Deep Policy Learning for Perfect Rectangle Packing

Boris Doux, Satya Tamby, Benjamin Negrevergne, Tristan Cazenave

LAMSADE, Université Paris Dauphine-PSL, CNRS
{boris.doux,satya.tamby,benjamin.negrevergne,tristan.cazenave}@lamsade.dauphine.fr

Abstract

Perfect Rectangle Packing consists in solving a puzzle where the goal is to fit many small rectangles in a larger rectangle without creating holes. It has been previously addressed with Monte Carlo Tree Search algorithms. In this paper we train a neural network on solved instances of Perfect Rectangle Packing that are easy to generate. The trained neural network enables to derive a policy that is used as a prior by Monte Carlo search. The learned policy outperforms a random policy.

Introduction

Given a set of rectangles with different widths and heights, solving the problem of *perfect rectangle packing* consists in finding a spacial arrangement of the rectangles that exactly covers a *board* whose area is equal to the sum of the area of all the rectangles.

Perhaps unsurprisingly, the rectangle packing problem is NP-hard and remains NP-hard even if we do not consider rectangle rotations, (x)or if we only consider rectangles with identical dimensions (Fowler, Paterson, and Tanimoto 1981). Despite its complexity, rectangle packing is an interesting problem to study, with a number of practical applications (e.g. in transports and logistics or in circuit boards design) as well as ties with more theoretical considerations such as interval graphs. In the context of this work, the problem is also an interesting candidate application for testing the applicability of Deep Learning methods for solving combinatorial problems, because it is relatively easy to generate a large number of solved instances, that can be used to supervise the training of neural networks.

In practice however, neural networks are not good to deal with hard constraints and rarely produce exact solutions (which explains why they remain challenging to apply to combinatorial optimization problems). To overcome this limitation it is common to use the neural network to drive a search algorithm such as Monte Carlo search. Monte Carlo search algorithms rely on random simulations to discover good sequences of actions according to a user-specified scoring function. The most natural way to combine deep learning with Monte Carlo search is to replace the random (or hand crafted) search heuristic with a neural network trained to output a probability distribution over all possible moves. The random simulations are thus biased using the output of the network and often offer much better performance in practice. Unfortunately this method slows down the throughput of the algorithm, since it requires computationally intensive neural inferences at each step of the simulation.

In this paper, we exploit the property of the rectangle packing problem to derive a more efficient approach. We consider a simple class of probability distribution over the solutions, and train the neural network to output a probability distribution for each given instance to solve. We can then derive a search policy for Monte Carlo search and discover a good solution without having to perform more than one inference for each instance to solve. As we will show, we can efficiently solve almost all the instances in our dataset using this technique.

We first discuss the neural network architecture, and the best performing representation for the inputs/outputs. We then consider a variety of branching heuristics, which can be used to reduce the branching factor and speed up the search without loss of generality (i.e. all solutions remain attainable). We also reintroduce this neural network based policy inside a Monte Carlo search algorithm called NMCS. Finally, we conduct thorough experiments and show that this new approach can very efficiently solve 80.7% instances, using the deep learning policy only and 99.5% instances using the deep learning policy combined with a simple NMCS implementation with only 1 recursive level, outperforming other approaches based on random policies such as the one described by Pejic and van den Berg (2020).

Related works

Monte Carlo search and combinatorial optimization problems: Monte Carlo search algorithms rely on random simulations to discover good sequences of actions according to a user-specified scoring function. They demonstrate good performance for puzzles games such as Morpion Solitaire, Crossword puzzles, Sudoku or SameGame, and have also been used for more general combinatorial optimization problems such as graph coloring problems or vehicle routing problems.

To improve the quality of the results, researchers have often replaced purely random simulations with hand-crafted

search heuristics, designed specifically for the problem at hand. However, building such heuristic is time-consuming and requires expert knowledge which often is difficult to encode in the search heuristic. To overcome this limitation and facilitate the adaptation of Monte Carlo search to new problems, several algorithms have replaced the hand-crafted heuristics with simple policies that are *learned* using data collected during the search (e.g. NRPA Rosin (2011)). The principle of NRPA is to adapt the playout policy so as to learn the best sequence of moves found so far at each level. NRPA has found new world records in Morpion Solitaire and crosswords puzzles (Rosin 2011). Other applications deal with Logistics (Cazenave et al. 2020), Graph Coloring (Cazenave, Negrevergne, and Sikora 2020) and RNA Design (Cazenave and Fournier 2020).

The approach discussed in this paper is based on Nested Monte Carlo Search (NMCS) (Cazenave 2009). NMCS biases its playouts using lower level playouts. At level zero NMCS adopts a uniform random playout policy. Note that the base NMCS algorithm does not rely on policy learning (which makes it a better candidate than NRPA for our approach) yet it works well for puzzles and optimization problems. Online learning of playout strategies combined with NMCS has given good results on optimization problems (Rimmel, Teytaud, and Cazenave 2011). Other applications of NMCS include Single Player General Game Playing (Méhat and Cazenave 2010), Cooperative Pathfinding (Bouzy 2013), Software testing (Poulding and Feldt 2014), heuristic Model-Checking (Poulding and Feldt 2015), the Pancake problem (Bouzy 2016), Games (Cazenave et al. 2016) and the RNA inverse folding problem (Portela 2018).

Monte Carlo Tree Search has been successfully combined with deep learning in the Alpha Zero program (Silver et al. 2018). The combination has reached superhuman level in many games including the game of Go. The approach we present also combines Monte Carlo search and deep learning. However the combination we use is quite different. We use NMCS instead of PUCT, we address a single player game and we train on perfectly solved instances that are easy to generate. Moreover the policy is determined with only one inference before using NMCS biased with this policy. The policy is generated once for all states and not for every state as in Alpha Zero.

Rectangle Packing Rectangle Packing is a well known computational geometry problem and exact approaches have been proposed. For instance, Korf, Moffitt, and Pollack (2010) propose a constraint satisfaction formulation, Simonis and O'Sullivan (2008) use a constraint programming solver (Prolog) and propose some search rules, Huang and Chen (2007) classify different kinds of empty region in order to construct an heuristic. More recently, Pejic and van den Berg (2020) have proposed an approach relying on a Monte Carlo search algorithm and introduce simple search heuristics for the problem.

Perfect rectangle packing

In its most general formulation, rectangle packing consists in placing a maximum number of rectangles on a polyhedral board (without overlapping), allowing arbitrary rectangle rotations. The *perfect* rectangle packing problem is a variant that assumes that every instance admits a solution involving no empty space and all the rectangles.

In this paper, we only consider a more specific setting where the board has a square shape, and where only $\frac{\pi}{2}$ rad rotation are allowed. We also assume that all rectangles positions are integral. As mentioned earlier, this problem remains NP-hard, even if rotating the rectangles is not allowed.

Random simulations for Rectangle Packing

Random simulations are the core principle underlying the approach we describe in this paper, and underlying the Monte Carlo search algorithms in general. In this section, we introduce the required formalism and describe how to perform random simulations for the Rectangle Packing problem.

Board state and reward: In the context of the perfect rectangle packing problem, a state is a representation of the board together with a valid position for some, or all, the rectangles that have been placed on the board. We call S the set of all possible board states. A state is terminal iff all the rectangles have been placed on the board, or if no new rectangle can be placed. We call $T \subseteq S$ the set of all terminal states. Each terminal state $t \in T$ has a real valued reward denoted $reward(t)$, corresponding to the number of rectangles that have been placed at t.

Legal moves and branching strategy In each state, $s \in S$, the search algorithm can perform a set of legal moves (or actions) denoted M_s. Each move consists in placing one additional rectangle at a given position on the board. We denote $\mathcal{M}$ the set of all possible moves across all states, i.e. $\mathcal{M} = \bigcup_{s \in S} M_s$.

The exact set of legal action M_s for a given state s depends on the branching strategy that is used. For example, a first naive branching strategy consists in considering every valid rectangle position as a valid move. Using a different branching strategy such as *top left* (see section Experiments), a move is only legal if the rectangle is placed at the upper left most position on the board. Note that a good branching strategy can reduce the number of legal moves at each state, without removing possible solutions (however, they can also make the training of the policy model more challenging and can induce an important computational overhead).

Sequence of moves: Given a sequence of moves $X = \langle m_1, \ldots, m_t \rangle$, where each m_i is a possible move in $\mathcal{M}$, we denote $state_s(X)$ the state reached after playing each move in X starting in from the initial state s (s is omitted when it is clear from context). We also note $state_s(X, k)$, the state reached after playing only the first k moves in X. Note that the sequence is only legal, if each move $m_i \in X$ belongs to the set of legal moves in the corresponding state, i.e. $\forall i \in 1 \ldots t, m_i \in M_{state(X,i-1)}$.

Stochastic policies: A policy is a probability distribution p over a set of moves $\mathcal{M}$ that is conditioned on the current

game state $s \in S$. For example, we often consider the uniform policy p_0, which assigns equal probability to all the moves that are legal in state s. I.e. $p_0(m|s) = \frac{1}{|M_s|}$.

In this paper, we also consider policies probability distributions p_W which are parameterized with a set of weights W. There is one real valued weight for each possible move, i.e. $W = w_{m_1}, \ldots, w_{m_{|\mathcal{M}|}}$, and the probability $p_W(m|s)$ is defined as follows:

$$p_W(m|s) = \frac{e^{w_m}}{\sum_{p \in M_s} e^{w_p}}$$

Random simulations (a.k.a. playouts) Given an initial state s and a policy p (e.g. the uniform policy p_0), a random playout X is a sequence of moves drawn from the policy until a terminal state in reached using Algorithm 1.

Algorithm 1: The playout algorithm

```
playout (s: state, p: policy)
X ← []
while s is not terminal do
    choose a move m ~ p
    X ← X ∪ {m}
    s ← state(s + m)
end while
return (reward(X), X)
```

Solving perfect rectangle packing using random simulations and deep learning

In this section, we demonstrate how to learn a model that can then be use to solve the perfect rectangle packing problem using simulations. We first train a model that can be used to predict rectangles boundaries on the board, then we show how to place rectangles on the board using the output of the network.

Input and output representation

Each rectangle to be placed on a board B is described using a tuple of two natural numbers R_w and R_h representing respectively the width and the height of the rectangle. Therefore, an input instance with n rectangles is characterized by a set $\mathcal{R} = \{(R^i_w, R^i_h), i \in \{1, \ldots, n\}\}$. From $\mathcal{R}$, we build an input vector $x \in \mathbb{R}^{2n}$ such x_{2i} is the width of the i^{th} rectangle and x_{2i+1} is the height of the i^{th} rectangle (the rectangles are sorted according to their areas to ensure that each permutation of the rectangles in R has the same corresponding vector x).

We then train a neural network to map each input vector x to two matrices H and V representing the horizontal and the vertical boundaries of the rectangles on the board. More formally $H_{i,j} = 1$ if and only if there is a boundary between the tiles $B_{i-1,j}$ and $B_{i,j}$ and $V_{i,j} = 1$ iff there is a boundary between $B_{i,j-1}$ and $B_{i,j}$ (as illustrated in Figure 2).

The neural network is then trained using a large number of supervised examples using binary cross entropy as a loss function.

$$loss(H, V, H^*, V^*) = \sum_{\substack{k=1 \\ \ell=1}}^{\substack{k=B_w \\ \ell=B_h}} \frac{H^*_{k,\ell} log(H_{k,\ell}) + (1 - H^*_{k,\ell}) log(1 - H_{k,\ell})}{B_w B_h} + \sum_{\substack{k=1 \\ \ell=1}}^{\substack{k=B_w \\ \ell=B_h}} \frac{V^*_{k,\ell} log(V_{k,\ell}) + (1 - V^*_{k,\ell}) log(1 - V_{k,\ell})}{B_w B_h}$$

Figure 1 presents the prediction of the network compared to the ground truth. To ease the reading, we merged both H and V matrices to produce an image depicting the boundaries of the solution. We can observe that the network has captured the linearity of the boundaries as well as the symmetrical aspect of the problem.

Scoring rectangle positions on the board

Given that coefficients in H and V each represent a probability of observing a rectangle boundary at the corresponding position on the board, there may not be a unique way of placing the rectangles. To place the rectangles on the board, we need a scoring function that scores different placements based on the probability in H and V. The scoring function should be maximal when all the rectangle edges perfectly match the boundaries on the board. In this paper, we consider two possible scoring functions called *boundary only* and *boundary-interior*.

Boundary only: This first scoring function consists in the summing the coefficient in H and V when the corresponding position on the board is covered by a rectangle edge. For a rectangle R, we have:

$$s_B(R) = \sum_{k=i}^{i+R_h} V_{k,j} + \sum_{k=i}^{i+R_h} V_{k,j+R_w} + \sum_{k=j}^{j+R_w} H_{i,k} + \sum_{k=j}^{j+R_w} H_{i+R_h,k}$$

This score prioritizes the widest rectangles since their frontier is larger.

Boundary-Interior: This score also takes into consideration the interior of the rectangle, which is defined by:

$$s_I(R) = \sum_{k=i}^{i+R_w-1} \sum_{\ell=j+1}^{j+R_h-1} V_{k,\ell} + \sum_{k=i+1}^{i+R_w-1} \sum_{\ell=j}^{j+R_h-1} H_{k,\ell}$$

Therefore, the final score is:

$$s_{BI}(R) = s_B(R) - s_I(R)$$

Observe that this score favours smallest rectangles. Indeed, since large rectangles have by definition a largest interior, their score can be penalized by non zero outputs.

Branching strategies

It is not always desirable to consider all the possible moves at each stage of the simulation, as it can dramatically increase the branching factor (i.e. the number of legal moves

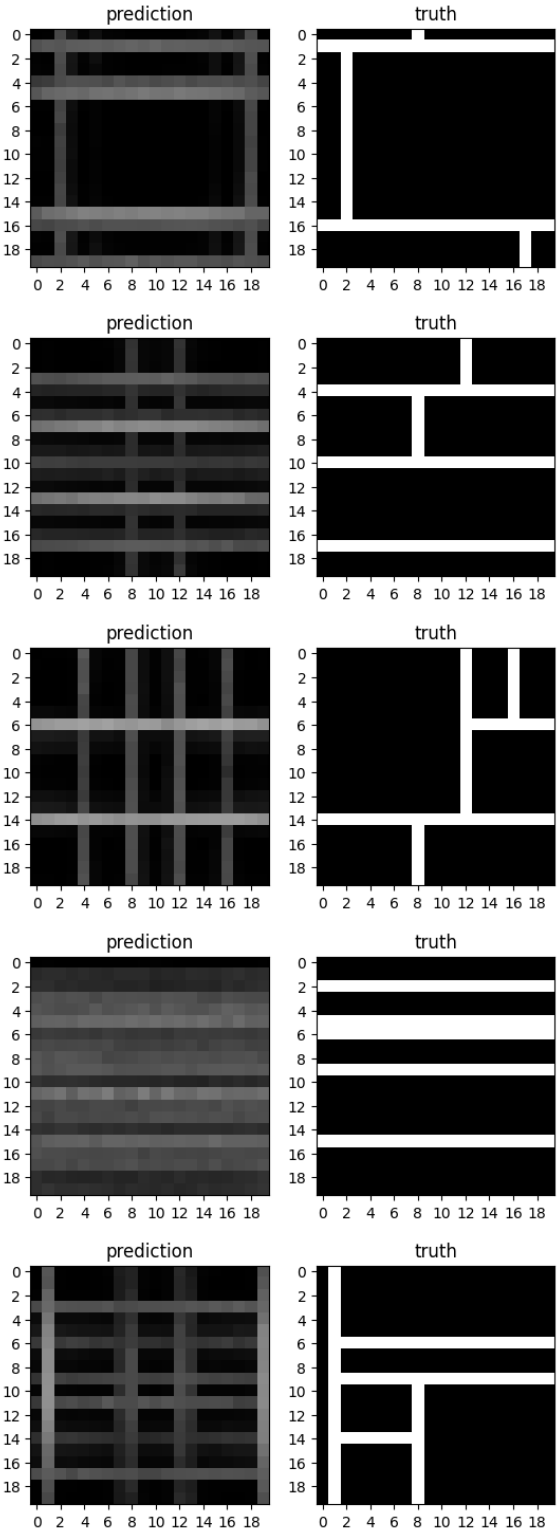
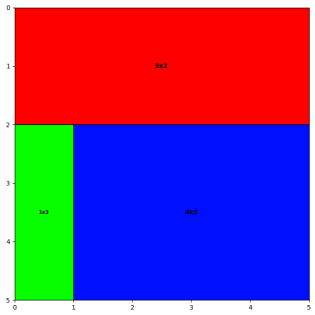


Figure 1: Left, Merged H and V net output Right, merged H and V from the ground truth.



$$H = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad V = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2: Top, an example of 3 rectangles perfectly packed on a 5x5 board. Bottom, corresponding H and V matrices representing rectangles frontiers.

at each step), impede the learning process and slow down the simulations.

However branching heuristics can also induce an important computational overhead to compute the set of legal moves and thus there is a trade-off between the extra computational cost induced by the branching heuristic and the benefit, from the learning perspective.

In this paper, we consider several heuristics in order to select which subset of the legal moves (i.e which remaining rectangles and which positions) will be selected at a given iteration.

Most constrained first. The first heuristic consists in trying to place the most constrained rectangle, i.e. the rectangle having the largest area. Ties are broken randomly. This heuristic relies on the idea that placing the largest rectangles first will drastically reduce the legal positions for the smaller ones, and so reduce the depth of the search tree.

Touching Edge first. A second heuristic consists in prioritizing positions adjacent to an edge of the board. It relies on the assumption that placing rectangles in the middle of the board have more chance to lead to an infeasible solution.

Top left Chazelle (1983). The last heuristic relies on the knowledge that the totality of the board will be covered (there will be no empty space). Thus, at each iteration, we consider every rectangles that can be placed on the free point that is the uppermost left. This allows to significantly reduce the number of available choices at each iteration.

Computing the policy

Given the current state, we compute all legal moves from the current branching strategy then from the score obtained by the current scoring function, we build a probability distribution using the softmax function.

	AM	MCF	TEF	MCF +TEF	TL
1 trial					
Uniform	4.54	4.47	4.69	4.59	4.33
Policy BI	5.03	5.43	5.05	**5.48**	5.42
Policy B	**5.04**	**5.45**	**5.07**	**5.48**	**5.75**
2 trials					
Uniform	4.75	4.73	4.86	4.85	4.8
Policy BI	**5.19**	5.56	**5.23**	**5.67**	5.58
Policy B	5.18	**5.58**	5.21	5.64	**5.84**
5 trials					
Uniform	4.94	5.02	4.99	5.12	5.32
Policy BI	**5.45**	**5.77**	**5.46**	5.77	5.73
Policy B	5.42	**5.77**	5.45	**5.79**	**5.89**
10 trials					
Uniform	4.98	5.17	5.03	5.3	5.61
Policy BI	5.61	**5.83**	5.65	**5.83**	5.83
Policy B	**5.62**	5.82	**5.68**	5.81	**5.91**
100 trials					
Uniform	5.10	5.66	5.22	5.81	**5.99**
Policy BI	5.97	**5.93**	5.97	**5.91**	5.93
Policy B	**5.99**	5.89	**5.99**	5.89	5.97

Table 1: Average number of positioned rectangle over 1000 random instances using different Monte Carlo search

	AM	MCF	TEF	MCF +TEF	TL
1 trial					
Uniform	0	0.039	0.002	0.078	0.124
Policy BI	0.117	0.525	0.122	0.574	0.626
Policy B	**0.120**	**0.536**	**0.150**	**0.581**	**0.807**
2 trials					
Uniform	0.002	0.079	0.006	0.120	0.197
Policy BI	**0.197**	**0.648**	0.237	**0.711**	0.702
Policy B	0.191	0.641	0.221	0.691	**0.872**
5 trials					
Uniform	0.006	0.159	0.018	0.219	0.425
Policy BI	**0.451**	0.791	0.458	0.796	0.806
Policy B	0.421	**0.795**	**0.495**	**0.809**	**0.911**
10 trials					
Uniform	0.006	0.236	0.034	0.346	0.616
Policy BI	0.614	**0.842**	0.653	**0.847**	0.842
Policy B	**0.648**	0.840	**0.672**	0.841	**0.926**
100 trials					
Uniform	0.103	0.663	0.224	0.809	**0.991**
Policy BI	0.986	**0.932**	0.986	**0.912**	0.929
Policy B	**0.994**	0.899	**0.996**	0.900	0.970

Table 2: Ratio of instances optimally solved using different Monte Carlo search

Preliminary results

Instance generation In our experiments, we focus on *guillotineable instances*. Starting with a single rectangle covering the whole board, the generator iteratively selects a random rectangle and cuts it in two, from one edge to the opposite edge. Observe that it is not possible to generate every perfect rectangle packing instances with this method, as mentioned in Pejic and van den Berg (2020).

Supervised learning We trained a residual fully-connected neural network of 20 layers of 2000 neurons each on a dataset containing a sample of 100 000 examples. We used the *ADAM* gradient descent to optimize the binary cross entropy between the ground truth and the prediction of our network.

Experiments We present in this section some preliminaries results on packing 6 rectangles on a 20×20 board

We compare the efficiency of the combinations of heuristics and scores presented above to guide rollouts. Each rollout takes 0.0085 seconds. The columns represent which subset of the legal moves is considered at each iteration, while the lines represent how the distribution of the legal moves is biased. In each situation, we take the best solution out of 1, 2, 5, 10 and 100 rollouts, referred to as trials. Column *AM* (*All Moves*) contains the whole set of legal moves, *MCF* (*Most Constrained First*) only considers the largest rectangle, *TEF* (*Touching Edge First*) prioritizes positions adjacent to at least an edge of the board and *TL* (*Top left*) only considers the uppermost left empty position on the board.

Table 1 and 2 respectively represent the average number of placed rectangles and the ratio of solved instances according to these rollouts, on 1000 problems.

We first observe that the *All Moves* heuristic struggles to solve instances using an uniform policy. This shows that the problem cannot be trivially solved without additional effort. A natural approach consists in increasing the number of rollouts (which is computationally more costly) but the evolution of the performance over the trials shows that increasing the search for the uniform policy is not enough to solve many instances. However, using the two policies deduced from the output of the neural network seems to significantly improve the average score and the number of problems that are solved.

A simple heuristic in order to reduce the number of legal moves available at each step is to act greedily with respect of rectangles area size. The *MCF* heuristic helps the uniform policy to solve a couple of problems but is still struggling. However, it does produce a huge improvement for policies. Regarding trials evolution, the computational effort added helps each policy but the leanrt policies even more by achieving more than 90% of problems solved.

Because placing a rectangle in the middle of the board may lead to an unsolvable instance , *TEF* forces policies to place rectangles near the edges. It helps the uniform policy to solve some problems and more across different trials but no improvement are visible for the learnt policies relatively to policies with *AM*. The relatively low number of rectangles may affect the quality of this heuristic because there is a high number of possible positions for the different rectangles which are touching an edge of the board. So only few bad

positions are pruned from the possibilities.

Combining *MCF* and *TEF* produce even better results for the policies. But as the number of trials grows, the benefit for the learnt policies is lesser than the benefit for the uniform policy which achieves the best score so far while staying weaker than the learnt policies.

Finally, *TL* helps policies the most. Policy B produces the best result so far with a single trial by solving 80% of the problems. With more search, every policies ended up solving almost all the instances. *TL* heuristics is reducing the most the number of possible moves: at each step, only one position is considered for each rectangle. From this experiment we can also notice that there is a difference between B score and BI score. Only this heuristic shows the penalty induced by BI score to bigger rectangles. BI score leads to a lesser discriminating policy than the one built from B score because of the very few number of possibilities at each step.

Combining the policy with NMCS

To further improve the quality of the solutions, we combine random simulations with *Nested Monte Carlo Search* (NMCS). NMCS improves on random search by running recursive (a.k.a. nested) simulations combined with a tree search.

At the lowest recursive level, moves are simply sampled using a stochastic policy. At the recursive level above, the move with the best estimated score is selected. The score of each possible move is estimated using playouts based on moves sampled from the policy of the level below. (See Algorithm 2.)

Nesting simulations greatly improves the quality of the solutions discovered, however it is generally impossible to run NMCS with more than 5 or 6 levels of recursion, due to the prohibitive cost of the recursive simulations.

NMCS can easily profit from deep learning by sampling moves at the lowest level from policies based on a deep neural network such as the ones we have presented in the previous sections. Doux, Negrevergne, and Cazenave (2021) have shown that replacing the stochastic policy at the lowest level with a deep neural network based policy can be used to obtain scores that are comparable to NMCS with 3 level of recursions using only one level of recursion.

Experiments

In this section, we use NMCS with only one level of recursion combined with the neural network based policy to solve 1000 instances of perfect rectangle packing. The instances involve 6 rectangles to be placed on a 20×20 board. Table 4 and 5 present the average number of rectangles placed, and the ratio of solved instances respectively. We also present the average time for each branching heuristic in Table 3.

In the tables, *NMCS* refers to NMCS with uniform random rollouts, *NMCS BI* and *NMCS B* refers to NMCS with rollouts driven by the branching strategies BI and B respectively.

First, we can observe that even though TL remains the most efficient branching strategy, the other branching strategies also perform well, thus using NMCS reduce the importance of choosing the correct branching strategy. If we look at the times from Table 3, we can also observe an important discrepancy between the branching strategies: even if MCF and TL are almost as good TF is three times faster and thus should be preferred. Finally, we can also see that the best configuration is able to solve 99.5% of the problems.

Algorithm 2: The NMCS algorithm.

```
NMCS (current-state, policy, level)
if level = 0 then
  return playout (current-state, policy)
else
  best-reward ← −∞
  while current-state is not terminal do
    for each move in M_current-state do
      next-state ← state(current-state + move)
      (reward, seq) ←
          NMCS (next-state, policy, level − 1)
      if reward > best-reward then
        next-best-state ← next-state
        best-reward ← reward
        best-sequence ← seq
      end if
    end for
    move ← move of best-sequence
    next-state ← state(current-state + move)
  end while
  return (best-reward, best-sequence)
end if
```

	AM	MCF	TEF	MCF +TEF	TL
NMCS	4.54	0.52	1.36	0.41	0.17

Table 3: Mean time (second) spent for 1 trial by NMCS over 1000 examples

Conclusion and further works

We show that the geometrical aspects of the *Perfect Rectangle Packing problem* can be exploited to efficiently supervise the training of a neural network. Even if its output is imperfect, we can significantly improve the quality of Monte Carlo search algorithms by taking the knowledge extracted by the network into consideration.

Beyond solving more complicated instances using this approach, some interesting improvements could consists in studying how a neural network can handle problems with various problem sizes. First, the number of rectangles can fluctuate across the instances, but also the size of the board can vary and finally, both of theses parameters can change.

	AM	MCF	TEF	MCF +TEF	TL
NMCS	5.38	5.83	5.45	5.86	5.82
NMCS BI	**5.88**	**5.97**	5.87	**5.96**	**5.99**
NMCS B	**5.88**	5.96	**5.89**	**5.96**	**5.99**

Table 4: Average number of positioned rectangle over 1000 random instances using NMCS

	AM	MCF	TEF	MCF +TEF	TL
NMCS	0.380	0.826	0.449	0.860	0.817
NMCS BI	0.878	**0.967**	0.870	**0.961**	0.987
NMCS B	**0.880**	0.962	**0.892**	0.955	**0.995**

Table 5: Ratio of instances optimally solved using NMCS

In order to deal with these fluctuations some other representations of the data may be relevant. It could also be interesting to use a policy derived from a neural network to guide other Monte Carlo search algorithms such as *Monte Carlo Tree Search* and *Nested Rollout Policy Adaptation* which are known to perform very well for solving combinatorial optimization problems.

References

Bouzy, B. 2013. Monte-Carlo Fork Search for Cooperative Path-Finding. In *Computer Games - Workshop on Computer Games, CGW 2013, Held in Conjunction with the 23rd International Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3, 2013, Revised Selected Papers*, 1–15.

Bouzy, B. 2016. Burnt Pancake Problem: New Lower Bounds on the Diameter and New Experimental Optimality Ratios. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016*, 119–120.

Cazenave, T. 2009. Nested Monte-Carlo Search. In Boutilier, C., ed., *IJCAI*, 456–461.

Cazenave, T.; and Fournier, T. 2020. Monte Carlo Inverse Folding. In *Monte Carlo Search at IJCAI*.

Cazenave, T.; Lucas, J.-Y.; Kim, H.; and Triboulet, T. 2020. Monte Carlo Vehicle Routing. In *ATT at ECAI*.

Cazenave, T.; Negrevergne, B.; and Sikora, F. 2020. Monte Carlo Graph Coloring. In *Monte Carlo Search at IJCAI*.

Cazenave, T.; Saffidine, A.; Schofield, M. J.; and Thielscher, M. 2016. Nested Monte Carlo Search for Two-Player Games. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, 687–693.

Chazelle, B. 1983. The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation. *IEEE Trans. Computers*, 32(8): 697–707.

Doux, B.; Negrevergne, B.; and Cazenave, T. 2021. Deep Reinforcement Learning for Morpion Solitaire. In Springer, ed., *Advances in Computer Games*, LNCS.

Fowler, R. J.; Paterson, M. S.; and Tanimoto, S. L. 1981. Optimal packing and covering in the plane are NP-complete. *Information processing letters*, 12(3): 133–137.

Huang, W.; and Chen, D. 2007. An efficient heuristic algorithm for rectangle-packing problem. *Simulation Modelling Practice and Theory*, 15(10): 1356–1365.

Korf, R. E.; Moffitt, M. D.; and Pollack, M. E. 2010. Optimal rectangle packing. *Annals of Operations Research*, 179(1): 261–295.

Méhat, J.; and Cazenave, T. 2010. Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4): 271–277.

Pejic, I.; and van den Berg, D. 2020. Monte carlo tree search on perfect rectangle packing problem instances. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 1697–1703.

Portela, F. 2018. An unexpectedly effective Monte Carlo technique for the RNA inverse folding problem. *bioRxiv*, 345587.

Poulding, S. M.; and Feldt, R. 2014. Generating structured test data with specific properties using nested Monte-Carlo search. In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014*, 1279–1286.

Poulding, S. M.; and Feldt, R. 2015. Heuristic Model Checking using a Monte-Carlo Tree Search Algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, 1359–1366.

Rimmel, A.; Teytaud, F.; and Cazenave, T. 2011. Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows. In *Applications of Evolutionary Computation*, volume 6625 of *Lecture Notes in Computer Science*, 501–510. Springer.

Rosin, C. D. 2011. Nested Rollout Policy Adaptation for Monte Carlo Tree Search. In *IJCAI*, 649–654.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Simonis, H.; and O'Sullivan, B. 2008. Search strategies for rectangle packing. In *International Conference on Principles and Practice of Constraint Programming*, 52–66. Springer.