Interpretability of rectangle packing solutions with Monte Carlo tree search

Yeray Galán López[1] · Cristian González García[1] · Vicente García Díaz[1] · Edward Rolando Núñez Valdez[1] · Alberto Gómez Gómez[2]

Abstract

Packing problems have been studied for a long time and have great applications in real-world scenarios. In recent times, with problems in the industrial world increasing in size, exact algorithms are often not a viable option and faster approaches are needed. We study Monte Carlo tree search, a random sampling algorithm that has gained great importance in literature in the last few years. We propose three approaches based on MCTS and its integration with metaheuristic algorithms or deep learning models to obtain approximated solutions to packing problems that are also interpretable by means of MCTS exploration and from which knowledge can be extracted. We focus on two-dimensional rectangle packing problems in our experimentation and use several well known benchmarks from literature to compare our solutions with existing approaches and offer a view on the potential uses for knowledge extraction from our method. We manage to match the quality of state-of-the-art methods, with improvements in time with respect to some of them and greater interpretability.

✉ Yeray Galán López
  uo257689@uniovi.es

  Cristian González García
  gonzalezcristian@uniovi.es

  Vicente García Díaz
  garciavicente@uniovi.es

  Edward Rolando Núñez Valdez
  nunezedward@uniovi.es

  Alberto Gómez Gómez
  albertogomez@uniovi.es

[1] Department of Computer Science, University of Oviedo, Leopoldo Calvo Sotelo, Oviedo 33007, Principality of Asturias, Spain

[2] Department of Business Administration, University of Oviedo, Pedro Puig Adam, Gijón 33204, Principality of Asturias, Spain

1 Introduction

The bin packing problem (BPP) is a combinatorial optimization problem that has been studied for decades. The widely approached one-dimensional bin packing problem, as described by Martello and Toth ([1990](#)), is formulated as follows: given n items, each of which is assigned a positive integer weight w_i, with $i \in I = \{1, \ldots, n\}$, and unlimited bins of maximum positive integer capacity c, assuming that $w_i < c$ for every i, find the packing for all the items in the minimum number of bins, so that the total weight of the items in each bin does not exceed the maximum capacity.

The problem can be generalized to higher dimensionality (Martello and Vigo [1998](#)). In two dimensions, the items are "cut" from the bins. The bins all have height H and width W, and each item $i \in I = \{1, \ldots, n\}$ has a height $h_i \leq H$ and a width $w_i \leq W$. The problem is enunciated as obtaining the cutting pattern that minimizes the number of required bins. In two dimensions, when a single bin has enough space for all the items, the problem is known as the rectangle packing problem (RPP), with the objective being minimizing the trim loss (i.e., the amount of space wasted) in the first bin. We study this problem, considering orthogonal rotations for the items.

However, no matter the variant of the problem, a wide range of industrial applications stem from obtaining the optimal way of placing a series of items inside containers, such as storage logistics (Martello et al [2000](#)), raw material processing (Hopper and Turton [1999](#)) or even resource allocation in cloud computing (Li et al [2015](#)). Despite the origin of the research dating from decades back, new applications and algorithms for this problem are still being studied as of today.

The regular BPP and its variants are known to be NP-hard problems (Lodi et al [1999](#)) (in case of the RPP, the problem of finding out whether there exists a solution with only one bin is itself an NP-hard problem), meaning that there is no knowledge of an exact algorithm that can solve it in polynomial time. Moreover, the size of real-world problems is in an ever-increasing state, which has oriented most of the research towards approximation algorithms that can obtain solutions in less time but cannot guarantee an optimal solution (Johnson [1973](#)) rather than exact algorithms.

We study Monte Carlo tree search (MCTS), which is a search tree algorithm that, in its basic form, samples random solutions and calculates their values over multiple iterations in order to approximate the best next action (Coulom [2006](#)). This algorithm, combined with machine learning, has reported phenomenal results, mainly on fields of artificial intelligence for games. Especially since Silver et al. managed to develop an algorithm that was able to beat the Go world champion (Silver et al [2016](#)), a great wave of research has emerged on the application of MCTS and machine learning to various different problems (Świechowski et al [2021](#)), not only to two-player board games, but also extended to a greater scope of problems provided that these can be modelled as a set of states and actions with a selection and objective function (Bellman [1957](#)).

However, we find that research on its application to optimization problems is still relatively scarce. In this work, we propose three different approaches that make use of the MCTS algorithm, standard MCTS in isolation and its integration with two methods: a deep learning approach, and a metaheuristic algorithm.

We perform the training of a deep learning structure via a reinforcement learning approach. This means that the system does not need any external training data, it starts from scratch and makes its own training data instead. Similarly, we propose a generic metaheuristic algorithm that does not require knowledge of the particular problem further than the evaluation function. We will use both of these approaches for attempting to improve the simulation of the MCTS algorithm.

In spite of the increasing interest in approximation and intelligent algorithms for NP-hard problems and in the interpretability of artificial intelligence separately, little research focuses on the application of interpretability methods to combinatorial optimization problems (Adadi and Berrada 2018). With this work, we aim to meet two objectives: obtaining solutions of great quality that can compete with or improve those of state-of-the-art algorithms and presenting interpretable solutions that show why the algorithm has chosen a certain way of packing items. We expect knowledge extracted from the algorithm to be very useful for guiding the exploration of heuristic algorithms and even for transmitting knowledge to humans, possibly helping in tasks like self-learning or the training of workers for a particular task.

The remainder of this work is structured as follows: Sect. 2 presents the background and related works. Section 3 introduces our proposal for obtaining solutions to the rectangle packing problem and details the different parts of the algorithm and the modelling of the problem. Section 4 shows the computational experiments that are carried out for the evaluation of our proposal. Lastly, Sect. 5 puts forward our reached conclusions and some ideas for future work.

2 Background

This section presents the background work that is related to the main different aspects of our study, packing problems, interpretability in artificial intelligence and Monte Carlo Tree Search.

2.1 Cutting and packing problems

The first known study of cutting and packing problems dates from many decades back, when Kantorovich first applied mathematical methods to industrial problems (Kantorovich 1960). The problem and its features receive different terminology throughout the years of literature. Dyckhoff attempts a typology unification in 1990 (Dyckhoff 1990), according to which, the multi-dimensional packing problem belongs to the field of geometric combinatorics in spatial dimensions, as opposed to the one-dimensional packing problem or knapsack problem, which is an abstract numeric problem. Dyckhoff offers a classification according to dimensionality, kind of assignment, assortment of large objects and assortment of small items. Years later, Wäscher et al (2007) pro-

pose a new revised typology with similar criteria but easier distinction in the kind of assignment and the assortment of small items.

One of the most studied variants is the BPP (Kantorovich 1960), classified as an input minimisation problem with identical large objects and strongly heterogeneous small items by Wäscher et al (2007). Other well known and widely studied variants are the multi-dimensional knapsack problem (KP), where items have value and the objective is maximizing the value in the packing, stemming from the classical numerical one-dimensional knapsack problem (Mathews 1896) or the strip-packing problem (SPP), which consists in finding the minimum height in which items can be packed in a bin of infinite height (Baker et al 1980). The cutting-stock problem (CSP) aims to minimize waste in the problem of cutting a set of items, in groups of identical items, from a large object, which is often equivalent to the BPP (Kantorovich 1960). The rectangle packing problem (RPP) fixes the dimensions of a bin and aims to minimize the wasted space after items have been placed inside (Huang et al 2007).

Variations to all these problems arise throughout the years of literature. Common ones include the consideration of orthogonal rotations of small items (Miyazawa and Wakabayashi 2004), variable-sized large objects (Friesen and Langston 1986), online item arrival in which not all items are available from the start (Ullman 1971) or the dynamic variation in which items also depart leaving empty space (Coffman et al 1983).

In this work, we study the two-dimensional RPP with orthogonal rotations of small items, without online item arrival or departure, since it is a problem with great applications in logistics and industrial problems and towards which a lot of benchmarks in literature for BPP can be adapted (Lodi et al 2002a; Iori et al 2022).

Originally proposed by Kantorovich (1960), the BPP is defined again by Martello and Toth (1990), who also propose a branch-and-bound algorithm as an exact algorithm to solve it. Later, they introduce a similar algorithm for the three-dimensional version of the same problem (Martello et al 2000). Throughout time, numerous studies have proposed new algorithms and improved lower bounds for this problem (Lodi et al 2002b; Christensen et al 2017). Approximation algorithms are proposed as an alternative to exact algorithms, presenting much faster but often non-optimal solutions (Johnson 1973). Lodi et al. (1999) evaluate existing heuristics and metaheuristics. Hopper and Turton (1999) a genetic algorithm for the two-dimensional packing problem. Later, a polynomial approximation algorithm is proposed for the two-dimensional BPP, CSP and SPP (Kenyon and Rémila 2000). More recently, Gonçalves and Resende (2013) create a genetic algorithm for solving two and three-dimensional bin packing problems. Blum and Schmid (2013) also apply evolutionary algorithms to the two-dimensional bin packing problem. Even in recent years, the problem is object of study for both modern approximation algorithms (Gomez and Terashima-Marín 2018; Laabadi et al 2020) and, to a lesser extent, exact algorithms, as reviewed by Iori et al (2021).

Popular instances that we study in this work are the ngcut instances (Beasley 1985b), the gcut instances (Beasley 1985a) and the cgcut instances (Christofides and Whitlock 1977). Recent results on these instances studied as RPP instances are offered by Delorme et al (2017) who create the exact DIM algorithm based on decomposition and integer programming methods and attempt to solve them as CSP instances and rectangle packing instances with orthogonal rotations, managing to solve many of

them. Similarly, Martin et al (2020) study the cgcut and gcut problems with guillotine cuts and again succeed in a lot of them, running out of time and memory in others. Li et al (2021) propose a hybrid genetic algorithm that obtains competitive solutions in most instances and optimals in some of them. We will study these same instances in our computational experiments and compare our results with some of these latest approaches.

2.2 Explainable artificial intelligence

Artificial intelligence has always been somewhat linked to a desire for explainable and interpretable solutions, as even some of the first studies on intelligent programs were partially oriented towards the extraction of knowledge (McCarthy et al 1960). However, in the last decades, more "opaque" methods with great complexity like deep neural networks have achieved predictions of extremely high accuracy and thus have been getting more attention, sacrificing some of the interpretability of simpler methods.

Nevertheless, multiple techniques for reaching some interpretation of these complex systems have been object of research throughout literature (Guidotti et al 2018). Interpretability is most often desired in fields where failure and inaccuracy are unacceptable and human supervision is often necessary, such as medicine (Shortliffe 2012), military (Van Lent et al 2004) or law (Berk and Bleich 2013).

In the last few years, many studies focus on the interpretability of complex artificial intelligence methods under the term Explainable Artificial Intelligence (XAI) (Adadi and Berrada 2018; Heuillet et al 2021). In the scope of deep neural networks, being one of the most prominent approaches in the last decade of artificial intelligence, some of the most relevant methods for explaining solutions propose the teacher-student model, which makes a distillation of a complex model into a simpler one, and the decomposition of final predictions into additions of features (Staniak and Biecek 2019). Other approaches use tree structures like decision trees, which are inherently interpretable (Blanco-Justicia and Domingo-Ferrer 2019; Rai 2020).

However, one of the greatest challenges in explaining complex intelligent systems remains the balance between accuracy and interpretability (Gunning et al 2019). Some recent works discuss the necessary trade-off of solution quality for explainability in optimization problems (Laber and Murtinho 2021). In this study, we attempt to make use of the inherent interpretability of MCTS and the power of deep learning and metaheuristic approaches to achieve both sides, competitive and interpretable solutions for the rectangle packing problem, towards which we could find little research in this field.

2.3 Monte Carlo tree search

Monte Carlo tree search (Coulom 2006) is a search algorithm based on the Monte Carlo method (Metropolis and Ulam 1949), which was developed in 1949 for the approximation of intractable mathematical results in physics when the number of variables is too big for a full exploration of every possible outcome. The method consists in the

random sampling of a smaller size in order to obtain an accurate approximation of the result.

The application of Monte Carlo methods to tree search gradually gives birth to the standard MCTS algorithm, taking the idea of approximation by random simulation to guide the decisions in the tree and approximate the best solution to a problem. Like other heuristic algorithms, it is applied to various combinatorial optimization problems, like the Travelling Salesman Problem (Powley et al 2012), throughout the years.

It is not however until Silver et al (2018) manage to employ it to beat the Go world champion that MCTS starts receiving great attention in literature. From there on, a lot of research has appeared on the application of MCTS to different aspects of artificial intelligence for a wide range of games, from general modern video game playing (Sironi et al 2018) and competitive games (Świechowski et al 2018; Baier and Cowling 2018) to traditional games like dots and boxes (Cotarelo et al 2021) and systems that aim to teach humans how to play the game (Gonzalo-Cristóbal et al 2021).

Furthermore, new applications different from games have also been widely researched (Walędzik and Mańdziuk 2018; Resende and Ribeiro 2019; Senington et al 2021; Clary et al 2018; Best et al 2019; Dieb et al 2019; Kajita et al 2020; Silver et al 2016; Leblond et al 2021).

In the past few years, the combination of machine learning and Monte Carlo methods and its applications to combinatorial optimization problems has appeared in some research. Laterre et al (2018) create the R2 algorithm with MCTS and reinforcement learning and obtain better results than classical linear programming methods for bigger instances.

Pejic and van den Berg (2020) attempt to use MCTS to obtain solutions to the two-dimensional perfect (zero-waste) variant of the rectangle packing problem, getting fast but rarely optimal solutions. MCTS is applied to the online packing problem in Zhao et al (2020), combined with a neural network that obtains action probabilities.

However, we find that the research does not particularly study the BPP, SPP or RPP or does not explore the integration of MCTS with other methods for solving the problem. Furthermore, none of the research found in the field of packing problems deals with the interpretability of solutions that MCTS provides in comparison to other machine learning, metaheuristic or exact methods, which is why we attempt to present some research on these fields in our methods.

3 Methods

In this section we detail all the aspects of our proposal, including the modelling of the rectangle packing problem, the item placement method, the MCTS algorithm, the deep learning architecture and the metaheuristic algorithm.

3.1 Problem modelling

As our goal is applying Monte Carlo tree search to the packing problem, we need to find a suitable representation for the problem, similarly to a Markov Decision Process (MDP) (Bellman 1957). We take a set of states S, a set of actions A_s from each state $s \in S$, a selection function $P(s, a)$ for action $a \in A_s$ from state s, and an objective function $R(s)$ that obtains the value of state s. Then:

- Each state s represents the state of the problem at s, that is: for each item $i \in I = \{1, \ldots, n\}$, the position x_{si} marks the coordinates of the bottom left corner of the item as it is placed in the bin, provided that it has already been placed. We also define t_s as the number of times state s has been visited and q_s as the cumulative value throughout those visits, which we define shortly.
- Each action a represents a placement of an item in the bin, and therefore the update of its position x_{si}. Rotations of the item, that is, permutations of h_i and w_i are considered. Thus, if item i is placed in a rotation different from the original one, h_i and w_i will also be updated. Executing an action from a state s produces another state s'.
- The Upper Confidence bounds applied to Trees (UCT) function (Kocsis and Szepesvári 2006) as the selection function $P(s, a)$ between states by means of an action. This function is developed from the Upper Confidence bounds proposed by Auer et al. (2002), it balances exploration and exploitation in the tree search taking into account both the value of the states and the number of visits. We have empirically determined that balancing value and visits leads to a more efficient exploration than purely guiding the search by the objective value. Given a state s and an action a that represents a transition from state s resulting in state s', the selection function is shown in Eq. 1.

$$P(s, a) = \overline{q}_{s'} + \sqrt{\frac{2 ln(t_s)}{t_{s'}}} \tag{1}$$

Exploration and exploitation can be balanced by weighing each of the addends, the first one for exploitation and the second one for exploration. Common practice is extracting $\sqrt{2}$ from the exploration addend and experimenting with different values for increasing or decreasing exploration depending on the problem (Silver et al 2016). However, through experimentation we have found that the originally proposed $\sqrt{2}$ works best for balancing our problem (Kocsis and Szepesvári 2006). Thus, we use the selection function as described, so the chosen action from state s will be given by Eq. 2.

$$a = \underset{a \in A_s}{argmax}(P(s, a)) \tag{2}$$

- The fill rate of the bin in a state as the objective function R (substracting the trim loss from 1 in order to transform the problem into a maximization problem). Let B be the set of items that have already been placed up to state s. Then function R can be formalized as seen in Eq. 3.

$$R(s) = \sum_{i \in B} \frac{h_i w_i}{H W} \tag{3}$$

3.2 Monte Carlo tree search

Monte Carlo Tree Search (Coulom 2006) is a heuristic algorithm based on the Monte Carlo method (Metropolis and Ulam 1949). It takes the idea of approximation by random sampling and applies it to tree search, attempting to approximate the best solution by randomly obtaining a great number of solutions and evaluating them.

The tree is traversed by means of some function, called the selection function, which often attempts to balance exploration and exploitation, until a state that has not been expanded is reached. The state is expanded by the actions that stem from it and simulated by random sampling of actions until a stop condition is met (in optimization problems, a solution is reached). The evaluation of the simulation is obtained by the objective function and backpropagated through the tree until the root, updating the value of every state that was traversed.

The general idea is that, by sampling random solutions and evaluating them, the tree will eventually find the path that achieves the best result, or at least one that is close. However, for large search spaces, approximating a good solution in this way might require too many simulations. For this reason, the random simulations are often replaced by other forms of evaluation, like machine learning approaches for predicting the result of the simulation (Silver et al 2018). With this method, the MCTS algorithm serves as a way of balancing exploration and exploitation of solutions in the search space of the problem and obtain more reliable evaluations over the course of iterations. Inspired by the success of the integration of MCTS and deep learning in games, and by the success of evolutionary metaheuristic algorithms in combinatorial optimization problems, we propose three approaches: isolated MCTS, the integration of MCTS with our deep learning model, and the integration of MCTS with our metaheuristic algorithm.

The outline of our basic MCTS algorithm is shown in Algorithm 1. Each iteration consists of four phases:

- Selection: the tree is traversed from the root by means of the selection function, choosing the most promising action, that is, the action with greatest $P(s, a)$ as defined above until an unexpanded state is reached.
- Expansion: the viable actions from the selected state are obtained and new states are created from executing each placement over it, expanding the state. This is the step where constraints are checked: only actions that meet the overlapping and fitting requirements are obtained and expanded. This way, there will never be actions in the tree that produce incorrect states and lead to non-viable solutions.
- Simulation: random actions are performed until all the items are placed, then the objective value from the random solution is obtained. This random sampling of solutions is replaced by a prediction from the deep learning model or by executions of our metaheuristic algorithm in our three approaches respectively. Our deep learning model receives an input representing the state and obtains the expected
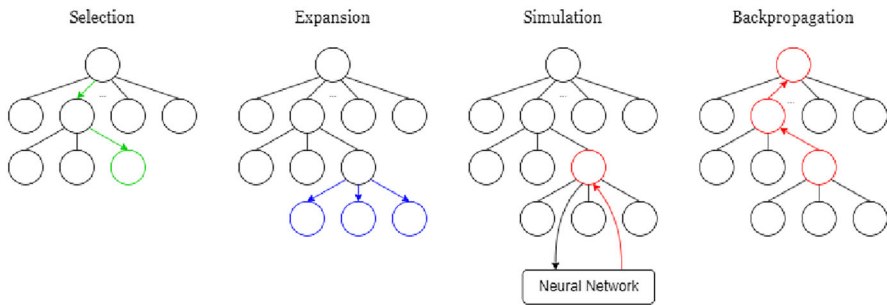
Fig. 1 Monte Carlo Tree Search with a deep learning model algorithm phases

final fill rate after placing all items. The metaheuristic algorithm receives the current placed item sequence and obtains a final sequence with its fill rate.
- Back-propagation: the value obtained in either method of the simulation phase is propagated from the selected state until the root state.

Figure 1 shows a visualization of these four phases with the deep learning approach.

Before the execution of the iterations as described, the root of the tree is initialized from a specific problem state by expanding all the viable actions from it. After that, the iterations are performed until the stop condition is met (a fixed number of iterations are reached or an optimal solution is reached), and lastly, the state with the highest average value among the children of the root state is chosen.

Thus, for each action to take (each item placement), the MCTS algorithm is executed. It will be executed as many times as items need to be placed in the specific problem. After every execution, the chosen state is taken as the root for the next iteration, reusing the subtree for the following iterations.

MCTS in isolation is a great algorithm for obtaining approximations to solutions with great time performance, but rarely the optimal ones. One of our main goals resides in obtaining interpretable solutions from which knowledge can be extracted (which items were placed first, which other actions were considered at each step, and so on), something that presents great difficulty in less transparent methods like deep learning models or standard metaheuristic algorithms in isolation. We expect the integration of those methods with a tree search algorithm like MCTS to produce solutions whose quality benefits from their complexity and learning capabilities while being transparent and explainable by means of MCTS' tree building.

3.3 Item placement method

For obtaining viable placements for items, we propose a "fitting space by dimension" method, consisting in the computation of all the free rectangular areas within the bin. Figure 2 shows an example of three free areas after the placement of two items.

The viability of placements is evaluated from the corner closest to the origin point (bottom-left corner) of the space. The origin of the item to place would then be placed in said corner. Thus, we only need to check whether each of the dimensions of the

Algorithm 1 MCTS algorithm

```
1:  procedure MCTS(n, r)                                          ▷ n iterations, r root state
2:      while i < n do
3:          s ← r
4:          while s has unexpanded children do                               ▷ Selection
5:              A ← actions from s
6:              a ← argmax(P(s, a))
                    a∈A
7:              s ← state from playing a
8:          end while
9:          S ← set of states from playing every a ∈ A                        ▷ Expansion
10:         s* ← s
11:         while ⋃_{B_j∈B_{s*}} ⊂ I do                                       ▷ Simulation
12:             a* ← random a ∈ A_{s*}
13:             s* ← state from playing a*
14:         end while
15:         q ← R(s*)
16:         while s ≠ r do                                                ▷ Backpropagation
17:             q_s ← q_s + q
18:             t_s ← t_s + 1
19:             s ← parent of s
20:         end while
21:         i ← i + 1
22:     end while
23:     S ← children of r
24:     b ← argmax(q_s/v_s)
              s∈S
25:     return b
26: end procedure
```
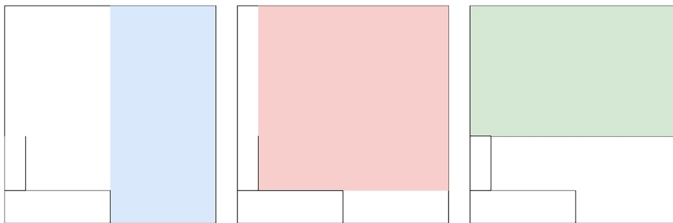

Fig. 2 Example of possible placement areas

item are less or equal than each of the dimensions of the space for the evaluation of whether an item is placeable in a space. The previously described MCTS algorithm evaluates all viable spaces and rotations as actions for each item.

After an item is placed in a space, the chosen space and the other spaces that collide with the newly placed item are updated. Depending on the relative position of the spaces, additional spaces might also need to be created. If we consider the configuration shown in Fig. 2, then Fig. 3 shows the updated spaces after placing a third item in the second space.

When only one possible placement needs to be obtained, as required for the objective function evaluation in the metaheuristic algorithm, the spaces are ordered by dimension, the height being the first dimension to be considered. Therefore, for a
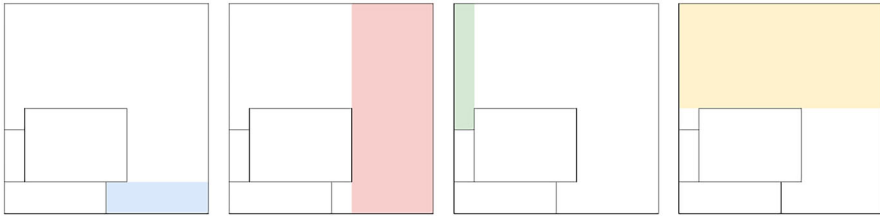
Fig. 3 Example of possible placement areas after placing an item

given sequence, the items will be placed in the left-most viable space of the lowest height spaces.

3.4 Deep learning model

We train a neural network model that attempts to predict the final fill rate of the problem instance after all items are packed. The input of the network models a problem instance by means of the following data: number of placed items, number of unplaced items, average unplaced item size, greatest unplaced item size, greatest unplaced item dimension, squarity of current packing and current fill rate.

The size refers to the product of height and width of an item or the bin. The dimension, on the other hand, refers to each individual dimension (i.e., the height and the width). The greatest unplaced item dimension is, therefore, the greatest individual dimension among all unplaced items. This measure provides information about how "long", at most, remaining items are.

We propose the squarity of current packing as a measure similar to the fill rate but more accurate for early stages of the problem where most items are still unplaced. This is used for informing the neural network of a fill rate that is independent of problem size and ratio of placed items. It is calculated as the fill rate of the smallest bin that could contain the items that are placed in the actual bin. The main intent is for this metric to be given greater importance when few items are placed (that is, when the smallest bin that could contain them is a lot smaller than the actual bin), while the current fill rate is given greater importance when most items are already placed, as the fill rate is very low and does not provide much information when only a small subset of items are placed, and they leave most of the bin empty.

Through experimental evaluation we have studied different structures and sizes for the deep learning network. We have determined that moderately low sized feed-forward layers obtain the best performance, as we need predictions to be both accurate and very fast. A "funnel" structure with descending layer sizes has reported the most accurate results. It consists on a simple model composed of three feed-forward layers, with 512, 128 and 32 neurons each respectively, which map the described input to a single sigmoid output. The output is meant to predict the final fill rate predicted by the model for the state of that instance.

We train our model with a reinforcement learning approach. Since the goal of our model is predicting the final fill rate of the problem, we use the real fill rate achieved

when all items are placed (obtained by our own MCTS algorithm) as the reward value for training. The training procedure is the same as solving a real problem via MCTS, with the exception that the chosen states for each item placement (i.e., at each tree depth) are recorded until all items are placed. When all items are placed and the solution is obtained, the actual final fill rate is computed and used as the reward output for training the model on each state with said output. The model is saved after each problem instance and used for the next one, while the MCTS algorithm, making use of the latest trained model, generates the data for the next training iteration, improving the quality of the predictions problem by problem until we consider training is finished.

We generate random problems from parameters resembling those of some popular benchmarks that we use for computational experiments (Christofides and Whitlock 1977; Beasley 1985a, b). This means that our approach does not rely on any external data for training and is able to train by itself for as many problems as needed. As the relations learned by the model do not depend on problem size (every input parameter is normalized), the transformations are applicable to any type of problem.

As the inputs are generated while executing the algorithm and thus, there is never a set of inputs to normalize, usual normalization is not possible. For overcoming this problem, we develop a method that allows inputs to be normalized within themselves, taking the bin data as the normalizing factor for similar item data. This normalization is done as shown in Eq. 4.

$$x_n = \frac{x}{F} \tag{4}$$

where x is the unnormalized input parameter and F is the corresponding bin parameter. Thus, F will be the bin size for normalizing sizes and bin greatest dimension for normalizing dimensions. This method ensures that not only is every input parameter between 0 and 1, but also that the model learns proportions. Normalizing inputs between different problems would result in, for example, a problem with bins of size 1000 having much bigger inputs that one of size 20. However, by using this method of normalization, our model will get "how big items are in comparison to the bins in the same problem" rather than "how big items are in comparison to the biggest items in a completely unrelated problem" as input parameters, which helps the model extrapolate knowledge to a wide range of different problems without having seen a similar problem ever before.

3.5 Metaheuristic algorithm

We propose an adaptive genetic algorithm (Gonçalves 2007) for a greater exploration of solutions in MCTS iterations. The genetic algorithm replaces the deep learning model in Fig. 1, that is, runs once every simulation phase in MCTS.

Our genetic algorithm implements adaptive crossover and mutation probabilities, making it evolve as the algorithm iterates. We adopt the method for calculation of evolving probabilities from the proposal by Li et al (2021). These probabilities are calculated as Eqs. 5 show.

$$P = \alpha P(t) + (1 - \alpha) P(f)$$

Fig. 4 Problem encoding with 5 items

4	2	5	1	3	0	0	1	0	1

Fig. 5 Crossover operators

4	2	5	1	3	0	0	1	0	1

2	1	5	3	4	1	1	0	1	1

1	2	5	3	4	0	1	0	1	1

4	1	5	2	3	1	0	1	0	1

$$P(t) = P_l + (P_u - P_l)\frac{1 - e^{-ct}}{1 + e^{-ct}}$$

$$P(f) = \begin{cases} P_u - (P_u - P_l)\frac{f - f_{avg}}{f_{max} - f_{avg}} & , f \geq f_{avg} \\ P_u, & , f < f_{avg} \end{cases} \tag{5}$$

In these equations α is a weight constant for balancing the importance of the number of iterations, with $\alpha = 0.6$ in practice. Value t is the number of iterations that have passed since the last improvement in fitness evaluation. Value f is the fitness of the best individual in the couple for the crossover operator and the fitness of the individual for the mutation operator. Values P_u and P_l are the constant upper and lower bounds of probability P respectively, with $P_u = 0.9$ and $P_l = 0.6$ for the crossover operator and $P_u = 0.5$, $P_l = 0.1$ for the mutation operator. Value c is a constant equal to 0.2 in practice. Finally, values f_{avg} and f_{max} are the average and maximum fitness values of the population respectively.

We encode the RPP as individuals with two different sequences: the order in which the items are to be packed and the rotation of each item. Thus, the sequence length is $2n$ for n items. The first half is a permutation of the integers from 1 to n, while the second half is a representation consisting in integers from 0 to $(2! - 1) = 1$. Figure 4 shows the representation of a possible solution for a simple problem with 5 items.

The items are packed in the order and rotation specified in the encoding. The viability of placing an item in the bin is determined by the closest fitting space method defined above. If the item cannot be placed in any of the spaces, the item will be skipped, negatively affecting the final fill rate.

We use the OX crossover operator, first proposed for the travelling salesman problem (TSP) and the exchange mutation operator for the first half of the sequence and 2-point crossover and bit-flip mutation for the second half (Srinivas and Patnaik 1994). Figure 5 and Fig. 6 show an example of these operators applied to an example of a small problem with 5 items.

Fig. 6 Mutation operators
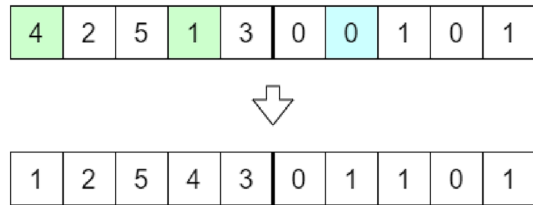
Selection of the new population is carried out in an elitist way by selecting the best individuals among the previous population and the newly generated children. We take the fill rate as the fitness function, as it is the function that we are aiming to maximize.

4 Experimentation

We conduct several computational experiments with known benchmarks from literature to evaluate the performance of our algorithms. After a first evaluation, we discuss the interpretability of the solutions obtained by our methods in contrast to less transparent methods like deep neural networks and attempt to extract knowledge from them. With the knowledge extracted from our methods, we repeat the experimentation with additional guidance for the solutions and see whether they can be improved on in terms of quality or speed.

4.1 Evaluation

In this subsection, we offer a comparison between our method and other known methods for solving the RPP. The metric used for comparison is the fill rate, equivalent to the more studied trim loss. We run ten executions for each problem instance and obtain the average and best fill rate of the obtained solutions.

The benchmarks used are the 12 ngcut instances and 13 gcut instances proposed by Beasley (Beasley 1985a, b) and the 3 cgcut instances proposed by Christofides and Whitlock (1977). Originally proposed for the 2-dimensional KP, throughout literature, these benchmarks have been studied for other problems, the most common ones being the SPP (Gonçalves 2007), and the BPP, including variants like the CSP (Delorme et al 2017). The instances have also seen several variations for the BPP like the RPP, and have been adapted so that items can in principle fit in a single bin (of minimum height), attempting then to minimize the wasted area or trim loss, equivalent to maximizing the filled area (Wei et al 2009). We study this latter variation in our experimentation. However, optimal solutions found are equivalent for the SPP as well, as the solution that minimizes the height will also present the greatest fill rate.

It should be noted that we study non-zero-waste instances, so the optimal fill rate is often lower than 1 since there is some space left even if all the items are placed in a single bin. Figure 7 shows a solution obtained for the instance ngcut3.

Table 1 shows the performance in terms of average and best fill rate (F%) and time. After weeks of training, our MCTS with deep learning approach could not manage

Fig. 7 Example of an optimal
solution found for ngcut3
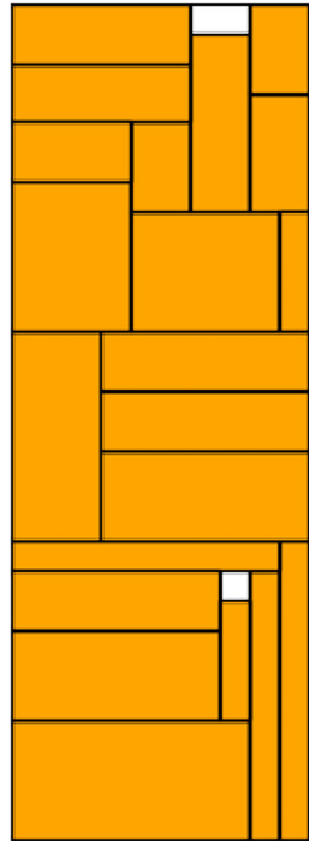
to reach solutions that are comparable to those of our other algorithms or to those of
recent literature. We believe that either a bigger input for the neural network model
that is able to capture more information about the problem or, mainly, more training,
could make the algorithm reach a point where predictions are reliable enough for it to
efficiently guide MCTS search and perform as well as the others. However, we could
not reach that point with our available time and computation power. We observe that
results are acceptable for easy instances, but when the number of items gets higher, it
struggles to find good solutions. The average solution and best solution has an average
of 0.0802 and 0.0543 lower fill rate than our genetic MCTS algorithm respectively,
which we consider a significant deterioration.

Therefore, we will focus on our other two approaches: standard MCTS and genetic
MCTS. As can be observed in Table 1, standard MCTS runs much faster but obtains
slightly worse solutions on most instances. Bold results represent a fill rate that matches
that of the optimal solution for the instance. The genetic MCTS algorithm manages
to find the optimal solution in 11 out of 12 ngcut instances, 1 out of 3 cgcut instances,
and 4 out of 13 gcut instances. Table 2 puts these results into perspective by showing a
comparison with other recent algorithms that deal with these instances (Delorme et al
2017; Li et al 2021).

Table 1 Fill rate and time by our three algorithms, standard MCTS, neural MCTS and genetic MCTS respectively

Instance	Optimal F%	MCTS			nMCTS			gMCTS		
		Avg F%	Best F%	t(s)	Avg F%	Best F%	t(s)	Avg F%	Best F%	t(s)
ngcut1	0.8261	**0.8261**	**0.8261**	0.01	**0.8261**	**0.8261**	242.97	**0.8261**	**0.8261**	0.03
ngcut2	0.9233	**0.9233**	**0.9233**	0.05	**0.9233**	**0.9233**	539.15	**0.9233**	**0.9233**	0.2
ngcut3	0.9893	0.985	**0.9893**	15.91	0.9454	0.9536	724.12	**0.9893**	**0.9893**	17.28
ngcut4	0.81	**0.81**	**0.81**	0.01	**0.81**	**0.81**	104.57	**0.81**	**0.81**	0.37
ngcut5	0.9805	**0.9805**	**0.9805**	0.35	0.9208	0.95	366.87	0.9775	**0.9805**	76.16
ngcut6	1	0.9945	1	9.86	0.9228	0.9448	452.53	0.9972	1	107.74
ngcut7	0.4375	**0.4375**	**0.4375**	0.01	**0.4375**	**0.4375**	170.72	**0.4375**	**0.4375**	0.01
ngcut8	0.9891	0.923	0.9328	12.18	0.9328	0.9328	402.31	0.9447	**0.9891**	324.67
ngcut9	0.9939	0.9432	**0.9939**	22.45	0.8894	0.9286	634.01	0.973	**0.9939**	469.38
ngcut10	0.7167	**0.7167**	**0.7167**	0.01	**0.7167**	**0.7167**	368.66	**0.7167**	**0.7167**	0.02
ngcut11	0.9887	0.9267	0.9427	16.96	0.9147	0.9147	478.93	0.9313	0.9427	715.83
ngcut12	0.8797	**0.8797**	**0.8797**	0.14	**0.8797**	**0.8797**	757.4	**0.8797**	**0.8797**	0.18
cgcut1	0.9783	**0.9783**	**0.9783**	6.08	0.9391	0.9391	533.4	**0.9783**	**0.9783**	6.04
cgcut2	0.985	0.8914	0.922	45.44	0.8284	0.9168	2638.67	0.9575	0.9644	1682.72
cgcut3	0.9996	0.8497	0.9417	496.16	0.8176	0.9291	814.16	0.9332	0.9393	28863.15
gcut1	0.6439	**0.6439**	**0.6439**	0.01	**0.6439**	**0.6439**	262.63	**0.6439**	**0.6439**	0.04
gcut2	0.9693	0.9212	0.9495	42.24	0.8949	0.9171	681.97	0.9387	0.9495	2170.56
gcut3	0.9044	**0.9044**	**0.9044**	21.46	0.8552	**0.9094**	1146.75	**0.9044**	**0.9044**	5.84
gcut4	0.9971	0.9467	0.9551	439.95	0.6333	0.6403	2127.31	0.9524	0.9618	31212.95
gcut5	0.9305	0.8668	0.8832	7.26	0.7893	0.8613	271.68	0.912	**0.9305**	269.44

Table 1 continued

Instance	Optimal F%	MCTS			nMCTS			gMCTS		
		Avg F%	Best F%	t(s)	Avg F%	Best F%	t(s)	Avg F%	Best F%	t(s)
gcut6	0.9802	0.9335	0.9481	38.98	0.8008	0.8061	698.64	0.9351	0.9481	2117.48
gcut7	0.8639	**0.8639**	**0.8639**	6.56	0.7826	0.8263	1146.4	**0.8639**	**0.8639**	1.23
gcut8	0.9839	0.9313	0.9438	433.73	0.6732	0.9102	2121.82	0.9427	0.9523	32820.59
gcut9	0.9999	0.8946	0.9421	7.46	0.8683	0.8768	258.8	0.9258	0.9421	275.36
gcut10	0.9999	0.9216	0.9451	41.46	0.7396	0.7976	715.91	0.9316	0.9445	2190.58
gcut11	0.9999	0.9258	0.9378	117.44	0.8843	0.9245	1144.6	0.9424	0.9548	7384.3
gcut12	0.9999	0.9351	0.9541	437.02	0.6376	0.6706	2119.38	0.9386	0.953	36620.42
gcut13	0.9999	0.9308	0.9494	133.85	0.8908	0.929	1264.26	0.9415	0.9533	7838.57

Table 2 Comparison by number of optimal solutions reached with gMCTS

Benchmark	# instances	Delorme et al.	Li et al.	Our gMCTS
ngcut	12	12	9	11
cgcut	3	1	2	1
gcut	13	3	2	4

Table 3 Comparison by number of optimal solutions and average time with standard MCTS

Benchmark	Delorme et al.		Li et al.		Our MCTS	
	# opt	**Avg t(s)**	**# opt**	**Avg t(s)**	**# opt**	**Avg t(s)**
ngcut	12	67.1	9	18.4	10	6.5
cgcut	1	734.6	2	99.6	1	182.6
gcut	3	742.7	2	108.4	3	132.9

It can be observed that our MCTS genetic algorithm manages to find better solutions for most instances. However, in some of them, it is at the cost of greater computational time.

If we compute the average difference between the solutions obtained by the standard and genetic variants of our algorithm, we obtain a 0.0122 average improvement on the average solutions and a 0.0056 average improvement on the best solutions in favour of the genetic MCTS algorithm. Therefore, in environments where solution quality is of maximum importance, the genetic MCTS algorithm would still be opted for. However, in other cases, the trade-off between solution quality and computational time could justify using the standard MCTS algorithm, since the decrease in fill rate is moderate, even negligible in some problems, and the increase in speed is greatly notable. Table 3 shows the comparison in number of optimal solutions reached and average time of our standard MCTS method and the previously mentioned two approaches (Delorme et al 2017; Li et al 2021).

We can observe a great improvement in time with respect to the decomposition-based algorithm by Delorme et al. with the trade-off of reaching two less optimal solutions in the ngcut instances. The genetic algorithm by Li et al. however, tends to run faster on bigger instances and slower on smaller instances, but reaches slightly worse solutions on most instances. Additionally, tree-based methods like our MCTS algorithm offer a far greater ease of interpretability of the obtained solutions which we will explore in more detail in the following subsection.

4.2 Interpretability of solutions

Aside from aiming to obtain solutions that make use of MCTS' guided exploration capabilities and genetic algorithms' gradual solution improvement, one of our main goals for using tree search is the obtention of interpretable and explainable solutions.

As such, we extract the tree that results after all items are placed in an instance, which contains the number of times each state has been visited, the average value
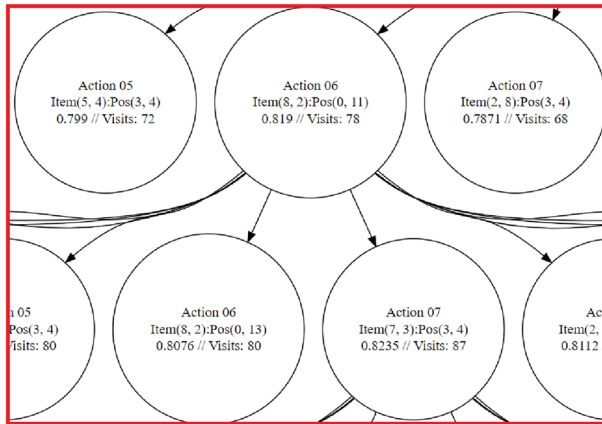
Fig. 8 Zoomed in part of the tree for ngcut1

achieved from each state across every visit, and the action that produced each state. With this information, we can discern how good each action was in comparison to other actions at the same tree depth, which other actions were considered, which paths produced the best results, and so on. That is, we can understand "why" and "how" our algorithm arrived at a particular solution.

Figure 8 shows part of the tree built by our algorithm for reaching an optimal solution instance ngcut1 as an example, where we can observe several states at two different depths of the tree. At each step of the solution, that is, at every decision for placing an item in a space of the bin, we can visualize which other actions were considered, the information about these actions (which item is placed in what position), the average value of the simulations and therefore, which actions were the best and the worst and how many simulations were performed for each actions (the "visit" number). We can also visualize the order in which the tree was developed during the algorithm execution, which items were placed first and which were placed last, and so on. From this information we can extract a great amount of valuable knowledge from our algorithm and discern "why" certain decisions led to an optimal solution, in contrast to non-transparent methods that, at best, only provide information about the final solution reached.

Making use of this information we conduct an additional study, this time on all the ngcut instances over the interpretability of solutions by representing data about the items in the order in which they are placed. Sizes and number of items vary from one instance to another, so we transform the data to fit in the same range so that we can study the tendencies in all the instances and obtain more representative data. That is, for an instance i with a number of items n_i, each point 1 to n_i is transformed by its product with $\frac{1}{n_i}$, so that every point for every instance is perfectly adjusted to the interval [0, 1]. On the other hand, every measure (size, greatest dimension...) is transformed by its product with $\frac{1}{max(m)}$, where $max(m)$ is the greatest measure among all the items in all instances, adjusting every measure to the interval [0, 1] as well.

With all the data, item order on the X-axis and feature measures on the Y-axis, we compute a linear regression and find the slopes of the adjusted lines obtained, which
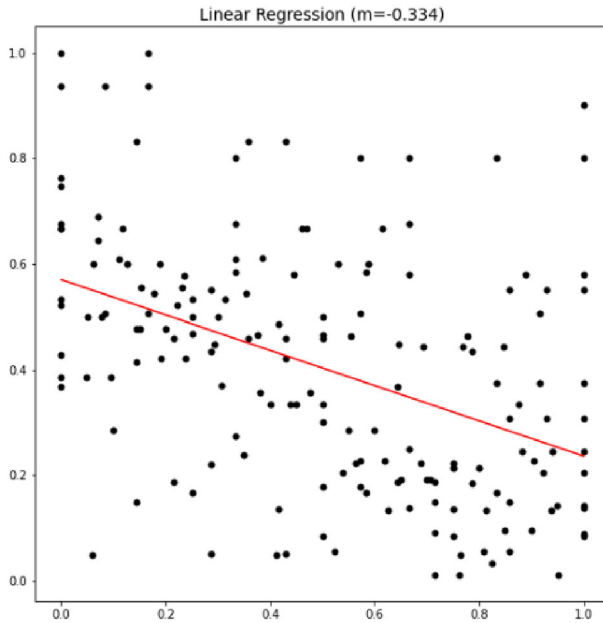
Fig. 9 Linear regression for the area feature

Table 4 Comparison of slopes by feature

Feature	Regression slope
Area	-0.3336
Greatest dimension	-0.0923
Diagonal	-0.1221
First dimension	-0.1621
Second dimension	-0.0925

will show what features of the items have the most influence in the order in which they are placed. Figure 9 shows the graph for the area of the items.

We can observe that the area of items is a very informative feature, as the value of the slope obtained in the linear regression is quite big. We compute the same for the greatest dimension, the diagonal, the first and second dimensions (we propose the features of ordered dimensions since the dimensions of the bins are not equal in these instances, so we imagine that the rotations of the items could be informative) and show the results on Table 4.

It can be observed that the area is the most informative feature in the order in which items are placed by a considerable margin. It should also be noted that every slope is negative, which means that there is a tendency for items to be placed in descending order of the features. This shows that in general, placing bigger sized items first leads to better solutions, with the area being the most reliable measure of size.

Table 5 Fill rate and time comparison of standard MCTS with and without additional knowledge

Instance	MCTS			MCTS (biggest first)		
	Avg F%	Best F%	t(s)	Avg F%	Best F%	t(s)
ngcut1	**0.8261**	**0.8261**	0.01	**0.8261**	**0.8261**	0.01
ngcut2	**0.9233**	**0.9233**	0.05	**0.9233**	**0.9233**	0.04
ngcut3	0.985	**0.9893**	15.91	**0.9893**	**0.9893**	8.38
ngcut4	**0.81**	**0.81**	0.01	**0.81**	**0.81**	0.01
ngcut5	**0.9805**	**0.9805**	0.35	**0.9805**	**0.9805**	0.13
ngcut6	0.9945	**1**	9.86	0.9945	**1**	7.84
ngcut7	**0.4375**	**0.4375**	0.01	**0.4375**	**0.4375**	0.01
ngcut8	0.923	0.9328	12.18	0.9084	0.9313	14.24
ngcut9	0.9432	**0.9939**	22.45	0.9523	0.9653	27.1
ngcut10	**0.7167**	**0.7167**	0.01	**0.7167**	**0.7167**	0.01
ngcut11	0.9267	0.9427	16.96	0.938	0.9427	17.4
ngcut12	**0.8797**	**0.8797**	0.14	**0.8797**	**0.8797**	0.08
cgcut1	**0.9783**	**0.9783**	6.08	**0.9783**	**0.9783**	0.92
cgcut2	0.8914	0.922	45.44	0.9354	0.9438	50.5
cgcut3	0.8497	0.9417	496.16	0.9675	0.9748	479.73
gcut1	**0.6439**	**0.6439**	0.01	**0.6439**	**0.6439**	0.01
gcut2	0.9212	0.9495	42.24	0.9302	0.9495	43.51
gcut3	**0.9044**	**0.9044**	21.46	**0.9044**	**0.9044**	10.74
gcut4	0.9467	0.9551	439.95	0.9525	0.9572	387.34
gcut5	0.8668	0.8832	7.26	0.8905	0.8914	7.91
gcut6	0.9335	0.9481	38.98	0.9338	0.9496	41.84
gcut7	**0.8639**	**0.8639**	6.56	**0.8639**	**0.8639**	2.77
gcut8	0.9313	0.9438	433.73	0.9396	0.9469	420.7
gcut9	0.8946	0.9421	7.46	0.8833	0.9421	8.35
gcut10	0.9216	0.9451	41.46	0.9283	0.9451	42.03
gcut11	0.9258	0.9378	117.44	0.9302	0.955	115.01
gcut12	0.9351	0.9541	437.02	0.9432	0.9494	411.05
gcut13	0.9308	0.9494	133.85	0.9348	0.9522	123.27

4.3 Evaluation with additional knowledge

We repeat the experimental study and try to obtain similar solutions on lower time by introducing knowledge extracted from the explained solutions into our algorithm, which will slightly guide the solutions towards placing items in the order of features previously obtained. Table 5 shows a comparison between our MCTS algorithm with and without the additional knowledge of prioritizing the placement of the three unplaced items with biggest areas first, with bold results representing optimal solutions.

We observe that times are, as expected, lower, since the MCTS algorithm needs to consider less actions per decision, especially in the early stages of the instance. As for the fill rate of the solutions, the results show that it slightly varies depending on

the instance in those where optimal solutions are not reached, but otherwise optimal solutions are mostly reached regardless. In some instances with a greater number of items, we find that the average fill rate even improves by guiding exploration towards placing the items with greater area first. In fact, if we compute the mean variation in fill rate between the average and best solutions obtained by both approaches, we obtain 0.0082 and 0.0019 respectively in favour of the guided approach, meaning that solutions with this extremely simple guidance are on average better. This is merely a simple case study, but we hope that these results serve as an example for the relevance of interpretable solutions and how extracting knowledge from solutions can lead to significant improvements in performance.

5 Conclusion

Monte Carlo Tree Search is a great method not only for finding very fast solutions to optimization problems, but also for generating a tree with information that can be analysed and interpreted.

In isolation, the algorithm can take just a few seconds to generally find great approximations. In addition to that, it is a method with great flexibility that can be integrated with other approaches like deep learning or metaheuristic algorithms to find solutions that are interpretable approximations of the optimal solution. Knowledge extracted from the solutions obtained by the method can be employed to find heuristics or guide the exploration of solutions in approximation algorithms, as we have shown during our experimentation, or analysed and conveyed to humans.

We have studied three different approaches: isolated MCTS, and its integration with deep learning and metaheuristic algorithms. We find that even without other methods substituting the simulation phase, MCTS can obtain near-optimal solutions for known problem instances in a matter of seconds. Improving the simulation phase with deep learning or a metaheuristic algorithm has reported a noticeable increase in computational time, but in the case of the metaheuristic approach, also a benefit in the quality of solutions obtained. Therefore, we can conclude that depending on whether speed or quality is more needed, one variant of our method would be a better fit than other.

We propose some of the following areas for future work. The study and development of a system that teaches people how to pack items in containers based on the solutions obtained by our MCTS method. The study and experimentation on multi-dimensional problems, especially three and four-dimensional problems, or other variants of packing problems. The evaluation of our method on bigger instances or real world-inspired problems. And lastly, a deeper analysis of hyperparameter adjustment in the deep learning model, and the consideration of a bigger deep learning model, possibly with variable-sized inputs, that can manage to find a better and more complex map between the current packing and the value of the solution.

Data availability The benchmarks studied in this work's experimentation are originally from various studies in cutting and packing literature: https://doi.org/10.1287/opre.33.1.49, https://doi.org/10.1057/jors.1985.51, https://doi.org/10.1287/opre.25.1.30. These are collected in https://doi.org/10.1007/s11590-021-01808-y and available, at the time of writing this paper, at their presented library.

Declarations

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this work.

Ethical approval The authors declare that they have no conflicts of interest, that this research does not involve human participants or animals, and that this work complies with the Ethical Rules of Journal of Heuristics.

References

Adadi, A., Berrada, M.: Peeking inside the black-box: a survey on explainable artificial intelligence (XAI). IEEE Access **6**, 52138–52160 (2018)

Auer, P.: Using confidence bounds for exploitation-exploration trade-offs. J. Mach. Learn. Res. **3**(Nov), 397–422 (2002)

Baier, H., Cowling, P.I.: Evolutionary MCTS for multi-action adversarial games. In: 2018 IEEE Conference on Computational Intelligence and Games (CIG), pp. 1–8. IEEE (2018)

Baker, B.S., Coffman, E.G., Jr., Rivest, R.L.: Orthogonal packings in two dimensions. SIAM J. Comput. **9**(4), 846–855 (1980)

Beasley, J.: Algorithms for unconstrained two-dimensional guillotine cutting. J. Oper. Res. Soc. **36**(4), 297–306 (1985)

Beasley, J.E.: An exact two-dimensional non-guillotine cutting tree search procedure. Oper. Res. **33**(1), 49–64 (1985)

Bellman, R.: A Markovian decision process. J. Math. Mech. **6**, 679–684 (1957)

Berk, R.A., Bleich, J.: Statistical procedures for forecasting criminal behavior: a comparative assessment. Criminol. Pub. Pol'y **12**, 513 (2013)

Best, G., Cliff, O.M., Patten, T., et al.: Dec-MCTS: decentralized planning for multi-robot active perception. Int. J. Robot. Res. **38**(2–3), 316–337 (2019)

Blanco-Justicia, A., Domingo-Ferrer, J.: Machine learning explainability through comprehensible decision trees. In: International Cross-Domain Conference for Machine Learning and Knowledge Extraction, pp. 15–26. Springer (2019)

Blum, C., Schmid, V.: Solving the 2d bin packing problem by means of a hybrid evolutionary algorithm. Procedia Comput. Sci. **18**, 899–908 (2013)

Christensen, H.I., Khan, A., Pokutta, S., et al.: Approximation and online algorithms for multidimensional bin packing: a survey. Comput. Sci. Rev. **24**, 63–79 (2017)

Christofides, N., Whitlock, C.: An algorithm for two-dimensional cutting problems. Oper. Res. **25**(1), 30–44 (1977)

Clary, P., Morais, P., Fern, A., et al: Monte-Carlo planning for agile legged locomotion. In: Twenty-Eighth International Conference on Automated Planning and Scheduling (2018)

Coffman, E.G., Jr., Garey, M.R., Johnson, D.S.: Dynamic bin packing. SIAM J. Comput. **12**(2), 227–258 (1983)

Cotarelo, A., García-Díaz, V., Núñez-Valdez, E.R., et al.: Improving Monte Carlo tree search with artificial neural networks without heuristics. Appl. Sci. **11**(5), 2056 (2021)

Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: International Conference on Computers and Games, pp. 72–83. Springer (2006)

Delorme, M., Iori, M., Martello, S.: Logic based benders' decomposition for orthogonal stock cutting problems. Comput. Oper. Res. **78**, 290–298 (2017)

Dieb, T.M., Ju, S., Shiomi, J., et al.: Monte Carlo tree search for materials design and discovery. MRS Commun. **9**(2), 532–536 (2019)

Dyckhoff, H.: A typology of cutting and packing problems. Eur. J. Oper. Res. **44**(2), 145–159 (1990)

Friesen, D.K., Langston, M.A.: Variable sized bin packing. SIAM J. Comput. **15**(1), 222–230 (1986)

Gomez, J.C., Terashima-Marín, H.: Evolutionary hyper-heuristics for tackling bi-objective 2d bin packing problems. Genet. Program. Evolvable Mach. **19**(1), 151–181 (2018)

Gonçalves, J.F.: A hybrid genetic algorithm-heuristic for a two-dimensional orthogonal packing problem. Eur. J. Oper. Res. **183**(3), 1212–1229 (2007)

Gonçalves, J.F., Resende, M.G.: A biased random key genetic algorithm for 2d and 3d bin packing problems. Int. J. Prod. Econ. **145**(2), 500–510 (2013)

Gonzalo-Cristóbal, V., Núñez-Valdez, E.R., García-Díaz, V., et al.: Monte Carlo tree search as a tool for self-learning and teaching people to play complete information board games. Electronics **10**(21), 2609 (2021)

Guidotti, R., Monreale, A., Ruggieri, S., et al.: A survey of methods for explaining black box models. ACM Comput. Surv. (CSUR) **51**(5), 1–42 (2018)

Gunning, D., Stefik, M., Choi, J., et al.: XAI-explainable artificial intelligence. Sci. Robot. **4**(37), eaay7120 (2019)

Heuillet, A., Couthouis, F., Díaz-Rodríguez, N.: Explainability in deep reinforcement learning. Knowl.-Based Syst. **214**(106), 685 (2021)

Hopper, E., Turton, B.: A genetic algorithm for a 2d industrial packing problem. Comput. Ind. Eng. **37**(1–2), 375–378 (1999)

Huang, W., Chen, D., Xu, R.: A new heuristic algorithm for rectangle packing. Comput. Oper. Res. **34**(11), 3270–3280 (2007)

Iori, M., de Lima, V.L., Martello, S., et al.: Exact solution techniques for two-dimensional cutting and packing. Eur. J. Oper. Res. **289**(2), 399–415 (2021)

Iori, M., de Lima, V.L., Martello, S., et al.: 2dpacklib: a two-dimensional cutting and packing library. Optim. Lett. **16**(2), 471–480 (2022)

Johnson, D.S.: Near-optimal bin packing algorithms. Ph.D. thesis, Massachusetts Institute of Technology (1973)

Kajita, S., Kinjo, T., Nishi, T.: Autonomous molecular design by Monte-Carlo tree search and rapid evaluations using molecular dynamics simulations. Commun. Phys. **3**(1), 1–11 (2020)

Kantorovich, L.V.: Mathematical methods of organizing and planning production. Manag. Sci. **6**(4), 366–422 (1960)

Kenyon, C., Rémila, E.: A near-optimal solution to a two-dimensional cutting stock problem. Math. Oper. Res. **25**(4), 645–656 (2000)

Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: European Conference on Machine Learning, pp. 282–293. Springer (2006)

Laabadi, S., Naimi, M., El Amri, H., et al.: A binary crow search algorithm for solving two-dimensional bin packing problem with fixed orientation. Procedia Comput. Sci. **167**, 809–818 (2020)

Laber, E.S., Murtinho, L.: On the price of explainability for some clustering problems. In: International Conference on Machine Learning, pp. 5915–5925. PMLR (2021)

Laterre, A., Fu, Y., Jabri, MK., et al.: Ranked reward: enabling self-play reinforcement learning for combinatorial optimization https://doi.org/10.48550/ARXIV.1807.01672 (2018)

Leblond, R., Alayrac, J.B., Sifre, L., et al.: Machine translation decoding beyond beam search (2021). https://doi.org/10.48550/ARXIV.2104.05336

Li, Y., Tang, X., Cai, W.: Dynamic bin packing for on-demand cloud resource allocation. IEEE Trans. Parallel Distrib. Syst. **27**(1), 157–170 (2015)

Li, Y.B., Sang, H.B., Xiong, X., et al.: An improved adaptive genetic algorithm for two-dimensional rectangular packing problem. Appl. Sci. **11**(1), 413 (2021)

Lodi, A., Martello, S., Vigo, D.: Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. INFORMS J. Comput. **11**(4), 345–357 (1999)

Lodi, A., Martello, S., Monaci, M.: Two-dimensional packing problems: a survey. Eur. J. Oper. Res. **141**(2), 241–252 (2002)

Lodi, A., Martello, S., Vigo, D.: Recent advances on two-dimensional bin packing problems. Discret. Appl. Math. **123**(1–3), 379–396 (2002)

Martello, S., Toth, P.: Bin-packing problem, pp. 221–245. Algorithms and computer implementations, Knapsack problems (1990)

Martello, S., Vigo, D.: Exact solution of the two-dimensional finite bin packing problem. Manag. Sci. **44**(3), 388–399 (1998)

Martello, S., Pisinger, D., Vigo, D.: The three-dimensional bin packing problem. Oper. Res. **48**(2), 256–267 (2000)

Martin, M., Birgin, E.G., Lobato, R.D., et al.: Models for the two-dimensional rectangular single large placement problem with guillotine cuts and constrained pattern. Int. Trans. Oper. Res. **27**(2), 767–793 (2020)

Mathews, G.B.: On the partition of numbers. Proc. Lond. Math. Soc. **1**(1), 486–490 (1896)

McCarthy, J., et al.: Programs with common sense. RLE and MIT computation center Cambridge, MA, USA (1960)

Metropolis, N., Ulam, S.: The Monte Carlo method. J. Am. Stat. Assoc. **44**(247), 335 (1949)

Miyazawa, F.K., Wakabayashi, Y.: Packing problems with orthogonal rotations. In: Latin American Symposium on Theoretical Informatics, pp. 359–368. Springer (2004)

Pejic, I., van den Berg, D.: Monte Carlo tree search on perfect rectangle packing problem instances. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, pp. 1697–1703 (2020)

Powley, EJ., Whitehouse, D., Cowling, P.I.: Monte Carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem. In: 2012 IEEE Conference on Computational Intelligence and Games (CIG), pp. 234–241. IEEE (2012)

Rai, A.: Explainable AI: from black box to glass box. J. Acad. Market. Sci. **48**(1), 137–141 (2020)

Resende, M.G., Ribeiro, C.C.: Greedy randomized adaptive search procedures: advances and extensions. In: Handbook of Metaheuristics, pp. 169–220. Springer (2019)

Świechowski, M., Godlewski, K., Sawicki, B., et al : Monte Carlo tree search: A review of recent modifications and applications. (2021) https://doi.org/10.48550/ARXIV.2103.04931

Senington, R., Schmidt, B., Syberfeldt, A.: Monte Carlo tree search for online decision making in smart industrial production. Comput. Ind. **128**(103), 433 (2021)

Shortliffe, E.: Computer-Based Medical Consultations: MYCIN, vol. 2. Elsevier (2012)

Silver, D., Huang, A., Maddison, C.J., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)

Silver, D., Hubert, T., Schrittwieser, J., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)

Sironi, CF., Liu, J., Perez-Liebana, D., et al: Self-adaptive MCTS for general video game playing. In: International Conference on the Applications of Evolutionary Computation, pp 358–375. Springer (2018)

Srinivas, M., Patnaik, L.M.: Genetic algorithms: a survey. Computer **27**(6), 17–26 (1994)

Staniak, M., Biecek, P.: Explanations of model predictions with live and breakDown packages. R J. **10**(2), 395 (2019). https://doi.org/10.32614/rj-2018-072

Świechowski, M., Tajmajer, T., Janusz, A.: Improving hearthstone AI by combining MCTS and supervised learning algorithms. In: 2018 IEEE Conference on Computational Intelligence and Games (CIG), pp. 1–8. IEEE (2018)

Ullman, J.: The performance of a memory allocation algorithm. Princeton University. Department of Electrical Engineering, Computer Science Laboratory (1971)

Van Lent, M., Fisher, W., Mancuso, M.: An explainable artificial intelligence system for small-unit tactical behavior. In: Proceedings of the national conference on artificial intelligence, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, pp 900–907 (2004)

Walędzik, K., Mańdziuk, J.: Applying hybrid Monte Carlo tree search methods to risk-aware project scheduling problem. Inf. Sci. **460**, 450–468 (2018)

Wäscher, G., Haußner, H., Schumann, H.: An improved typology of cutting and packing problems. Eur. J. Oper. Res. **183**(3), 1109–1130 (2007)

Wei, L., Zhang, D., Chen, Q.: A least wasted first heuristic algorithm for the rectangular packing problem. Comput. Oper. Res. **36**(5), 1608–1614 (2009)

Zhao, H., She, Q., Zhu, C., et al.: Online 3d bin packing with constrained deep reinforcement learning (2020) https://doi.org/10.48550/ARXIV.2006.14978

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.