Ranked Reward: Enabling Self-Play Reinforcement Learning for Combinatorial Optimization

Alexandre Laterre a.laterre@instadeep.com

Yunguan Fu y.fu@instadeep.com

Mohamed Khalil Jabri mk.jabri@instadeep.com

Alain–Sam Cohen as.cohen@instadeep.com

David Kas d.kas@instadeep.com

Karl Hajjar k.hajjar@instadeep.com

Torbjørn S. Dahl t.dahl@instadeep.com

Amine Kerkeni ak@instadeep.com

Karim Beguir kb@instadeep.com

Abstract

Adversarial self-play in two-player games has delivered impressive results when used with reinforcement learning algorithms that combine deep neural networks and tree search. Algorithms like AlphaZero and Expert Iteration learn *tabula-rasa*, producing highly informative training data on the fly. However, the self-play training strategy is not directly applicable to single-player games. Recently, several practically important combinatorial optimization problems, such as the traveling salesman problem and the bin packing problem, have been reformulated as reinforcement learning problems, increasing the importance of enabling the benefits of self-play beyond two-player games. We present the Ranked Reward (R2) algorithm which accomplishes this by ranking the rewards obtained by a single agent over multiple games to create a relative performance metric. Results from applying the R2 algorithm to instances of a two-dimensional bin packing problem show that it outperforms generic Monte Carlo tree search, heuristic algorithms and reinforcement learning algorithms not using ranked rewards.

1 Introduction and Motivation

Reinforcement learning (RL) algorithms that combine neural networks and tree search have delivered outstanding successes in two-player games such as Go, Chess, Shogi, and Hex. One of the main strengths of algorithms like AlphaZero [16] and Expert Iteration [1] is their capacity to learn *tabula rasa* through *self-play*. Historically, self-play has also produced great results in the game of Backgammon [18]. Using this strategy removes the need for training data from human experts and provides an agent with a well-matched adversary, which facilitates learning.

While self-play algorithms have proven successful for two-player games, there has been little work on applying similar principles to single-player games [11]. These games include several well-known combinatorial problems that are particularly relevant to industry and represent real-world optimization challenges, such as the traveling salesman problem (TSP) and the bin packing problem.

This paper describes the *Ranked Reward* (R2) algorithm and results from its application to a 2D bin packing problem formulated as a single-player Markov decision process (MDP). The R2 algorithm uses a deep neural network to estimate a policy and a value function, as well as Monte Carlo tree search (MCTS) for policy improvement. In addition, it uses a reward ranking mechanism to build a single-player training curriculum that provides advantages comparable to those produced by self-play in competitive multi-agent environments.

The R2 algorithm offers a new generic method for producing approximate solutions to NP-hard optimization problems. Generic optimization approaches are typically based on algorithms such as integer programming [20], that provide optimality guarantees at a high computational expense, or heuristic methods that are lighter in terms of computation but may produce unsatisfactory suboptimal solutions. The R2 algorithm has the advantage of outperforming heuristic approaches while scaling better than optimization solvers. We present results showing that it surpasses a range of existing algorithms on a 2D bin packing problem, including MCTS [5], the Lego heuristic algorithm [7], as well as RL algorithms such as A3C [10] and PPO [15].

In Section 2 of this paper, we summarize the current state-of-the-art in deep learning for games with large search spaces. Then, in Section 3, we present a single-player MDP formulation of the 2D bin packing problem. In Section 4 we describe the R2 algorithm using deep RL and tree search along with a reward ranking mechanism. Section 5 presents our experiments and results, and discusses the implications of using different reward ranking thresholds. Finally, Section 6 summarizes current limitations of our algorithm and future research directions.

2 Deep Learning for Combinatorial Optimization

Combinatorial optimization problems are widely studied in computer science and mathematics. A large number of them belongs to the NP-hard class of problems. For this reason, they have traditionally been solved using heuristic methods [13, 4]. However, these approaches may need hand-crafted adaptations when applied to new problems because of their problem-specific nature.

Deep learning algorithms potentially offer an improvement on traditional optimization methods as they have provided remarkable results on classification and regression tasks [14]. 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 solutions.

One way to overcome this challenge was introduced by Vinyals et al. [19] 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 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 [3].

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.* [8] 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.

2.1 Deep Learning with Tree Search and Self-Play

Policy iteration algorithms that combine deep neural networks and tree search in a self-training loop, such as AlphaZero [16] and Expert Iteration [1], have exceeded human performance on several two-player games. These algorithms 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, helps to raise the performance of the neural network by providing improved MCTS-based policies during training.

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.

The main contribution of the R2 algorithm is 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 Bin Packing as a Markov Decision Problem

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.* [8], 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 1 and illustrated in Figure 1, 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 the bin L_b:

$$r = \begin{cases} \frac{L_b - L}{L_b - L^*}, & \text{if all items have been placed,} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

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.

An initial analysis of the problem shows its exponential complexity in the number of items. Figure 3a illustrates how the number of legal moves changes at each step of the game and Figure 3b illustrates

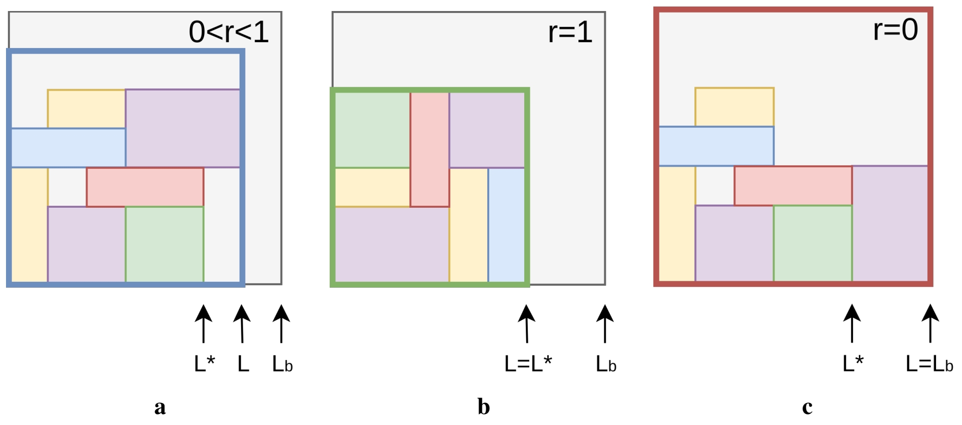


Figure 1: **MDP-reward.** Figure 1a shows a sub-optimal solution with all items placed. The bounding box side, L, lies between the optimal side, L^*, and the bin side, L_b. The exact reward, $0 < r < 1$, is given by Equation 1. Figure 1b shows an optimal solution with a bounding box of minimal side,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

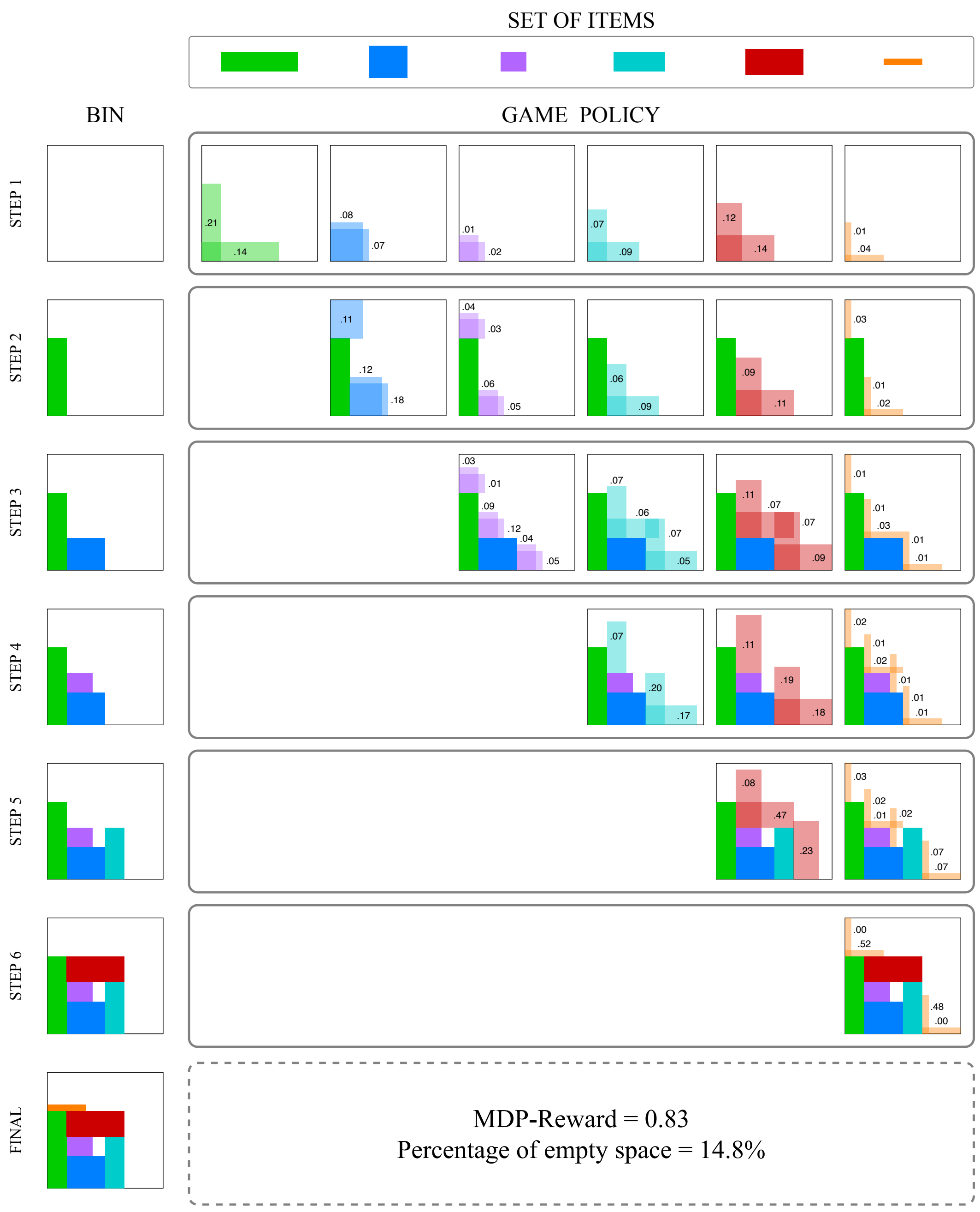


Figure 2: **Progressive construction of a solution to a 6-item problem.** The action space is depicted with the associated game policy, and the action executed is selected greedily with respect to this policy.

how the number of possible games grows with the number of items. A conservative upper bound for the number of possible games is:

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

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 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 the best possible solution.

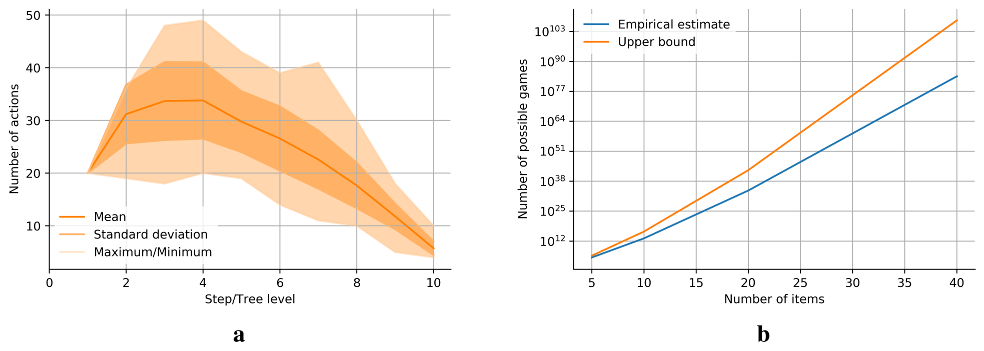


Figure 3: **Bin packing complexity.** The values shown are estimated empirically by playing 50 random games and taking the average of the number of possible next actions (Figure 3a), and the number of possible games in the final MCTS tree (Figure 3b). Figure 3a describes the number of available actions after each item is placed in a problem with 10 items. Figure 3b 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 2.

4 The Ranked Reward Algorithm

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 1.

4.1 Ranked Rewards

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 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 used to compute the threshold 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 [17]. The 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., $(p, v) = f_\theta(s)$. Finally, the neural network is updated to minimize the cross-entropy loss between predicted ranked reward v and true ranked reward $\tilde{r}$, as well as the cross-entropy loss between the neural network policy p and the MCTS-based improved policy π, plus an L^2 regularization term.

4.2 Neural Network Architecture

The neural network architecture used in this work has been 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.

Algorithm 1: The R2 (Ranked Reward) 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, ..., $M$ do
        Sample an initial state $s_0$
        for t = 0, ..., $N$-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
```

Our 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 4a and 4b. Similarly, each item is represented by two binary feature planes, one for each rotation, as illustrated in Figures 4c and 4d. 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[1] followed by separate policy and value heads representing 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.

5 Experiments and Results

In order to evaluate the effectiveness of our approach, we considered the 2D bin packing problem described above, with ten items and a bin of side $L_b = 20$. Problem instances were created by progressively and randomly dividing a quarter of the bin area into items to produce an optimal solution with no empty spaces and side $L^* = 10$.

For each experiment, we ran the R2 algorithm for 200 iterations[2]. At each iteration, $M = 50$ new games were randomly generated. The neural network parameters were optimized using the Adam

[1]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 [6].

[2]Each experiment was run using an NVIDIA V100 GPU for the training and inference of the neural network, and an Intel Xeon to execute the search algorithm.

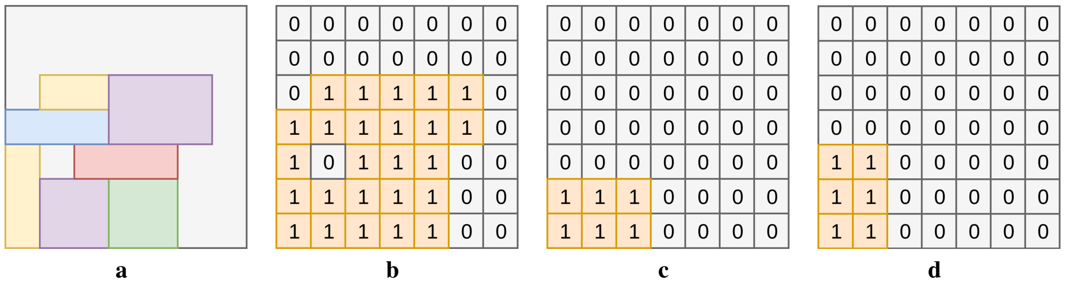


Figure 4: **Visual State Representation.** Figure 4a shows seven items placed in the bin with $L_b = 7$. Figure 4b shows the binary matrix indicating the presence, 1, or absence, 0, of items in the bin. Figure 4c shows the first binary representation of an item of size 3×2, without rotation. Figure 4d shows the second binary representation of the same item when rotated. Both options are included in the problem state space.

optimizer [9] with a learning rate of 0.0005. Mini-batches of size $b = 64$ were sampled from a buffer of size $25{,}000$. At each step of a game, MCTS used $S = 300$ simulations to select moves. The algorithm was then evaluated on a set of 50 new games. To ensure the diversity during training, actions were sampled from $\pi(\cdot|s)$, whereas, during evaluation, they were selected greedily, i.e. the action with the largest visit count was executed. Since the problem is deterministic, when evaluating the algorithm, the tree search returned the sequence of actions leading to the best game outcome reached during the entire search rather than the best outcome from the last 300 simulations only.

5.1 Ranked Reward Performance

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[3] in Figure 5.

The results show that R2 outperforms its rank-free counterpart. The latter quickly plateaued at a value close to 0.88, whereas R2 surpassed that, with the 75% threshold version reaching 0.97. This represents an improvement of 10%, with 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%)	$\mathbf{0.97}$ **($\pm \mathbf{0.05}$)**	$\mathbf{1.00}$	$\mathbf{72\%}$
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.**

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 [5]; the *Lego* heuristic search algorithm [7]; two successful reinforcement learning methods: the *asynchronous ad-*

[3]The learning curve for the 90% reward threshold is not included in Figure 5 to improve the readability of the graph.

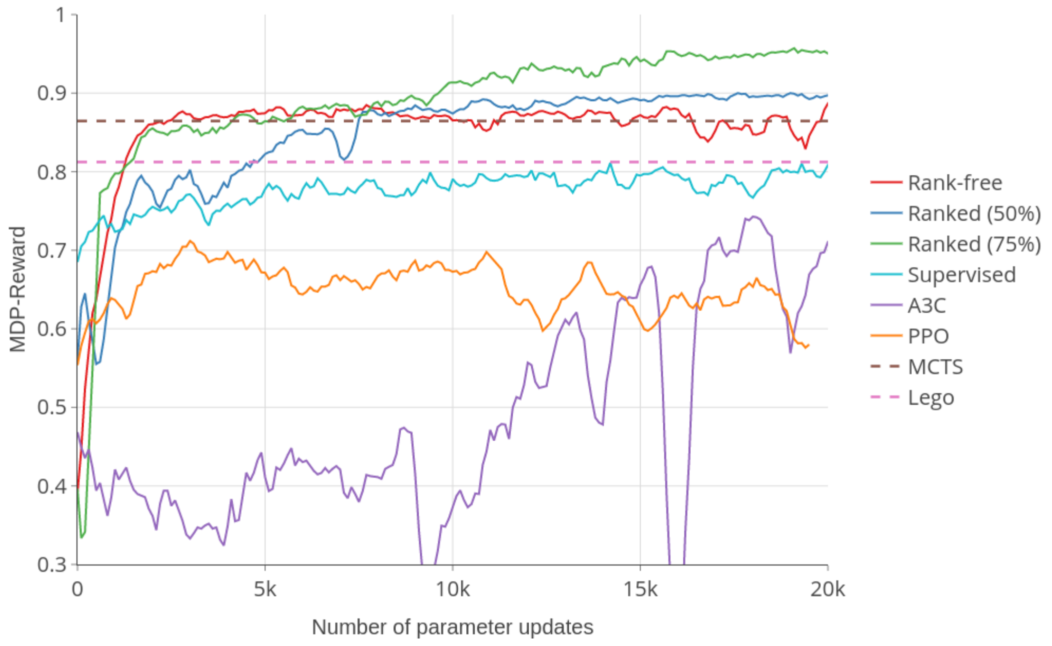


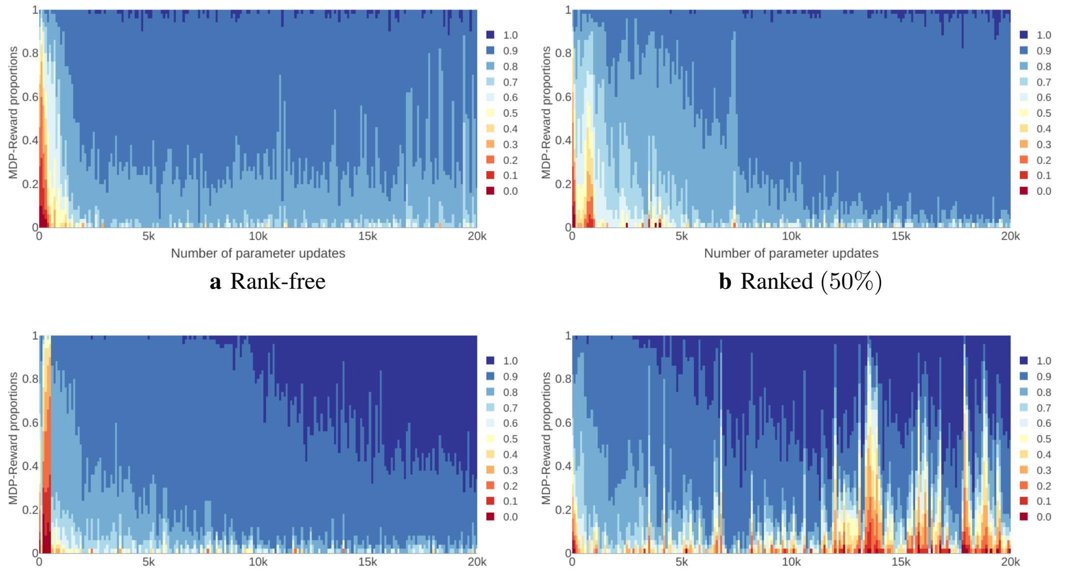
Figure 5: **Evolution of mean MDP-reward.**

vantage actor-critic (A3C) algorithm [10] and the *proximal policy optimization* (PPO) algorithm [15]; 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 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 [10] and PPO algorithms [15], and adapted the implementations provided in the Ray package [12] 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 $\{a_i\}_{0\le i\le N-1}$ 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 5. 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%) **d** Ranked (90%)

Figure 6: **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.

5.2 The Effects of Ranking Thresholds on Learning

The performance level and the learning behavior of R2 are both sensitive to the α-percentile value. Figure 6 illustrates the effect of different reward thresholds on the distribution of rewards received across 50 games.

The impact of the percentile α on the performance follows our intuitive understanding of human learning. Setting the threshold at 50% is equivalent to making the agent play against an opponent of exactly the same level, as it has a predetermined 50% chance of winning. Increasing the percentile value corresponds to improving the opponent's level, as it makes it harder to obtain a reward of 1. In our context, when the percentile changes from 50% to 75%, the probability of winning falls to 25%. Interestingly, this threshold produces a faster learning and attains a better final level of performance. Taking inspiration from sports, we can expect that the learning process would be improved by playing against a slightly stronger adversary, because it would push learners to the limit of their abilities.

In general, higher thresholds lead to faster learning, i.e. the proportion of high-reward games increases faster. However, Figure 6d shows that, for a threshold of 90%, large amounts of low-reward games re-appear, especially during the last $10{,}000$ parameter updates. These instabilities result in weaker final performance despite some good short-lived peaks. To explain this, we can hypothesize that if the opponent is too strong, the learning process will suffer because the agent can very rarely affect the outcome of the game even when it manages to play significantly better than its current mean performance level.

6 Discussion and Future Work

The results presented above show that R2 outperforms the selected alternatives on the given problem. Yet, these results have limitations that we discuss here.

First, our implementation of the 2D bin packing problem only produces problems with known optimal solutions that do not contain any empty space, i.e., square packings with no gaps. Even though this helps us to evaluate the algorithm's performance, it introduces an undesirable bias. Future research should evaluate the algorithm on a wider range of problems, for which the optimal solution is unknown and not necessarily square.

Secondly, our results are presented for instances of ten items only. Although this represents a problem space of 10^{16} possible solutions, more than this can be handled by current optimizers used in industry, such as the IBM CPLEX Optimizer[4]. Therefore, experimenting on larger problems is a necessary step towards demonstrating the superiority of R2 over the other algorithms from Section 5.1.

Furthermore, regarding the scalability of our approach, the capacity of our network can be increased at an acceptable computational cost. In particular, we only use two residual blocks for the policy-value network which is significantly less than what was used to master the game of Go [17]. A more thorough exploration of the threshold space may also improve performance and scalability.

7 Conclusion

In this paper, we introduced the R2 algorithm and compared its performance to other algorithms on a bin packing problem of ten items. By ranking the rewards obtained over recent games, R2 provides a threshold-based relative performance metric. This enables it to reproduce the benefits of self-play for single-player games, removing the requirement for training data and providing a well-suited adversary throughout the learning process.

Consequently, R2 outperforms the selected alternatives as well as its rank-free counterpart, improving on the performance of the best alternative, plain MCTS, by more than 10% when using a threshold value of 75%. An analysis of the effects of different percentiles α has shown that higher thresholds perform better up to a point after which learning becomes unstable and performance decreases.

The R2 algorithm is potentially applicable to a wide range of optimization tasks, though it has so far been used only on the bin packing. In the future, we will consider other optimization problems to further evaluate its effectiveness.

[4]https://www.ibm.com/analytics/cplex-optimizer.

References

[1] 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.

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

[3] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] Gerald Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[19] 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.

[20] L. A. Wolsey. *Integer programming*. Wiley-Interscience, New York, NY, USA, 1998.