization problems) that is always between 0 and 1 and that represents the quality of the obtained solutions. We define the value $normalizedScore(v,w)$ for all edges between parent node v and child node w as follows: let v be a node, let $visitedChildren(v)$ denote the set of children of v that have been visited at least once and let $unvisitedChildren(v)$ denote the set of children of v that have not been visited before. We define $score(v,w)$ as the rank of node w that we obtain by sorting the nodes in $visitedChildren(v)$ in increasing order of quality according to the value $averageObjectiveFunctionValue(v,w)$ (i.e. edges that seem more promising get a higher score). Now $normalizedScore(v,w)$ is defined as:

$$normalizedScore(v,w) = \frac{score(v,w)}{\sum_{w' \in visitedChildren(v)} score(v,w')} \tag{2}$$

Hence, during the selection phase the algorithm will select the child for which the following expression is maximized (with ties broken in an arbitrary fashion):

$$normalizedScore(parent,child) + \sqrt{\frac{2 \cdot \ln(numberVisits(parent))}{numberVisits(child)}} \tag{3}$$

As mentioned in the previous section, the algorithm balances exploitation and exploration by maximizing this expression (the first term of this expression is high for promising value assignments, whereas the second term is high for infrequently used value assignments). We cope with the issue that this term is undefined for children that have not been visited before (because of division by 0) as follows: let $k_1 = |unvisitedChildren(v)|$ and let $k_2 = |visitedChildren(v)|$, where $|.|$ represents the size of a set. To choose which child node to descend to, the algorithm selects with probability $\frac{k_1}{k_1+k_2}$ a child uniformly at random from the set of unvisited children and with probability $1 - \frac{k_1}{k_1+k_2}$ the child for which expression (3) is maximized.

A numerical example is given in Table 1 for a minimization problem in which a parent node has five children to choose from. The parent node has already been visited seven times and during these iterations, three different children were visited while two children have never been visited before. These three children have an average objective function value of 759.3, 753.0 and 751.3 (ordered in increasing order of quality). The second and the third column in this table (*numberVisits* and *averageObjectiveFunctionValue*) completely determine the values in all other columns.

Table 1: Numerical example of calculating the selection probabilities for a minimization problem

	numberVisits	*averageObjectiveFunctionValue*	*score*	*normalizedScore*	expression (3)	selection probability
Child 1	3	751.3	3	$\frac{3}{1+2+3}$	1.64	0 %
Child 2	3	759.3	1	$\frac{1}{1+2+3}$	1.31	0 %
Child 3	1	753.0	2	$\frac{2}{1+2+3}$	2.31	60 %
Child 4	0	-	-	-	-	20 %
Child 5	0	-	-	-	-	20 %

Special care must be taken when the values $averageObjectiveFunctionValue(v,w)$ are updated if an iteration produces an infeasible solution, because the objective function value of an infeasible solution is not meaningful. In this case, the algorithm will add a flag to the node that was added in the expansion phase, which prevents the algorithm from revisiting this node in any further iterations. Hence, the subtree rooted at this node is heuristically removed from the search space tree and the values $averageObjectiveFunctionValue(v,w)$ remain unchanged, because the parent node is regarded as having one less feasible child. This special case also illustrates the need for a well-performing heuristic simulation policy, because the algorithm punishes infeasible solutions by removing the node from which the heuristic simulation policy was started.

4.5. Directing the search by using a beam width

To be able to learn for a given parent node which child node will lead to promising solutions, the child nodes should each be visited several times. However, the deeper the algorithm descends in the tree the more nodes there are at a given depth. For large search space trees it is not computationally feasible to visit all nodes several times, because if this were the case one could simply enumerate the whole search space. For this reason, we will shrink the search space to focus only on the most promising parts. This is done as follows: let d be the number of decision variables (i.e. the depth of the search space tree) and t be the number of seconds for which the algorithm is run. The algorithm will work in d stages such that each stage takes $\frac{t}{d}$ seconds. After stage i, the algorithm will only keep a beam of the w most promising nodes at depth i of the search space tree (according to the value *averageObjectiveFunctionValue*), where w is an algorithm parameter that denotes the width of the beam. For the sake of computational efficiency, this should be implemented in a lazy way such that one should not explicitly delete all but the w most promising nodes, but instead indicate for those w nodes that they were not deleted and treat all other nodes as if they were deleted. Note that if there are at most w nodes at a given depth, all nodes at this depth will be kept.

4.6. Generalizability

Although it is clear that the proposed algorithm could in principle be applied to any optimization problem where the decision variables all have finite domains, it might work better on certain problems than on others. In this subsection

we will briefly highlight several desirable properties of a problem that make it well-suited for the proposed algorithm. First, the algorithm can be expected to perform better when the domains of the decision variables are not too big. This is the case, because in order to learn for a particular decision variable which value assignments tend to yield better solutions, all possible value assignments to that variable should be tried out at least once and preferably several times. Recall that for every edge (v,w) the value $averageObjectiveFunctionValue(v,w)$ is stored to guide the algorithm in the selection phase and this value gets more accurate as more iterations are performed. Hence, the average branching factor of the tree should not be too high. However, the number of decision variables (i.e. the depth of the tree) is a somewhat less limiting factor, as will also become more clear in Section 7. Second, using stronger domain reduction rules will likely have a positive effect on the algorithm's performance and problems for which strong domain reduction rules are available are better suited for the algorithm. This is immediately linked to the first point, because strong domain reduction rules can eliminate more values from the domains of variables. Third, the tightness of the bounds used for pruning subtrees and the complexity of computing them both play an important role in the algorithm. In case tight bounds are available, large parts of the search space can be eliminated, allowing the algorithm to focus on more promising parts of the search space. The complexity of computing the bounds has an immediate effect on the number of iterations that the algorithm can perform in a given time. Usually it takes more time to compute stronger bounds, so it is important that a good trade-off between tightness and complexity is found.

5. Case study A: quay crane scheduling problem with non-crossing constraints

The proposed algorithm was validated on two specific combinatorial optimization problems. In the first case study, we focus on the quay crane scheduling problem with non-crossing constraints. In the crane scheduling literature, it is classified with the name $[1D||C_{Max}]$ according to the classification scheme proposed in [11]. In this problem, we are given a set of n bays $B = \{0,1,\ldots,n-1\}$ on a container vessel and a set of m quay cranes $K = \{0,1,...,m-1\}$. Both the bays and the quay cranes are placed from left to right on a one-dimensional line, where the lower numbered bays and quay cranes are more towards the left. An integer processing time p_b is associated with every bay. The time it takes for the quay cranes to move parallel to the bays is assumed to be negligible in comparison with the processing times. The goal of the problem is to find a schedule that determines which quay crane has to process which bay at which time such that the makespan (the last completion time of a bay) is minimized. Furthermore, there are several constraints that need to be satisfied. The schedule has to be non-preemptive, which means that once a quay crane starts processing a bay, the quay crane keeps on processing this bay until it is finished. Hence, every bay is processed by exactly one quay crane. Because the quay cranes are located along a line from left to right, the quay cranes cannot cross each other. More specifically, if quay crane k_1 is processing bay b_1 at the same time as quay crane k_2 is processing bay b_2, with $k_1 < k_2$, then it must hold that $b_1 < b_2$. Finally, there are also constraints that indicate that at all times the quay cranes must leave enough space for the other quay cranes. An illustration of a toy problem instance and its solution is given in Figure 3, which will be discussed in more detail in Subsection 5.1.

5.1. Mathematical model

To formalize the quay crane scheduling problem with non-crossing constraints, we use the mathematical model proposed by Santini et al. [10]. In this model, the problem variables are the number of bays n in the set B, the number of quay cranes m in the set K and the processing times of the bays p_b ($\forall b \in B$). The value M denotes a sufficiently large constant. The decision variables are the binary variables x_{bk}, the binary variables $y_{bb'}$, the real variables c_b and the real variable c. The meaning of these decision variables is as follows: x_{bk} indicates whether bay b will be processed by quay crane k, $y_{bb'}$ indicates whether the processing on bay b finishes before the processing on bay b' starts, c_b denotes

the completion time of bay b and c indicates the makespan. The model is given by:

$$\text{minimize} \quad c \tag{4a}$$

$$\text{subject to} \quad c \geq c_b, \qquad \forall b \in B, \tag{4b}$$

$$c \geq \sum_{b \in B} x_{bk} p_b, \qquad \forall k \in K, \tag{4c}$$

$$c_b \geq p_b, \qquad \forall b \in B, \tag{4d}$$

$$\sum_{k \in K} x_{bk} = 1, \qquad \forall b \in B, \tag{4e}$$

$$c_b \leq c_{b'} - p_{b'} + M(1 - y_{bb'}), \forall b, b' \in B, b \neq b', \tag{4f}$$

$$\sum_{k \in K} k x_{bk} - \sum_{k \in K} k x_{b'k} + 1 \leq M(y_{bb'} + y_{b'b}), \qquad \forall b, b' \in B, b < b', \tag{4g}$$

$$\sum_{k \in K} k x_{b'k} - \sum_{k \in K} k x_{bk} \leq b' - b + M(y_{bb'} + y_{b'b}), \forall b, b' \in B, b < b', \tag{4h}$$

$$x_{bk} = 0, \qquad \forall b \in B, k \in K, k > b, \tag{4i}$$

$$x_{bk} = 0, \qquad \forall b \in B, k \in K, n - b < m - k, \tag{4j}$$

$$x_{bk} \in \{0, 1\}, \qquad \forall b \in B, k \in K, \tag{4k}$$

$$y_{bb'} \in \{0, 1\}, \qquad \forall b, b' \in B, b \neq b', \tag{4l}$$

$$c_b \in \mathbb{R}, \qquad \forall b \in B, \tag{4m}$$

$$c \in \mathbb{R} \tag{4n}$$

The meaning of these constraints is as follows. Constraints (4b) express that the makespan is at least as large as the completion time of all bays while constraints (4c) express that the makespan is also at least as large as the sum of the processing times of all bays that are processed by the same quay crane. Constraints (4d) express that the completion time of a bay is at least as large as its processing time. The fact that every bay is processed by exactly one quay crane is enforced by constraints (4e). Constraints (4f) define the decision variables $y_{bb'}$: they are equal to 0 if the processing on bay b finishes later than the processing on bay b' starts. Next, constraints (4g) ensure both that a quay crane can only process one bay at a given time and that two quay cranes that are processing a different bay at the same time have not crossed each other. Constraints (4h)-(4j) all ensure that quay cranes have enough space at all times. More specifically, constraints (4h) enforce that two quay cranes that are working at the same time leave enough space (at least one bay) for all quay cranes that are in between them, while constraints (4i) and (4j) enforce that quay cranes will not be pushed off the left and right side of the ship, respectively. Finally, constraints (4k)-(4n) indicate the domain of the decision variables.

5.2. Relaxation

The model above contains decision variables whose domain is not finite (c_b and c) and hence it cannot be directly used for the algorithm proposed in this article. However, the approach that we will take to solve this problem is to instead generate solutions for a relaxation of this problem (using a model whose decision variables are all discrete) and choose the best found solution for this relaxation that is also feasible for the original problem. The relaxation that we will consider, is the relaxation studied by Lim et al. [53]. This relaxation is obtained by taking the original model and removing the constraints that indicate that the quay cranes must leave enough space for other quay cranes (constraints (4h)-(4j)). The decision variables of this relaxation are the same as the decision variables of the original problem, so doing this does not immediately help. However, Lim et al. have also shown that this relaxation is equivalent with another model in which all decision variables are discrete. For this equivalent model, the only decision variables are

the n decision variables σ_b ($\forall b \in B$), which indicate which quay crane will process bay b. Hence, the domain of σ_b is $K = \{0, 1, ..., m-1\}$. They have shown that if we know the values of these decision variables (i.e. if we know which quay cranes will process which bays), we can determine the schedule with minimal makespan by letting all quay cranes process their assigned bays from left to right, where each quay crane moves to its next assigned bay as soon as its path to the next assigned bay is free (no other quay crane is blocking its path). This schedule (and the resulting objective function value) can be calculated as follows: let $earliestTime[k][b]$ denote the earliest time that quay crane k can move past bay b. Now all $earliestTime$ values can be computed with a time complexity of $O(n \cdot m)$ by dynamic programming by iteratively updating $earliestTime$ for subsequent bays b (the outer loop of the algorithm) and quay cranes k (the inner loop of the algorithm). The algorithm will first increase $earliestTime[k][b]$ by the processing time p_b of bay b in case bay b is processed by quay crane k (i.e. $\sigma_b = k$) and later it might be increased further in case $earliestTime[k+1][b] > earliestTime[k][b]$, because every quay crane has to wait for the quay crane immediately to its right to avoid that quay cranes cross each other. This is shown in the pseudocode of Algorithm 1:

Algorithm 1: Computes *earliestTime*

```
Function computeEarliestTime(m, n, σ, p)
    Data:
    m: the number of quay cranes
    n: the number of bays
    σ: an array representing the allocated quay crane for every bay
    p: an array representing the processing time for every bay
    Result:
    The matrix earliestTime
    /* Start of code                                                    */
    earliestTime ← emptyMatrix(m,n);
    for b in {0,1,...,n−1} :
        for k in {m−1,m−2,...,0} :
            /* Initialization                                           */
            if b ≥ 1 then
                earliestTime[k][b] ← earliestTime[k][b−1];
            else
                earliestTime[k][b] ← 0;
            /* Bay b is processed by quay crane k                       */
            if σ[b] == k then
                earliestTime[k][b] ← earliestTime[k][b] + p[b];
            /* Quay crane k has to wait for quay crane k+1              */
            if k+1 < m and earliestTime[k+1][b] > earliestTime[k][b] then
                earliestTime[k][b] ← earliestTime[k+1][b];
    return earliestTime;
```

Because every quay crane waits for the quay crane immediately to its right, it holds that $earliestTime[0][b] \geq earliestTime[1][b] \geq \ldots \geq earliestTime[m-1][b]$ ($\forall b \in B$). Hence, the makespan c is equal to the earliest time that the first quay crane can move past the last bay ($earliestTime[0][n-1]$).

A toy problem instance and its optimal solution is given in Figure 3. In this problem instance, there are $m = 2$ quay cranes that have to process $n = 4$ bays with a processing time of 5, 9, 2 and 1 time units (from left to right). A possible solution for this problem instance is for the first quay crane to process the first and the third bay and for the second quay crane to process the second and the fourth bay. The $earliestTime$ values are calculated according to the pseudocode of Algorithm 1. For example, to calculate the $earliestTime$ values in the second column, the algorithm first sets $earliestTime[1][1]$ to $0+9=9$, because $earliestTime[1][0] = 0$, $p[1] = 9$ and $\sigma_1 = 1$. The algorithm then sets $earliestTime[0][1]$ to 9, because $max(earliestTime[0][0], earliestTime[1][1]) = 9$. The corresponding schedule

that is associated with this solution is depicted on the Gantt chart, where a dotted line represents that a quay crane is waiting and a full line represents that a quay crane is processing a bay. The makespan of this schedule is 11 (equal to $earliestTime[0][3]$ as desired), which is optimal. This solution is feasible for both the original problem (Subsection 5.1) and the relaxed version (Subsection 5.2).

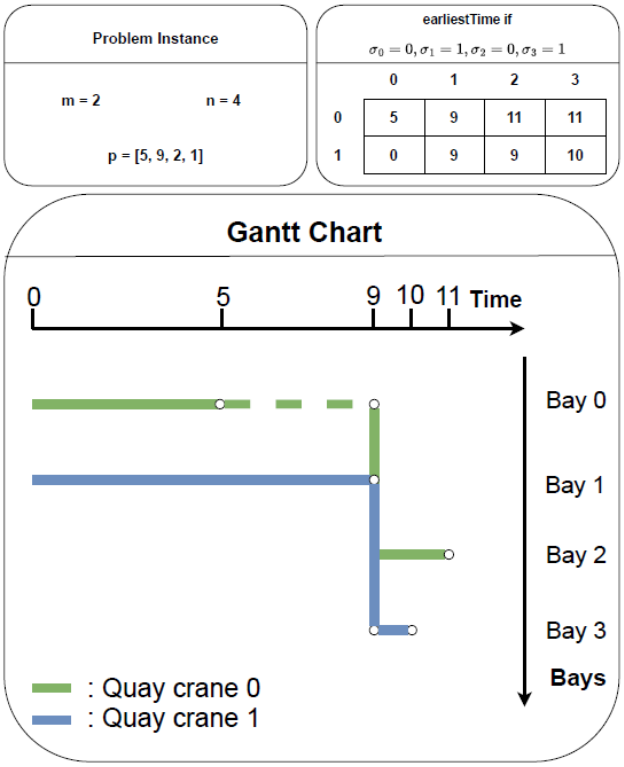

Figure 3: A toy problem instance and an optimal solution with an objective function value of 11

5.3. Enhancements used in our algorithm

Finally, we will demonstrate for the relaxation above how we can enhance Monte Carlo tree search by reducing the domain of the decision variables, pruning subtrees by calculating bounds and using a heuristic solution completion policy.

5.3.1. Domain reduction

The generated solutions of the relaxation will be tested for feasibility for the original problem. Some of the constraints that were left out in the relaxation can however be directly satisfied by appropriately reducing the domains of the decision variables. More specifically, constraints (4i) and (4j) are constraints that prohibit certain value assignments to the decision variables and can easily be satisfied by eliminating those values from the domains of the decision variables.

Another possibility for domain reduction makes use of Theorem 1, which was proven in [53]:

Theorem 1. *Suppose that the values of the first b decision variables $\sigma_0, \sigma_1, \ldots, \sigma_{b-1}$ are known and the algorithm currently has to choose the value for decision variable σ_b. If there exists a quay crane $k \in K$ such that $earliestTime[k][b-1] = earliestTime[k+1][b-1]$, then quay crane $k+1$ can be removed from the domain of σ_b without changing the optimum.*

In order to reduce the domains by using this theorem, the *earliestTime* values should be known before a complete solution is constructed, at the moment where the value for σ_b is chosen. This can be done by updating the *earliestTime* values on the fly, by executing the inner loop of Algorithm 1, every time a value is assigned to a decision variable.

5.3.2. Pruning subtrees by calculating bounds

In the proposed algorithm, it is possible to prune subtrees by keeping track of the best solution found so far and comparing the associated objective function value with a lower bound on the objective function. In case the best found solution so far cannot be improved by further descending into the current subtree, it is pruned. Two lower bounds were proposed by Lim et al. [53] for the relaxation that was discussed earlier. Suppose that the values of the first b decision variables $\sigma_0, \sigma_1, \ldots, \sigma_{b-1}$ are known and the algorithm currently has to choose the value for decision variable σ_b. Now, the first lower bound is given by:

$$earliestTime[m-1][b-1] + \max(p_b, p_{b+1}, ..., p_{n-1}) \tag{5}$$

This bound can easily be proven by realising that every bay must be assigned to a quay crane and the fact that $earliestTime[0][b] \geq earliestTime[1][b] \geq \ldots \geq earliestTime[m-1][b]\ \forall b \in B$ (see earlier). Hence, assigning the remaining bay with the largest processing time to the earliest available quay crane is indeed a lower bound for the objective function value.

The second lower bound that was proposed, is given by:

$$earliestTime[0][b-1] + \max\left(0, \left\lceil \frac{\sum_{i=b}^{n-1}(p_i) - \sum_{i=1}^{m-1}(earliestTime[0][b-1] - earliestTime[i][b-1])}{min(m, n-b)} \right\rceil\right) \tag{6}$$

The fraction in this lower bound represents the following: first the sum of the processing times of all remaining bays is distributed over the quay cranes until the first quay crane does not have to wait anymore. After doing this, all the quay cranes have the same *earliestTime* values, equal to $earliestTime[0][b-1]$. Now there are at most $(n-b)$ bays left to process for m quay cranes and the total remaining processing time is evenly spread over $min(m, n-b)$ quay cranes. Finally this result can be rounded up, because all processing times are integers and hence the real optimum has to be an integer as well. When spreading the processing times over the quay cranes, the constraint that every bay has to be assigned to exactly one quay crane is ignored and hence the obtained value is indeed a lower bound for the real optimal objective function value.

Both lower bounds can be calculated efficiently by doing an appropriate preprocessing step in $O(n)$ time, once before the start of the algorithm. The final lower bound that is used to compare against the current best solution is the maximum of both lower bounds. To calculate the first lower bound (expression (5)), we have to be able to calculate $\max(p_b, p_{b+1}, ..., p_{n-1})$ efficiently without having to iterate over all the values inside the maximum. This can be achieved by calculating and storing the maximum of every suffix of $p_0, p_1, \ldots, p_{n-1}$ during a preprocessing step. These values can then be retrieved in $O(1)$ every time the lower bound is calculated. Similarly, to calculate the second lower bound (expression (6)), we have to be able to calculate $\sum_{i=b}^{n-1} p_i$ and $\sum_{i=1}^{m-1} earliestTime[0][b-1] - earliestTime[i][b-1]$ efficiently. Here, $\sum_{i=b}^{n-1} p_i$ can be calculated for every suffix in the preprocessing step, completely analogous to the approach that was used to calculate $\max(p_b, p_{b+1}, ..., p_{n-1})$. To calculate $\sum_{i=1}^{m-1} earliestTime[0][b-1] - earliestTime[i][b-1]$, however, the *earliestTime* values depend on the choices that were made in the search space tree and hence cannot be calculated in the preprocessing step. Instead, a variable representing the value of this expression is stored and this value is updated on the fly when the *earliestTime* values are changed.

5.3.3. Using a heuristic simulation policy

When the algorithm arrives in a node which it has never visited before, the values for the remaining decision variables are chosen according to a heuristic. Suppose that the value of the first b decision variables $\sigma_0, \sigma_1, \ldots, \sigma_{b-1}$ are already known, then the algorithm will subsequently choose the values for $\sigma_b, \sigma_{b+1}, \ldots, \sigma_{n-1}$ as follows. The

algorithm consecutively tries to assign every value from the domain of σ_b to σ_b and updates the *earliestTime* values and calculates the lower bound from the previous subsection. The final value that will be assigned to σ_b is the value for which the calculated lower bound is as low as possible, which reflects the fact that we want the objective function value of the solution that is obtained in the end to be as low as possible. After doing this, the first $b+1$ decision variables are already instantiated and this process is repeated until all n decision variables are instantiated.

6. Case study B: 0-1 knapsack problem

In the second part of this case study, we focus on a classical problem: the 0-1 knapsack problem. In this problem, we are given a knapsack with capacity c and n items. With every item, we associate a positive integer profit p_i and a positive integer weight w_i. The goal of the problem is to select a set of items to include in the knapsack such that the sum of the profits of these items is maximized and the sum of the weights of these items does not exceed the knapsack capacity. Hence, this problem can be written concisely as the following maximization problem:

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i \tag{7a}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq c, \tag{7b}$$

$$x_i \in \{0,1\}, \ \forall i \in \{1,2,\ldots,n\} \tag{7c}$$

Here, the problem variables are the capacity c, the number of items n, the profits of the items p_i and the weights of the items w_i. The decision variables are the n binary variables x_i. The meaning of these decision variables is as follows: x_i is equal to 1 if and only if the i-th item is included in the knapsack. We will further assume that $x_i \leq c$ ($\forall i \in \{1,2,\ldots,n\}$) and $\sum_{i=1}^{n} w_i x_i > c$ to avoid trivial solutions. Finally, we also assume that the items are ordered according to the ratio between the profit and the weight such that:

$$\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}}, \ \forall i \in \{1,2,\ldots,n-1\} \tag{8}$$

This last assumption is introduced for convenience and will make the notation in the rest of the paper easier. In the following paragraphs, we will discuss for the case of the 0-1 knapsack problem which particular choices we have made for the enhancements that were proposed for the Monte Carlo tree search algorithm in the context of combinatorial optimization.

6.1. Enhancements used in our algorithm

6.1.1. Domain reduction

Suppose that the first $k < n$ decision variables $x_1, x_2, \ldots, x_k$ have already been instantiated. If $w_{k+1} + \sum_{i=1}^{k} w_i x_i > c$, we can eliminate the value 1 from the domain of x_{k+1}. This is the case, because if we assign 1 to x_{k+1}, the knapsack capacity is exceeded and the sum of the weights cannot decrease by adding zero or more items in the knapsack (recall that the weights are positive integers).

6.1.2. Pruning subtrees by calculating bounds

The 0-1 knapsack problem is a maximization problem. Hence, the bound that will be calculated for every node v of the search space tree is an upper bound for the best objective function value amongst all leaf nodes in the subtree rooted at v. In case the current best objective function value is greater than or equal to this upper bound, the subtree rooted at v can be pruned, because no solution in this subtree can be better than the current best solution.

It is clear that if the first $k < n$ decision variables $x_1, x_2, \ldots, x_k$ have already been instantiated, the algorithm should try to assign values to the remaining $(n-k)$ decision variables such that the sum of their profits is maximized and the sum of their weights does not exceed $c - \sum_{i=1}^{k} w_i x_i$. Hence, assigning a value to the next decision variable x_{k+1} gives rise to a new, slightly different knapsack problem instance in which one less item than before is available and the knapsack capacity is changed to $c - \sum_{i=1}^{k+1} w_i x_i$. Because of this reason, the upper bound in every node of the search space tree is an upper bound for a slightly different problem instance. The upper bound that the algorithm uses is due to Dantzig [59] and is obtained from solving the linear programming relaxation of the 0-1 knapsack problem. If constraint (7c) is replaced by $0 \leq x_i \leq 1, \forall i \in \{1,2,\ldots,n\}$ (with $x_i \in \mathbb{R}$), the resulting optimization problem can be solved exactly in a greedy manner by consecutively adding the largest possible fraction of the next item to the knapsack. More specifically, let b be an index of an item such that $\sum_{i=1}^{b-1} w_i \leq c$ and $\sum_{i=1}^{b} w_i > c$. Item b is called the break item (or sometimes also split item), because it is the first item that does not fit entirely in the knapsack. The optimal solution to the linear programming relaxation of the 0-1 knapsack problem is obtained by setting:

$$\begin{aligned} x_i &= 1, \ \ \forall i \in \{1,2,\ldots,b-1\} \\ x_b &= \frac{c - \sum_{i=1}^{b-1} w_i}{w_b} \\ x_i &= 0, \ \ \forall i \in \{b+1, b+2, \ldots, n\} \end{aligned} \tag{9}$$

The corresponding objective function value is given by:

$$p_b \cdot \frac{c - \sum_{i=1}^{b-1} w_i}{w_b} + \sum_{i=1}^{b-1} p_i \tag{10}$$

As indicated earlier, this upper bound has to be calculated in all consecutive nodes of the search space tree that are on the path of a single iteration of the Monte Carlo tree search algorithm. Since the index of the break item b can only increase as more decision variables are instantiated, the algorithm can avoid doing redundant computations by remembering the index of the break item for the previous node of a path and increasing this index for the current node if this is necessary. Since the index of the break item b can only move at most n times to the right and there are at most n nodes on a path that is generated by the Monte Carlo tree search algorithm, all upper bounds for the path can be calculated efficiently with a total amortized time complexity of $O(n)$.

6.1.3. Using a heuristic simulation policy

When the algorithm arrives in a node that it has never visited before, the values for the remaining decision variables are chosen according to a heuristic. This heuristic considers the remaining items one by one and greedily adds an item to the knapsack if the item still fits in the knapsack. This heuristic is quite similar to the algorithm for the linear relaxation of the 0-1 knapsack problem which was discussed in the previous paragraph. However, the heuristic simulation policy will not add a fraction of the break item to the knapsack, but will instead leave the break item out of the knapsack and continue to add the remaining items one by one as long as there is still enough space. Because of the close resemblance between these two algorithms, the gap between the upper bound from the previous paragraph and the lower bound obtained by the heuristic simulation policy is usually very small. Later in Subsection 7.2, it will become clear that this heuristic works remarkably well for many problem instances.

7. Computational results

The proposed Monte Carlo tree search algorithm was extensively tested on two different case studies, requiring approximately 1,050 CPU-hours to run all the tests. In this section, we will consider five research questions: (1) What is the impact of varying the algorithm's parameters? (2) How does the performance of the algorithm compare

with state-of-the-art algorithms? (3) Is the proposed algorithm able to improve the heuristic simulation policy? (4) What is the impact of the different components of the algorithm? (5) How do different selection policies affect the performance of the algorithm? We will answer all five questions for the first case study, but for the second case study we limit ourselves to the first four questions to limit the length of the paper. The results for the two case studies will be discussed in two different subsections.

To keep the computational burden manageable, the experiments were conducted on the ThinKing cluster of the Flemish Supercomputer Center (VSC), using powerful CPUs with a clock rate of 2.5 GHz and 10 GB RAM memory. The C++ code for our algorithms and the datasets that we generated are available online at *https://github.com/JorikJooken/MCTSQuayCraneSchedulingNonCrossingConstraints* and *https://github.com/JorikJooken/MCTS01Knapsack*. In the tables of this section, the numbers between brackets indicate the deviations from the best known dual bound (i.e. the dual bound refers to a lower bound for the quay crane scheduling problem with non-crossing constraints and an upper bound for the 0-1 knapsack problem) and the best found primal bounds (objective function values) for every experiment will be marked in bold. In the main section of this paper, we provide summaries of the experiments, whereas the tables in the appendix (Tables A11-B24) contain the detailed results. For every table in the appendix, we also perform Wilcoxon signed-rank tests (non-parametric statistical hypothesis tests) in which we compare paired samples formed by the objective function value of the algorithm that we propose in the current paper (i.e. the reference algorithm, abbreviated as REF. ALG.) and the objective function value of another algorithm, which is different for every experiment. The null hypothesis of these tests states that the median of $X - Y$ is greater (less) than or equal to 0 for the first case study (second case study), where X and Y are random variables that represent the objective function value of the reference algorithm and the other algorithm, respectively. In other words, for each case study small p-values in the tables indicate that the reference algorithm tends to produce a better objective function value more often than the other algorithm.

7.1. Computational results: quay crane scheduling problem with non-crossing constraints

7.1.1. Data generation

The set of problem instances that we will use to answer these questions consists of two separate datasets (A and B). Dataset A was proposed by Lee and Chen [9] and later also used by Santini et al. [10]. Most of these problem instances represent a realistic situation with respect to the number of quay cranes and bays. To give the reader an idea of realistic numbers, we refer to Santini et al. [10] who report that there were 23 bays in one of the largest container vessels in 2,014 and Ng et al. [78] who report that two to seven quay cranes are deployed for a ship in a typical terminal. Apart from these realistic instances, this set also consists of a few problem instances for which the number of quay cranes and bays is unrealistically large. The number of bays in problem instances from dataset A is between 16 and 100, while the number of quay cranes is between 4 and 10. The processing times of the bays are uniformly distributed between 30 and 100, leading to sums of processing times between 2,893 and 16,519.

Dataset B was generated by us in the same way as described in the previous paragraph, but consists of much bigger problem instances that represent unrealistic situations. The purpose of these unrealistically large problem instances is to demonstrate that heuristic algorithms are still able to produce good results in a reasonable time. The number of bays in these instances is between 200 and 3,000, the number of quay cranes is between 4 and 10 and the sums of the processing times are between 33,532 and 495,726.

7.1.2. Experiments

In the first experiment, we have investigated the impact of varying the parameters of the Monte Carlo tree search algorithm (question (1)). This algorithm, which we will denote by $MCTS\_t\_w$, has two parameters that have to be chosen: the execution time t (in seconds) and the beam width w. All combinations of parameters were tested, where $t \in \{10; 100\}$ and $w \in \{1; 10; 100\}$, yielding 6 different possibilities. The possible values for the execution time t were chosen in such a way that they are comparable with the times that were reported in state-of-the-art algorithms (see the second experiment), while the values for the beam width w were chosen to be reasonably small integers. Because of

the stochasticity of the algorithm, every problem instance was solved 25 times for every parameter setting, each time using a different seed for the random number generator (a Mersenne Twister with a state size of 19,937 bits). The average objective function values over these 25 runs were recorded for every problem instance and these values were again averaged over the whole dataset. These values are summarized in Table 2 for both datasets. Since the quay crane scheduling problem with non-crossing constraints is a minimization problem, lower values are better.

Table 2: Average objective function values for varying parameters

Dataset average	$MCTS\_10\_1$	$MCTS\_10\_10$	$MCTS\_10\_100$	$MCTS\_100\_1$	$MCTS\_100\_10$	$MCTS\_100\_100$
Avg. A	847.9	847.4	846.9	845.7	845.5	**845.3**
Avg. B	38,195.2	38,192.8	38,193.0	38,180.0	38,180.5	**38,178.5**

As can be seen from this table, the different parameter settings obtain very comparable results. The biggest relative differences between the different parameter settings are 0.31% and 0.04% for dataset A and B, respectively, where $MCTS\_10\_1$ had the worst performance and $MCTS\_100\_100$ had the best performance. For a fixed execution time t, the effect of varying the beam width w is in general quite small and there is no single choice of beam width w that is always at least as good as another choice for every single problem instance. For a fixed beam width w, the effect of changing the execution time t is a little larger, although still quite small. On average, the relative improvement that could be gained by changing the execution time t from 10 seconds to 100 seconds was 0.26%, 0.22% and 0.19% (dataset A) and 0.04%, 0.03% and 0.04% (dataset B) for a beam width of 1, 10 and 100, respectively. As we will see later in the paper (in Table 3), the solutions produced by these six different algorithms are all nearly optimal (even using the worst parameter settings) and this explains why the observed differences are relatively small.

To answer the second question we have compared the performance of our algorithm (using the best obtained result from the previous experiment, where ties are broken in favour of less time in case of equal results) with the performance of the algorithms described by Lee and Chen [9], Santini et al. [10] and Zhang et al. [57]. Lee and Chen proposed a deterministic, heuristic algorithm, named *enhanced best partition* (EBP), that consists of two phases. In the first phase, a more constrained version of the problem is solved in polynomial time by dynamic programming. In the second phase, this solution is iteratively improved by reassigning quay cranes to different bays. Santini et al. proposed the mathematical model that was introduced earlier in Subsection 5.1 and used the mixed integer programming solver CPLEX to solve it. Zhang et al. proposed another deterministic, heuristic algorithm, named *selected partition-based algorithm* (SPA), in which the bays are first partitioned and next consecutive areas of bays are merged according to several rules. The resulting algorithm achieves an approximation ratio of $2 - \frac{2}{m+1}$, where m denotes the number of quay cranes. This is the best approximation ratio that is known in literature.

We have contacted the corresponding authors of these three papers to inquire about the availability of the source code of their algorithms and the problem instances that were used. We have received the source code of Santini et al. [10] and the problem instances of Lee and Chen [9] and Santini et al. [10] (dataset A), but we were unable to obtain the source code of Lee and Chen [9] and Zhang et al. [57]. Fortunately, these two missing algorithms were described in great detail without any unclear aspects in both papers and we have taken the effort to reimplement these algorithms. For the algorithm by Lee and Chen, we were able to verify that our reimplementation obtained exactly the same objective function values on the problem instances from dataset A as in the original paper [9]. The execution times are faster in our reimplementation, but this is very likely to be due to more modern hardware and a potentially faster programming language (MATLAB in the original paper versus C++ in our reimplementation). For the algorithm by Zhang et al. [57], there were unfortunately no common problem instances available for which we could test the correctness of our reimplementation. However, the optimality gaps that we obtained on our problem instances are comparable to the optimality gaps that were reported in [57], which gives us confidence that there were no significant errors in our reimplementation of this algorithm.

The detailed results of these algorithms can be found in appendix A (Table A11 for dataset A and Table A12

for dataset B) and are summarized in Table 3. In these tables, the first column contains the name of the problem instance (or the dataset in case of Table 3). In the remainder of this article, we will denote the problem instances by $n-m-s$, where n represents the number of bays, m represents the number of quay cranes and s represents the sum of the processing times of all bays. These numbers are indicative of the size of the problem instance according to the mathematical model from Subsection 5.1 (the higher these numbers, the larger the size). The second and the third column contain the lower bounds on the objective function values that were computed using expressions (5)+(6) and CPLEX, respectively. Hence, the solutions that were computed using the different algorithms cannot have an objective function value that is below these lower bounds and equality would imply that the solution is optimal. Finally, the eight remaining columns contain the objective function value of the best found solution (with relative deviations from the best known lower bound indicated between brackets) and the execution time (with a maximum allotted time of 3,600 seconds) for all four algorithms. Table 3 contains the averages over the two datasets. In case not all algorithms were able to compute a feasible solution for every problem instance, we have made a distinction between the average over all problem instances (Avg.) and the average over all problem instances for which all algorithms produced a feasible solution (Avg. feasible). For dataset A, all algorithms were able to produce a feasible solution for every problem instance, but this was not the case for dataset B where CPLEX was not always able to find a feasible solution within 3,600 seconds.

Table 3: Summary of the comparison between the performances of different algorithms for the quay crane scheduling problem with non-crossing constraints

		CPLEX (Santini et al. [10])			MCTS (this paper)		EBP (Lee and Chen [9])		SPA (Zhang et al. [57])	
Dataset average	Lower bound (5)+(6)	Lower bound	Value	Time	Value	Time	Value	Time	Value	Time
Avg. A	840.4	842.9	846.1 (0.4%)	1,937.7 s	**843.9 (0.1%)**	32.5 s	883.1 (4.8%)	<0.1 s	1,327.6 (57.5%)	<0.1 s
Avg. B	38,166.7	-	-	-	**38,169.9 (0.0 %)**	51.3 s	38,220.3 (0.1 %)	2.1 s	64,239.2 (68.3 %)	<0.1 s
Avg. B feasible	7,137.0	7,137.0	33,487.8 (369.2 %)	3,600.0 s	**7,138.7 (0.0 %)**	47.5 s	7,191.3 (0.8 %)	<0.1 s	11,941.3 (67.3 %)	<0.1 s

By inspecting Table 3 and Table A11, it becomes clear that for dataset A both EBP and SPA produce their answer very fast, while both MCTS and CPLEX take a longer time to do so, but also obtain results with a higher quality. The results of SPA are quite different from the other three algorithms. The relative deviations from the lower bound range between 22.5% and 79.9% for SPA, while they are always below 15% for the other algorithms. This shows that although SPA obtains the best known approximation ratio ($2-\frac{2}{m+1}$), its practical performance is not so good. For EBP the relative deviations range from 1.2% to 13.5%, while for both MCTS and CPLEX they are always below 1.5%. Recall that both SPA and EBP are deterministic, heuristic algorithms such that executing them multiple times or increasing their computational budget will not affect the results that they produce. Hence, the best results for dataset A are produced by CPLEX (average relative deviation of 0.4%) and MCTS (average relative deviation of 0.1%). There were 9 problem instances for which the result that was produced by MCTS was strictly better than the result of CPLEX and 1 problem instance for which CPLEX was better than MCTS. For 8 out of these 9 cases, MCTS improved the previously best known solution in literature (for problem instance $23-4-3{,}544$, MCTS matches the previously best known solution that was also obtained in [10]).

The problem instances of dataset B (Table 3 and Table A12) are much bigger in comparison with dataset A and in this case the results of CPLEX are no longer satisfactory, with relative deviations ranging between 145.6% and 807.7%.

For the 12 biggest problem instances, CPLEX was no longer able to even produce any solution or lower bound at all. For these bigger instances, the results obtained by MCTS were always strictly better than the results obtained by the other three algorithms. It should also be noted that the lower bounds that are used by MCTS (expressions (5)+(6)) seem to be tighter for dataset B than for dataset A. For dataset A, MCTS produced a solution with an objective function value equal to the lower bound (i.e. an optimal solution) in 5 cases, while there were 13 such cases for dataset B. When there are more bays it is easier to find a partition of the bays into contiguous areas such that the bay areas all roughly have the same processing time. Such a partition into contiguous areas gives rise to a feasible solution and since there are more partitions in case the number of bays rises, the quality of the best solution with this property also increases. This is illustrated by the fact that the results of EBP, which are based on finding the best partition into contiguous areas, also get better (average relative deviation of 4.8 % for dataset A versus average relative deviation of 0.1 % for dataset B). Because the lower bounds used by MCTS become tighter, its results also get slightly better in comparison with dataset A (average relative deviation of 0.1 % for dataset A versus average relative deviation of 0.0 % for dataset B). The results of SPA, however, get slightly worse (average relative deviation of 57.5 % for dataset A versus average relative deviation of 68.3 % for dataset B). Hence, it can be concluded that in general the best results are obtained by MCTS and it surpasses the state-of-the-art results for this case study. The statistical tests that we performed also support this. At a significance level of $\alpha = 5\%$, we reject the null hypothesis of the Wilcoxon signed-rank test in favour of the alternative hypothesis (the p-values can be found in Tables A11 and A12). Note that all p-values are very small.

We answer the third question by providing evidence for the claim that MCTS is able to improve the heuristic that is used as its solution completion policy. Using the heuristic solution completion policy of MCTS as a standalone heuristic corresponds to stopping MCTS after its first iteration. This is the case because in the first iteration of the algorithm, the root node has never been visited before and hence the algorithm switches from using its selection policy to using its simulation policy. For this reason, the performance of MCTS is always at least as good as the performance of the heuristic solution completion policy. The averaged performance of the heuristic simulation policy for both datasets can be found in Table 4. We have also again added the best known lower bound and the results of $MCTS\_100\_100$ from the first experiment (because this parameter setting yielded the best average performance) to this table to make it easier for the reader to follow the comparison. The relative deviations from the lower bound are indicated between brackets. The problem instances for which the heuristic solution completion policy found an infeasible solution have been left out for computing the averages. The best found value for every row is marked in bold.

Table 4: Comparison between MCTS and its heuristic solution completion policy used as a standalone heuristic

Dataset average	Lower bound	$MCTS\_100\_100$	Time	Heuristic solution completion policy	Time
Avg. A feasible	828.7	**830.8 (0.3 %)**	100.0 s	922.9 (11.4 %)	<0.1 s
Avg. B feasible	42,805.3	**42,811.8 (0.0 %)**	100.0 s	42,898.4 (0.2 %)	<0.1 s

From this table, one can observe that the heuristic simulation policy can be executed very fast and the quality of the produced solution is moderate to good. In some cases, the solution produced by the heuristic simulation policy is infeasible (there were 4 out of 24 infeasible solutions for dataset A and 7 out of 24 for dataset B). Recall that the followed strategy of MCTS is to attempt to solve the original problem by solving a relaxation and discarding infeasible solutions. MCTS is able to learn to correct all of these solutions and is able to improve the heuristic on average by 10.0% for dataset A and by 0.2% for dataset B. Note that for dataset B, this improvement is quite small, because the performance of the heuristic solution completion policy is already quite good and there is less room for improvement. MCTS did not produce any infeasible solutions for both datasets. Hence, one can indeed conclude that MCTS can also be seen as a powerful method to improve the performance of a constructive heuristic algorithm.

To answer the fourth question we implemented 16 different versions of MCTS, where every subset of the modifications that we proposed in Section 4 is omitted once (see Table 5). The first version contains all of the modifications, whereas for the other versions at least one modification is omitted (so the first version corresponds to the version that was discussed in the previous paragraphs). The used parameters for every version are the same ($t = 100$ and $w = 100$ as in the previous experiment). The versions where a certain modification is not present work as follows. If a version does not follow the heuristic simulation policy, the simulation policy assigns domain values uniformly at random to the decision variables in the simulation phase of the algorithm. Not using the idea of directing the search by using a beam width corresponds to using an infinitely large beam width. Finally, not using dominance rules or pruning rules will result in no domain values or subtrees being pruned from the search space. The detailed results of this experiment can be found in appendix A (Tables A13-A15 for dataset A and Tables A16-A18 for dataset B) and are summarized in Table 5. We can see that version 16 (where none of the modifications that we propose are present) performs consistently worse than the other versions. For dataset B, there were even 8 problem instances for which version 16 was not able to find any feasible solutions. We can also see that usually the versions which contain more modifications perform better, but this is not always the case. For example, for problem instance $16-5-2,893$ from dataset A we can see that version 10 has a worse objective function value than version 12. The versions which include the heuristic simulation policy also perform much better than the versions that exclude it. Finally, also note that the first version, which contains all of the modifications that we propose, always performs at least as well as the other versions for both datasets for every problem instance, except for two problem instances ($21-5-3,033$ and $25-5-4,334$). Hence, the first version is clearly the best version for this case study. This is also supported by the statistical tests that we performed. For dataset A and B, there are respectively 13 and 15 (out of 15) MCTS variants for which we reject the null hypothesis of the Wilcoxon signed-rank test in favour of the alternative hypothesis (the p-values can be found in Tables A13-A18; they are often even below 0.01). The null hypothesis cannot be rejected at a significance level of 5% for version 5 (p-value of 0.063) and version 7 (p-value of 0.063) based on dataset A, although version 1 performs better on average and produces a provably optimal solution for all but two problem instances.

Table 5: Average objective function values for different versions of MCTS

MCTS Version	Heuristic simulation policy	Beam	Domain reduction	Pruning subtrees by calculating bounds	Avg. A	Avg. B	Avg. B feasible
1	✓	✓	✓	✓	**843.9**	**38,169.9**	**13,385.5**
2	✓	✓	✓	✗	849.5	38,176.7	13,386.4
3	✓	✓	✗	✓	847.0	38,185.3	13,386.0
4	✓	✓	✗	✗	852.5	38,189.3	13,387.3
5	✓	✗	✓	✓	844.8	38,174.3	13,385.7
6	✓	✗	✓	✗	849.6	38,176.2	13,386.3
7	✓	✗	✗	✓	845.5	38,185.5	13,386.2
8	✓	✗	✗	✗	851.5	38,196.5	13,390.0
9	✗	✓	✓	✓	874.2	-	13,669.3
10	✗	✓	✓	✗	902.4	-	13,669.5
11	✗	✓	✗	✓	927.7	-	14,097.5
12	✗	✓	✗	✗	949.5	-	13,990.1
13	✗	✗	✓	✓	865.8	-	13,696.0
14	✗	✗	✓	✗	902.2	-	13,698.0
15	✗	✗	✗	✓	912.9	-	13,954.8
16	✗	✗	✗	✗	937.0	-	14,105.3

Finally to answer the fifth research question, we compared the selection policy that we propose in the current paper with two alternative ones. For the first alternative, we replaced expression (3) used to select a child in the selection phase with the following expression (which is more close to expression (1) from Monte Carlo tree search in the context

of game playing):

$$averageObjectiveFunctionValue(parent, child) + \sqrt{\frac{2 \cdot \ln(numberVisits(parent))}{numberVisits(child)}} \tag{11}$$

For the second alternative, we implemented[2] the OCBA-MCTS selection policy proposed by Li et al. [27]. This is based on the Optimal Computing Budget Allocation (OCBA) framework [79] and has nice theoretical properties. More specifically, Li et al. [27] prove that the algorithm optimally allocates a limited computing budget such that a lower bound on the probability of correctly selecting the best action at each node is maximized. Note that this goal is different from the MCTS variant used in the current paper (which attempts to maximize the cumulative rewards).

The results of the three algorithms considered in this experiment are summarized in Table 6 (see Tables A19 and A20 for the detailed results). For each algorithm, we again use the best found parameter setting from the first experiment ($w = 100$ and $t = 100$). As we can see, the three algorithms have a relatively comparable performance in the sense that all algorithms tend to produce nearly optimal solutions (the average deviation from the lower bound is at most 0.3% for all algorithms). The best average objective function values for both data sets are given by the algorithm which uses expression (3) from the current paper, the algorithm which uses expression (11) and OCBA-MCTS (in that order). The algorithm from the current paper produces a solution that is always at least as good as the other two algorithms on all problem instances, except for problem instance $25-5-4,334$. The Wilcoxon signed-rank test also confirms for both datasets that at a significance level of $\alpha = 5\%$ the null hypothesis should be rejected in favour of the alternative hypothesis (the p-values can be found in Tables A19 and A20).

Table 6: Comparsion between different selection policies of MCTS.

Dataset average	Lower bound	MCTS (selection policy uses (3))	MCTS (selection policy uses (11))	OCBA-MCTS (Li et al. [27])
Avg. A	842.9	**843.9 (0.1%)**	844.7 (0.2%)	845.1 (0.3%)
Avg. B	38,166.7	**38,169.9 (0.0%)**	38,178.3 (0.0%)	38,178.5 (0.0%)

7.2. Computational results: 0-1 knapsack problem

7.2.1. Data generation

For the 0-1 knapsack problem, we have also generated two datasets of problem instances (dataset A and dataset B). It is well known that the 0-1 knapsack problem can be exactly solved with a worst-case time complexity of $O(n \cdot c)$ (see e.g. [66]), where n denotes the number of items in the knapsack and c denotes the knapsack capacity. Therefore, most of the problem instances were generated such that $n \cdot c$ is large.

The problem instances from dataset A are strongly correlated spanner instances, which were introduced by Pisinger in [72]. This article has introduced several classes of problem instances that were empirically shown to be hard to solve. Amongst all these classes, the strongly correlated spanner instances were the hardest class of problem instances. The weights and the profits of the n items in a strongly correlated spanner instance are all multiples of the weights and profits of a small set of items (the spanner set). Pisinger has described in [72] that the problem instances become harder for smaller spanner sets, leading us to use 2 items for the spanner set. The weights w_i of the items in the spanner set are generated independently from each other, uniformly at random between 1 and 10^8. The corresponding profits p_i of the items in the spanner set are strongly correlated with the weights and are generated by setting $p_i = w_i + 10^7$. The remaining $(n-2)$ items are generated by selecting a random item j from the spanner set and generating an integer multiplier k uniformly at random between 1 and 10 for every item. The profit and weight of the i-th item

[2] We used the same parameters as those used for the inventory control problem in [27], namely $n_0 = 2$, $\sigma_0^2 = 100$ and $\alpha_{N(\mathbf{x})} = 1 - \frac{1}{5N(\mathbf{x})}$.

are then obtained by setting $p_i = k \cdot p_j$ and $w_i = k \cdot w_j$, respectively. Finally, the knapsack capacity is chosen to be a certain fraction f of the sum of the weights of all the items. In the instances from dataset A, we have chosen $f \in \{0.25; 0.50; 0.75\}$ and $n \in \{50; 200; 500; 2,000; 5,000; 20,000; 50,000\}$.

The state-of-the-art Combo algorithm for the 0-1 knapsack problem [67] is still able to exactly solve the largest problem instances from dataset A in a reasonable time, despite the fact that these instances are the hardest problem instances described in [72]. For dataset B on the contrary, we have been able to create problem instances which consist of relatively few items in comparison with dataset A, but the time needed by the Combo algorithm to solve these problem instances is considerably higher. These problem instances are so-called noisy multi-group exponential problem instances and these instances have been shown to be difficult to solve for exact algorithms in [73]. In the current paper we show that our heuristic algorithm is able to quickly produce nearly optimal solutions. An instance is generated as follows: the knapsack capacity c is equal to 10^{10}. There are n items, which can be divided into 10 different groups. The first 9 groups (groups $1, 2, \ldots, 9$) consist of items with exponentially decreasing weights and profits. These 9 groups consist of around $\frac{2}{3} \cdot n$ items, where every group is equally large. All items in a certain group i ($i = 1, \ldots, 9$) are generated independently from each other, by setting the profit p_j of item j as $p_j = (\frac{1}{2^i} + 10^{-4}) \cdot c + r_1$ and setting the weight as $w_j = (\frac{1}{2^i} + 10^{-4}) \cdot c + r_2$ (note that p_j and w_j are integers because of the choice of c). Here, r_1 and r_2 are small integers which are chosen uniformly at random between 1 and 300 for each item. Note that by defining the items precisely like this, the profit-weight ratios of the items in the first 9 groups are all very close to each other. Another consequence of the definition is that the optimal solution usually has to combine items from several different groups, because including 2^i items from group i would slightly exceed the knapsack capacity, whereas including $2^i - 1$ items from group i would leave a large part of the knapsack unfilled. Finally the tenth group consists of the remaining $\frac{1}{3} \cdot n$ items and these items have very small weights and profits with a very diverse range of profit-weight ratios. These items are generated independently from each other by setting the profit p_j of item j as $p_j = r_1$ and setting the weight as $w_j = r_2$, where r_1 and r_2 are again small integers which are chosen uniformly at random between 1 and 300 for each item. Hence, the items in the tenth group introduce more variability in the profit-weight ratios and they are useful to fill a small part of the knapsack. The problem instances from dataset B were generated by choosing $n \in \{100; 125; 150; \ldots; 700\}$.

7.2.2. Experiments

To answer the first research question from Section 7, we investigated to which extent the results are affected by different parameter settings of the Monte Carlo tree search algorithm. As in the previous case study, we tested 6 different parameter settings that were obtained by choosing the beam width $w \in \{1; 10; 100\}$ and the execution time $t \in \{10; 100\}$. For every parameter setting, we solved every instance 25 times independently from each other, using a different random seed in every run. For both datasets, we computed the average objective function values over these 25 runs for every problem instance, and next these values were averaged over the whole dataset. The results of this experiment can be found in Table 7. Recall that the 0-1 knapsack problem is a maximization problem, such that higher objective function values are better. The best found average for every dataset is marked in bold.

Table 7: Objective function values for varying parameters

Dataset average	*MCTS*_10_1	*MCTS*_10_10	*MCTS*_10_100	*MCTS*_100_1	*MCTS*_100_10	*MCTS*_100_100
Avg. A	**334,940,368,920.9**	**334,940,368,920.9**	**334,940,368,920.9**	**334,940,368,920.9**	**334,940,368,920.9**	**334,940,368,920.9**
Avg. B	9,999,791,698.2	9,999,846,037.4	9,999,721,277.9	9,999,875,981.1	**9,999,892,314.5**	9,999,829,188.7

Interestingly, we see that changing the parameter settings has no effect at all for dataset A. The obtained objective function values are all exactly the same for the 6 different parameter settings and the 25 different runs, given a fixed problem instance of dataset A. For dataset B, different parameter settings yield different results, but only very slightly. For a fixed execution time t, the results are ordered in increasing order of quality by choosing the beam width w as

100, 1 and 10 (in this order). Changing the execution time t from 10 seconds to 100 seconds for a fixed beam width w improves the results, albeit only slightly (the biggest relative difference with respect to the objective function value is equal to 1.1×10^{-3} %). These results reveal that the best parameter settings for both datasets are obtained by choosing $t = 100$ and $w = 10$, although the differences are very small.

We answer the second question by comparing the performance of our algorithm (using the best obtained result from the previous experiment, where ties are broken in favour of less time in case of equal results) with the performance of the famous Combo algorithm [67] (source code[3] available online at: http://hjemmesider.diku.dk/~pisinger/codes.html). Although this algorithm was invented more than twenty years ago, it still represents the current state-of-the-art (see e.g. [68], [69], [70] and [71] for relatively recent articles that support this claim). This is not surprising, given that the authors of Combo have been conducting research about the 0-1 knapsack problem for at least two decades prior to the publication of the article in which Combo was described [67]. The stellar performance of this algorithm is reconfirmed in our experiments. From these experiments, it becomes clear that many very large knapsack problem instances can be solved in only a few seconds, despite Combo being an exact algorithm. Because of the very limited room for improvement, there do not exist heuristic algorithms which are able to outperform Combo in this case. However, as will be indicated later, even the simple heuristic solution completion policy that we used in our algorithm produces objective function values that are extremely close to the optimal values. With this knowledge, the second research question is less interesting for the 0-1 knapsack problem (Combo has been the undisputed best algorithm for more than two decades), but we want to answer it anyways because we also addressed this question for the first case study. Nevertheless, we are also able to show that certain problem instances with specific characteristics exist (dataset B) for which the large execution times might make it impractical to use Combo.

The detailed results of Combo and MCTS can be found in appendix B (Table B21 for dataset A and Table B22 for dataset B) and are summarized in Table 8. The first column contains the name of the problem instance (or name of the dataset in case of Table 8). The instances will be denoted as strCorrSpan-*n*-*c* and exp-*n*-*c* for dataset A and B, respectively, where n denotes the number of items in the knapsack and c denotes the knapsack capacity. The four other columns contain the found objective function value and the time needed to obtain this result by both algorithms. Table 8 contains the averages over both datasets. The Combo algorithm computes the exact optimum while MCTS is a heuristic, and thus the obtained result by MCTS is a lower bound for this optimum. The relative deviation from the optimum is indicated between brackets.

Table 8: Summary of the comparison between the performances of Combo and MCTS for the 0-1 knapsack problem

	Combo		MCTS	
Dataset	Value	Time	Value	Time
Avg. A	**334,942,167,924.2**	2.8 s	334,940,368,920.9 (5.3×10^{-4} %)	10.0 s
Avg. B	**9,999,968,161.2**	866.9 s	9,999,920,718.0 (4.7×10^{-4} %)	67.6 s

If we take a closer look at Table B21, we see that Combo is able to compute the optimal answer in less than 10 seconds for almost all problem instances of dataset A, except for the three largest problem instances where there are 50,000 items in the knapsack. For two of these three instances, it obtains a better objective function value than MCTS and requires more or less the same time to do so. However, it should also be noted that the objective function value produced by MCTS is always very close to the optimal answer: amongst all problem instances of dataset A the largest relative deviation from the optimum is equal to 0.029 % and the average relative deviation is equal to 5.3×10^{-4}

[3] We changed the value of the variable *MAXSTATES* from 1.5×10^6 to 4.5×10^8 to avoid the program crashing when the number of states in the dynamic programming algorithm exceeds this variable.

%. For 14 problem instances, MCTS was able to produce an objective function value that was equal to the optimal one. Thus we can conclude that for dataset A both algorithms produce very similar solutions, but Combo has a slight advantage.

However, the conclusion for dataset B is somewhat different. These results can be found in Table B22. Despite the fact that there are only several hundreds of items, the execution time needed by Combo to produce the optimal answer increases rapidly. For example, it took Combo more than 2,500 seconds to solve exp-700-10,000,000,000 from dataset B, while it took less than 0.1 second to solve strCorrSpan-2,000-17,052,969,836 from dataset A (note that the number of items n and the knapsack capacity c of both problem instances have a comparable order of magnitude). For the problem instances of dataset B, MCTS is still able to produce objective function values that are very close to the optimal ones, while only using at most 100 seconds to do so. Hence, for these instances it might be advisable to use a heuristic like MCTS that sacrifices the optimality guarantee for a decrease in execution time. On average, MCTS used 67.6 seconds of execution time, while Combo used 866.9 seconds on average. The cost of this decrease in execution time is not so big: the average optimality gap attained by MCTS is equal to 4.7×10^{-4} %.

Table 9: Comparison between MCTS and its heuristic solution completion policy used as a standalone heuristic

Dataset average	Optimum	*MCTS*_100_10	Time	Heuristic solution completion policy	Time
Avg. A	334,942,167,924.2	**334,940,368,920.9 (5.3×10^{-4} %)**	100.0 s	334,940,285,925.1 (5.6×10^{-4} %)	<0.1 s
Avg. B	9,999,968,161.2	**9,999,920,718.0 (4.7×10^{-4} %)**	100.0 s	9,992,194,843.4 (7.7×10^{-2} %)	<0.1 s

To answer the third question, we compare *MCTS*_100_10 (the best parameter setting in the first experiment) with the heuristic that is used as its solution completion policy. Recall that the result of MCTS will always be at least as good as the result of its solution completion policy. The average performance of both algorithms can be found in Table 9. We also repeated the average optimal objective function values and the relative deviations between brackets for the reader's convenience. The heuristic solution completion policy can be executed very fast (in less than 0.1 second), because it corresponds to the first iteration of MCTS. From this table we can also see that the objective function values obtained by the heuristic solution completion policy are very close to the optimal ones. For both datasets, MCTS is able to improve these values. For dataset A, this improvement is not so big (from 5.6×10^{-4} % to 5.3×10^{-4} %), because the initial optimality gap is already very small. For dataset B, the relative improvement is bigger. The optimality gap is reduced from 7.7×10^{-2} % to 4.7×10^{-4} %, which corresponds to a decrease with a factor of approximately 163.8. We conclude that MCTS is indeed able to learn to correct the solutions produced by its heuristic simulation policy, despite the fact that this policy already achieves small optimality gaps.

Finally, to answer the fourth question we implemented five different versions of MCTS. The modifications that we proposed in Section 4 are introduced one by one in these five versions. The used parameters for every version are the same ($t = 100$ and $w = 10$ as in the previous experiment). The detailed results of this experiment can be found in appendix B (Table B23 for dataset A and Table B24 for dataset B) and are summarized in Table 10. From these tables, it becomes clear that the fifth version of MCTS (where none of the modifications that we propose are present) is not suitable at all for this problem and it performs much worse than the four other versions. It can often not improve the initial solution in which none of the items are selected. This was the case for 7 problem instances of dataset A and all problem instances of dataset B. The four other versions were always able to produce nearly optimal solutions. For dataset A, there was only one problem instance ($strCorrSpan - 2,000 - 28,036,577,373$) for which the solution produced by the four other versions was different. Hence, for dataset A these four other versions were equally good (apart from this one problem instance where version 1 and version 4 produced an equally good solution that was better than version 2 and version 3). For dataset B, however, the solutions produced by the four other versions were often slightly different. For this dataset, version 1 produced the best solutions on average and there were 14 problem instances for which it was strictly better than all other versions. However, this dataset also shows that version 1 was

not always at least as good as the other versions (as was previously the case). The first version tends to produce a better solution most of the time. This was also confirmed by the Wilcoxon signed-rank test: at a significance level of $\alpha = 5\%$, the null hypothesis should be rejected in favour of the alternative hypothesis (the p-values can be found in Table B24).

Table 10: Average objective function values for different versions of MCTS

Dataset average	MCTS Version 1	MCTS Version 2	MCTS Version 3	MCTS Version 4	MCTS Version 5
Heuristic simulation policy	✓	✓	✓	✓	✗
Beam	✓	✓	✓	✗	✗
Domain reduction	✓	✓	✗	✗	✗
Pruning subtrees by calculating bounds	✓	✗	✗	✗	✗
Avg. A	**334,940,368,920.9**	334,940,326,098.6	334,940,326,098.6	**334,940,368,920.9**	187,232,935,624.5
Avg. B	**9,999,920,718.1**	9,999,914,190.1	9,999,914,267.6	9,999,697,742.2	0.0

8. Conclusions and further work

In this article, we have proposed a heuristic algorithm to explore search space trees that is based on Monte Carlo tree search. By leveraging the combinatorial structure of the problem, the algorithm was enhanced in several ways. These enhancements were demonstrated on two case studies: the quay crane scheduling problem with non-crossing constraints and the 0-1 knapsack problem. The computational results for these problems have shown that the proposed algorithm is able to compete with state-of-the-art algorithms for both problems and eight new best solutions were found for the set of problem instances proposed by Lee and Chen [9]. The computational results also provided further insight into the sensitivity of the algorithm's parameters, the ability to learn to correct the choices made by the heuristic simulation policy and the added value of the proposed modifications to Monte Carlo tree search.

An interesting avenue for further work is to research the possibility of learning across different problem instances. In the proposed algorithm, the selection policy is learned in an online fashion, making it unable to benefit from information that was learned from previous problem instances. However, by using information that was learned offline (e.g. features that capture all relevant information of a node in the search space tree), the time necessary to learn a well-performing selection policy for a given problem instance could potentially be significantly reduced. It is very likely that integrating such information into the proposed algorithm would require significant changes and this problem deserves further attention.

Another interesting idea that was not further explored in this article is the ability to integrate exact solvers into the algorithm and, related to this, compare it with other search paradigms (e.g. heuristic tree search algorithms or matheuristics). In the deeper levels of the search space tree, the number of decision variables whose values are not yet known is low, making exact approaches feasible. However, the time for a single iteration of the algorithm could potentially be significantly increased by using exact solvers. In this case, a good balance between the accuracy and time increase must be found. The question of how to do this in an optimal way is a challenging one and requires further attention. It is likely that this will also depend on the problem that one is solving and another interesting opportunity for further work is to implement the proposed algorithm for different problems than the ones presented in the current article.

Finally, another topic that was not further explored in this article is the suitability of the proposed algorithm to be parallelized. There are several possibilities to do this. The most simple option is to let the algorithm have multiple independent runs in parallel. Another option would be to run multiple simulations in parallel in the simulation phase of the algorithm. The most sophisticated option would be to combine the learned selection policies of multiple runs into a single selection policy. It is not immediately clear how to combine these policies in an optimal way and this question requires further research.

Acknowledgments

The computational resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government - department EWI. We gratefully acknowledge the support provided by the ORDinL project (FWO-SBO S007318N, Data Driven Logistics, 1/1/2018 - 31/12/2021). This research received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme. Pieter Leyman is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO) with contract number 12P9419N. We are also grateful to Stefan Røpke (Technical University of Denmark) for providing dataset A for the quay crane scheduling problem with non-crossing constraints. Editorial consultation has been provided by Luke Connolly (KU Leuven).

References

[1] O. Vinyals, M. Fortunato, N. Jaitly, Pointer networks, in: Advances in Neural Information Processing Systems, Vol. 28, 2015, pp. 2692–2700.
URL https://proceedings.neurips.cc/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf

[2] I. Bello, H. Pham, Q. V. Le, M. Norouzi, S. Bengio, Neural combinatorial optimization with reinforcement learning, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings, 2017.
URL https://openreview.net/forum?id=Bk9mxlSFx

[3] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, L. Song, Learning combinatorial optimization algorithms over graphs, in: Advances in Neural Information Processing Systems, Vol. 30, 2017.
URL https://proceedings.neurips.cc/paper/2017/file/d9896106ca98d3d05b8cbdf4fd8b13a1-Paper.pdf

[4] A. Hottung, K. Tierney, A biased random-key genetic algorithm for the container pre-marshalling problem, Computers & Operations Research 75 (2016) 83–102.

[5] D. Karapetyan, A. P. Punnen, A. J. Parkes, Markov chain methods for the bipartite boolean quadratic programming problem, European Journal of Operational Research 260 (2) (2017) 494–506.

[6] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: H. J. van den Herik, P. Ciancarini, H. H. L. M. Donkers (Eds.), Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers, Vol. 4630 of Lecture Notes in Computer Science, Springer, 2006, pp. 72–83. doi:10.1007/978-3-540-75538-8_7.

[7] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[8] H. Baier, M. H. Winands, Beam Monte-Carlo tree search, in: 2012 IEEE Conference on Computational Intelligence and Games (CIG), IEEE, 2012, pp. 227–233.

[9] D.-H. Lee, J. H. Chen, An improved approach for quay crane scheduling with non-crossing constraints, Engineering Optimization 42 (1) (2010) 1–15. doi:10.1080/03052150902943020.

[10] A. Santini, S. Røpke, H. Friberg, A note on a model for quay crane scheduling with non-crossing constraints, Engineering Optimization 47. doi:10.1080/0305215X.2014.958731.

[11] N. Boysen, D. Briskorn, F. Meisel, A generalized classification scheme for crane scheduling with interference, European Journal of Operational Research 258 (1) (2017) 343–357. doi:10.1016/j.ejor.2016.08.041.

[12] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack problems, Berlin, DE: Springer 57.

[13] J. Duguépéroux, A. Mazyad, F. Teytaud, J. Dehos, Pruning playouts in Monte-Carlo tree search for the game of Havannah, in: International Conference on Computers and Games, Springer, 2016, pp. 47–57.

[14] R. J. Lorentz, Amazons discover Monte-Carlo, in: Proceedings of the 6th International Conference on Computers and Games, CG 08, Springer-Verlag, Berlin, Heidelberg, 2008, p. 1324. doi:10.1007/978-3-540-87608-3_2.

[15] M. H. Winands, Y. Björnsson, J.-T. Saito, Monte Carlo tree search in lines of action, IEEE Transactions on Computational Intelligence and AI in Games 2 (4) (2010) 239–250.

[16] B. Arneson, R. B. Hayward, P. Henderson, Monte Carlo tree search in Hex, IEEE Transactions on Computational Intelligence and AI in Games 2 (4) (2010) 251–258.

[17] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Mastering the game of Go with deep neural networks and tree search, Nature 529 (7587) (2016) 484.

[18] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al., Mastering chess and shogi by self-play with a general reinforcement learning algorithm, arXiv preprint arXiv:1712.01815.

[19] T. L. Lai, H. Robbins, Asymptotically efficient adaptive allocation rules, Advances in Applied Mathematics 6 (1) (1985) 4–22.

[20] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: European conference on machine learning, Springer, 2006, pp. 282–293.

[21] P. Auer, N. Cesa-Bianchi, P. Fischer, Finite-time analysis of the multiarmed bandit problem, Machine Learning 47 (2-3) (2002) 235–256.

[22] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Transactions on Computational Intelligence and AI in games 4 (1) (2012) 1–43.

[23] C. Bierwirth, F. Meisel, A follow-up survey of berth allocation and quay crane scheduling problems in container terminals, European Journal of Operational Research 244 (3) (2015) 675–689.

[24] G. M. J. Chaslot, M. H. Winands, H. J. v. d. Herik, J. W. Uiterwijk, B. Bouzy, Progressive strategies for Monte-Carlo tree search, New Mathematics and Natural Computation 4 (03) (2008) 343–357.

[25] M. H. Winands, Y. Björnsson, Evaluation function based Monte-Carlo LOA, in: Advances in Computer Games, Springer, 2009, pp. 33–44.

[26] S. Gelly, D. Silver, Combining online and offline knowledge in UCT, in: Proceedings of the 24th international conference on Machine learning, 2007, pp. 273–280.

[27] Y. Li, M. C. Fu, J. Xu, An optimal computing budget allocation tree policy for Monte Carlo tree search, IEEE Transactions on Automatic Control 67 (6) (2021) 2685–2699.

[28] G. Zhang, Y. Peng, Y. Xu, An efficient dynamic sampling policy for monte carlo tree search, arXiv preprint arXiv:2204.12043.

[29] M. H. Winands, Y. Björnsson, J.-T. Saito, Monte-Carlo tree search solver, in: International Conference on Computers and Games, Springer, 2008, pp. 25–36.

[30] P. Drake, S. Uurtamo, Move ordering vs heavy playouts: Where should heuristics be applied in Monte Carlo Go, in: Proceedings of the 3rd North American Game-On Conference, Citeseer, 2007, pp. 171–175.

[31] S. Gelly, D. Silver, Monte-Carlo tree search and rapid action value estimation in computer Go, Artificial Intelligence 175 (11) (2011) 1856–1875.

[32] F. Xie, Z. Liu, Backpropagation modification in Monte-Carlo game tree search, in: 2009 Third International Symposium on Intelligent Information Technology Application, Vol. 2, IEEE, 2009, pp. 125–128.

[33] A. Previti, R. Ramanujan, M. Schaerf, B. Selman, Monte-Carlo style uct search for boolean satisfiability, in: Congress of the Italian Association for Artificial Intelligence, Springer, 2011, pp. 177–188.

[34] B. Satomi, Y. Joe, A. Iwasaki, M. Yokoo, Real-time solving of quantified CSPs based on Monte-Carlo game tree search, in: Twenty-Second International Joint Conference on Artificial Intelligence, 2011.

[35] D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, S. M. Lucas, Solving the physical traveling salesman problem: Tree search and macro actions, IEEE Transactions on Computational Intelligence and AI in Games 6 (1) (2013) 31–45.

[36] Y. Tanabe, K. Yoshizoe, H. Imai, A study on security evaluation methodology for image-based biometrics authentication systems, in: 2009 IEEE 3rd International Conference on Biometrics: Theory, Applications, and Systems, IEEE, 2009, pp. 1–6.

[37] T. M. Dieb, S. Ju, K. Yoshizoe, Z. Hou, J. Shiomi, K. Tsuda, MDTS: automatic complex materials design using Monte Carlo tree search, Science and Technology of Advanced Materials 18 (1) (2017) 498–503.

[38] N. R. Sabar, G. Kendall, Population based Monte Carlo tree search hyper-heuristic for combinatorial optimization problems, Information Sciences 314 (2015) 225–239.

[39] P. Ross, Hyper-heuristics, search methodologies: Introductory tutorials in optimization and decision support techniques (E. K. Burke and G. Kendall, eds.) (2005).

[40] A. Sabharwal, H. Samulowitz, C. Reddy, Guiding combinatorial optimization with UCT, in: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, 2012, pp. 356–361.

[41] G. Chaslot, S. De Jong, J.-T. Saito, J. Uiterwijk, Monte-carlo tree search in production management problems, in: Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Vol. 9198, 2006.

[42] K. Liu, Z. Wu, Q. Wu, Y. Cheng, Smart DAG task scheduling with efficient pruning-based MCTS method, in: 2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), IEEE, 2019, pp. 348–355.

[43] C. D. Rosin, Nested rollout policy adaptation for monte carlo tree search, in: Twenty-Second International Joint Conference on Artificial Intelligence, 2011.

[44] C. Grelier, O. Goudet, J.-K. Hao, On monte carlo tree search for weighted vertex coloring, in: European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar), Springer, 2022, pp. 1–16.

[45] T. Cazenave, B. Negrevergne, F. Sikora, Monte carlo graph coloring, in: Monte Carlo Search International Workshop, Springer, 2020, pp. 100–115.

[46] T. Cazenave, Nested monte-carlo search., in: IJCAI International Joint Conference on Artificial Intelligence, 2009, pp. 456–461.

[47] T. P. Runarsson, M. Schoenauer, M. Sebag, Pilot, rollout and Monte Carlo tree search methods for job shop scheduling, in: International Conference on Learning and Intelligent Optimization, Springer, 2012, pp. 160–174.

[48] P. J. Van Laarhoven, E. H. Aarts, J. K. Lenstra, Job shop scheduling by simulated annealing, Operations Research 40 (1) (1992) 113–125.

[49] S. Edelkamp, M. Gath, C. Greulich, M. Humann, O. Herzog, M. Lawo, Monte-carlo tree search for logistics, in: Commercial Transport, Springer, 2016, pp. 427–440.

[50] C. F. Daganzo, The crane scheduling problem, Transportation Research Part B: Methodological 23 (3) (1989) 159–175.

[51] R. I. Peterkofsky, C. F. Daganzo, A branch and bound solution method for the crane scheduling problem, Transportation Research Part B: Methodological 24 (3) (1990) 159–172.

[52] Y. Zhu, A. Lim, Crane scheduling with non-crossing constraint, Journal of the Operational Research Society 57 (12) (2006) 1464–1471.

[53] A. Lim, B. Rodrigues, Z. Xu, A m-parallel crane scheduling problem with a non-crossing constraint, Naval Research Logistics 54 (2) (2007) 115–127.

[54] D.-H. Lee, H. Q. Wang, L. Miao, Quay crane scheduling with non-interference constraints in port container terminals, Transportation Research Part E: Logistics and Transportation Review 44 (1) (2008) 124–135.

[55] D.-H. Lee, H. Q. Wang, Integrated discrete berth allocation and quay crane scheduling in port container terminals, Engineering Optimization 42 (8) (2010) 747–761.

[56] L. Tang, J. Zhao, J. Liu, Modeling and solution of the joint quay crane and truck scheduling problem, European Journal of Operational Research 236 (3) (2014) 978–990.

[57] A. Zhang, W. Zhang, Y. Chen, G. Chen, X. Chen, Approximate the scheduling of quay cranes with non-crossing constraints, European Journal of Operational Research 258 (3) (2017) 820–828.

[58] M. R. Garey, D. S. Johnson, Computers and intractability, Vol. 174, freeman San Francisco, 1979.

[59] G. B. Dantzig, Discrete-variable extremum problems, Operations Research 5 (2) (1957) 266–288.

[60] E. L. Lawler, Fast approximation algorithms for knapsack problems, Mathematics of Operations Research 4 (4) (1979) 339–356.

[61] M. J. Magazine, O. Oguz, A fully polynomial approximation algorithm for the 0–1 knapsack problem, European Journal of Operational Research 8 (3) (1981) 270–273.

[62] H. Kellerer, U. Pferschy, A new fully polynomial time approximation scheme for the knapsack problem, Journal of Combinatorial Optimization 3 (1) (1999) 59–71.

[63] H. Kellerer, U. Pferschy, Improved dynamic programming in connection with an fptas for the knapsack problem, Journal of Combinatorial Optimization 8 (1) (2004) 5–11.

[64] S. Martello, P. Toth, A new algorithm for the 0-1 knapsack problem, Management Science 34 (5) (1988) 633–644.

[65] D. Pisinger, An expanding-core algorithm for the exact 0–1 knapsack problem, European Journal of Operational Research 87 (1) (1995) 175–187.

[66] D. Pisinger, A minimal algorithm for the 0-1 knapsack problem, Operations Research 45 (5) (1997) 758–767.

[67] S. Martello, D. Pisinger, P. Toth, Dynamic programming and strong bounds for the 0-1 knapsack problem, Management Science 45 (3) (1999) 414–424.

[68] M. Büther, D. Briskorn, Reducing the 0-1 knapsack problem with a single continuous variable to the standard 0-1 knapsack problem, International Journal of Operations Research and Information Systems (IJORIS) 3 (1) (2012) 1–12.

[69] M. Monaci, U. Pferschy, P. Serafini, Exact solution of the robust knapsack problem, Computers & Operations Research 40 (11) (2013) 2625–2631.

[70] D. Pisinger, A. Saidi, Tolerance analysis for 0–1 knapsack problems, European Journal of Operational Research 258 (3) (2017) 866–876.

[71] I. I. Huerta, D. A. Neira, D. A. Ortega, V. Varas, J. Godoy, R. Asín-Achá, Anytime automatic algorithm selection for knapsack, Expert Systems with Applications (2020) 113613.

[72] D. Pisinger, Where are the hard knapsack problems?, Computers & Operations Research 32 (9) (2005) 2271–2284.

[73] J. Jooken, P. Leyman, P. De Causmaecker, A new class of hard problem instances for the 0–1 knapsack problem, European Journal of Operational Research 301 (3) (2022) 841–854.

[74] C. H. Papadimitriou, K. Steiglitz, Combinatorial optimization: algorithms and complexity, Courier Corporation, 1998.

[75] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, E. C. Sewell, Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning, Discrete Optimization 19 (2016) 79–102.

[76] J. Tromp, The number of legal Go positions, in: International Conference on Computers and Games, Springer, 2016, pp. 183–190.

[77] S. Chinchalkar, An upper bound for the number of reachable positions, ICGA Journal 19 (3) (1996) 181–183.

[78] W. Ng, K. Mak, Quay crane scheduling in container terminals, Engineering Optimization 38 (6) (2006) 723–737.

[79] C.-H. Chen, J. Lin, E. Yücesan, S. E. Chick, Simulation budget allocation for further enhancing the efficiency of ordinal optimization, Discrete Event Dynamic Systems 10 (3) (2000) 251–270.

Appendix A: Detailed results for the quay crane scheduling problem with non-crossing constraints

Table A11: Comparison between performance of the different algorithms for dataset A. Smaller objective function values are better.

		CPLEX (Santini et al. [10])			MCTS (this paper)		EBP (Lee and Chen [9])		SPA (Zhang et al. [57])	
Problem instance	Lower bound (5)+(6)	Lower bound	Value	Time	Value	Time	Value	Time	Value	Time
16-4-2,893	724	726	**726 (0.0%)**	2.1 s	**726 (0.0%)**	10.0 s	781 (7.6%)	<0.1 s	1,073 (47.8%)	<0.1 s
16-5-2,893	579	586	**586 (0.0%)**	0.7 s	**586 (0.0%)**	10.0 s	665 (13.5%)	<0.1 s	928 (58.4%)	<0.1 s
17-4-2,941	736	741	**741 (0.0%)**	19.0 s	**741 (0.0%)**	10.0 s	761 (2.7%)	<0.1 s	1,135 (53.2%)	<0.1 s
17-5-2,941	589	600	**600 (0.0%)**	1.8 s	**600 (0.0%)**	10.0 s	636 (6.0%)	<0.1 s	902 (50.3%)	<0.1 s
18-4-2,856	714	720	**720 (0.0%)**	17.9 s	**720 (0.0%)**	10.0 s	745 (3.5%)	<0.1 s	882 (22.5%)	<0.1 s
18-5-2,856	572	579	**579 (0.0%)**	6.4 s	**579 (0.0%)**	10.0 s	619 (6.9%)	<0.1 s	882 (52.3%)	<0.1 s
19-4-2,804	701	702	**702 (0.0%)**	177.3 s	**702 (0.0%)**	10.0 s	711 (1.3%)	<0.1 s	1,069 (52.3%)	<0.1 s
19-5-2,804	561	567	**567 (0.0%)**	65.3 s	**567 (0.0%)**	10.0 s	584 (3.0%)	<0.1 s	878 (54.9%)	<0.1 s
20-4-3,688	922	925	**925 (0.0%)**	846.8 s	**925 (0.0%)**	10.0 s	979 (5.8%)	<0.1 s	1,337 (44.5%)	<0.1 s
20-5-3,688	738	739	**739 (0.0%)**	754.1 s	**739 (0.0%)**	10.0 s	781 (5.7%)	<0.1 s	1,136 (53.7%)	<0.1 s
21-4-3,033	759	759	**759 (0.0%)**	1,408.4 s	**759 (0.0%)**	100.0 s	801 (5.5%)	<0.1 s	1,173 (54.5%)	<0.1 s
21-5-3,033	607	612	**612 (0.0%)**	3,376.3 s	614 (0.3%)	100.0 s	622 (1.6%)	<0.1 s	906 (48.0%)	<0.1 s
22-4-3,027	757	757	**757 (0.0%)**	3,343.5 s	**757 (0.0%)**	10.0 s	766 (1.2%)	<0.1 s	1,015 (34.1%)	<0.1 s
22-5-3,027	606	611	**611 (0.0%)**	485.8 s	**611 (0.0%)**	10.0 s	643 (5.2%)	<0.1 s	931 (52.4%)	<0.1 s
23-4-3,544	886	886	887 (0.1%)	3,600.0 s	**886 (0.0%)**	10.0 s	910 (2.7%)	<0.1 s	1,297 (46.4%)	<0.1 s
23-5-3,544	709	709	713 (0.6%)	3,600.0 s	**712 (0.4%)**	10.0 s	740 (4.4%)	<0.1 s	1,159 (63.5%)	<0.1 s
24-4-3,430	858	858	**858 (0.0%)**	3,600.0 s	**858 (0.0%)**	10.0 s	874 (1.9%)	<0.1 s	1,337 (55.8%)	<0.1 s
24-5-3,430	686	686	693 (1.0%)	3,600.0 s	**691 (0.7%)**	10.0 s	712 (3.8%)	<0.1 s	1,135 (65.5%)	<0.1 s
25-4-4,334	1,084	1,084	1,087 (0.3%)	3,600.0 s	**1,084 (0.0%)**	10.0 s	1,129 (4.2%)	<0.1 s	1,623 (49.7%)	<0.1 s
25-5-4,334	867	867	872 (0.6%)	3,600.0 s	**870 (0.3%)**	100.0 s	921 (6.2%)	<0.1 s	1,392 (60.6%)	<0.1 s
50-8-7,986	999	999	1,006 (0.7%)	3,600.0 s	**1,001 (0.2%)**	100.0 s	1,046 (4.7%)	<0.1 s	1,700 (70.2%)	<0.1 s
50-10-7,986	799	799	804 (0.6%)	3,600.0 s	**803 (0.5%)**	100.0 s	897 (12.3%)	<0.1 s	1,365 (70.8%)	<0.1 s
100-8-16,519	2,065	2,065	2,087 (1.1%)	3,600.0 s	**2,067 (0.1%)**	100.0 s	2,124 (2.9%)	<0.1 s	3,635 (76.0%)	<0.1 s
100-10-16,519	1,652	1,652	1,675 (1.4%)	3,600.0 s	**1,656 (0.2%)**	10.0 s	1,747 (5.8%)	<0.1 s	2,972 (79.9%)	<0.1 s
Avg.	840.4	842.9	846.1 (0.4%)	1,937.7 s	**843.9 (0.1%)**	32.5 s	883.1 (4.8%)	<0.1 s	1,327.6 (57.5%)	<0.1 s
p-value	-	-	0.010	-	REF. ALG.	-	5.960×10^{-8}	-	5.960×10^{-8}	-

Table A12: Comparison between performance of the different algorithms for dataset B. Smaller objective function values are better.

		CPLEX (Santini et al. [10])			MCTS (this paper)		EBP (Lee and Chen [9])		SPA (Zhang et al. [57])	
Problem instance	Lower bound (5)+(6)	Lower bound	Value	Time	Value	Time	Value	Time	Value	Time
200-4-33,532	8,383	8,383	23,852 (184.5%)	3,600.0 s	**8,383 (0.0%)**	10.0 s	8,471 (1.0%)	<0.1 s	13,270 (58.3%)	<0.1 s
200-5-33,532	6,707	6,707	21,433 (219.6%)	3,600.0 s	**6,707 (0.0%)**	10.0 s	6,744 (0.6%)	<0.1 s	11,107 (65.6%)	<0.1 s
200-8-33,532	4,192	4,192	10,294 (145.6%)	3,600.0 s	**4,193 (0.0%)**	100.0 s	4,237 (1.1%)	<0.1 s	7,259 (73.2%)	<0.1 s
200-10-33,532	3,354	3,354	9,884 (194.7%)	3,600.0 s	**3,356 (0.1%)**	100.0 s	3,440 (2.6%)	<0.1 s	6,019 (79.5%)	<0.1 s
250-4-42,573	10,644	10,644	35,550 (234.0%)	3,600.0 s	**10,644 (0.0%)**	10.0 s	10,655 (0.1%)	0.1 s	16,925 (59.0%)	<0.1 s
250-5-42,573	8,515	8,515	40,087 (370.8%)	3,600.0 s	**8,515 (0.0%)**	10.0 s	8,588 (0.9%)	<0.1 s	14,184 (66.6%)	<0.1 s
250-8-42,573	5,322	5,322	37,483 (604.3%)	3,600.0 s	**5,323 (0.0%)**	100.0 s	5,384 (1.2%)	<0.1 s	9,377 (76.2%)	<0.1 s
250-10-42,573	4,258	4,258	38,097 (794.7%)	3,600.0 s	**4,267 (0.2%)**	10.0 s	4,319 (1.4%)	<0.1 s	7,697 (80.8%)	<0.1 s
300-4-50,767	12,692	12,692	44,909 (253.8%)	3,600.0 s	**12,692 (0.0%)**	10.0 s	12,724 (0.3%)	0.1 s	20,168 (58.9%)	<0.1 s
300-5-50,767	10,154	10,154	46,183 (354.8%)	3,600.0 s	**10,154 (0.0%)**	10.0 s	10,192 (0.4%)	0.1 s	16,869 (66.1%)	<0.1 s
300-8-50,767	6,346	6,346	47,995 (656.3%)	3,600.0 s	**6,349 (0.0%)**	100.0 s	6,393 (0.7%)	0.1 s	11,218 (76.8%)	<0.1 s
300-10-50,767	5,077	5,077	46,086 (807.7%)	3,600.0 s	**5,081 (0.1%)**	100.0 s	5,148 (1.4%)	<0.1 s	9,202 (81.2%)	<0.1 s
2,000-4-323,427	80,857	-	-	-	**80,857 (0.0%)**	10.0 s	80,910 (0.1%)	2.0 s	129,256 (59.9%)	<0.1 s
2,000-5-323,427	64,686	-	-	-	**64,686 (0.0%)**	10.0 s	64,729 (0.1%)	2.5 s	107,801 (66.7%)	<0.1 s
2,000-8-323,427	40,429	-	-	-	**40,429 (0.0%)**	100.0 s	40,483 (0.1%)	1.4 s	71,825 (77.7%)	<0.1 s
2,000-10-323,427	32,343	-	-	-	**32,360 (0.1%)**	100.0 s	32,401 (0.2%)	0.9 s	58,730 (81.6%)	<0.1 s
2,500-4-410,999	102,750	-	-	-	**102,750 (0.0%)**	10.0 s	102,829 (0.1%)	3.3 s	164,365 (60.0%)	<0.1 s
2,500-5-410,999	82,200	-	-	-	**82,200 (0.0%)**	10.0 s	82,241 (0.0%)	4.4 s	136,956 (66.6%)	<0.1 s
2,500-8-410,999	51,375	-	-	-	**51,376 (0.0%)**	100.0 s	51,427 (0.1%)	3.1 s	91,208 (77.5%)	<0.1 s
2,500-10-410,999	41,100	-	-	-	**41,115 (0.0%)**	100.0 s	41,137 (0.1%)	1.5 s	74,668 (81.7%)	<0.1 s
3,000-4-495,726	123,932	-	-	-	**123,932 (0.0%)**	10.0 s	123,967 (0.0%)	12.5 s	198,256 (60.0%)	<0.1 s
3,000-5-495,726	99,146	-	-	-	**99,146 (0.0%)**	10.0 s	99,184 (0.0%)	9.0 s	165,225 (66.6%)	<0.1 s
3,000-8-495,726	61,966	-	-	-	**61,978 (0.0%)**	100.0 s	62,051 (0.1%)	4.6 s	110,155 (77.8%)	<0.1 s
3,000-10-495,726	49,573	-	-	-	**49,585 (0.0%)**	100.0 s	49,632 (0.1%)	4.2 s	90,000 (81.6%)	<0.1 s
Avg.	38,166.7	-	-	-	**38,169.9 (0.0 %)**	51.3 s	38,220.3 (0.1 %)	2.1 s	64,239.2 (68.3 %)	<0.1 s
Avg. feasible	7,137.0	7,137.0	33,487.8 (369.2 %)	3,600.0 s	**7,138.7 (0.0 %)**	47.5 s	7,191.3 (0.8 %)	<0.1 s	11,941.3 (67.3 %)	<0.1 s
p-value	-	-	-	-	REF. ALG.	-	5.96×10^{-8}	-	5.96×10^{-8}	-

Table A13: Dataset A: comparison between different versions of MCTS where a subset of enhancements are omitted (versions 1-6). Smaller objective function values are better.

Problem instance	MCTS Version 1	MCTS Version 2	MCTS Version 3	MCTS Version 4	MCTS Version 5	MCTS Version 6
Heuristic simulation policy	✓	✓	✓	✓	✓	✓
Beam	✓	✓	✓	✓	✗	✗
Domain reduction	✓	✓	✗	✗	✓	✓
Pruning subtrees by calculating bounds	✓	✗	✓	✗	✓	✗
16-4-2,893	**726**	739	**726**	739	**726**	739
16-5-2,893	**586**	611	**586**	611	**586**	611
17-4-2,941	**741**	744	**741**	**741**	**741**	744
17-5-2,941	**600**	608	**600**	608	**600**	608
18-4-2,856	**720**	722	**720**	722	**720**	722
18-5-2,856	**579**	**579**	**579**	**579**	**579**	**579**
19-4-2,804	**702**	703	**702**	703	**702**	703
19-5-2,804	**567**	**567**	**567**	**567**	**567**	**567**
20-4-3,688	**925**	926	**925**	**925**	**925**	926
20-5-3,688	**739**	755	**739**	755	**739**	755
21-4-3,033	**759**	**759**	**759**	762	**759**	760
21-5-3,033	614	616	618	619	614	619
22-4-3,027	**757**	758	**757**	758	**757**	758
22-5-3,027	**611**	619	**611**	**611**	**611**	621
23-4-3,544	**886**	888	**886**	888	**886**	888
23-5-3,544	**712**	713	**712**	713	**712**	**712**
24-4-3,430	**858**	859	**858**	**858**	**858**	**858**
24-5-3,430	**691**	693	**691**	693	**691**	693
25-4-4,334	**1,084**	1,085	1,085	**1,084**	**1,084**	**1,084**
25-5-4,334	870	874	872	874	869	874
50-8-7,986	**1,001**	1,007	1,004	1,019	1,004	1,010
50-10-7,986	**803**	825	824	830	813	823
100-8-16,519	**2,067**	2,071	2,079	2,091	2,071	2,071
100-10-16,519	**1,656**	1,666	1,687	1,710	1,660	1,666
Avg.	**843.9**	849.5	847.0	852.5	844.8	849.6
p-value	REF. ALG.	4.768×10^{-7}	0.008	7.629×10^{-6}	0.063	1.907×10^{-6}

Table A14: Dataset A: comparison between different versions of MCTS where a subset of enhancements are omitted (versions 7-12). Smaller objective function values are better.

Problem instance	MCTS Version 7	MCTS Version 8	MCTS Version 9	MCTS Version 10	MCTS Version 11	MCTS Version 12
Heuristic simulation policy	✓	✓	✗	✗	✗	✗
Beam	✗	✗	✓	✓	✓	✓
Domain reduction	✗	✗	✓	✓	✗	✗
Pruning subtrees by calculating bounds	✓	✗	✓	✗	✓	✗
16-4-2,893	**726**	739	**726**	739	**726**	742
16-5-2,893	**586**	611	**586**	628	**586**	625
17-4-2,941	**741**	743	**741**	752	**741**	748
17-5-2,941	**600**	608	**600**	615	**600**	608
18-4-2,856	**720**	722	**720**	750	**720**	731
18-5-2,856	**579**	**579**	**579**	586	**579**	608
19-4-2,804	**702**	703	**702**	728	**702**	722
19-5-2,804	**567**	**567**	**567**	591	**567**	588
20-4-3,688	**925**	926	**925**	952	**925**	944
20-5-3,688	**739**	755	**739**	785	**739**	769
21-4-3,033	**759**	**759**	**759**	769	760	771
21-5-3,033	**612**	619	**612**	630	**612**	622
22-4-3,027	**757**	758	759	771	758	780
22-5-3,027	**611**	618	**611**	654	**611**	646
23-4-3,544	**886**	889	887	903	889	898
23-5-3,544	**712**	713	**712**	749	716	727
24-4-3,430	**858**	859	**858**	894	**858**	878
24-5-3,430	**691**	693	**691**	730	**691**	717
25-4-4,334	**1,084**	1,085	1,085	1,130	**1,084**	1,099
25-5-4,334	871	874	**868**	916	869	895
50-8-7,986	1,003	1,021	1,071	1,081	1,087	1,224
50-10-7,986	823	830	889	974	1,129	1,291
100-8-16,519	2,071	2,091	2,322	2,272	2,408	2,445
100-10-16,519	1,669	1,675	1,972	2,058	2,908	2,711
Avg.	845.5	851.5	874.2	902.4	927.7	949.5
p-value	0.063	4.768×10^{-7}	0.047	5.960×10^{-8}	0.014	5.960×10^{-8}

Table A15: Dataset A: comparison between different versions of MCTS where a subset of enhancements are omitted (versions 13-16). Smaller objective function values are better.

Problem instance	MCTS Version 13	MCTS Version 14	MCTS Version 15	MCTS Version 16
Heuristic simulation policy	✗	✗	✗	✗
Beam	✗	✗	✗	✗
Domain reduction	✓	✓	✗	✗
Pruning subtrees by calculating bounds	✓	✗	✓	✗
16-4-2,893	**726**	740	**726**	741
16-5-2,893	**586**	616	**586**	625
17-4-2,941	**741**	752	**741**	743
17-5-2,941	**600**	641	**600**	611
18-4-2,856	**720**	760	**720**	739
18-5-2,856	**579**	595	**579**	628
19-4-2,804	**702**	723	**702**	703
19-5-2,804	**567**	594	**567**	605
20-4-3,688	**925**	938	**925**	942
20-5-3,688	**739**	779	**739**	763
21-4-3,033	**759**	772	**759**	773
21-5-3,033	**612**	634	**612**	629
22-4-3,027	758	792	**757**	765
22-5-3,027	614	645	**611**	637
23-4-3,544	887	890	887	889
23-5-3,544	713	731	**712**	735
24-4-3,430	**858**	879	**858**	875
24-5-3,430	**691**	720	**691**	718
25-4-4,334	1,085	1,114	**1,084**	1,110
25-5-4,334	870	918	873	919
50-8-7,986	1,076	1,149	1,139	1,179
50-10-7,986	911	936	1,019	1,121
100-8-16,519	2,188	2,279	2,458	2,319
100-10-16,519	1,873	2,055	2,564	2,720
Avg.	865.8	902.2	912.9	937.0
p-value	0.012	5.960×10^{-8}	0.023	5.960×10^{-8}

Table A16: Dataset B: comparison between different versions of MCTS where a subset of enhancements are omitted (versions 1-6). Smaller objective function values are better.

Problem instance	MCTS Version 1	MCTS Version 2	MCTS Version 3	MCTS Version 4	MCTS Version 5	MCTS Version 6
Heuristic simulation policy	✓	✓	✓	✓	✓	✓
Beam	✓	✓	✓	✓	✗	✗
Domain reduction	✓	✓	✗	✗	✓	✓
Pruning subtrees by calculating bounds	✓	✗	✓	✗	✓	✗
200-4-33,532	**8,383**	**8,383**	**8,383**	**8,383**	**8,383**	**8,383**
200-5-33,532	**6,707**	**6,707**	**6,707**	**6,707**	**6,707**	**6,707**
200-8-33,532	**4,193**	4,199	4,197	4,216	4,194	4,196
200-10-33,532	**3,356**	3,364	3,382	3,390	3,360	3,361
250-4-42,573	**10,644**	**10,644**	**10,644**	**10,644**	**10,644**	**10,644**
250-5-42,573	**8,515**	**8,515**	**8,515**	**8,515**	**8,515**	**8,515**
250-8-42,573	**5,323**	5,333	5,326	5,334	**5,323**	5,333
250-10-42,573	**4,267**	4,277	4,283	4,289	4,270	4,277
300-4-50,767	**12,692**	**12,692**	**12,692**	**12,692**	**12,692**	**12,692**
300-5-50,767	**10,154**	**10,154**	**10,154**	**10,154**	**10,154**	**10,154**
300-8-50,767	**6,349**	6,353	6,352	6,358	6,351	6,354
300-10-50,767	**5,081**	5,089	5,097	5,112	5,089	5,087
2,000-4-323,427	**80,857**	**80,857**	**80,857**	**80,857**	**80,857**	**80,857**
2,000-5-323,427	**64,686**	**64,686**	**64,686**	**64,686**	**64,686**	**64,686**
2,000-8-323,427	**40,429**	40,442	40,446	40,453	40,438	40,440
2,000-10-323,427	**32,360**	32,388	32,400	32,400	32,386	32,367
2,500-4-410,999	**102,750**	**102,750**	**102,750**	**102,750**	**102,750**	**102,750**
2,500-5-410,999	**82,200**	**82,200**	**82,200**	82,201	**82,200**	**82,200**
2,500-8-410,999	**51,376**	51,385	51,421	51,430	51,391	51,384
2,500-10-410,999	**41,115**	41,130	41,181	41,181	41,122	41,130
3,000-4-495,726	**123,932**	**123,932**	**123,932**	**123,932**	**123,932**	**123,932**
3,000-5-495,726	**99,146**	**99,146**	**99,146**	**99,146**	**99,146**	**99,146**
3,000-8-495,726	**61,978**	61,982	62,017	62,017	61,982	62,005
3,000-10-495,726	**49,585**	49,632	49,678	49,697	49,611	49,629
Avg.	**38,169.9**	38,176.7	38,185.3	38,189.3	38,174,3	38,176.2
Avg. feasible	**13,385.5**	13,386.4	13,386.0	13,387.3	13,385.7	13,386.3
p-value	REF. ALG.	2.441×10^{-4}	2.441×10^{-4}	1.221×10^{-4}	4.883×10^{-4}	2.441×10^{-4}

Table A17: Dataset B: comparison between different versions of MCTS where a subset of enhancements are omitted (versions 7-12). Smaller objective function values are better.

Problem instance	MCTS Version 7	MCTS Version 8	MCTS Version 9	MCTS Version 10	MCTS Version 11	MCTS Version 12
Heuristic simulation policy	✓	✓	✗	✗	✗	✗
Beam	✗	✗	✓	✓	✓	✓
Domain reduction	✗	✗	✓	✓	✗	✗
Pruning subtrees by calculating bounds	✓	✗	✓	✗	✓	✗
200-4-33,532	**8,383**	**8,383**	8,435	8,452	8,459	8,398
200-5-33,532	**6,707**	**6,707**	6,771	6,826	6,779	6,797
200-8-33,532	4,200	4,223	4,558	4,665	4,873	5,602
200-10-33,532	3,382	3,393	4,010	3,938	4,218	-
250-4-42,573	**10,644**	**10,644**	10,667	10,694	10,741	10,682
250-5-42,573	**8,515**	**8,515**	8,613	8,655	8,661	8,644
250-8-42,573	5,328	5,361	5,589	5,660	5,807	5,927
250-10-42,573	4,304	4,367	4,753	6,428	-	-
300-4-50,767	**12,692**	**12,692**	12,732	12,755	12,706	12,735
300-5-50,767	**10,154**	**10,154**	10,318	10,333	10,336	10,287
300-8-50,767	6,353	6,388	6,858	6,771	8,166	7,028
300-10-50,767	5,124	5,136	5,442	-	6,450	-
2,000-4-323,427	**80,857**	**80,857**	82,162	82,252	83,495	82,902
2,000-5-323,427	**64,686**	**64,686**	66,751	66,647	72,744	70,741
2,000-8-323,427	40,451	40,453	49,946	-	-	-
2,000-10-323,427	32,400	32,400	-	-	-	-
2,500-4-410,999	**102,750**	**102,750**	104,610	104,359	105,573	106,020
2,500-5-410,999	**82,200**	**82,200**	85,388	85,337	-	86,603
2,500-8-410,999	51,388	51,430	-	63,915	-	-
2,500-10-410,999	41,197	41,198	-	-	-	-
3,000-4-495,726	**123,932**	**123,932**	126,264	125,980	-	128,654
3,000-5-495,726	**99,146**	99,153	105,683	103,811	-	105,960
3,000-8-495,726	61,982	62,017	-	-	-	-
3,000-10-495,726	49,678	49,678	-	-	-	-
Avg.	38,185.5	38,196.5	-	-	-	-
Avg. feasible	13,386.2	13,390.0	13,669.3	13,669.5	14,097.5	13,990.1
p-value	2.441×10^{-4}	1.221×10^{-4}	-	-	-	-

Table A18: Dataset B: comparison between different versions of MCTS where a subset of enhancements are omitted (versions 13-16). Smaller objective function values are better.

Problem instance	MCTS Version 13	MCTS Version 14	MCTS Version 15	MCTS Version 16
Heuristic simulation policy	✗	✗	✗	✗
Beam	✗	✗	✗	✗
Domain reduction	✓	✓	✗	✗
Pruning subtrees by calculating bounds	✓	✗	✓	✗
200-4-33,532	8,478	8,477	8,447	8,476
200-5-33,532	6,843	6,826	6,744	6,845
200-8-33,532	4,565	4,523	4,902	4,796
200-10-33,532	3,933	3,858	-	-
250-4-42,573	10,692	10,716	10,736	10,774
250-5-42,573	8,667	8,654	8,692	8,694
250-8-42,573	5,788	5,901	7,547	7,706
250-10-42,573	5,068	4,902	-	-
300-4-50,767	12,747	12,789	12,794	12,790
300-5-50,767	10,249	10,305	10,394	10,269
300-8-50,767	6,858	6,874	7,111	7,358
300-10-50,767	5,789	5,652	-	8,775
2,000-4-323,427	82,261	81,961	84,351	84,693
2,000-5-323,427	66,982	67,373	67,703	68,871
2,000-8-323,427	50,409	46,155	-	-
2,000-10-323,427	-	-	-	-
2,500-4-410,999	104,575	104,354	105,493	107,255
2,500-5-410,999	85,271	85,206	-	89,061
2,500-8-410,999	-	-	-	-
2,500-10-410,999	-	-	-	-
3,000-4-495,726	126,368	127,351	128,852	128,819
3,000-5-495,726	114,056	101,415	107,256	106,724
3,000-8-495,726	-	-	-	-
3,000-10-495,726	-	-	-	-
Avg. feasible	13,696.0	13,698.0	13,954.8	14,105.3

Table A19: Dataset A: comparsion between different selection policies of MCTS.

Problem instance	Lower bound	MCTS (selection policy uses (3))	MCTS (selection policy uses (11))	OCBA-MCTS (Li et al. [27])
16-4-2,893	726	**726 (0.0%)**	**726 (0.0%)**	**726 (0.0%)**
16-5-2,893	586	**586 (0.0%)**	**586 (0.0%)**	**586 (0.0%)**
17-4-2,941	741	**741 (0.0%)**	**741 (0.0%)**	**741 (0.0%)**
17-5-2,941	600	**600 (0.0%)**	**600 (0.0%)**	**600 (0.0%)**
18-4-2,856	720	**720 (0.0%)**	**720 (0.0%)**	**720 (0.0%)**
18-5-2,856	579	**579 (0.0%)**	**579 (0.0%)**	**579 (0.0%)**
19-4-2,804	702	**702 (0.0%)**	**702 (0.0%)**	**702 (0.0%)**
19-5-2,804	567	**567 (0.0%)**	**567 (0.0%)**	**567 (0.0%)**
20-4-3,688	925	**925 (0.0%)**	**925 (0.0%)**	**925 (0.0%)**
20-5-3,688	739	**739 (0.0%)**	**739 (0.0%)**	**739 (0.0%)**
21-4-3,033	759	**759 (0.0%)**	**759 (0.0%)**	762 (0.4%)
21-5-3,033	612	**614 (0.3%)**	616 (0.7%)	**614 (0.3%)**
22-4-3,027	757	**757 (0.0%)**	**757 (0.0%)**	**757 (0.0%)**
22-5-3,027	611	**611 (0.0%)**	**611 (0.0%)**	**611 (0.0%)**
23-4-3,544	886	**886 (0.0%)**	**886 (0.0%)**	887 (0.1%)
23-5-3,544	709	**712 (0.4%)**	**712 (0.4%)**	**712 (0.4%)**
24-4-3,430	858	**858 (0.0%)**	**858 (0.0%)**	**858 (0.0%)**
24-5-3,430	686	**691 (0.7%)**	**691 (0.7%)**	**691 (0.7%)**
25-4-4,334	1,084	**1,084 (0.0%)**	**1,084 (0.0%)**	1,085 (0.1%)
25-5-4,334	867	870 (0.3%)	869 (0.2%)	**868 (0.1%)**
50-8-7,986	999	**1,001 (0.2%)**	1,004 (0.5%)	1,006 (0.7%)
50-10-7,986	799	**803 (0.5%)**	812 (1.6%)	813 (1.8%)
100-8-16,519	2,065	**2,067 (0.1%)**	2,068 (0.1%)	2,070 (0.2%)
100-10-16,519	1,652	**1,656 (0.2%)**	1,660 (0.5%)	1,663 (0.7%)
Avg.	842.9	**843.9 (0.1%)**	844.7 (0.2%)	845.1 (0.3%)
p-value	-	REF. ALG.	0.469	0.195

Table A20: Dataset B: comparsion between different selection policies of MCTS.

Problem instance	Lower bound	MCTS (selection policy uses (3))	MCTS (selection policy uses (11))	OCBA-MCTS (Li et al. [27])
200-4-33,532	8,383	**8,383 (0.0%)**	**8,383 (0.0%)**	**8,383 (0.0%)**
200-5-33,532	6,707	**6,707 (0.0%)**	**6,707 (0.0%)**	**6,707 (0.0%)**
200-8-33,532	4,192	**4,193 (0.0%)**	4,194 (0.0%)	**4,193 (0.0%)**
200-10-33,532	3,354	**3,356 (0.1%)**	3,361 (0.2%)	3,360 (0.2%)
250-4-42,573	10,644	**10,644 (0.0%)**	**10,644 (0.0%)**	**10,644 (0.0%)**
250-5-42,573	8,515	**8,515 (0.0%)**	**8,515 (0.0%)**	**8,515 (0.0%)**
250-8-42,573	5,322	**5,323 (0.0%)**	**5,323 (0.0%)**	5,336 (0.3%)
250-10-42,573	4,258	**4,267 (0.2%)**	4,269 (0.3%)	4,269 (0.3%)
300-4-50,767	12,692	**12,692 (0.0%)**	**12,692 (0.0%)**	**12,692 (0.0%)**
300-5-50,767	10,154	**10,154 (0.0%)**	**10,154 (0.0%)**	**10,154 (0.0%)**
300-8-50,767	6,346	**6,349 (0.0%)**	**6,349 (0.0%)**	**6,349 (0.0%)**
300-10-50,767	5,077	**5,081 (0.1%)**	**5,081 (0.1%)**	5,088 (0.2%)
2,000-4-323,427	80,857	**80,857 (0.0%)**	**80,857 (0.0%)**	**80,857 (0.0%)**
2,000-5-323,427	64,686	**64,686 (0.0%)**	**64,686 (0.0%)**	**64,686 (0.0%)**
2,000-8-323,427	40,429	**40,429 (0.0%)**	40,447 (0.0%)	40,446 (0.0%)
2,000-10-323,427	32,343	**32,360 (0.1%)**	32,400 (0.2%)	32,367 (0.1%)
2,500-4-410,999	102,750	**102,750 (0.0%)**	**102,750 (0.0%)**	**102,750 (0.0%)**
2,500-5-410,999	82,200	**82,200 (0.0%)**	**82,200 (0.0%)**	**82,200 (0.0%)**
2,500-8-410,999	51,375	**51,376 (0.0%)**	51,430 (0.1%)	51,398 (0.0%)
2,500-10-410,999	41,100	**41,115 (0.0%)**	41,130 (0.1%)	41,162 (0.2%)
3,000-4-495,726	123,932	**123,932 (0.0%)**	**123,932 (0.0%)**	**123,932 (0.0%)**
3,000-5-495,726	99,146	**99,146 (0.0%)**	**99,146 (0.0%)**	**99,146 (0.0%)**
3,000-8-495,726	61,966	**61,978 (0.0%)**	61,982 (0.0%)	62,017 (0.1%)
3,000-10-495,726	49,573	**49,585 (0.0%)**	49,647 (0.1%)	49,632 (0.1%)
Avg.	38,166.7	**38,169.9 (0.0%)**	38,178.3 (0.0%)	38,178.5 (0.0%)
p-value	-	REF. ALG.	0.002	9.776×10^{-4}

Appendix B: Detailed results for the 0-1 knapsack problem

Table B21: Comparison between Combo and MCTS for dataset A

	Combo		MCTS	
Problem instance	Value	Time	Value	Time
strCorrSpan-50-371,096,080	**743,839,440**	<0.1 s	**743,839,440 (0.0 %)**	10.0 s
strCorrSpan-50-646,374,982	**953,046,980**	<0.1 s	**953,046,980 (0.0 %)**	10.0 s
strCorrSpan-50-2,134,353,461	**2,527,368,539**	<0.1 s	**2,527,368,539 (0.0 %)**	10.0 s
strCorrSpan-200-1,707,778,944	**3,246,231,168**	<0.1 s	**3,246,231,168 (0.0 %)**	10.0 s
strCorrSpan-200-2,817,370,945	**4,061,278,176**	<0.1 s	**4,061,278,176 (0.0 %)**	10.0 s
strCorrSpan-200-9,187,378,001	**10,985,275,013**	<0.1 s	**10,985,275,013 (0.0 %)**	10.0 s
strCorrSpan-500-4,265,423,637	**8,261,226,266**	<0.1 s	**8,261,226,266 (0.0 %)**	10.0 s
strCorrSpan-500-7,384,719,424	**10,421,994,304**	<0.1 s	**10,421,994,304 (0.0 %)**	10.0 s
strCorrSpan-500-22,936,987,823	**27,556,403,070**	<0.1 s	**27,556,403,070 (0.0 %)**	10.0 s
strCorrSpan-2,000-17,052,969,836	**32,783,499,328**	<0.1 s	**32,783,499,328 (0.0 %)**	10.0 s
strCorrSpan-2,000-28,036,577,373	**40,157,803,058**	<0.1 s	40,146,187,862 (2.9×10^{-2} %)	10.0 s
strCorrSpan-2,000-92,148,671,511	**110,409,143,147**	<0.1 s	110,409,094,321 (4.4×10^{-5} %)	10.0 s
strCorrSpan-5,000-41,267,918,845	**81,368,124,258**	0.1 s	**81,368,124,258 (0.0 %)**	10.0 s
strCorrSpan-5,000-71,023,181,263	**100,217,939,868**	0.1 s	100,217,640,112 (3.0×10^{-4} %)	10.0 s
strCorrSpan-5,000-226,199,500,674	**271,790,411,102**	0.2 s	271,781,871,491 (3.1×10^{-3} %)	10.0 s
strCorrSpan-20,000-166,246,484,639	**324,652,926,326**	3.5 s	**324,652,926,326 (0.0 %)**	10.0 s
strCorrSpan-20,000-284,753,555,248	**401,750,506,702**	0.5 s	**401,750,506,702 (0.0 %)**	10.0 s
strCorrSpan-20,000-902,111,715,974	**1,083,393,835,104**	3.9 s	1,083,393,761,865 (7.0×10^{-6} %)	10.0 s
strCorrSpan-50,000-412,563,911,323	**809,153,725,808**	26.7 s	809,149,037,830 (5.8×10^{-4} %)	10.0 s
strCorrSpan-50,000-713,142,682,378	**1,005,107,183,744**	11.8 s	1,005,094,669,280 (1.2×10^{-3} %)	10.0 s
strCorrSpan-50,000-2,251,823,185,869	**2,704,243,765,007**	12.2 s	**2,704,243,765,007 (0.0 %)**	10.0 s
Avg.	**334,942,167,924.2**	2.8 s	334,940,368,920.9 (5.3×10^{-4} %)	10.0 s
p-value	0.992	-	REF. ALG.	-

Table B22: Comparison between Combo and MCTS for dataset B

	Combo		MCTS	
Problem instance	Value	Time	Value	Time
exp-100-10,000,000,000	**9,999,949,957**	2.7 s	9,999,478,296 (4.7×10^{-3} %)	10.0 s
exp-125-10,000,000,000	**9,999,952,568**	5.8 s	9,999,479,807 (4.7×10^{-3} %)	10.0 s
exp-150-10,000,000,000	**9,999,954,224**	10.5 s	9,999,951,727 (2.5×10^{-5} %)	100.0 s
exp-175-10,000,000,000	**9,999,955,429**	12.1 s	9,999,953,940 (1.5×10^{-5} %)	10.0 s
exp-200-10,000,000,000	**9,999,958,144**	18.4 s	9,999,956,638 (1.5×10^{-5} %)	10.0 s
exp-225-10,000,000,000	**9,999,959,636**	25.7 s	9,999,958,340 (1.3×10^{-5} %)	10.0 s
exp-250-10,000,000,000	**9,999,960,633**	47.8 s	9,999,959,300 (1.3×10^{-5} %)	10.0 s
exp-275-10,000,000,000	**9,999,962,074**	72.3 s	9,999,960,122 (2.0×10^{-5} %)	10.0 s
exp-300-10,000,000,000	**9,999,964,031**	135.3 s	9,999,962,434 (1.6×10^{-5} %)	10.0 s
exp-325-10,000,000,000	**9,999,965,949**	150.9 s	9,999,963,242 (2.7×10^{-5} %)	10.0 s
exp-350-10,000,000,000	**9,999,967,542**	194.7 s	9,999,957,621 (9.9×10^{-5} %)	100.0 s
exp-375-10,000,000,000	**9,999,969,532**	233.2 s	9,999,960,927 (8.6×10^{-5} %)	100.0 s
exp-400-10,000,000,000	**9,999,969,939**	347.6 s	9,999,967,171 (2.8×10^{-5} %)	100.0 s
exp-425-10,000,000,000	**9,999,969,103**	445.0 s	9,999,961,354 (7.7×10^{-5} %)	100.0 s
exp-450-10,000,000,000	**9,999,969,729**	696.0 s	9,999,960,321 (9.4×10^{-5} %)	100.0 s
exp-475-10,000,000,000	**9,999,971,162**	759.4 s	9,999,957,931 (1.3×10^{-4} %)	100.0 s
exp-500-10,000,000,000	**9,999,972,478**	891.5 s	9,999,959,782 (1.3×10^{-4} %)	100.0 s
exp-525-10,000,000,000	**9,999,974,083**	1,429.6 s	9,999,958,458 (1.6×10^{-4} %)	100.0 s
exp-550-10,000,000,000	**9,999,975,475**	1,928.9 s	9,999,958,398 (1.7×10^{-4} %)	100.0 s
exp-575-10,000,000,000	**9,999,976,688**	2,031.7 s	9,999,958,138 (1.9×10^{-4} %)	100.0 s
exp-600-10,000,000,000	**9,999,978,232**	2,185.7 s	9,999,957,600 (2.1×10^{-4} %)	100.0 s
exp-625-10,000,000,000	**9,999,978,747**	2,284.9 s	9,999,958,763 (2.0×10^{-4} %)	100.0 s
exp-650-10,000,000,000	**9,999,980,785**	2,354.7 s	9,999,958,972 (2.2×10^{-4} %)	100.0 s
exp-675-10,000,000,000	**9,999,982,708**	2,533.2 s	9,999,959,169 (2.3×10^{-4} %)	100.0 s
exp-700-10,000,000,000	**9,999,985,183**	2,874.6 s	9,999,959,501 (2.6×10^{-4} %)	100.0 s
Avg.	**9,999,968,161.2**	866.9 s	9,999,920,718.0 (4.7×10^{-4} %)	67.6 s
p-value	1.000	-	REF. ALG.	-

Table B23: Dataset A: comparison between different versions of MCTS where the modifications that we propose are introduced one by one

Problem instance	MCTS Version 1	MCTS Version 2	MCTS Version 3	MCTS Version 4	MCTS Version 5
Heuristic simulation policy	✓	✓	✓	✓	✗
Beam	✓	✓	✓	✗	✗
Domain reduction	✓	✓	✗	✗	✗
Pruning subtrees by calculating bounds	✓	✗	✗	✗	✗
strCorrSpan-50-371,096,080	**743,839,440**	**743,839,440**	**743,839,440**	**743,839,440**	729,907,992
strCorrSpan-50-646,374,982	**953,046,980**	**953,046,980**	**953,046,980**	**953,046,980**	0
strCorrSpan-50-2,134,353,461	**2,527,368,539**	**2,527,368,539**	**2,527,368,539**	**2,527,368,539**	2,518,853,341
strCorrSpan-200-1,707,778,944	**3,246,231,168**	**3,246,231,168**	**3,246,231,168**	**3,246,231,168**	3,014,716,388
strCorrSpan-200-2,817,370,945	**4,061,278,176**	**4,061,278,176**	**4,061,278,176**	**4,061,278,176**	0
strCorrSpan-200-9,187,378,001	**10,985,275,013**	**10,985,275,013**	**10,985,275,013**	**10,985,275,013**	10,874,504,200
strCorrSpan-500-4,265,423,637	**8,261,226,266**	**8,261,226,266**	**8,261,226,266**	**8,261,226,266**	7,390,128,594
strCorrSpan-500-7,384,719,424	**10,421,994,304**	**10,421,994,304**	**10,421,994,304**	**10,421,994,304**	0
strCorrSpan-500-22,936,987,823	**27,556,403,070**	**27,556,403,070**	**27,556,403,070**	**27,556,403,070**	27,130,936,126
strCorrSpan-2,000-17,052,969,836	**32,783,499,328**	**32,783,499,328**	**32,783,499,328**	**32,783,499,328**	28,535,538,918
strCorrSpan-2,000-28,036,577,373	**40,146,187,862**	40,145,288,594	40,145,288,594	**40,146,187,862**	0
strCorrSpan-2,000-92,148,671,511	**110,409,094,321**	**110,409,094,321**	**110,409,094,321**	**110,409,094,321**	82,427,400,137
strCorrSpan-5,000-41,267,918,845	**81,368,124,258**	**81,368,124,258**	**81,368,124,258**	**81,368,124,258**	69,624,382,394
strCorrSpan-5,000-71,023,181,263	**100,217,640,112**	**100,217,640,112**	**100,217,640,112**	**100,217,640,112**	0
strCorrSpan-5,000-226,199,500,674	**271,781,871,491**	**271,781,871,491**	**271,781,871,491**	**271,781,871,491**	188,661,664,560
strCorrSpan-20,000-166,246,484,639	**324,652,926,326**	**324,652,926,326**	**324,652,926,326**	**324,652,926,326**	278,047,453,072
strCorrSpan-20,000-284,753,555,248	**401,750,506,702**	**401,750,506,702**	**401,750,506,702**	**401,750,506,702**	0
strCorrSpan-20,000-902,111,715,974	**1,083,393,761,865**	**1,083,393,761,865**	**1,083,393,761,865**	**1,083,393,761,865**	735,514,657,412
strCorrSpan-50,000-412,563,911,323	**809,149,037,830**	**809,149,037,830**	**809,149,037,830**	**809,149,037,830**	689,606,401,084
strCorrSpan-50,000-713,142,682,378	**1,005,094,669,280**	**1,005,094,669,280**	**1,005,094,669,280**	**1,005,094,669,280**	0
strCorrSpan-50,000-713,142,682,378	**2,704,243,765,007**	**2,704,243,765,007**	**2,704,243,765,007**	**2,704,243,765,007**	1,807,815,103,896
Avg.	**334,940,368,920.9**	334,940,326,098.6	334,940,326,098.6	**334,940,368,920.9**	187,232,935,624.5
p-value	REF. ALG.	0.5	0.5	-	4.768×10^{-7}

Table B24: Dataset B: comparison between different versions of MCTS where the modifications that we propose are introduced one by one

Problem instance	MCTS Version 1	MCTS Version 2	MCTS Version 3	MCTS Version 4	MCTS Version 5
Heuristic simulation policy	✓	✓	✓	✓	✗
Beam	✓	✓	✓	✗	✗
Domain reduction	✓	✓	✗	✗	✗
Pruning subtrees by calculating bounds	✓	✗	✗	✗	✗
exp-100-10,000,000,000	**9,999,478,296**	9,999,477,943	9,999,477,490	9,995,948,086	0
exp-125-10,000,000,000	9,999,479,807	9,999,478,408	9,999,478,953	**9,999,950,674**	0
exp-150-10,000,000,000	9,999,951,727	9,999,950,977	9,999,952,052	**9,999,952,695**	0
exp-175-10,000,000,000	9,999,953,940	9,999,952,702	9,999,952,805	**9,999,954,418**	0
exp-200-10,000,000,000	**9,999,956,638**	9,999,954,791	9,999,955,730	9,999,956,362	0
exp-225-10,000,000,000	**9,999,958,340**	9,999,955,338	9,999,956,529	9,999,958,206	0
exp-250-10,000,000,000	9,999,959,300	9,999,958,217	9,999,957,838	**9,999,959,339**	0
exp-275-10,000,000,000	9,999,960,122	9,999,954,732	9,999,955,590	**9,999,960,630**	0
exp-300-10,000,000,000	9,999,962,434	9,999,955,238	9,999,955,033	**9,999,962,552**	0
exp-325-10,000,000,000	9,999,963,242	9,999,956,790	9,999,956,260	**9,999,964,220**	0
exp-350-10,000,000,000	9,999,957,621	9,999,957,165	**9,999,957,848**	9,999,438,255	0
exp-375-10,000,000,000	**9,999,960,927**	9,999,956,985	9,999,958,320	9,999,439,701	0
exp-400-10,000,000,000	**9,999,967,171**	9,999,958,236	9,999,958,238	9,999,439,923	0
exp-425-10,000,000,000	**9,999,961,354**	9,999,956,638	9,999,956,386	9,999,438,907	0
exp-450-10,000,000,000	**9,999,960,321**	9,999,957,733	9,999,956,951	9,999,913,087	0
exp-475-10,000,000,000	9,999,957,931	**9,999,959,269**	9,999,956,930	9,999,913,823	0
exp-500-10,000,000,000	**9,999,959,782**	9,999,956,796	9,999,955,712	9,999,915,599	0
exp-525-10,000,000,000	**9,999,958,458**	9,999,957,062	9,999,957,229	9,999,917,757	0
exp-550-10,000,000,000	**9,999,958,398**	9,999,958,350	9,999,957,141	9,999,918,271	0
exp-575-10,000,000,000	**9,999,958,138**	9,999,957,132	9,999,957,725	9,999,919,510	0
exp-600-10,000,000,000	**9,999,957,600**	9,999,903,674	9,999,903,976	9,999,920,919	0
exp-625-10,000,000,000	9,999,958,763	9,999,904,395	**9,999,959,011**	9,999,922,069	0
exp-650-10,000,000,000	**9,999,958,972**	9,999,957,427	9,999,957,462	9,999,923,888	0
exp-675-10,000,000,000	9,999,959,169	**9,999,960,158**	9,999,959,740	9,999,926,379	0
exp-700-10,000,000,000	**9,999,959,501**	9,999,958,597	9,999,905,742	9,999,928,286	0
Avg.	**9,999,920,718.1**	9,999,914,190.1	9,999,914,267.6	9,999,697,742.2	0.0
p-value	REF. ALG.	5.037×10^{-6}	1.639×10^{-6}	7.264×10^{-4}	2.980×10^{-8}