Internship Report, Master MVA, 2017/2018

Self-play Reinforcement Learning for single-player games using Neural Networks and MCTS

Karl Hajjar[1]
karl.hajjar@polytechnique.edu

InstaDeep,[2] Ecole Polytechnique,[3] Ecole Normale Supérieure de Cachan[4]

Abstract

Monte Carlo tree search has recently had a substantial impact on a number of sequential decision-making problems. Based on the popular Upper Confidence Bound for Trees algorithm, different variants of Monte Carlo tree search have proven to perform particularly well in different kinds of games. Building on this progress, adversarial self-play in two-player games has delivered impressive results when using reinforcement learning algorithms that interleave deep neural networks and tree search. Algorithms like AlphaZero and Expert Iteration, which learn *tabula-rasa*, have been able to reach milestones in AI. However, the self-play training strategy is not directly applicable to single-player games. What is more, several practically important combinatorial optimization problems, such as the traveling salesman problem, the bin packing problem and the cutting-stock problem can be reformulated as reinforcement learning problems, increasing the importance of enabling the benefits of self-play beyond two-player games. We present our research which accomplishes this by ranking the rewards obtained by a single agent over multiple games to create a relative performance metric. Results from applying this method to instances of a two-dimensional bin packing problem show that it outperforms generic Monte Carlo tree search, heuristic algorithms and classic reinforcement learning algorithms. These results would also suggest that our method could produce approximate solutions to NP-hard problems of practical importance with large domain search spaces, potentially going beyond classical operations research algorithms which do not learn from their past experience.

[1]All the work was done while an intern in the AI research team at InstaDeep, and benefited from the help of Yun Guan Fu (y.fu@instadeep.com), Torbjorn Dahl (t.dahl@instadeep.com) and Alexandre Laterre (a.laterre@instadeep.com).

[2]2 Eastbourne Terrace, W6 LG, London, UK

[3]Route de Saclay, 91128, Palaiseau, France

[4]61 Avenue du Président Wilson, 94230 Cachan, France

Acknowledgements

First and foremost, I would specially like to thank the whole AI team at InstaDeep for welcoming me and for showing so much passion for AI related topics and sharing any new article or result in the field with great enthusiasm. I also wish to extend my gratitude to anyone at InstaDeep I have met, chatted, joked, laughed or teamed with.

I would like to address particular thanks to Yun Guan for helping me out in hours of darkness and for our regular discussions and exchange of ideas, and to Marta for her immense kindness and for being so lively, joyful and genuine every single day.

Finally, I thank Alain-Sam for sharing my passion for football and jokes, Karim for welcoming me warmly in his company and for his insatiable appetite for developing cutting-edge research, my desk neighbor Torbjorn for his light-hearted humor and easy-goingness and Alex for his rigor and dedication.

1 Introduction

This report aims to present and describe in depth all the work produced as an intern at InstaDeep. The scope of this work is primarily focused on self-learning systems which use *a combination of neural networks and tree search* to produce solutions to single-player problems. Needless to say that this represents a large range of methods since tree search itself has many variants, and there are numerous ways to use a neural network to improve tree searches.

The techniques used take great inspiration from DeepMind's *AlphaZero* algorithm [1], but also widely benefit from recent advances in Reinforcement Learning and Deep Learning. For instance, the benefits of adversarial self-play in multi-agent competition, described in [2] were instrumental in our attempt to reproduce adversity in a single-player setting and the architecture[1] used in OpenAI Five[2] for the Dota 2 competition helped a lot in designing network architectures, specially when it comes to representing actions.

This work presents the research conducted in order to produce novel methods for solving two well-known problems : the **2D bin packing problem**, and the **glass cutting** problem. The first problem deals with placing a given number items of given shapes into a fixed sized bin while leaving minimal unused space. The second problem is very similar but is a lot more constrained. In the glass cutting problem (a particular instance of the more generic stock cutting problem), the goal is to cut a certain number of items of given shapes out of plates of glass of fixed size while leaving minimal wasted space and satisfying a certain number of constraints. These two problems classically fall in the field of operations research and optimization, and are known to be **NP-hard**. Given the very similar formulation of both problems, our research focused on devising a general algorithm that could - with minor adjustments - be used for the bin packing and the glass cutting problems, and even extend to other types of single-player decision making problems.

I took part to both the work on the bin packing problem and the glass cutting problem, with a stronger focus on the latter since I was the only one working on this project, whereas the bin packing problem was a shared effort. My contribution to the bin packing project was more on the modeling side than the implementation, suggesting new ideas and discussing potential improvements and variations of the approach. On the other hand, I took on the glass cutting problem from scratch and implemented the environment, the agent, and the network myself, and the choices for the algorithm used resulted from a joint research and discussions with Yun Guan Fu, Torbjorn Dahl and Alexandre Laterre to design a general adversarial approach for single-player games using neural networks and tree search.

The work on the bin packing problem resulted in a significant achievement on instances of the problem with 10 items. Learning from scratch, our approach was able to produce solutions which were optimal more than 70% of the time, and proved to achieve higher average reward than plain MCTS, the Lego heuristic [3] and other reinforcement learning algorithms. A first version [4] of our work on the 2D bin packing problem was submitted

[1]A sketch of the architecture used is available here : `https://s3-us-west-2.amazonaws.com/openai-assets/dota_benchmark_results/network_diagram_08_06_2018.pdf`

[2]https://blog.openai.com/openai-five/

at NIPS[3]. Although some changes have been made since then, much of the work presented here will use content from this article.

Our focus on the glass cutting problem resulted from the fact that the French Operational Research and Decision Support Society[4] (ROADEF), jointly with the European Operational Research Society[5] (EURO), organized a challenge dedicated to a cutting optimization problem in collaboration with Saint-Gobain[6]. This competition is open to anyone from any background and aims at encouraging individuals or companies to suggest new approaches to tackle the glass cutting problem. Given the nature of the challenge, the instances of the problem represent real world problems at industrial scale with heavy production constraints, and thus represent a much more complex setting than the 2D bin packing problem with a limited number of items. Given the scale of the problem and the fact that Monte Carlo tree search requires playing a very large number of games during simulations, we chose to implement the environment of the glass cutting problem using the Go programming language[7] to ensure efficient episode generation. The neural network was implemented in Python and a Python process controlled the whole training pipeline and thus the communication between Go and Python.

Being able to produce high quality solutions to this kind of problem is the natural extension of our work on the 2D bin packing and would definitely validate our approach as a novel way of efficiently solving these kinds of problem and might even extend to a number of optimization problems which can be represented as single-player games.

While we have applied our algorithm to the 2D bin packing and the glass cutting problems only, our research aims at developing a general algorithm using neural networks and tree search that could potentially be applicable to any single-player game.

Section 2 summarizes the background our work is based on, section 3 presents related work, sections 4 and 5 present our approach to tackling respectively the bin packing and the glass cutting problems.

2 Background

2.1 Markov Decision Processes

In this section, we introduce the notations we will use throughout this report and the first principles on which our work is based. In doing so, we will make extensive use of Sutton and Barto's book [5] on reinforcement learning and use the same formalism as them.

2.1.1 The agent-environment interface

The framework we consider here is that of an *agent* interacting with an *environment*. The environment corresponds to everything outside of the agent, i.e. the context in which the

[3]the corresponding article is available on arXiv : https://arxiv.org/pdf/1807.01672.pdf
[4]http://www.roadef.org/societe-francaise-recherche-operationnelle-aide-decision
[5]https://www.euro-online.org/web/pages/1/home
[6]Saint-Gobain is a French industrial company https://www.saint-gobain.com/fr
[7]the Go programming language was created by Google in 2009

agent evolves. The goal of the agent is to maximize a *reward signal* which is provided by the environment. The agent can change the state of the environment by making decisions sequentially. Each time it takes an action, the environment is modified, and it observes a reward signal resulting from its action.

In all that follows we consider finite, fully-observable Markov Decision Processes (MDP) (after each action, the agent can observe the new state of the environment and the reward) where both the number of states and the number of possible actions are finite. Formally, an MDP is defined by :

- the set $\mathcal{S}$ of all possible states of the environment
- the set $\mathcal{A}$ of all possible actions
- the set $\mathcal{R}$ of all possible rewards
- a transition probability distribution p

We consider the setting where $\mathcal{S}$ and $\mathcal{A}$ are finite, $\mathcal{R} \subset \mathbb{R}$. For each state $s \in \mathcal{S}$, only a subset of actions are legal, and we denote by $\mathcal{A}(s)$ this subset, such that $\mathcal{A} = \bigcup\limits_{s \in \mathcal{S}} \mathcal{A}(s)$.
At each time step $t = 0, 1, 2 \dots$ the agent observes the state $S_t \in \mathcal{S}$ of the environment and takes an action $A_t \in \mathcal{A}(S_t)$, and observes the new state S_{t+1} of the environment, and the reward R_{t+1}. We consider the setting where the state, action and reward variables are random variables. The stochasticity comes both from the environment (selecting twice the same action in a given state may result in 2 different pairs of next state and reward) and the agent (the choice of an action can be stochastic and not deterministic). We will use capital letters to denote random variables and small letters to denote their possible values or other non stochastic variables.

The transition probability distribution p describes the dynamics of the environment :

$$p(s', r|s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a) \tag{2.1}$$

It gives the probability of observing the pair next state - reward (s', r) after selecting action a in state s. This transition probability distribution is inherent to the environment and is not affected by the choices of the agent, which, most of the time, does not have access to those dynamics through the complete distribution p, but only through observations of sequences of state and reward pairs. By giving the definition above, we implicitly assume the environment to be stationary : these transition probabilities do not depend on the time step t and are thus always the same.

The interaction between the agent and the environment results in a sequence of state, action, reward triplets : $S_0, A_0, S_1, R_1, A_1, S_2, R_2, A_2, \dots$ Such a sequence depends on the actions taken by the agent, which we represent by a *policy* π mapping states to distributions over actions :

$$\pi(a|s) = \mathbb{P}(A_t = a | S_t = s) \tag{2.2}$$

Again, this definition implicitly assumes the policy of the agent to be stationary. Depending on the environment and the choices of the agent, the sequence of (state, action, reward) triplets can be finite or infinite. We call this sequence an episode or a game (depending on the context). In the finite case, the environment has special states called terminal states which define the end of an episode. A state s_T is terminal if no legal action is available in this state, and thus the game or the episode ends. The finite case can always be considered as an instance of the infinite setting where, upon reaching a terminal state s_T, any subsequent action will result in no change in the environment and the rewards observed will always be $0 : \forall\, t \geq T+1,\ S_t = s_T,\ R_t = 0$.

2.1.2 Returns

Informally, the goal of the agent is to select actions, i.e. to choose a policy π in order to maximize the expected sum of the rewards observed during an episode. This sum is called the return G and is defined as follows :

$$G_0 = R_1 + R_2 + R_3 + \dots$$

In fact, the return can be defined from any time step, as the accumulated sum of rewards starting from a certain state S_t at time t :

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots$$

In the case of infinite episodes however, this sum might not converge, and we therefore consider a discounted version of the return where each reward observed is weighted by a factor decaying exponentially over time. Provided that the set $\mathcal{R}$ of possible rewards is bounded, and given a discount factor $\gamma \in [0, 1]$, the total discounted return of an episode starting from time t is :

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots\ = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

In the case of finite episodes, we do not need to discount the rewards since the sum will always be finite. However, the above formulation of the return still holds when setting $\gamma = 1$ since the rewards after the terminal state are all 0. In the infinite case, γ must be < 1 to ensure convergence.

Let us note here that the expected value of the return always depends on the transition dynamics p of the environment (over which the agent has no control) and, more importantly, on the choice of the agent policy π, since, as we already mentioned, the observations of an episode are themselves policy dependent. We will thus denote the expected return as $\mathbb{E}_\pi[G_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right]$.

2.2 Value functions

Now that we have formalized the setting in which we develop our work, we can state a few first basic principles of reinforcement learning. We will start by defining two different, but very similar, types of value functions which aim to evaluate how good or bad a state or an action is, given a certain policy.

First we define the *state value function* or more simply the *value function* of a policy π which gives the value of a state $s \in \mathcal{S}$ when following the policy π :

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[G_0|S_0 = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \tag{2.4}$$

The value function thus gives, for any state s, the expected return when starting an episode from state s and following policy π to select actions to perform at each step throughout the episode.

Similarly, we define the *action-state value function* or more simply the *action value function* of a policy π as being, for any state $s \in \mathcal{S}$ and any legal action $a \in \mathcal{A}(s)$, the expected return when starting in state s, taking a as first action and following policy π thereafter :

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \tag{2.5}$$

Given these new definitions we can formalize the goal of the agent as to find an *optimal policy* which maximizes the value of every state, i.e. the expected return starting in any state : the agent seeks to find a policy π^*, if there is any, such that, for any given policy π :

$$\forall s \in \mathcal{S},\ v_{\pi^*}(s) \geq v_\pi(s)$$

Which is to say that an optimal policy, if one exists, has a higher expected return in any state than any other policy. If such a policy exists, it is easy to see that it must also satisfy:

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s),\ q_{\pi^*}(s, a) \geq q_\pi(s, a)$$

It therefore follows that any optimal policy for v is also optimal for q and the converse is also true, such that v and q have the same optimal function(s).

Now we define the optimal state value function v_* and action value function q_* as being the maximal values obtained for any policy, when fixing the arguments of the functions :

$$v_*(s) = \max_\pi\ v_\pi(s) \tag{2.6}$$

$$q_*(s, a) = \max_\pi\ q_\pi(s, a) \tag{2.7}$$

With these definitions, we easily get that any policy π is optimal if and only if :
$\forall s \in \mathcal{S},\ v_\pi(s) = v_*(s)$, or, equivalently, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s),\ q_\pi(s, a) = q_*(s, a)$.

In other words, an agent seeking to find an optimal policy can either try to maximize the value function v or the action value function q. In fact, these two functions are in close relation as shows the following equality, true for any $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \tag{2.8}$$

In the finite setting which we consider in this work, there is always at least one optimal policy π_* and though there might be more than one, those share the same state value and action value functions.

2.3 Policy improvement and policy iteration

Here we discuss methods by which a current policy π can be made better by changing the decisions made in order to obtain a new policy π' such that $\forall s \in \mathcal{S},\ v_{\pi'}(s) \geq v_{\pi}(s)$. These type of methods are known as *policy improvement* methods.

In the tabular case, i.e. the case where the number of states is sufficiently small to be explored fully many times, and when a model for the transitions probability distribution p is provided, the policy evaluation algorithm (see more details in section 4.1 of [5]) provides an estimation, to a certain degree of error, of the value of each state for a given policy π. Knowing that, a simple method for providing an improvement over a current policy π is to select, for every state s, the action greedily with respect to the action value function to obtain a new, *deterministic policy* π' :

$$\pi'(a|s) = \begin{cases} 1 & \text{if } a = \underset{a'}{\text{argmax}}\ q_{\pi}(s, a') \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

It is easy to verify that this new policy π' satisfies $v_{\pi'}(s) \geq v_{\pi'}(s)$ for any s. This operation is however only possible if the action value function q_{π} of the current policy π is known. Nevertheless, in the case where p is known, given the fact that the policy evaluation gives us access to v_{π}, the expectation in 2.8 can be computed and the values of q are known. Thus, in this simplistic case where p is known and the number of states is small enough so that the value of the policy can be accurately estimated (to a small error margin) for any state via a simple algorithm, a policy improvement method can easily be provided. Starting from a random policy, iterating on such an improvement would eventually lead to an optimal policy. This kind of iterative algorithm which provides an improvement on a current policy at each step is called a *policy iteration* algorithm.

Traditional policy iteration algorithms usually work by making a certain number of iterations, each consisting of two steps : policy evaluation and policy improvement. The policy evaluation step consists in evaluating the value of a policy, either by applying an iterative algorithm that evaluates the value of every single state (when the number of possible states is not too large), or by estimating this value by averaging the results of Monte Carlo roll-outs, or by learning a function which approximates this value. The policy improvement step is done using equation 2.9 where $q_{\pi(s,a')}$ is replaced by its estimate computed in the evaluation step.

Outside the tabular case, it is too computationally expensive to loop over all possible states a great number of times, and other methods such as Monte Carlo estimation allows for similar policy improvement methods. In this type of method, the action value function q_{π} is approximated by another function Q by generating episodes using a current policy π and by updating the approximation Q via Monte-Carlo averaging over returns observed for each state action pair visited. The current policy is in turned improved at the end of all episodes by selecting actions greedily with respect to the approximate action value function Q.

When the number of possible states or actions is too large, these Monte-Carlo methods are not efficient because of the exploration problem (in Monte-Carlo methods, the approximate value $Q(s, a)$ of q_{π} needs to visit many times the state action pair (s, a) in order to be accurate), and thus learning an approximate state or action value function can provide a way to use policy improvement methods using the approximator as a proxy for the true value function. This is typically what is done in deep Q-learning where the policy used to generate episodes is ϵ-greedy policy to ensure exploration instead of a fully greedy policy with respect to the approximate action value function Q (see section 6.5 of [5] for more details).

2.4 Monte Carlo Tree Search

This section is dedicated to the presentation in depth of Monte Carlo tree search (MCTS) methods. These methods can be seen as a form of policy iteration, and use a tree to perform episode simulations before selecting actions taking into account averages of returns observed during simulation. Much of what is presented here is based on the survey of MCTS methods conducted by Browne *et. al* [6].

According to their definition, *"MCTS is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree corresponding to the results"*. MCTS is particularly effective on domains with large search spaces since it provides a way of cleverly exploring the domain by searching deeper (i.e. far ahead in terms of number of moves) in directions which look promising. It already has had a great impact on Artificial Intelligence (AI), particularly for games and planning problems.

2.4.1 Description of the basic principle of MCTS

Here we give an overview of what is Monte Carlo tree search by describing briefly its different components and the general strategy of the method. We consider, as in the rest of this work, the setting where the games end in a finite number of moves, i.e. a terminal state is always reached during an episode.

The basic MCTS process builds a tree starting from a root node in an incremental fashion. Each node of the tree represents a state of the game, and holds statistics concerning its children, namely visit-counts (how many times each children has been visited, each children corresponding to a different action) and accumulated value (that is, for each child, the total sum of returns observed at the end of the game when selecting that child). The tree is built while playing a game, and at each step of the game, a move is selected according to the current tree.

At each step of the game, the root node of the tree is taken to be the node representing the current state of the game (if anything comes before that node, it can be discarded). Then, a certain number of simulations are run from this root node. At the end of each simulation, the tree is grown, and at the end of all simulations an action is selected (i.e. a child is chosen from the root node) for the next move to play. Each simulation consists of four distinct phases:

(i) *tree traversing* : the current grown tree is traversed starting from the root node until a node which is not *expanded* (in a sense defined below). The policy driving child selection during the traversing phase is called the *tree policy*, and the visited counts of the nodes traversed are incremented.

(ii) *expanding* : the leaf node is *expanded* (its children are initialized in a sense to be defined later on)
(iii) *roll-out* : the value of the node to be expanded is estimated by playing a number of games from that node until the end using a *default policy*, and taking the average of the returns observed.
(iv) *back-up* : the value from the roll-out phase is *backed-up* (in a sense to be defined in later on) to all the nodes encountered during the simulation (traversing and roll-out phases) in order to update their statistics.

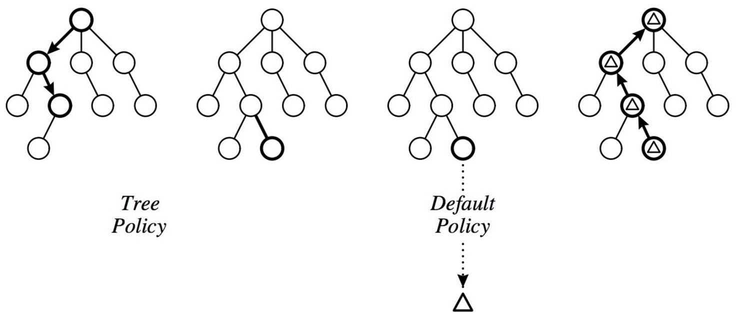


Figure 1: **MCTS simulations**. The different steps of a simulation in MCTS. From left to right : traversing, expanding, roll-out, back-up (Δ represents the value v to be backed-up).

The definition of being *expandable* (and thus of not being expanded) can vary from one implementation of MCTS to another. Here we take the basic definition of expandable to mean being a non terminal state and having (at least one) un-visited children. The different steps of a simulation in the MCTS are depicted in Figure 1. As shown, each simulation both grows the tree and updates the node statistics. At the end of all simulations, a child node is selected from the root in function of the root node statistics (e.g. most visited child, i.e. the child with the highest visit-counts is selected) and the next state of the game is the state corresponding to that node. The phase regrouping all the simulations is called the search phase.

Let us note here that, first each node in the search tree corresponds to a state of the domain and we will use both terms interchangeably. Secondly, each directed link to a child node represents a legal action leading to a subsequent state.

In MCTS, each node stores statistics on its children. More specifically, a node of the tree, representing state s of the game, holds two attributes for each legal action a in that state :
(i) $N(s,a)$: the number of times action a has been selected from state s
(ii) $W(s,a)$: the total cumulative sum of returns observed so far when taking action a in state s

Therefore, for each state s, $W(s,a)/N(s,a)$ forms a Monte Carlo estimate of the expected return from taking action a in state s.

For each simulation, during the traversing phase, the value of $N(s,a)$ is incremented for every state s encountered and action a selected at this state. At the end of a the simulation,

the value v obtained from the roll-out phase is backed-up to all the nodes encountered during the traversing phase, and the values of $W(s, a)$ are updated accordingly via
$W(s, a) = W(s, a) + v$.

Given what was presented above, it becomes clear why MCTS can be seen as a form of policy iteration : $W(s, a)/N(s, a)$ estimates the action value function q_π of the tree policy π for the pair (s, a), and selecting a move after having run a number of simulations in function of the statistics of the root node (visit counts of children and/or estimated value of each children) acts as the policy improvement step (the root selection policy is better than the tree policy because it makes a decision informed by the future results of following the tree policy from the root s_0).

2.4.2 Multi-Armed Bandits

Multi-arm bandits problems are sequential decision-making problems where a learner has to choose between K actions (or arms) at each step, and the player observes a reward a each step after selection an action. The goal is to maximize the total sum of rewards received after a certain number of steps. The connection to MDPs and MCTS is that, for each state s of a game, the problem of selecting an action can be seen as multi-armed bandit problem where there are as many arms as there are legal actions in sate s. This section does not intend to give an exhaustive overview of bandit problems, but simply to introduce what they are, as well as a few notations and basic algorithms. For more information on bandits, [7] is a very detailed and well structured resource from which we have used content in this section.

More formally, a bandit problem is defined by :
(i) a given number of arms K
(ii) a time horizon n
(iii) K probability distributions over the set of real numbers, of respective means $\mu_1, \mu_2, \dots \mu_K$.

The means are unknown to the player. At each step t, this one chooses an arm $A_t \in [\![1, K]\!]$ and observes a reward X_t drawn from the corresponding distribution of mean μ_{A_t}

Traditionally, in bandit problems, the objective is formulated not in terms of maximizing the expected value of a return $S_T = \sum_{t=1}^{T} X_t$, but rather in terms of minimizing a cost named the *regret*, defined as follows :

$$R_T = \mu^* T - \sum_{t=1}^{T} \mathbb{E}[X_t] \tag{2.10}$$

where $\mu^* \in \operatorname{argmax}_k \mu_k$.

This regret represents the loss incurred at each step by choosing a different arm than the one with maximal (unknown) mean. For a given bandit problem (given probability distributions of given means), minimizing the regret is of course equivalent to maximizing the expected return.

An obvious comment is that if the horizon is infinite, we can estimate with great precision each mean by using Monte Carlo average estimation, and then play only the arm of highest estimated mean. However, there is often a computational budget (time, memory, number of iterations, damage caused by choosing sub-optimal arms, etc) which prevents from doing this. The objective is thus to learn a policy to select arms in order to minimize the regret over a finite episode. A large number of algorithms exist to attempt to solve that problem, and each of those could be a potential way of selecting children in the tree policy of the MCTS. However, we will only present one of those here, which is among the most popular and very often used in MCTS methods in practice.

2.4.3 Upper Confidence Bound for Trees

One of the most famous bandit algorithms is the *Upper Confidence Bound* (UCB), and it has been adapted to MCTS, leading to an algorithm named *Upper Confidence Bound for Trees* (UCT). As previously stated, if the mean of each arm could be easily estimated with good precision, then the policy would be easy to design. However, estimating precisely the means usually implies having a very large time horizon T, and instead, what can be done is to estimate a value that will be an upper bound on the mean with good probability.

Using the confidence interval formula following from Hoeffding's inequality, the upper bound of UCT is given, for any arm k, at time step t by :

$$\text{UCB}_k(t) = \overline{X}_k^{(t)} + \sqrt{\frac{2\ln t}{n_k(t)}} \tag{2.11}$$

where $n_k(t)$ is the number of times arm k has been chosen over the first t time steps, and $\overline{X}_k^{(t)} = \frac{1}{n_k(t)} \sum_{s=1}^{t} \mathbb{1}_{\{A_s=k\}} X_s$ is the empirical mean reward of arm k at time t.

Given this upper bound, the UCB algorithm simply consists in choosing at each time step t the arm k with the highest upper bound :

$$A_t = \pi^{\text{UCB}}(t) = \underset{k\in[\![1,K]\!]}{\text{argmax}} \frac{1}{n_k(t)} \sum_{s=1}^{t} \mathbb{1}_{\{A_s=k\}} X_s \tag{2.12}$$

Let us point out that the second term in the UCB bound biases actions positively towards arms that have not been selected much so far, because of the the denominator $n_k(t)$. This second term is thus called an *exploration term.* This way, if an action has not been selected much, it is more likely to be selected, and actually selecting it would improve the current average estimate of the true mean reward for the corresponding arm. Therefore, the more an arm has been selected, the more the upper bound is actually close to the true mean, and the exploration term will encourage selecting arms that have rarely been chosen so far, and therefore improve the estimate of the corresponding mean.

Now that we have explained in what the UCB algorithm consists we can fully describe the UCT algorithm. MCTS works by trying to accurately estimate the true value of actions in a way that favors current most promising directions. In this respect, the choice of a tree policy is crucial for the performance of the algorithm. The UCT algorithm uses a slightly modified version of the UCB policy at each node of the tree as a tree policy to select actions during the traversing phase. Given the statistics $N(s,a)$ and $W(s,a)$ of a node s, the tree policy in UCT is given by :

$$\pi^{\text{UCT}}(a|s) = \frac{W(s,a)}{N(s,a)} + C_p\sqrt{\frac{2\ln N(s)}{N(s,a)}} \tag{2.13}$$

where $N(s) = \sum_{a\in\mathcal{A}(s)} N(s,a)$ is the number of times node s has been visited, and $C_p > 0$ is a constant parameter driving the strength of exploration. Let us note that this formula needs for each child of a node to be visited a least once. According to our definition a node is expandable only if all its children have been visited at least once. Therefore, since the tree policy selects actions only for nodes which have already been expanded, the term $N(s,a)$ in the UCT policy is never zero. However, we might want to change the definition of what is a expandable node, and take it, for instance, to mean that the node has never been visited before. In such a case, to ensure that division by zero is avoided in the UCT policy, a positive term is added to $N(s,a)$ in the denominators (e.g. a small ε or simply 1).

As discussed in the next section, the UCT formula can have a number of variants, and different values for C_p can result in very different performances. Here, we stick the formulation above. Now that we have detailed the tree policy used in UCT, we can describe the rest of the algorithm. The tree traversing phase ends on a not yet expanded node. From this node, a certain number of roll-outs are performed until terminal states are reached, but this number can be only one in the most basic version. The default policy used in the roll-outs is simply a random policy which selects children uniformly over legal actions. The complete UCT algorithm is fully described in Algorithm 1.

2.4.4 MCTS variants

In this section we discuss possible variations to the basic MCTS process. Changes can happen at different steps of the process : in the tree policy, in the way of expanding a node, in the selection of the best child at the end of the simulation phase, etc.

We start by examining changes in the tree policy. A variant of UCT which encourages more exploration is sometime used, and the child are selected during the traversing phase as :

$$\underset{a\in\mathcal{A}(s)}{\text{argmax}} \; \frac{W(s,a)}{N(s,a)} + C_p\frac{\sqrt{N(s)}}{1+N(s,a)} \tag{2.14}$$

This is just an example but the exploration term can be modified in many other ways. A third term can even be added to account for the variance of the returns at a given node.

The expansion of a node can also be dealt with in many different ways. As previously mentioned, we can consider that as soon as a node is selected once, it is expanded. None of its children has yet been visited and the statistics of its children are thus not initialized. To get around that, an idea is to use the value of the parent node to initialize the value of all its children. More precisely, let us consider a node s that needs to be expanded, and denote s^- its parent, a^- the action leading to s from s^-. We then take the initial value of each child a of s to be $W(s,a) = W(s^-,a^-)$. Then, when having to select a child from s for the first time, we can use any variant of the UCT formula (for example) where a positive term has been added to $N(s,a)$ to ensure non-zero division.

Algorithm 1: The UCT Algorithm.

```
Parameters : exploration parameter $c$, number of roll-outs $N_r$

Function UCTSearch($s_0$):
    while within computational budget do
        $s_l$ = TreePolicy($s_0$)
        $v$ = RollOut($s_l$)
        BackUp($s_l$, $v$)
    end
    return BestChild($v$, 0)

Function TreePolicy($s$):
    while $s$ is non terminal do
        if $s$ not fully expanded then
            return Expand($s$)
        else
            s = BestChild($s$)
        end
    end
    return $s$

Function Expand($s$):
    choose $a \in$ untried actions from $\mathcal{A}(s)$
    add a new child $s'$ to $s$
    return $s'$

Function BestChild($s$, $c$):
    return $\operatorname*{argmax}_{a \in \mathcal{A}(s)} \frac{W(s,a)}{N(s,a)} + c\sqrt{\frac{2 \ln N(s)}{N(s,a)}}$

Function RollOut($s$):
    $v = 0$
    for $it = 1 \ldots N_r$ do
        while $s$ is non terminal do
            choose $a \in \mathcal{A}(s)$ uniformly at random
            $s$ = new state after taking action $a$
        end
        $v = v$ + reward for state $s$
    end
    return $v/N_r$

Function BackUp($s$, $v$):
    while $s$ is not null do
        $N(s,a) = N(s,a) + 1$
        $Q(s,a) = Q(s,a) + v$
    end
```

Another variant consists in just setting all the $W(s,a)$ to zero, and simply adding 1 to $N(s,a)$ in the UCT formula (doing so ensures that the first time a child is selected from s, it is selected uniformly at random because all children have the same value).

Now we discuss child selection at the end of the simulation phase when having to take an action that will lead to the next state. A popular strategy is to chose the most visited child from the root node at the end of all simulations. That is, select from the root node s_0 the action a that has the highest $N(s_0,a)$. Another option would be to select the child with highest average value $W(s_0,a)/N(s_0,a)$. Yet another approach would be to make a "safe choice" by selecting a child which maximizes a lower confidence bound.

Among the variants of MCTS, some methods make more structural changes to the whole process. Again here, there are quite a few such variants but we will mainly focus on the one used in the *AlphaZero* algorithm in [1], which is of particular interest to us, and which we will discuss in detail in the section 2.5. In this algorithm, a neural network is combined with MCTS to help the search and provide valuable information on the look-ahead search performed in MCTS.

2.4.5 Hyper-parameters of MCTS

In this section we discuss the importance an flexibility of certain parameters of MCTS, such as the number of searches, the number of roll-outs, and exploration parameter C_p. All of these impact differently the way the tree search is done and thus have an impact on its outcome.

As an obvious first remark, it is clear that a greater number of roll-outs or searches will improve the performance. An increased number of roll-outs will mean more confidence that the value which is backed-up accurately estimates the value of the leaf node (the one which is being expanded), and therefore more precise estimation of action values for the nodes. Similarly, an increasing number of searches will mean both better value estimation and potentially more exploration.

The changes in performance due to changes of the parameter C_p are less easy to predict. It is obvious that $C_p = 0$ will probably leave out many interesting search directions while a very large C_p would definitely prevent from exploring deeper directions which appear promising. The optimal value of C_p is problem dependent and not much can be said in general other than it should not be “too small” or “too large”, where, of course, these two notions are very much game-dependent.

However, a more interesting question would be to know if the hyper-parameters can be changed during the the game to improve performance ? For instance, for a fixed computational budget (for example the computation time for playing an entire game with MCTS), is it useful to modify the number of searches or roll-outs as the game goes on and moves are made ? Indeed, towards the end of the game, the number of actions becomes more limited and there is less value in doing more searches and roll-outs, and the time spent on those might be more usefully spent at the beginning of the game, or for states where the number of possible actions is large. It could reveal useful, for example, to slowly increase the number of searches to reach a maximum towards the middle of the game where the number of moves is probably the highest, and then start slowly decreasing it until the end of the game. The same thing could be applied for the parameter C_p : performance might benefit from adjusting it at each node according to the number of children of that node.

To our knowledge, there are not much studies conducted to answer this question. We will however mention [8] which actually makes those parameters learnable, and thus allows for some flexibility in the parameters as the game is played, and as well allows for adaptation to game specific parameters.

To conclude this section on MCTS, let us state that it is a powerful method which can nevertheless be enhanced by adding a learning component to it which can use previous searches to inform future searches as well as help in the estimation of state values.

2.5 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning which deals with sequential decision-making systems, where an agent interacts with an environment and tries to maximize a scalar reward signal. Therefore, Markov decision processes are very well suited to formalize the reinforcement learning setting. This section will reuse notations and definitions from sections 2.1 and 2.2.

In reinforcement learning, an agent tries to learn a policy which maximizes the expected return of an episode. In order to learn this policy, an agent must be able to estimate the value of its actions, i.e. given the current state, what return can it expect to observe when selecting a given action. Most reinforcement learning algorithms (but not all, e.g. policy gradient algorithms) use a form of generalized policy iteration where first the agent learns to correctly estimate the value of its current policy (policy evaluation), and then, from that knowledge, it designs a new policy which is better than the previous one (policy improvement). As previously discussed, when the state domain of the MDP is prohibitively large to go through many times, learning methods can be employed in order to form estimates that can generalize across states (i.e. a good estimate of the value of a state -or a state-action pair- can be provided by learning from observations of similar states), instead of Monte Carlo methods.

RL methods are of various kinds and we will not detail all the different methods existing, but will simply state that most modern methods make use of neural networks to learn either a policy, the action value function, the state value function, or even a policy and one of the two value functions jointly. We very briefly summarize below a few of the most popular methods :

- *policy gradient* methods such as REINFORCE : a policy π_θ is parameterized with weights θ (those of a neural network for instance), and a gradient ascent method is used to maximize an empirical estimate of the expected return of the parameterized policy $\theta \leftarrow \theta + \eta \nabla_\theta \mathbb{E}_{\pi_\theta}[v_{\pi_\theta}(s_0)]$
- *actor-critic* methods : an actor learns a parameterized policy π_θ and a critic learns an estimate of the corresponding state value function v_{π_θ} through a parameterized function $\hat{v}_w$ with weights w
- *Q-learning methods* such as deep Q-learning : an estimator $\hat{q}_\theta(s, a)$ of the optimal action value function q^* is learned through gradient descent on the following loss $[\hat{q}_\theta(s_t, a_t) - r_t - \gamma \max_{a \in \mathcal{A}(s_t)} \ \hat{q}_\theta(s_t, a)]^2$, where episodes are generated using an ε-greedy policy, meaning that from state s_t action $\text{argmax}_{a \in \mathcal{A}(s_t)} \ \hat{q}_\theta(s_t, a)$, is selected with probability $1 - \varepsilon$, and the action is selected uniformly at random with probability ε.

Neural networks are often used for parameterizing policies or functions in those methods because of their ability to approximate complex functions and their power of representation. These networks learn from data generated using a policy to select actions at each time step, either by having the agent play alone in case of single player games, or sometimes in two-player games by having the agent play against itself, in which case this data-generation process is called *self-play*. If the policy learned is also the one used to generate the data, the method is called *on-policy* (e.g. SARSA) and it is called *off-policy* otherwise (e.g. Q-learning).

2.6 Self-play with Deep Learning and tree search

Policy iteration algorithms that combine deep neural networks and tree search in a self-training loop, such as AlphaZero [1] and Expert Iteration [9], have exceeded human performance on several two-player games. These algorithms are given only the rules of a game and train solely by self-play.

They use a neural network with weights θ to provide a policy $p_\theta(\cdot|s)$ and/or a state value estimate $v_\theta(s)$ for every state s of the game. The tree search uses the neural network's output to focus on moves with both high probabilities according to the policy and high-value estimates. The value function also removes any need for Monte Carlo roll-outs when evaluating leaf nodes. Therefore, using a neural network to guide the search reduces both the breadth and the depth of the searches required, leading to a significant speedup. The tree search, in turn, provides improved MCTS policies which the network can then learn from, thus guiding the search even better in the next round of MCTS.

The MCTS used is a variant of UCT where the exploration term is changed a little, and the back-up uses value estimates from the neural network instead of roll-outs as previously stated. More precisely, the tree policy used in *AlphaZero* is :

$$\pi^{\mathrm{AZ}}_{tree}(a|s) = \frac{W(s,a)}{N(s,a)} + p_\theta(a|s)\ C_p\ \frac{\sqrt{N(s)}}{1+N(s,a)} \tag{2.15}$$

When an unexpanded node is reached at the end of the traversing phase, its value is estimated not via roll-outs but simply using the value $v_\theta(s)$ from the neural network. This value is then backed-up to all nodes traversed. Finally, at the end of the simulation phase, the MCTS policy is :

$$\pi^{\mathrm{AZ}}_{\mathrm{MCTS}}(a|s_0) = \frac{N(s_0,a)^{1/\tau}}{\sum_b N(s_0,b)^{1/\tau}} \tag{2.16}$$

where τ is a temperature parameter ($\tau \to 0$ makes the policy deterministic and $\tau \to +\infty$ make the policy uniform). Moves are then sampled from this policy to obtain the next state of the game.

Self-play is used to generate data. The MCTS plays games against itself, and at the end of the game, the data for each state of the game from the perspective of both players is added to a dataset : for every state s of the game both triplets $(s, \pi^1_{\mathrm{MCTS}}(\cdot|s), z_1)$ and $(s, \pi^2_{\mathrm{MCTS}}(\cdot|s), z_2)$ are added to the dataset, where 1 denotes player one and 2 player two, and z_i is the outcome of the game from the perspective of player $i \in \{1,2\}$ (-1 for a loss, $+1$ for a win). After playing a certain number of games and adding new data, the network is trained to minimize the loss :

$$l = (z-v)^2 - \pi^T_{\mathrm{MCTS}} \log p + c||\theta||^2 \tag{2.17}$$

where $(p,v) = f_\theta(s)$ is the output of the network.

Self-play allows these algorithms to learn from the games played by both players. It also removes the need for potentially expensive training data, often produced by human experts. Such data may be biased towards human strategies, possibly away from better solutions. Another significant benefit of self-play is that an agent will always face an opponent with a similar performance level. This facilitates learning by providing the agent with just the right curriculum in order for it to keep improving [2]. If the opponent is too weak, anything the agent does will result in a win and it will not learn to get better. If the opponent is too strong, anything the agent does will result in a loss and it will never know what changes in its strategy could produce an improvement.

One of the main value of self-play is that it produces instances of games that persistently push the algorithm just beyond its current abilities, providing meaningful rewards based on the agent's *relative performance*. In that context, small changes in policy can affect the relative performance, enabling the agent to learn more easily what changes in its behavior will result in an improved performance. The main contribution of our research is in the design of a relative reward mechanism for single-player games, providing the benefits of self-play in single-player MDPs and potentially making policy iteration algorithms with deep neural networks and tree search effective on a range of combinatorial optimization problems.

3 Related Work

Combinatorial optimization problems are widely studied in computer science and mathematics. A large number of them belongs to the class of NP-hard problems. For this reason, they have traditionally been solved using heuristic methods [10, 11]. However, these approaches may need hand-crafted adaptations when applied to new problems because of their problem-specific nature.

Machine learning algorithms potentially offer an improvement on traditional optimization methods as they can help learn part of problem and provide valuable information to these methods, preventing them from restarting from scratch on each new instance. Particularly, deep learning methods might bring a lot of value since they have achieved remarkable results on classification and regression tasks [12]. Nevertheless, their application to combinatorial optimization is not straightforward. A particular challenge is how to represent these problems in ways that allow the deployment of deep learning techniques.

One way to overcome this challenge was introduced by Vinyals et al. [13] through *Pointer Networks*, a neural architecture representing combinatorial optimization problems as sequence-to-sequence learning problems. Early Pointer Networks were trained using supervised learning methods and yielded promising results on the Traveling Salesman Problem (TSP) but required datasets containing optimal solutions which can be expensive, or even impossible, to build. Using the same network architecture, but training with actor-critic methods, removed this requirement [14].

Unfortunately, the constraints inherent to the bin packing problem prohibit its representation as a sequence in the same way as the TSP. In order to get around this, Hu *et al.* [15] combined a heuristic approach with RL to solve a 3D version of the problem. The main role of the heuristic is to transform the output sequence produced by the RL algorithm into a feasible solution so that its reward signal can be computed. This technique outperformed previous well-designed heuristics.

While self-play algorithms have proven successful for two-player games, to the best of our knowledge, there has been little work on applying similar principles to single-player games. We will however cite [16], who attempts to extend the *AlphaZero* algorithm to the continuous action space setting, which contains several single-player problems. [17] also makes use of MCTS and neural networks to provide solutions to the retrosynthetic analysis of organic molecules which represents a single-player problem. However the policy networks learned are trained in a supervised fashion and do not learn from the MCTS policy, so that the whole process is very different in nature to our approach.

4 The Bin Packing Problem

The bin packing problem is a well-known NP-hard optimization problem where the goal is to place items in a fixed sized bin while minimizing the unused area. We present here our approach using MCTS and neural networks to solve this problem. Our method provides a way of ranking rewards observed in a way which reproduces the relative performance signal that an agent would experience when playing against himself self-play in two-player games. What is described below corresponds to what was published in [4]. Substantial changes have been made since then in different parts of our method (neural architecture, ranking mechanism, and reward formulation) and we will discuss those further in Section 5 when presenting our approach to solving the glass cutting problem.

4.1 Bin Packing as an MDP

The bin packing problem consists of a set of items to be packed into fixed-sized bins in a way that minimizes a cost function, e.g., the number of bins required. The work presented here considers an alternative version of the 2D bin packing problem. Like in the work of Hu *et al.* [15], this problem involves a set of N rectangular items $I = \{(w_i, h_i)\}_{i=1}^{N}$ where w_i and h_i denote the width and height of item i. Items can be rotated of $90°$ and $o_i \in \{0, 1\}$ denotes whether the i-th item is rotated or not. The bottom-left corner of an item i placed inside the bin is denoted by (x_i, y_i) with the bottom-left corner of the bin set to $(0, 0)$. The problem also includes additional constraints, complexifying the environment and reducing the number of available positions in which an item can be placed. In particular, items may not overlap and an item's center of gravity needs physical support. A solution to this problem is a sequence of triplets $((x_i, y_i, o_i))_{i=1}^{N}$ where all items are placed inside the bin while satisfying all the constraints. An example of how the solution is constructed is shown on Figure 2.

We formulate the problem as an MDP in which the state encodes the items and their current placement while the actions encode the possible positions and rotations of the unplaced items. The goal of the agent is to select actions in a way that minimizes the side of the minimal square bounding box, L. This is reflected in the terminal reward, r, after all items have been placed. As defined in Equation 4.1 and illustrated in Figure 3, all non-terminal states receive a reward of 0 while terminal states receive a reward which is a function of the side of the optimal bounding box L^*, the minimal square bounding box L, and the side of

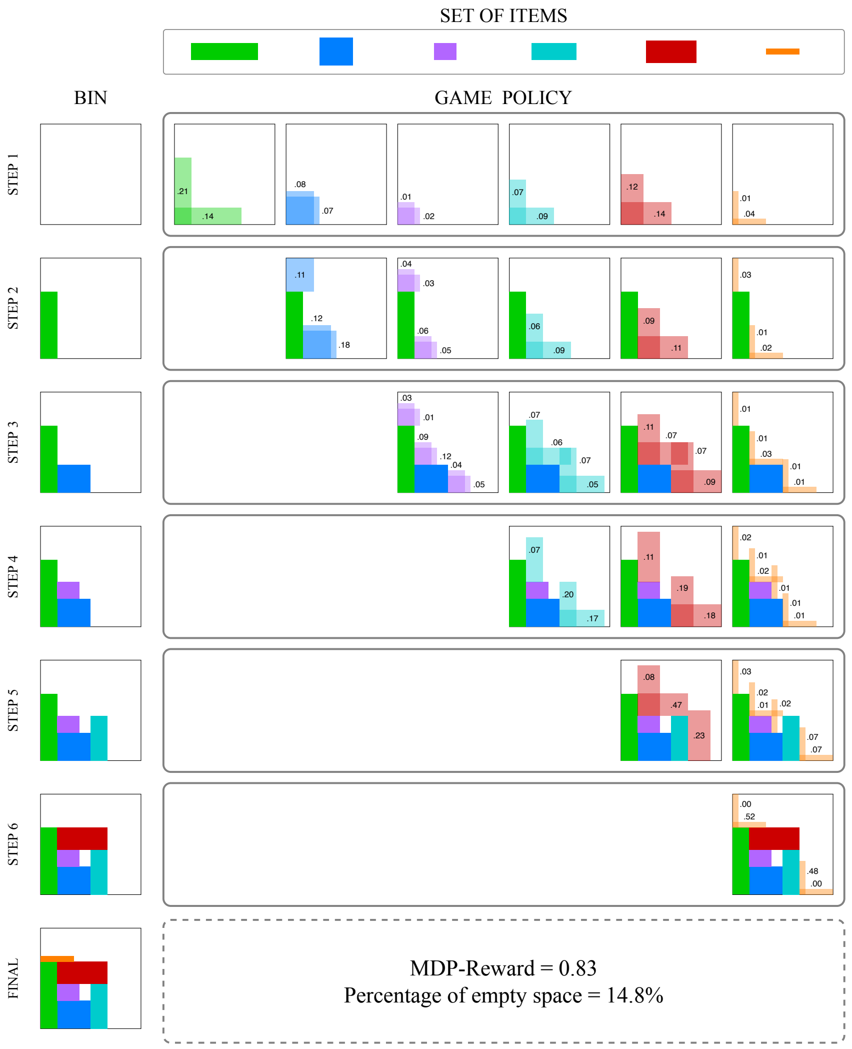
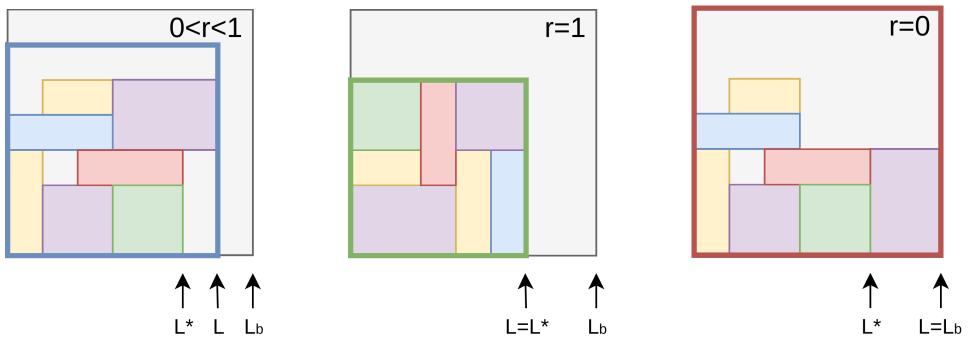


Figure 2: **Progressive construction of a solution to a 6-item problem.** The action space is depicted with the associated game policy, and the action executed is selected greedily with respect to this policy.

the bin L_b:

$$r = \begin{cases} \frac{L_b - L}{L_b - L^*}, & \text{if all items have been placed,} \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

Note that we only use the knowledge of the side of the optimal packing L^* to compute the reward. Information related to the optimal position of the items is not exploited in the algorithm. Knowing this allows us to calculate how close to the optimum a given solution is. The algorithm can be made generally applicable by changing the reward function when the size of the optimal solution to the problem is not known, e.g. a function of the percentage of the empty space.



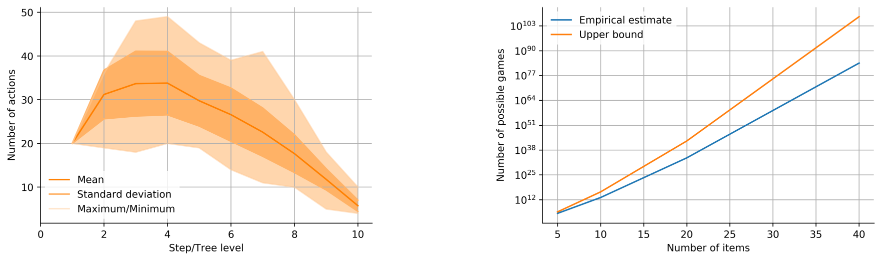
(a) sub-optimal solution (b) optimal solution (c) worst case solution

Figure 3: **MDP-reward.** Figure 3a shows a sub-optimal solution with all items placed. The bounding box side, L, lies between the optimal side, L^*, and the bin side, L_b. The exact reward, $0 < r < 1$, is given by Equation 4.1. Figure 3b shows an optimal solution with a bounding box of minimal side, $L = L^*$ and reward $r = 1$. Figure 3c shows a worst case scenario with the square bounding box filling the bin, $L = L_b$, and reward $r = 0$.

An initial analysis of the problem shows its exponential complexity in the number of items. Figure 4a illustrates how the number of legal moves changes at each step of the game and Figure 4b illustrates how the number of possible games grows with the number of items. A conservative upper bound for the number of possible games is:

$$G_N = \prod_{i=0}^{N-1} 2(N-i)(1+i). \tag{4.2}$$

The term $N - i$ represents the number of items left to play, while the term $1 + i$ stands for the maximum number of playable positions for move number i and the factor 2 accounts for the possible rotations. Decision problems with large branching factors cannot be solved optimally by brute force search. Tree search algorithms have thus emerged as a general method for identifying high reward solutions, and are thus well-suited for the bin packing problem.



(a) number of legal actions (b) number of possible games

Figure 4: **Bin packing complexity**.The values shown are estimated empirically by playing 50 random games and taking the average of the number of possible next actions (Figure 4a), and the number of possible games in the final MCTS tree (Figure 4b). Figure 4a describes the number of available actions after each item is placed in a problem with 10 items. Figure 4b shows how the number of possible games grows exponentially with the number of items. The upper bound G_N of the number of possible games for N items is calculated according to Equation 4.2.

4.2 The Ranked Reward algorithm

In this section we present in detail the method designed as a result of our first work of research on the bin packing problem, which we have named the **Ranked Reward** (R2) algorithm.

4.2.1 Ranked Rewards

When using self-play in two-player games, an agent faces a perfectly suited adversary at all times because no matter how weak or strong it is, the opponent always provides just the right level of opposition for the agent to learn from [2]. The R2 algorithm reproduces the benefits of self-play for generic single-player MDPs by reshaping the rewards of a single agent according to its relative performance over recent games. A detailed description is given by Algorithm 2.

The ranked reward mechanism compares each of the agent's solutions to its recent performance so that no matter how good it gets, it will have to surpass itself to get a positive reward. Recent MDP rewards, as given in Equation 4.1, are used to compute a threshold value r_α. This value is based on a given percentile $\alpha \in [0, 100]$ of the recent rewards, e.g. the threshold value r_{75} is the reward value at the 75th percentile of the recent rewards. The agent's recent solutions are each given a *ranked reward* $\tilde{r}$ of 0 or 1 according to whether or not it surpasses the threshold value: $\tilde{r} = \mathbb{1}_{\{r \geq r_\alpha\}}$. Doing this ensures that $(100 - \alpha)\%$ of the games will get a ranked reward of 1 and the rest a ranked reward of 0. This way, the player is provided with samples of recent games labeled relatively to the agent's current performance, providing information on which policies will improve its present capabilities.

The ranked rewards are then used as targets for the value head of a policy-value network and as the value of the end-game nodes of the MCTS. More precisely, we consider a policy-value network f_θ with parameters θ and MCTS which uses f_θ for guiding the move selection during the search and evaluating states without performing Monte Carlo roll-outs [18]. The

Algorithm 2: The Ranked Reward (R2) Algorithm.

```
Input: a percentile $\alpha$ and a mini-batch size $b$
Initialize fixed size buffers $\mathcal{D} = \{\}$ and $\mathcal{R} = \{\}$
Initialize parameters $\theta_0$ of the neural network, $f_{\theta_0}$
for $k = 0, 1, \ldots$ do
    for episode $= 1, \ldots, M$ do
        Sample an initial state $s_0$ ;
        for $t = 0, \ldots, N-1$ do
            Perform a Monte Carlo tree search consisting of $S$ simulations guided by $f_{\theta_k}$
            Extract MCTS-improved policy $\pi(\cdot|s_t)$
            Sample action $a_t \sim \pi(\cdot|s_t)$
            Take action $a_t$ and observe new state $s_{t+1}$
        end
        Compute MDP reward $r \leftarrow R(s_N)$ and store it in $\mathcal{R}$
        Compute threshold $r_\alpha$ based on the MDP rewards in $\mathcal{R}$
        Reshape to ranked reward $\tilde{r} \leftarrow \mathbb{1}_{\{r \geq r_\alpha\}}$
        Store all triplets $(s_t, \pi(\cdot \mid s_t), \tilde{r})$ in $\mathcal{D}$ for $t = 0, \ldots, N-1$
    end
    $\theta \leftarrow \theta_k$ for step=1, ..., $J$ do
        Sample mini-batch $\mathcal{B}$ of size $b$ uniformly from from $\mathcal{D}$
        Update $\theta$ by performing one optimization step using mini-batch $\mathcal{B}$
    end
    $\theta_{k+1} \leftarrow \theta$
end
```

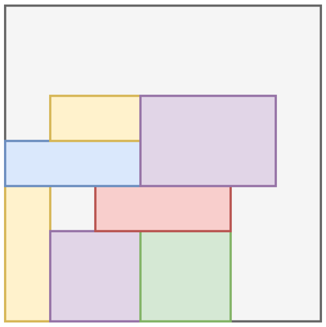
network takes a state s as input, and outputs probabilities p over the action space as well as an estimate v of the ranked reward of the current game, i.e., $(p, v) = f_\theta(s)$.

4.2.2 Neural Network architecture

The neural network architecture presented here was the one we used in the first version of our work, and was kept general to emphasize the wider applicability of our approach. This was in spite of more problem-specific architectures performing better and converging faster on the problem considered.

The network uses a visual representation of the bin and items. To represent the bin we use a binary occupancy grid indicating the presence or absence of items at discrete locations, as illustrated in Figures 5a and 5b. Similarly, each item is represented by two binary feature planes, one for each rotation, as illustrated in Figure 5c. If an item has already been placed in the bin, both planes are set to zero. The complete network input consists of the bin representation of size $L_b \times L_b \times 1$ and an $L_b \times L_b \times 2N$ feature stack representing the $2N$ individual items. Historical features (previous bin states) are not necessary as the environment is fully observable and strictly Markov.

An embedding of the bin representation is produced by feeding it to a number of convolutional layers and the item features are processed by multiple in-plane convolutional layers—with each item and its rotation processed independently. This is followed by aggregate operations ensuring that the embedding doesn't depend on the order of the items. The embeddings of the bin and of the items are then concatenated and fed to a residual tower[8] followed by separate policy and value heads representing respectively the full joint probability distribution over the action space $(L_b \times L_b \times 2N)$ and a state value estimate. This architecture contains approximately 420,000 parameters.



(a) state of the game

$$\begin{matrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0
\end{matrix}$$

(b) representation of the state

$$\begin{matrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0
\end{matrix}$$

(c) representation of an item

Figure 5: **Visual State Representation.** Figure 5a shows seven items placed in the bin with $L_b = 7$. Figure 5b shows the binary matrix indicating the presence, 1, or absence, 0, of items in the bin. Figure 5c shows the first binary representation of an item of size 3×2, without rotation.

The network architecture has been modified in a newer version of our algorithm and we dropped the visual representation and the heavy residual blocks for a simpler architecture where the state as well as actions attributes are fed as input to the network and both are processed to provide meaningful embeddings. The change in performance resulting from those changes (and some other changes in the algorithm) are further discussed in Section 4.4.

4.3 Experiments and results

Our experiments compare the performance of the R2 algorithm for α-percentiles of 50%, 75% and 90%. The experiments also include a version of the algorithm that used the MDP-reward without ranking as the target for the value estimate. The performances of the different algorithms are presented in Table 1 and the learning curves are displayed[9] in Figure 6.

The results show that our initial R2 algorithm outperforms its rank-free counterpart. Since then we have run more tests with an updated version of the algorithm, and the results are briefly summarized in the following section. The results presented here are thus those of the first - and less performant - version.

The rank-free version quickly plateaued at a value close to 0.88, while R2 surpassed that, with the 75% threshold version reaching 0.97. This represents an improvement of 10%, with

[8]One residual block applies the following transformations sequentially to the input: a convolution of 64 filters of kernel size 5x5 with stride 1, batch normalization, an ELU non-linearity, a convolution of 64 filters of kernel size 5x5 with stride 1, batch normalization, a skip connection that adds the input to the layer and an ELU non-linearity [19].

[9]The learning curve for the 90% reward threshold is not included in Figure 6 to improve the readability of the graph.

more than half of the problems solved optimally. In addition, faster and more stable learning is observed for R2 compared to its rank-free version. These results validate the importance of the ranking mechanism within the algorithm.

Algorithm	Mean (± std)	Median	Optimality
Rank-free	0.90 (±0.02)	0.90	4%
Ranked (50%)	0.92 (±0.04)	0.90	18%
Ranked (75%)	**0.97 (±0.05)**	**1.00**	**72%**
Ranked (90%)	0.94 (±0.07)	0.90	48%
Supervised	0.84 (±0.18)	0.90	16%
A3C	0.77 (±0.11)	0.80	0%
PPO	0.73 (±0.14)	0.80	0%
MCTS	0.88 (±0.05)	0.90	8%
Lego	0.82 (±0.10)	0.80	4%

Table 1: **Algorithms' best performance.**

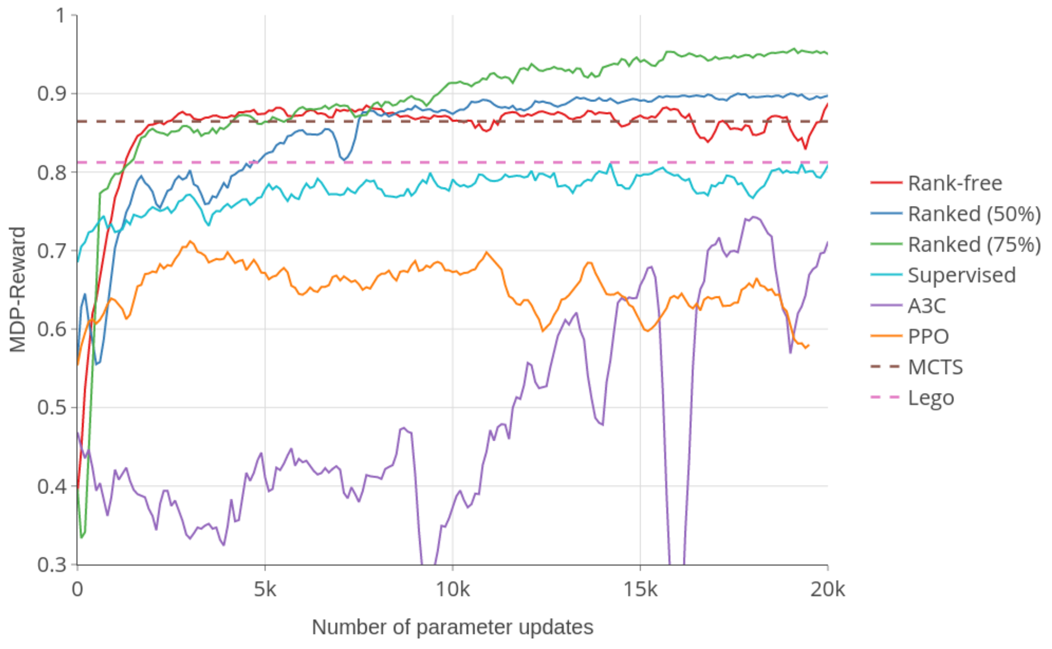


Figure 6: **Evolution of mean MDP-reward.**

In order to compare the performance of the R2 algorithm to existing approaches, our experiments also included a plain MCTS agent using Monte-Carlo roll-outs for state value estimation [6]; the *Lego* heuristic search algorithm [3]; two successful reinforcement learning methods: the *asynchronous advantage actor-critic* (A3C) algorithm [20] and the *proximal policy optimization* (PPO) algorithm [21]; and a supervised learning algorithm:

- **Plain MCTS** The plain MCTS agent used 300 simulations per move just like R2 and executed a single Monte Carlo roll-out per simulation to estimate state values.

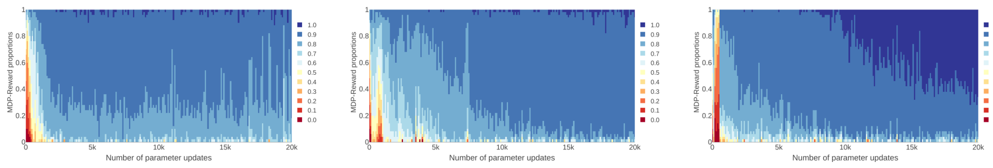
- **Lego Heuristic** The Lego algorithm [3] worked sequentially by first selecting the item minimizing the wasted space in the bin, and then selecting the orientation and position of the chosen item to minimize the bin size.

- **Reinforcement Learning** We considered the A3C [20] and PPO algorithms [21], and adapted the implementations provided in the Ray package [22] to our problem. In each experiment, we used exactly the same network as in the R2 algorithm. We ran 250 iterations for both A3C and PPO, and each iteration performed 100 steps of optimization with a mini-batch size of 64.

- **Supervised Learning** Because the bin packing problem instances are generated in a way that provides a known optimal solution for each problem, we designed a Lego-like heuristic algorithm defining a corresponding optimal sequence of actions $\{a_i\}_{0\leq i\leq N-1}$ resulting in this optimal solution. The state-action pairs (s_i, a_i) were used to train the policy-head of the R2 neural network as a one-class classification problem: given state s_i, the policy network should choose action a_i with maximum probability, i.e. the target is a one-hot encoding of the action a_i.

The performances of these algorithms are also given in Table 1 and in Figure 6. Both A3C and PPO reached a significantly lower performance level than R2 and MCTS, suggesting there is a clear advantage in using a tree search algorithm as a policy improvement mechanism. The same neural network was used in A3C, PPO, and R2, and was also trained in a supervised fashion as described above. The supervised learning policy was superior in performance to A3C and PPO, but relies on knowledge of optimal sequences of actions which are in practice unavailable.

Lego is faster to run than the other algorithms but performs worse than R2. The rank-free version of R2 achieves the same level of performance as MCTS, which suggests that the combination of its trained neural network with tree search provides neither an advantage nor a disadvantage. On the other hand, the neural network trained using ranked rewards as target for the value head leads to a significant improvement in the MCTS performance.



(a) **Rank-free** (b) **Ranked 50%** (c) **Ranked 75%**

Figure 7: **Proportion of MDP rewards.** Evolution of the reward proportions according to the ranking percentile α. Dark blue denotes the maximum reward of 1 and red denotes the minimum reward of 0. Ranked (75%) achieves better performance and stability than others.

4.4 Limitations and further work

Despite the good performance of our algorithm, we highlight here some potential limitations of our work. Some of these issues were addressed by the new version of our algorithm and we will also describe some of the changes we made in the algorithm.

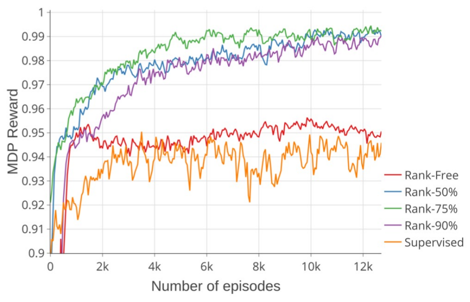
First, our results were produced only for instances of the problem with small number of items, and validating our approach would require comparing the performance on larger instances.

To really prove that our approach compares to state-of-the-art solutions to the bin packing problem, we would need to compare our results to the performance of the best optimization solvers used in practice. Unfortunately, there are not many baselines or free solvers available online for this problem, and it is thus difficult to evaluate how our method compares to the latest operations research techniques.

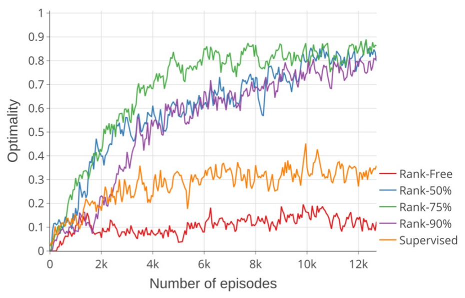
In addition, our current reward formulation includes the size of the optimal bounding box, which is in practice not known in advance. What is more, the best value for the threshold might be problem dependent and the performance might vary a lot from one value to another for different problems. Finally, our algorithm might run faster to solve a given instance compared to classical optimization solvers, but the amount of training needed to achieve such performance comes with high computational expense (days of training on thousands of different games). Limitations which optimization solvers do not have.

Addressing the issue regarding the reward is pretty straightforward : we need only use the percentage of wasted space an no additional information to compute the reward, thereby removing the knowledge of the optimal bounding box.

When it comes to the high computational expense of the method, we have moved to a lighter network architecture where the visual representation of the bin has been dropped. The bin and the items are simply represented by a number of attributes, and we also went for representing actions individually with similar attributes. Results from the new version of the algorithm on instances of the bin packing problem with 10 items are presented below.



(a) Evolution of mean MDP-reward



(b) Evolution of percentage of optimality

Figure 8: **RR (new version) comparative performance**

Figure 8a shows the the evolution of the mean MDP reward of different algorithms on the bin packing problem, while Figure 8b shows the percentage of games which produce an optimal solution out of 50 evaluation games. The RR algorithm with a threshold of 75% achieves a mean reward greater than 0.99 while being optimal nearly 84% of the time.

With the same version of the algorithm, we have also tried solving instances with 20 and 30 items, using the network which had been trained on instances with 10 items to guide the MCTS. The results on games with 30 items are detailed in Table 2 below, and show that our algorithm was able to produce high quality solutions even when the number of items increased. This demonstrates the potential of the algorithm to perform well on instances with a large number of items.

Algorithm	Mean (± std)	Min	Max	Median
Rank-free	0.70 (±0.082)	0.48	0.85	0.70
Ranked (50%)	**0.95 (±0.021)**	**0.90**	**0.98**	**0.95**
Ranked (75%)	0.94 (±0.028)	0.85	0.98	0.95
Ranked (90%)	0.94 (±0.025)	0.85	0.98	0.95

Table 2: **R2 Performance (30 items in 2D).** The values shown are calculated over 50 different instances of the 2D bin packing problem with 30 items. For each method, the neural network was restored from 250-th checkpoint when learning on data generated playing instances of 10 items.

5 The Glass Cutting Problem

Building on our previous work on the bin packing problem, we tried a similar approach for the glass cutting problem, which represents a good test to validate the performance of our algorithm at scale.

5.1 Presentation of the problem and context

As previously mentioned, the glass cutting problem is very similar to the bin packing problem, but it has a lot more constraints. The presentation of the challenge and the precise description of the problem can be found on the challenge's web-page[10,11]. We will summarize here in what consists the glass cutting problem and present its specificities compared to the bin packing problem.

The glass-cutting problem, as given in the challenge, consists of two elements : the plates and the items. The number of plates for a given instance is always the same and fixed to 100. The shape of the plates is also constant and fixed to 6000×3210 (in width, height order). Each plate contains a certain number of defects which are rectangles of variable shapes and positions on the plate (defects arise during the production of glass in the melting phase). A plate can have no defect, and a maximum of 8 defects.

Items, on the other hand, are given in a certain number of stacks. The number of stacks can

[10]challenge : http://www.roadef.org/challenge/2018/files/Challenge_Rules.pdf
[11]problem description : http://www.roadef.org/challenge/2018/files/Challenge_ROADEF_EURO_SG_Description.pdf

vary from an instance to another and the number of items in a stack can also vary. There are between 1 and 700 items per instance. Each item belongs to a stack, and there can be as many stacks as one wants. Therefore, the number of stacks per instance is also between 1 and 700. Plates are ordered from 1 to 100 and have to be used in this order to cut-out the items.

The objective formulation for the glass cutting problem is as follows : let $(H, W) = (6000, 3210)$ be the shape of the plates, $\mathcal{I}$ the set of items of the instance considered, and $\{(w_i, h_i)\}_{i \in \mathcal{I}}$ their shapes. Finally, let m be the number of plates used to cut-out all the items in $\mathcal{I}$, and r_m be the width of the residual part (defined later on) on the last bin. Then the glass cutting problem is formulated as a minimization problem under constraints :

$$\min \ HWm - Hr_m - \sum_{i \in \mathcal{I}} w_i h_i \tag{5.1}$$

The objective thus corresponds to the unused area of wastes, since $HWm - Hr_m$ represents the total area used to cut-out items, and $\sum_{i \in \mathcal{I}} w_i h_i$ represents the area of all items in the instance. This minimization is under a lot of constraints which we describe below.

The constraints are of different types : there are constraints on the items which can be played, and constraints on the cuts needed to cut-out those items. First, items are ordered in the stacks, and only the first item of each stack (the one on top) can be played, i.e. items below in the stacks can be played only once those above have been cut in order to ensure that the items are produced in the correct order. On the other hand, stacks themselves are unordered and an item can be played from any stack. Secondly, items cut cannot contain any defect, and cuts made cannot go through any defects. There are additional constraints which need specific vocabulary to be understood, and we thus introduce the appropriate notions below.

To cut-out an item, a certain number of cuts are allowed. Each cut has to be a *guillotine cut*, which is to say that it has to cut a given piece from edge to edge, parallel to the remaining edge, and thus divides a piece of glass in two. Depending on the cuts already made, the two resulting pieces can be in turn cut, or not. Only 4 guillotine cuts are allowed to cut-out items. The items have to be cut in order from left to right, and from bottom to top. The first cut to be made on a plate has to be vertical and is called a 1-cut, thus dividing the initial plate into two smaller pieces. Then the left piece resulting from the 1-cut has to be cut first, and the first cut to be made on it has to be a horizontal guillotine cut, called a 2-cut. This 2-cut in turn divides the left piece into two pieces, one on the top and one on the bottom. The bottom piece has to be cut before the top piece, and the next guillotine cut on it, called a 3-cut, has to be vertical. This succession of cuts dividing the initial plate in smaller pieces is called a cutting pattern. While performing cuts on the initial plate, some parts cut are not used to cut-out items and are called *wastes*. Finally, the last unused piece resulting from a 1-cut on the last plate used is called a *residual*. Figure 9 shows a visual example of how a plate can be cut using successive cuts. The small black rectangle on the top right corner is a defect.

A cutting pattern can be represented by a tree. This is possible since a guillotine cut always divides a rectangular plate in two smaller rectangular plates. Its root corresponds to the initial plate, its leafs are either the items, wastes (pieces of glass which are not cut further

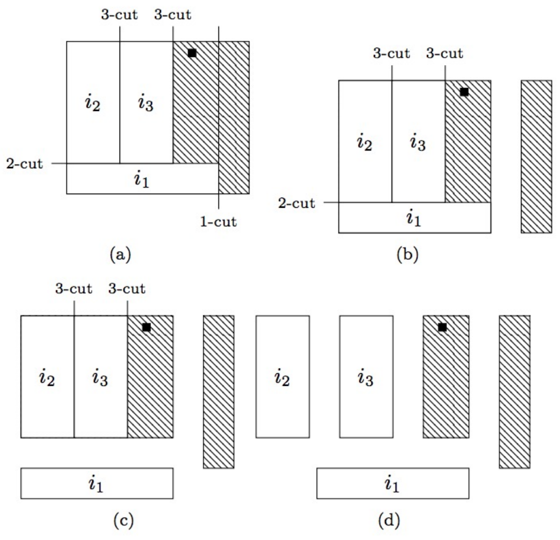


Figure 9: An initial pattern (a) and its variants by performing cuts (b-c-d). Due to the cut constraints (from left to right, and bottom up) if i_1, i_2, i_3 belong to the same stack, they must be cut in this order (i_1, i_2, i_3).

and which do not correspond to any item) or the residual. The children of a given node except the leafs are the ones obtained after performing a cut to divide the rectangle piece represented by the node into smaller pieces. An example of a tree built from a cutting pattern is shown in Figure 10.

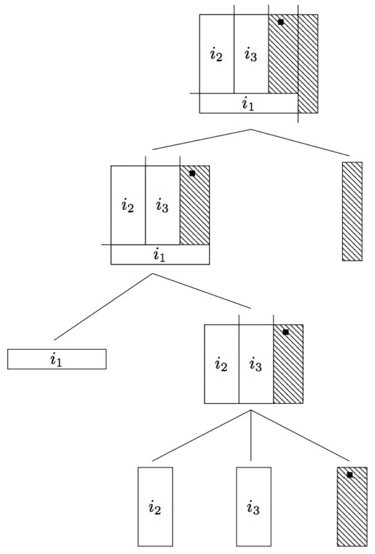


Figure 10: Tree representation of the cutting pattern from Figure 9

A solution to the problem consists in a list of successive cuts. Each cut is a rectangle representing the piece which is being cut, and has 9 attributes : the id of the plate on which the cut was made, an id for the cut itself, the coordinates x, y of the lower-left corner of the rectangle, its width w, its height h, the type of object cut (item, waste or residual if it is a leaf, and just a branch in the tree otherwise), the cut level (1-cut, 2-cut, 3-cut or 4-cut), and finally the id of the parent piece from which the current rectangle was cut. As previously mentioned, items have to be cut in a given order, and to ensure this order is respected, constraints on the cuts force items to be cut from left to right and from bottom up. This constraint ensures that from a given solution, the constraint on the order of the items can be checked.

Finally, additional constraints on cuts are imposed. Cutting patterns are two-dimensional and can be obtained using guillotine cuts only. The number of cuts allowed to obtain an item is at most 3 (1,2,3-cuts only). However, it is possible to perform at most one 4-cut in a sub-plate obtained after a 3-cut. This means that only one 4-cut is allowed only if no additional cut is required to get two items or one item and a waste. This is known in the literature as allowing trimming. An example is given in Figure 11.

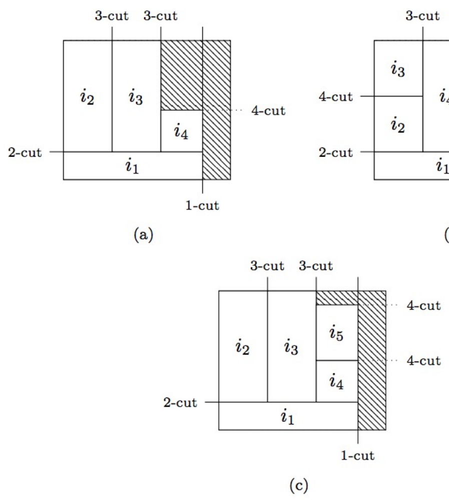
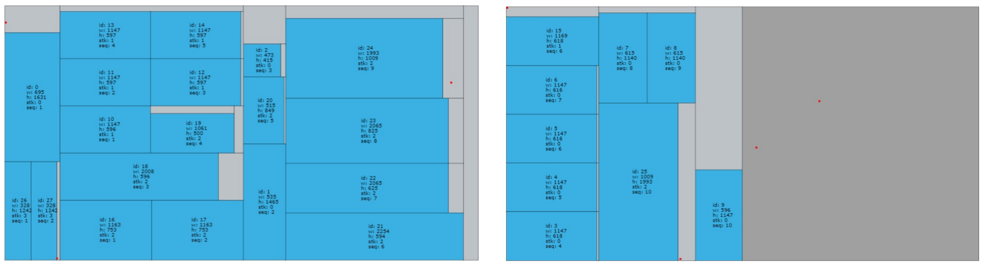


Figure 11: Example of valid cutting patterns (a-b) and a forbidden pattern (c). In pattern (a), the 4-cut is used only to remove the waste to get item i4. In pattern (b), two 4-cuts are used to cut items i2, i3 and items i5, i6. This configuration is allowed since a 4-cut is performed in each plate obtained after a 3-cut. In pattern (c), two 4-cuts are performed to cut items i4, i5 and a waste. This configuration is forbidden since two 4-cuts are performed in the same plate obtained after a 3-cut.

It is possible to cut an item in less than 3 cuts (see item i1 in Figure 4-(a)). The minimal width between two consecutive 1-cuts is 100, except for wastes. The maximal width between two consecutive 1-cuts is 3500, except for the residual. Due to the minimal and maximal widths between two consecutive 1-cuts and from technical limitation, a cutting pattern must contain at least one 1-cut. The minimal height between two consecutive 2-cuts is 100, except for wastes.

Only rotation by 90° of items is allowed, i.e. the items can be set horizontally or vertically on cutting patterns. All items must be cut, i.e. a solution must contain all the items in all stacks. Overlapping of items between them is not allowed. Finally, it is forbidden to cut through a defect. An example of a solution is provided in Figure 12.



(a) first plate (b) second plate

Figure 12: Example of a solution with two cutting patterns ($m = 2$) p1 (a) and p2 (b) for bins b1 and b2. Red points are defects. Light grey part represents the loss area (wastes), the dark grey one is the residual area (r_m is the width of this area).

5.2 Approach to the problem

We use the same approach to tackle this problem as we did for bin packing and use a combination of neural networks and MCTS in a self-learning loop. In order to do so, we have to define an environment and an agent which will interact in order to produce a solution to the problem.

5.2.1 Creating the environment

The first element is to be able to design an environment which takes into account all the constraints and stills allows for all legal actions to be performed by the agent, modifying the environment accordingly. As previously stated, since MCTS does a lot of simulations before actually taking an action, the interaction with the environment has to be fast in order to ensure that our training loop can be run for many iterations in a reasonable amount of time. This is why we chose Go to implement the environment.

The environment was implemented in a way such that items are played on the plate, and the cuts which gave birth to those items can be retrieved at some later point in the game. Actions are thus represented as placing items in the plate, and the only difficulty with this is that to produce a solution, we have to provide the cuts which were performed on the plate, and not simply the position of the items played. To get around that, we allow for 1-cuts to be performed from time to time, in order to be able then to retrieve the cuts which led to an item. The environment thus begins by allowing only a 1-cut on a new plate, then items are placed on the left sub-plate resulting from the 1-cut until no more items can be placed. Cuts leading to all the items are then retrieved at this point and the second sub-plate on the right resulting from the 1-cut can be in turn cut in the same fashion (the part on the left will be filled with items, and the one on the right will await further cutting).

Implementing the environment in this way allows to play items most of the time just like in bin packing, and still be able to produce a valid solution to the problem as required by the rules, building a solution tree in which the order of the items in the stacks is respected. The environment is able to implement the constraints in any given state, and therefore can provide a list of legal actions for the next step. Given the scale of the problem and the great number of constraint and their relative complexity, being able to design an environment which actually implements all the constraints properly, and is able to produce a valid solution (a tree of cuts) from item placements and 1-cuts only was already a huge step in designing an algorithm to tackle the glass cutting problem. Implementing this environment in Go enabled quick interaction with the agent, so that a large number of games could be played without taking too much time.

To account for possible rotations of the items, the number of stacks was artificially doubled, each stack having a "duplicate" containing the same items which had however been rotated. Each time an item was placed, its corresponding rotated duplicate was removed of the corresponding stack to make sure no item could be played twice.

To help the development of the environment and to check the solution produced, a small visualization script was written in Python in order to be able to visually render the actions taken by an agent, and actually see what a solution produced by an agent would look like. An upgraded version of this script was written to be able to show a demonstration of an agent playing, and an example of such a demonstration is available by clicking this link[12].

5.2.2 Network architecture

The agent we consider here is very similar to the one used in the bin packing problem. It consists of an MCTS to select moves in each state and actually play the game, and of a neural network to guide the search and evaluate states. The only difference is in the architecture of the network and the input it takes. On the other hand, when it comes to the environment, the bin packing environment and the glass cutting environment are really quite different.

The network receives five inputs from the environment, and has three outputs : the value v, the cut policy p^c and the item policy p^i. The input state s of the neural network is provided by the environment given its current state, and consists in five different inputs.

- *items input* : we consider the first d items in each stack and compute features for every one of those items based on their widths and heights. The items input is a matrix where each line corresponds to an item, and columns correspond to the attributes computed. Since the number of stacks can vary, we padd the lines of this matrix using the maximum number of stacks found in the different instances given (namely 72), to get a matrix having $2 \times 72 \times d$ lines.
- *defects input* consists of a matrix where each line represents a defect of the current plate being played, and the columns are the defects attribute (x, y, w, h). Again,

[12]copy-pasting http://34.241.184.167:8080/GlassCutting in a browser should produce the live demonstration in any case if the link does not work

the lines of this matrix are padded with zeros to reach a number of 8, which is the maximum number of defects per plate.

- *cuts input* consists of information concerning the last cuts that have been made. Namely, the position and shape of the last 1-cut, of the last 2-cut, and of the area left on the right of the last 1-cut.
- *environment information input* consists of basic information about the state of the environment such as the index of the current plate played, the number of items already played, the number of items left to play and whether the next available move is placing an item or making a 1-cut.
- *item actions input* is a matrix where each line represents an actions and columns represent attributes for each action. The illegal actions are padded with zeros.

The neural network is thus a function f_θ parameterized by some weights θ which takes as input a state s as described above, and outputs a triplet :

$$f_\theta(s) = (v_\theta(s),\ p^c_\theta(\cdot|s),\ p^i_\theta(\cdot|s))$$

corresponding to the estimated value of state s, a probability distribution on 1-cuts given state s, and a probability distribution on items' placement given state s.

A sketch of the neural architecture is depicted in Figure 13. Each input is processed separately. We use 1×1 convolution layers to treat each line of the input in the same way. This way, all items are processed the same way. The same is true for the defects in the defects input and the cuts in the cuts input. This processing yields embeddings for the cuts and the defects. As for the items, after a first processing, each d consecutive lines are stacked together to form a representation of the stacks, and then all stacks are processed in the same way using a second block of 1×1 convolution layers. And embedding of the stacks and their items is produced this way. All of the above-metnioned embeddings are stacked to produce an embedding of the state.
From there, this embedding is fed to two different heads of the network : a value head which outputs a value in $[-1, 1]$ after a tanh non-linearity, and a cut-policy head which outputs a distribution over cuts after a $softmax$ non-linearity. For the distribution over item actions, a different strategy is used. The item actions input is processed by a block of 1×1 convolution layers treating all actions the same way to produce an embedding of the actions. Then, a matrix-vector product between the item actions embedding and the state embedding is performed to obtain the logits of the distribution over items' placements.

5.2.3 Description of the method

The environment is implemented in such a way that either cuts or items are not legal moves for a given state. The neural network will guide the search of the MCTS by using one of the two distributions output by the network depending on the legal moves for the next action. In the distributions output by the network, all entries are non-zero, and those corresponding to illegal actions are thus zeroed-out, and the resulting vector is re-normalized to produce an actual distribution on legal moves.

The reward is not zero only for terminal states, and is defined as the ratio of the area of all the items to the total area used to cut-out all those items :

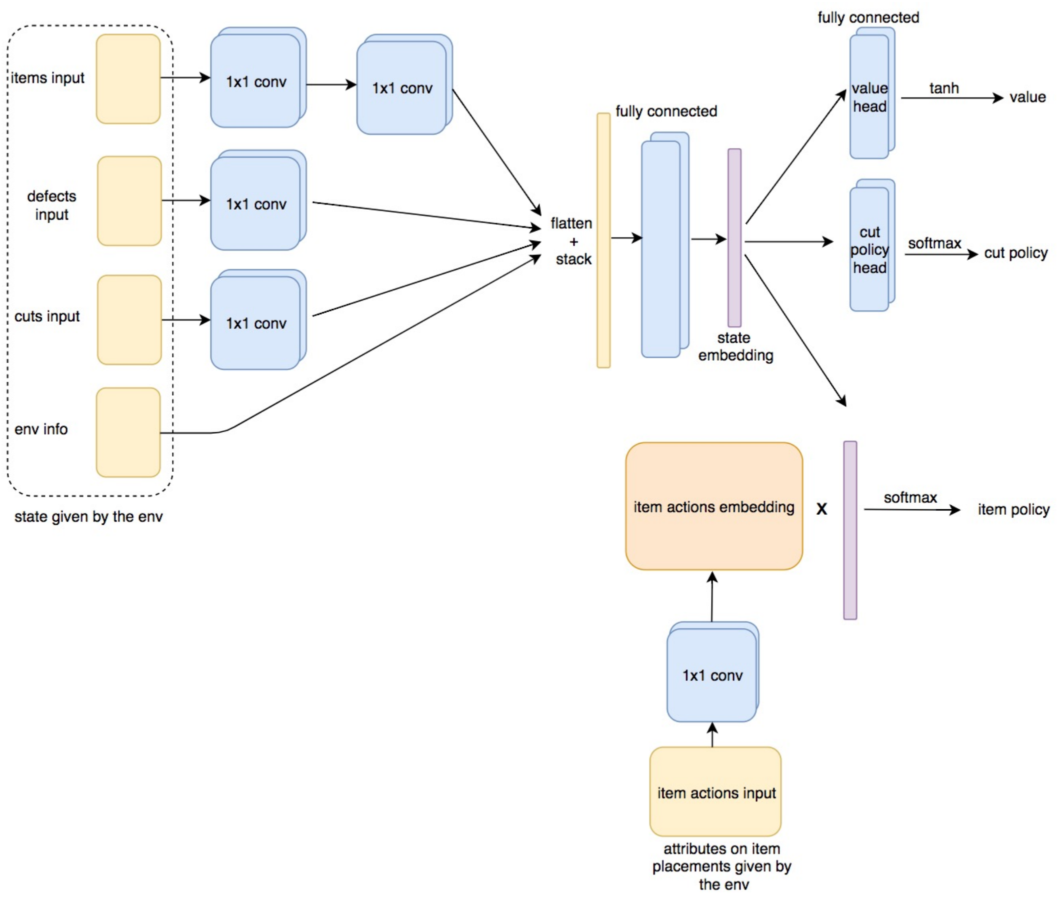


Figure 13: **Neural network architecture**

$$r = \begin{cases} \frac{\sum_{i \in \mathcal{I}} w_i h_i}{HWm - Hr_m}, & \text{if all items have been placed,} \\ 0, & \text{otherwise.} \end{cases} \tag{5.2}$$

If we define $o = \; HWm - Hr_m - \sum_{i \in \mathcal{I}} w_i h_i$ as being the objective of the problem, and $\mathcal{A}_I = \sum_{i \in \mathcal{I}} w_i h_i$ the total area of the items for a given instance, we then have that :

$$r = \frac{1}{o/\mathcal{A}_I + 1} \tag{5.3}$$

so that, for any given instance, maximizing the reward is equivalent to minimizing the objective ($\mathcal{A}_I$ is a constant for a given instance). We can observe furthermore that $r \in]0, 1]$ is a decreasing function of o, and $r = 1 \iff o = 0$.

The ranking method used is changed a little compared to what was described in Section 4. We keep the notion of having a threshold α to give a relative performance metric, but we now reshape the ranked reward $\tilde{r}$ in $\{-1, 0, 1\}$ using :

$$\tilde{r} = \mathbb{1}_{\{r > r_\alpha\}} - \mathbb{1}_{\{r < r_\alpha\}} \tag{5.4}$$

so that the ranked reward is 1 if it is greater than the reward threshold, 0 if it is equal to it, and -1 otherwise.

Other than that, the method is the same as before : we use MCTS to play games from which the neural network will learn the policies and the values. The network will in turn guide the search in the MCTS and be used to evaluate states. Actions from the MCTS are still selected by sampling from the visit-count distribution of the root node during training, and selected greedily as the most visited child for evaluation.

5.2.4 Plain MCTS performance and baselines

To begin with, we compared the performance of plain MCTS (without the help of a neural network and using roll-outs to evaluate states) to the baseline of the best scores obtained during the qualification phase of the challenge. This qualification phase was optional and since we joined the challenge quite late we did not take part to this phase. The best scores were always obtained by the *Local Solver* team who developed in-house operations research methods to provide an optimization solver which performed really well on the glass cutting challenge. We do not have access, however, to the amount of computation or time which was required by their algorithm to produce solutions. Comparative results on a few instances are summarized in Table 3. The performance of the MCTS is evaluated by playing 10 times the same instance, using 100 searches, 500 roll-outs and $C = 0$. This version of the MCTS could of course be improved by increasing C and increasing the number of searches, but those values were chosen in order to make runs faster. The results shown for MCTS are given in the mean (± std) format since they were obtained by averaging the score on different runs. For the Local Solver results however, we only had access to the best performance and could not compare the mean performance to that of MCTS.

Algorithm \ Instance	A1 obj	A5 obj	A12 obj	A20 obj
MCTS	**425,486 (±0.0)**	10,761,348 (±7.8e6)	5,126,227 (±4.8e6)	4,785,781 (±1.7e6)
Local Solver	**425,486**	**4,824,453**	**2,226,634**	**1,467,925**

Table 3: **Objective value of MCTS vs Local Solver on different instances**

For the easy instance A1 (low objective), MCTS was able to achieve the same score as the Local Solver at every one of the 10 runs (zero standard deviation), which tends to show that it is able to compete with Local Solver on small instances. However, when the instances become increasingly difficult, Local Solver is more than twice as good as MCTS. Nevertheless, MCTS seems to be of the same order of magnitude as Local Solver and still appears as a potential competitive approach if its parameters are chosen more carefully. As an example, we have tested an improved version of the MCTS on instance A20, using $C = 10$, 500 searches and 100 roll-outs. Out of 10 runs, the best solution found had an objective value of **1,673,365** (with a mean of 2,071,405) which is actually quite close to the value 1,467,925 obtained by the local solver. We hope that, with the help of a neural network properly trained, the performance of MCTS will increase and reach or surpass that of Local Solver.

5.3 Performance of the algorithm

Given the scale of the problem, the amount of time needed to play multiple games is a lot greater than what was needed for the bin packing. Therefore, the training takes a lot of time, and the last results are still pending at this time. With less than 10 iterations a day, it takes nearly a week to reach 100 iterations, by which we will not even be sure that the algorithm has converged (convergence might take more time than for the bin packing).

The latest results are awaited with great impatience and will be added to this report as soon as they are available.

5.4 Going further

We discuss here the potential improvements we have thought of but have not had time to try. First, finding the best threshold in the Ranked Reward algorithm might be problem or scale dependent, and choosing a value which makes the algorithm perform well may be a hassle. To overcome this, we thought of having two different agents play the same games, and give each of them a final reward $\tilde{r} \in \{-1, 0, 1\}$ depending on who achieved the highest reward. The agents would both use MCTS with the same parameters and neural networks with the same architecture, but initialized differently. Theses agents would play different moves due to the slight difference in their networks but also because of the randomness when sampling moves during training in MCTS. Going even further, we could imagine a population of agents with different initializations for the networks which would train on the same instances at each iteration and for which we would give a ranked reward of 1 if the agent achieved the best score among the agents, and -1 if it achieved a lower score. This would naturally place a threshold which could vary with time and adapt to the problem at hand by itself.

Another problem which arises however when using this kind of method is that the compute power needed grows (specially when having a population of agents). On this point, had we had more time, we would have liked to re-implement parts of the algorithm to make them parallelizable. For instance, as done in [18] we could have made the queries to the neural network for a policy and a value come in batch. This is done by delaying the moment when the value is backed-up and the policy from the network is incorporated to the UCT formula. This has both the effect of accelerating the whole process and favoring exploration. Similarly, at other stages in the process, be it in the environment itself, or when playing games, we could have changed the process a little to make things run in parallel and thus save time. For example, during training, since we play a number of games to generate data, we could make it so that those games are played in parallel to save time.

We would also have liked to try different versions of the MCTS in our algorithm to test the performance of each one. Specifically, we could have varied C and the number of searches during games to see if there was a way to optimize for the choice of those parameters as actions were performed. Finally, to evaluate how general our method is, it would be meaningful to compare its performance on other single-player games. To make it a fair comparison however, it would need to be ones with large branching factors since we believe it is in this type of setting that our algorithm has a competitive advantage and would prove most effective.

6 Conclusion

We have presented here our method to approach combinatorial optimization problems by reformulating them as Markov Decision Processes and using an algorithm interleaving MCTS and neural networks, as well as a ranking mechanism. The latest version of our algorithm has shown exceptional performance on instances of the bin packing problem with 10 items, finding an optimal solution 84% of the time, and has also proven to achieve high mean reward of 0.95 for instances of 20 and 30 items. The ranking mechanism provides a relative performance metric, acting as an artificial adversary which always pushes the agent beyond its current abilities. Results of applying our method to the more constrained problem of glass cutting at scale are still pending, but we expect our method to still perform well on this problem as it was already effective on the bin packing problem. It would be a true achievement to be able to reach or beat the baseline set by the Local Solver. Our work on the glass cutting problem is a first version that we have had to produce in a limited amount of time due to the deadline of the challenge, but more adjustments in the parameters, the network architecture or the MCTS can be made to improve this first draft. This first version may suffer from some remaining bugs in the code, and further work should produce a better performing version of the algorithm which would provide a new type of method to solve the glass cutting problem which is competitive compared to classical optimization methods.

References

[1] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv:1712.01815*, 2017.

[2] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv:1710.03748*, 2017.

[3] Haoyuan Hu, Lu Duan, Xiaodong Zhang, Yinghui Xu, and Jiangwen Wei. A multi-task selected learning approach for solving new type 3D bin packing problem. *arXiv:1804.06896*, 2018.

[4] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain–Sam Cohen, David Kas, Karl Hajjar, Torbjørn S. Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. https://arxiv.org/pdf/1807.01672.pdf, 2018. URL `https://arxiv.org/pdf/1807.01672.pdf`.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* A Bradford Book, 2017.

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

[7] Tor Lattimore and Csaba Szepesvári. *Bandit Algorithms.* 2018.

[8] Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. *CoRR*, abs/1802.04697, 2018. URL `http://arxiv.org/abs/1802.04697`.

[9] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems (NIPS) 30*, pages 5360–5370. 2017.

[10] César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research*, 211(3):427–441, 2011. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2010.09.010. URL `http://www.sciencedirect.com/science/article/pii/S0377221710006065`.

[11] V. Boyer, M. Elkihel, and D. El Baz. Heuristics for the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, 199(3):658–664, 2009. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2007.06.068. URL `http://www.sciencedirect.com/science/article/pii/S0377221708003639`.

[12] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61: 85–117, 2015.

[13] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems (NIPS) 28*, page 2692–2700, Montreal, Quebec, Canada, December 7-12 2015.

[14] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. URL `http://arxiv.org/abs/1611.09940`.

[15] Haoyuan Hu, Xiaodong Zhang, Xiaowei Yan, Longfei Wang, and Yinghui Xu. Solving a new 3D bin packing problem with deep reinforcement learning method. *arXiv:1708.05930*, 2017.

[16] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. A0C: Alpha zero in continuous action space. *arXiv:1805.09613*, 2018.

[17] Marwin H. S. Segler, Mike Preuss, and Mark P. Waller. Planning chemical syntheses with deep neural networks and symbolic ai. *Nature*, 555:604–610, 2018.

[18] David Silver, Julian Schrittandieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, (550):354–359, 2017.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, June 27-30 2016.

[20] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning (ICML)*, pages 1928–1937, New York City, NY, USA, June 19-24 2016.

[21] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

[22] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *arXiv:1712.05889*, 2017.