Learning Self-Game-Play Agents for Combinatorial Optimization Problems

Ruiyang Xu
Khoury College of Computer Sciences
Northeastern University
Boston, MA
ruiyang@ccs.neu.edu

Karl Lieberherr
Khoury College of Computer Sciences
Northeastern University
Boston, MA
lieber@ccs.neu.edu

ABSTRACT

Recent progress in reinforcement learning (RL) using self-game-play has shown remarkable performance on several board games (e.g., Chess and Go) as well as video games (e.g., Atari games and Dota2). It is plausible to consider that RL, starting from zero knowledge, might be able to gradually approximate a winning strategy after a certain amount of training. In this paper, we explore neural Monte-Carlo-Tree-Search (neural MCTS), an RL algorithm which has been applied successfully by DeepMind to play Go and Chess at a super-human level. We try to leverage the computational power of neural MCTS to solve a class of combinatorial optimization problems. Following the idea of Hintikka's Game-Theoretical Semantics, we propose the Zermelo Gamification (ZG) to transform specific combinatorial optimization problems into Zermelo games whose winning strategies correspond to the solutions of the original optimization problem. A specially designed neural MCTS algorithm is also provided to train Zermelo game play agents. We use a prototype problem for which the ground-truth policy is efficiently computable to demonstrate that ZG is promising.

KEYWORDS

Reinforcement Learning; neural MCTS; Self-game-play; Combinatorial Optimization; Tabula rasa

1 INTRODUCTION

The past several years have witnessed the progress and success of reinforcement learning (RL) in the field of game-play. The combination of classical RL algorithms with newly developed deep learning techniques gives stunning performance on both traditional simple Atari video games [10] and modern complex RTS games (like Dota2 [12]), and even certain hard board games like Go and Chess [17]. One common but outstanding feature of those learning algorithms is the tabula-rasa style of learning. In terms of RL, all those algorithms are model-free[1] and learn to play the game with zero knowledge (except the game rules) in the beginning. Such tabula-rasa learning can be regarded as an approach towards a general artificial intelligence.

Although there are lots of achievements in game, there is little literature on how to apply those techniques to general problems in other domains. It is tempting to see whether those game-play agents' superhuman capability can be used to solve problems in other realms. In this work, we transform a family of combinatorial optimization problems into games via a process called Zermelo Gamification, so that an AlphaZero style (i.e., neural MCTS [15, 17]) game-play agent can be leveraged to play the transformed game and solve the original problem. Our experiment shows that the two competitive agents gradually, but with setbacks, improve and jointly arrive at the optimal strategy. The tabula-rasa learning converges and solves a non-trivial problem, although the Zermelo game is fundamentally different from Go and Chess. The trained game-play agent can be used to approximate[2] the solution (or show the non-existence of a solution) of the original problem through competitions against itself based on the learned strategy.

We make three main contributions: 1. We introduce the Zermelo Gamification a way to transform combinatorial problems to Zermelo games using a variant of Hintikka's Game-Theoretical Semantics [6]; 2. We implemented a modification of the neural MCTS algorithm[3] designed explicitly for those Zermelo games; 3. We experiment with a prototype problem (i.e., *HSR*). Our result shows that, for problems under a certain size, the trained agent does find the optimal strategy, hence solving the original optimization problem in a tabula-rasa style.

The remainder of this paper is organized as follows. Section 2 presents essential preliminaries on neural MCTS and combinatorial optimization problems which we are interested in. Section 3 introduces Zermelo game and a general way to transform the given type of combinatorial optimization problems into Zermelo games, where we specifically discuss our prototype problem *HSR*. Section 4 gives our correctness measurement and presents experimental results. 5 and 8 made a discussion and conclusions.

2 PRELIMINARIES

2.1 Monte Carlo Tree Search

The PUCT (Predictor + UCT) algorithm implemented in AlphaZero [16, 17] is essentially a neural MCTS algorithm which uses PUCB Predictor + UCB [11] as its confidence upper bound [1, 8] and uses the neural prediction $P_\phi(a|s)$ as the predictor. The algorithm usually proceeds through 4 phases during each iteration:

(1) SELECT: At the beginning of each iteration, the algorithm selects a path from the root (current game state) to a leaf (either a terminal state or an unvisited state) in the tree according to the PUCB (see [14] for detailed explanation for terms used in the formula). Specifically, suppose the root is

[1]Considering the given problem as an MDP (Markov Decision Process), the learning algorithm doesn't have to know in advance the transition probabilities and rewards after each action is taken

[2]For problems with small sizes, one can achieve an optimal solution by providing the learning algorithm enough computing resources.

[3]Our implementation is based on an open-source, lightweight framework: AlphaZero-General, https://github.com/suragnair/alpha-zero-general

s_0, we have [4]:

$$a_{i-1} = \arg\max_a \left[Q(s_{i-1}, a) + cP_\phi(a|s_{i-1}) \frac{\sqrt{\sum_{a'} N(s_{i-1}, a')}}{N(s_{i-1}, a) + 1} \right]$$

$$Q(s_{i-1}, a) = \frac{W(s_{i-1}, a)}{N(s_{i-1}, a) + 1}$$

$$s_i = next(s_{i-1}, a_{i-1})$$

(2) EXPAND: Once the select phase ends at a non-terminal leaf, the leaf will be fully expanded and marked as an internal node of the current tree. All its children nodes will be considered as leaf nodes during next iteration of selection.
(3) ROLL-OUT: Normally, starting from the expanded leaf node chosen from previous phases, the MCTS algorithm uses a random policy to roll out the rest of the game [4]. The algorithm simulates the actions of each player randomly until it arrives at a terminal state which means the game has ended. The result of the game (winning information or ending score) is then used by the algorithm as a result evaluation for the expanded leaf node.
However, a random roll-out usually becomes time-consuming when the tree is deep. A neural MCTS algorithm, instead, uses a neural network V_ϕ to predict the result evaluation so that the algorithm saves the time on rolling out.
(4) BACKUP: This is the last phase of an iteration where the algorithm recursively backs-up the result evaluation in the tree edges. Specifically, suppose the path found in the Select phase is $\{(s_0, a_0), (s_1, a_1), ...(s_{l-1}, a_{l-1}), (s_l, \_)\}$. then for each edge (s_i, a_i) in the path, we update the statistics as:

$$W^{new}(s_i, a_i) = W^{old}(s_i, a_i) + V_\phi(s_l)$$

$$N^{new}(s_i, a_i) = N^{old}(s_i, a_i) + 1$$

However, in practice, considering the +1 smoothing in the expression of Q, the following updates are actually applied:

$$Q^{new}(s_i, a_i) = \frac{Q^{old}(s_i, a_i) \times N^{old}(s_i, a_i) + V_\phi(s_l)}{N^{old}(s_i, a_i) + 1}$$

$$N^{new}(s_i, a_i) = N^{old}(s_i, a_i) + 1$$

Once the given number of iterations has been reached, the algorithm returns a vector of action probabilities of the current state (root s_0). And each action probability is computed as $\pi(a|s_0) = \frac{N(s_0, a)}{\sum_{a'} N(s_0, a')}$. The real action played by the MCTS is then sampled from the action probability vector π. In this way, MCTS simulates the action for each player alternately until the game ends, this process is called MCTS simulation (self-play).

[4]Theoretically, the exploratory term should be $\sqrt{\frac{\sum_{a'} N(s_{i-1}, a')}{N(s_{i-1}, a)+1}}$, however, the AlphaZero used the variant $\frac{\sqrt{\sum_{a'} N(s_{i-1}, a')}}{N(s_{i-1}, a)+1}$ without any explanation. We tried both in our implementation, and it turns out that the AlphaZero one performs much better.

2.2 Combinatorial Optimization Problems

The combinatorial optimization problems studied in this paper can be described with the following logic statement:

$$\exists n : \{G(n) \wedge (\forall n' > n \neg G(n'))\}$$

$$G(n) := \forall x \exists y : \{F(x, y; n)\}$$

or

$$G(n) := \exists y \forall x : \{F(x, y; n)\}$$

In this statement, n is a natural number and x, y can be any instances depending on the concrete problem. F is a predicate on n, x, y. Hence the logic statement above essentially means that there is a maximum number n such that for all x, some y can be found so that the predicate $F(x, y; n)$ is true. Formulating those problems as interpreted logic statements is crucial to transforming them into games (Hintikka [6]). In the next section, we will introduce our gamification method in details.

3 ZERMELO GAMIFICATION

3.1 General Formulation

We introduce the Zermelo Gamification (ZG) to transform a combinatorial optimization problem into a Zermelo game that is fit for being used by a specially designed neural MCTS algorithm to find a winning strategy. The winning strategy can be used to find a solution to the original combinatorial problem. We will illustrate the Zermelo Gamification by deriving a prototype game: *HSR*.

A Zermelo game is defined to be a two-player, finite, and perfect information game with only one winner and loser, and during the game, players move alternately (i.e., no simultaneous move). Leveraging the logic statement (see section 2.2) of the problem, the Zermelo game is built on the Game-Theoretical Semantic approach (Hintikka [6]). We introduce two roles: the Proponent (P), who claims that the statement is true, and the Opponent (OP), who argues that the statement is false. The original problem can be solved if and only if the P can propose some optimal number n so that a perfect OP cannot refute it. To understand the game mechanism, let's recall the logic statement in section 2.2, which implies the following Zermelo game (Fig. 1):

(1) Proposal Phase: in the initial phase of the Zermelo game player P will propose a number n. Then the player OP will decide whether to accept this n, or reject it. OP will make his decision based on the logic statement: $A \wedge B, A := G(n), B := \forall n' > n \neg G(n')$. Specifically, the OP tries to refute the P by attacking either on the statement A or B. The OP will accept n proposed by the P if she confirms $A = False$. The OP will reject n if she is unable to confirm $A = False$. In this case, The OP treats n as non-optimal, and proposes a new $n' > n$ (in practice, for integer n, we take $n' = n + 1$) which makes $B = False$. To put it in another way, $B = False$ implies $\neg B = True$ which also means that the OP claims $G(n')$ holds. Therefore, the rejection can be regarded as a role-flip between the two players. To make the Zermelo non-trivial, in the following game, we require that the P has to accept the new n' and tries to figure out the corresponding y to defeat the OP. Notice that since this is an adversarial game, the OP will never agree with the P (namely, the OP

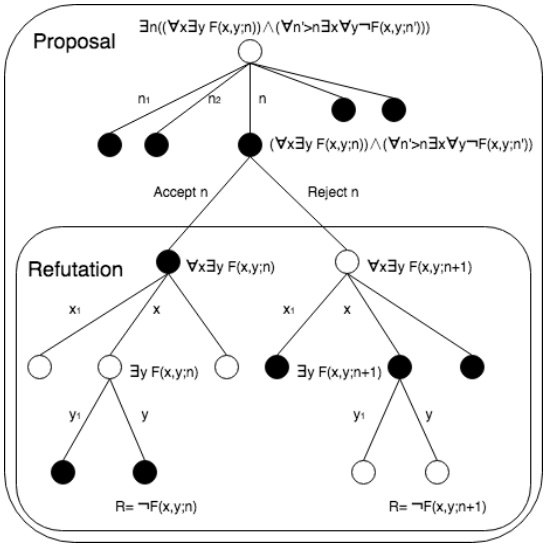

Figure 1: An overall Zermelo game where white nodes stand for the P's turn and black nodes stand for the OP's turn. A role-flip happened after OP's rejecting of n. The refutation game can be treated uniformly where, depending on the order of the quantifiers, the OP takes the first move. The OP wins if and only if the P fails to find any y to make $F(x, y; n)$ holds, hence the game result $R = \neg F(x, y; n)$.

will either decide that the n is too small or too large because the OP has to regard the move made by the P as incorrect). Therefore, the OP is in a dilemma when the P is perfect, i.e., the P chooses the optimal n.

(2) Refutation Phase: it is the phase where the two players search for evidence and construct strategies to attack each other or defend themselves. Generally speaking, regardless of the role-flip, we can treat the refutation game uniformly: the P claims $G(n)$ holds for some n, the OP will refute this claim by giving some instances of x (for existential quantifier) so that $\neg G(n)$ holds. If the P successfully figures out the exceptional y (for universal quantifier) which makes $F(x, y; n)$ hold, the OP loses the game; otherwise, the P loses.

The player who takes the first move is decided by order of the quantifiers, namely, for $G(n) := \exists x \forall y : \{F(x, y; n)\}$ the P will take the first move; for $G(n) := \forall y \exists x : \{F(x, y; n)\}$ the OP will take the first move. The game result is evaluated by the truth value of $F(x, y; n)$, specifically, if the P takes the last move then she wins when $F(x, y; n)$ holds; otherwise, if the OP makes the last move then she wins when $F(x, y; n)$ doesn't hold. It should be noticed that the OP is in a dilemma when the P is perfect.

3.2 HSR Problem

In this section, we first introduce our prototype, the Highest Safe Rung (*HSR*) problem. Then, we will see how to perform Zermelo gamification on it.

The *HSR* problem can be described as follows: consider throwing jars from a specific rung of a ladder. The jars could either break or not. If a jar is unbroken during a test, it can be used next time. A highest safe rung is a rung that for any test performed above it, the jar will break. Given k identical jars and q test chances to throw those jars, what is the largest number of rungs a ladder can have so that there is always a strategy to locate the highest safe rung with at most k jars and q tests?

To formulate *HSR* problem in predicate logic, we utilize the recursive property. Notice that, after performing a test, depends on whether the jar is broken or not, the highest safe rung should only be located either above the current testing level, or below or equal to the current testing level. This fact means that the next testing level should only be located in the upper partial ladder or the lower partial ladder. Therefore, the original problem can be divided into two sub-problems. We introduce the predicate $G_{k,q}(n)$ which means there is a strategy to find the highest safe rung on an n-level ladder with at most k jars and q tests. Specifically, using the recursive property we have mentioned, $G_{k,q}(n)$ can be written as:

$$G_{k,q}(n) = \exists\ 0 < m \leq n\ : \{G_{k-1,q-1}(m-1) \wedge G_{k,q-1}(n-m)\}$$

$$G_{k,q}(0) = True, G_{0,q}(n) = False, G_{k,0}(n) = False, G_{0,0}(n) = False$$

$$n > 0, k > 0, q > 0$$

This formula can be interpreted as following: if there is a strategy to locate the highest safe rung on an n-level ladder, then it must tell you a testing level m so that, no matter the jar breaks or not, the strategy can still lead you to find the highest safe rung in the following sub-problems. More specifically, for sub-problems, we have $G_{k-1,q-1}(m-1)$ if the jar breaks, that means we only have $k-1$ jars and $q-1$ tests left to locate the highest safe rung in the lower partial ladder (which has $m-1$ levels). Similarly, $G_{k,q-1}(n-m)$ for upper partial ladder. Therefore, the problem is solved recursively, and until there is no ladder left, which means the highest safe rung has been located, or there is no jars/tests left, which means one has failed to locate the highest safe rung. With the notation of $G_{k,q}(n)$, the *HSR* problem now can be formulated as:

$$HSR_{k,q} = \exists n : \{G_{k,q}(n) \wedge (\forall n' > n\ \neg G_{k,q}(n'))\}$$

Next, we show how to perform Zermelo gamification on the *HSR* problem. Notice that the expression of $G_{k,q}(n)$ is slightly different with the ones used in section 2.2: there is no universal quantifiers in the expression. To introduce the universal quantifier, we regard the environment as another player who plays against the tester so that, after each testing being performed, the environment will tell the tester whether the jar is broken or not. In this way, locating the highest safe can be formulated as following:

$$G_{k,q}(n) = \exists m \leq n\ \forall a \in \{\text{“break”}, \text{“not break”}\} : \{F(m, a; n)\}$$

$$F(m, a; n) = \begin{cases} \text{True, if } n = 0 \\ \text{False, if } n > 0 \wedge (k = 0 \vee q = 0) \\ G_{k-1,q-1}(m-1), \text{ if } a = \text{“break”} \\ G_{k,q-1}(n-m), \text{ if } a = \text{“not break”} \end{cases}$$

Now, with the formula above, one can perform the standard Zermelo gamification (section 3.1) to get corresponding Zermelo game (Fig. 3). Briefly speaking, the tester now becomes the P player in the game, and the environment becomes OP. In the proposal phase, P will propose a number n for which P thinks it is the largest number of levels a ladder can have so that she can locate any

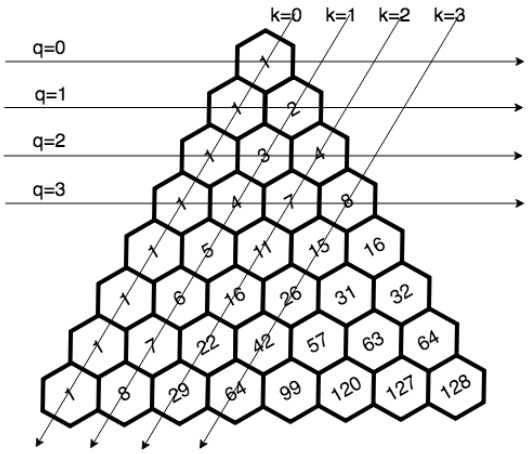

Figure 2: Theoretical values for maximum n in *HSR* problem with given k, q, which can be represented as a Bernoulli's Triangle.

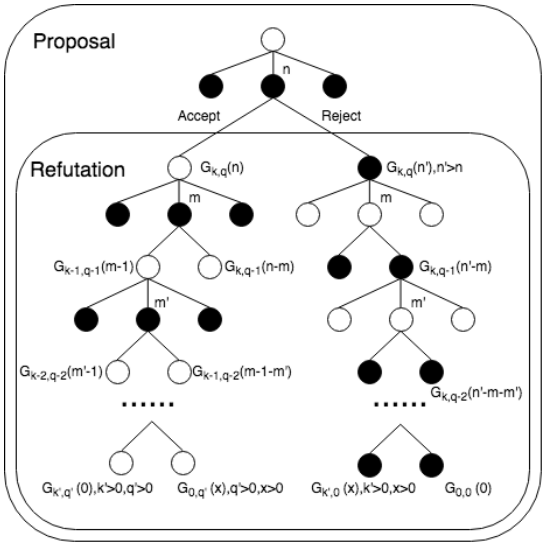

Figure 3: The Zermelo gamification of *HSR* problem. The game recursively played between two players until the highest safe rung being loacted or all resources have been used up.

highest safe rung using at most k jars and q tests. The OP will decide whether to accept or reject this proposal by judging whether n is too small or too large. In the refutation phase, P and OP will give a sequence of testing levels and testing results alternately, until the game ends. In this game, both P and OP will improve their strategy during the game so that they always play adversarial against each other and adjust one's strategy based on the reaction from the other one.

It should be mentioned that due to *HSR* problem, as a prototype to test our idea, itself is not a hard problem, the solution for the *HSR* problem can be computed and represented efficiently with a Bernoulli's Triangle (Fig. 2). We use the notation $N(k, q)$ to represent the solution for *HSR* problem given k jars and q tests. In other words, $G_{k,q}(N(k, q)) \wedge (\forall n' > N(k, q)\ \neg G_{k,q}(n'))$ always holds.

4 EXPERIMENT

4.1 Neural MCTS implementation

In this section, we will discuss our neural MCTS implementation on the *HSR* game. Since the Zermelo game has two phases and the learning tasks are quite different between these two phases, we applied two independent neural networks to learn the proposal game and refutation game respectively. The neural MCTS will access the first neural network during the proposal game and then the second neural network during the refutation game. There are also two independent replay buffers which store the self-play information generated from each phase, respectively.

Our neural network consists of four layers of 1-D convolution neural networks and two dense layers. The input is a tuple (k, q, n, m, r) where k, q are resources, n is the number of rungs on the current ladder, m is the testing point, and r indicates the current player. The output of the neural network consists of two vectors of probabilities on the action space for each player as well as a scalar as the game result evaluation.

During each iteration of the learning process, there are three phases: 1. 100 episodes of self-play will be executed through a neural MCTS using the current neural network. Data generated during self-play will be stored and used for the next phase. 2. the neural networks will be trained with the data stored in the replay buffer. And 3. the newly trained neural network and the previous old neural network are put into a competition to play with each other. During the competition phase, the new neural network will first play as the OP for 20 rounds, and then it will play as the P for another 20 rounds. We collect the correctness data for both of the neural networks during each iteration. [5]

We shall mention that since it is highly time-consuming to run a complete Zermelo game on our machines, to save time and as a proof of concept, we only run the entire game for $k = 7, q = 7$ and $n \in [1...130]$. Nevertheless, since the refutation game, once n is given, can be treated independently from the proposal game, we run the experiment on refutation games for various parameters.

4.2 Correctness Measurement

Informally, an action is correct if it preserves a winning position. It is straightforward to define the correct actions using the Bernoulli Triangle (Fig. 2).

4.2.1 P's correctness. Given (k, q, n), correct actions exist only if $n \leq N(k, q)$. In this case, all testing points in the range $[n - N(k, q - 1), N(k - 1, q - 1)]$ are acceptable. Otherwise, there is no correct action.

4.2.2 OP's correctness. Given (q, k, n, m), When $n > N(k, q)$, any action is regarded as correct if $N(k-1, q-1) \leq m \leq n-N(k, q-1)$, otherwise, the OP should take "not break" if $m > n - N(k, q - 1)$ and "break' if $m < N(k - 1, q - 1)$; when $n \leq N(k, q)$, the OP should

[5] It should be mentioned that the arena phase can be used only to obtain experimental data while the model can be continuously updated without the arena phase, as AlphaZero.

take the action "not break" if $m < n - N(k, q-1)$ and take action "break" if $m > N(k-1, q-1)$. Otherwise, there is no correct action.

4.3 Complete Game

In this experiment, we run a full Zermelo game under the given resources $k = 7, q = 7$. Since there are two neural networks which learn the proposal game and the refutation game respectively, we measure the correctness separately: Fig. 4 shows the ratio of correctness for each player during the proposal game. And Fig. 5 shows the ratio of correctness during the refutation game. The horizontal axis is the number of iterations, and it can be seen that the correctness converges extremely slow (80 iterations). It is because, for $k = 7, q = 7$, the game is relatively complex and the neural MCTS needs more time to find the optimal policy.

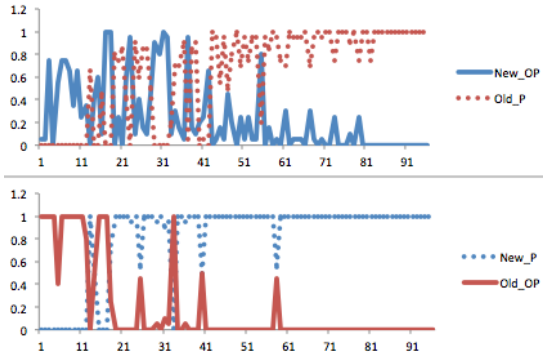

Figure 4: Correctness ratio measured for the proposal game on $k = 7, q = 7$. The legend "New_OP" means that the newly trained neural network plays as an OP; "Old_P" means that the previously trained neural network plays as a P. The same for the following graphs.

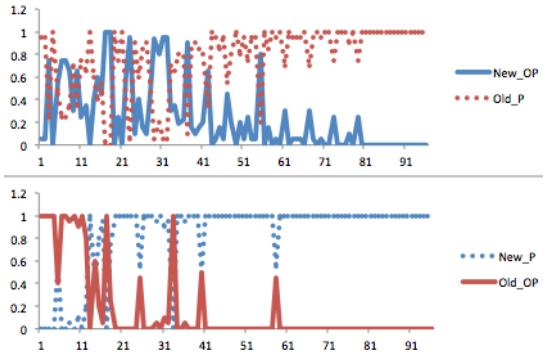

Figure 5: Correctness ratio measured for the refutation game on $k = 7, q = 7$.

4.4 Refutation Game

To test our method further, we focus our experiment only on refutation games with a given n. We first run the experiment on an extreme case where $k = 7, q = 7$. Using the Bernoulli Triangle (Fig. 2), we know that $N(7, 7) = 2^7$. We set $n = N(k, q)$ so that the learning process will converge when the P has figured out the optimal winning strategy which is binary search: namely, the first testing point is 2^6 then 2^5, 2^4 and so on. Fig. 6 verified that the result is as expected. Then, to study the behavior of our agents under extreme conditions, we run the same experiment on a resource-insufficient case where we keep k, q unchanged and set $n = N(k, q) + 1$. In this case, theoretically, no solution exists. Fig. 7, again, verified our expectation and one can see that the P can never find any winning strategy no matter how many iterations it has learned.

In later experiments, we have also tested our method in two more general cases where $k = 3, q = 7$ for $n = N(3, 7)$ (Fig. 8) and $n = N(3, 7) - 1$ (Fig. 9). All experimental results are conforming to the ground-truth as expected.

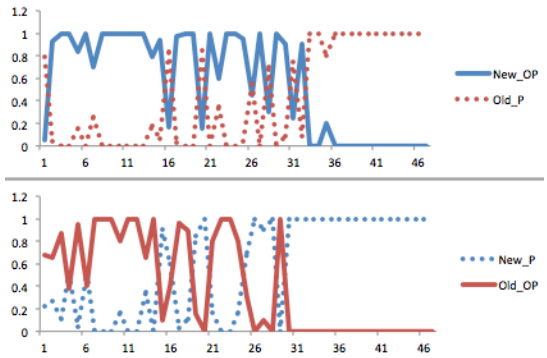

Figure 6: Refutation game on $k = 7, q = 7, n = 128$

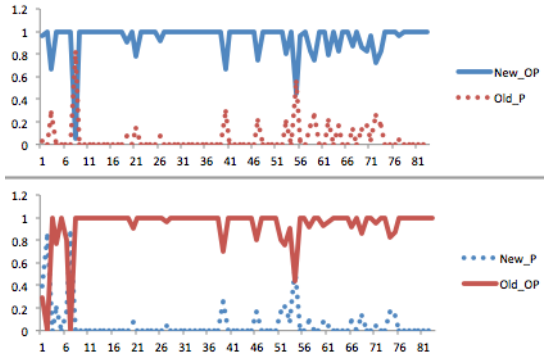

Figure 7: Refutation game on $k = 7, q = 7, n = 129$. Notice that in this game, the P is doomed for there is no winning strategy exists.

The $HSR_{k,q}$ game is also intrinsically asymmetric in terms of training/learning because the OP always takes the last step before the end of the game. This fact makes the game harder to learn for the P. Specifically, considering all possible consequences (in the view of the P) of the last action, there are only three cases: win-win, win-lose, and lose-lose. The OP will lose the game if and only if the consequence is win-win. If the portion of such type of result

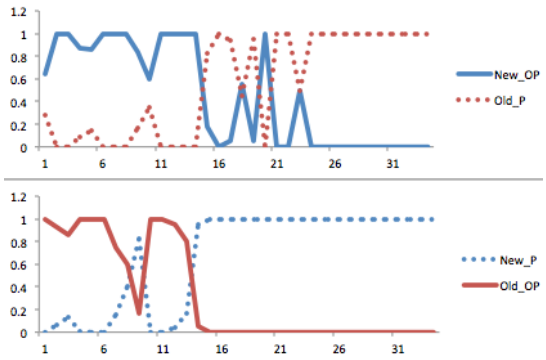

Figure 8: Refutation game on $k = 3, q = 7, n = 64$

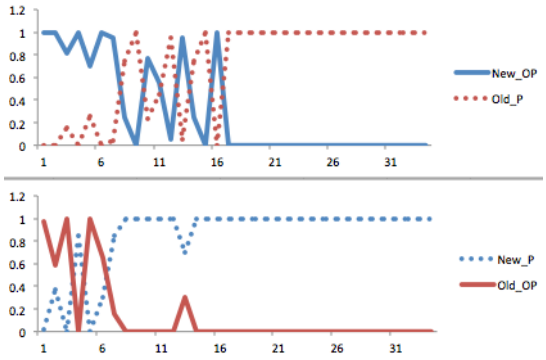

Figure 9: Refutation game on $k = 3, q = 7, n = 63$

is tiny, then the OP could exclusively focus on learning the last step while ignoring other steps. However, the P has to learn every step to avoid possible paths which lead him to either win-lose or lose-lose, which, theoretically, are more frequently encountered in the end game.

5 DISCUSSION

5.1 State Space Coverage

Neural MCTS is capable of handling a large state space [17]. It is necessary for such an algorithm to search only a small portion of the state space and make the decisions on those limited observations. To measure the state space coverage ratio, we recorded the number of states accessed during the experiment, Specifically, in the refutation game $k = 7, q = 7, n = 128$, we count the total number of states accessed during each self-play, and we compute the average state accessed for all 100 self-plays in each iteration. It can be seen in Fig. 10 that the maximum number of state accessed is roughly 1500 or 35% (we have also computed the total number of possible states in this game, which is 4257). As indicated in Fig. 10, at the beginning of the learning, neural MCTS accessed a large number of states, however, once the learning converged, it looked at a few numbers of state and pruned all other irrelevant states. It can also be seen that the coverage ratio will bounce back sometimes, which is due to the exploration during self-play. Our experimental result indicates that changes in coverage ratio might be evidence of adaptive self-pruning in a neural MCTS algorithm, which can be regarded as a justification of its capability of handling large state spaces.

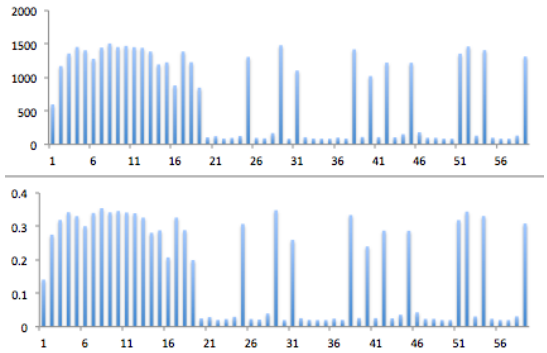

Figure 10: States accessed (top) and state space coverage ratio (bottom) during self-play for each iteration, in refutation game $k = 7, q = 7, n = 128$.

5.2 Perfectness

This discussion is in the context where the ground truth is known. Since the correct solution is derived from the optimal policy, it is important to question whether the players are perfect after the training converged (i.e., the correctness of each player becomes flat without further changes). The experimental result shows that, after convergence, for a problem which has a solution, the P always keeps 100% correctness while the OP rests at 0%. On the other hand, for a problem which has no solution, the opposite happens. Notice that a consistent 100% correctness indicates that the player is perfect because, otherwise, the other player will quickly find out the weakness in her adversary. However, there is no guarantee that a consistent 0% correctness player is also perfect. Since after one player becoming perfect, the other one will always lose no matter what decisions have been made. In this case, all game results are the same, and there is no reward to be gained from further training. Even though, from our experimental observation, the doomed loser is still a robust sub-optimal player after being competitively trained from tabula rasa. The question of when to stop training and how to guarantee that both P and OP become perfect are further topics for future research.

5.3 Asymmetry

One can observe some asymmetry in the charts we presented in section 4, and notice that it is always the case that during the beginning iterations the OP is dominating until the P has gained enough experience and learned enough knowledge. Two facts cause this asymmetry: 1. the action space of the P is entirely different from the one of the OP. 2. the OP always takes the last step before the end of the game. These two facts make the game harder to learn for the P but easier for the OP.

5.4 Limitations

Our neural MCTS algorithm is time-consuming. It usually takes a large amount of time to converge and we have to use more resources (more CPUs, distributed parallel computing) to make it run faster. That's the reason why we don't experience the amazing performance of AlphaZero for Chess and Go on huge game trees. Another limitation is that to learn the correct action in a discrete action space; the neural MCTS algorithm has to explore all possible actions before learning the correct action. This fact makes the action space a limitation to MCTS like algorithms: the larger the action space, the lower the efficiency of the algorithm.

6 FUTURE WORK

As we have mentioned, *HSR* is only a prototype for us to apply neural MCTS to problems in other domains. It is still unknown to us whether neural MCTS can be used to solve more complex problems. Our next plan is to try neural MCTS on Quantified Boolean Formulas (QBFs), which is considered to be PSPACE complexity. Since it is quite a natural way to turn a solving process of a QBF into gameplay, we think it could be another touchstone to the capability of neural MCTS. However, since the fundamental symmetry among those QBFs, we plan to turn QBFs into graphs and use a graph neural network to embed them so that symmetry would not be an issue.

7 RELATED WORK

Imagination-Augmented Agents (I2As [19]), an algorithm invented by DeepMind, is used to handle complex games with sparse rewards. Although the algorithm has performed well, it is not model-free. Namely, one has to train, in a supervised way, an imperfect but adequate model first, then use that model to boost the learning process of a regular model-free agent. Even though I2As, along with a trained model, can solve games like Sokoban to some level, I2As can hardly be applied to games where even the training data is limited and hard to generate or label.

By formulating a combinatorial problem as an MDP, Ranked reward [9] binarized the final reward of an MDP based on a certain threshold, and improves the threshold after each training episode so that the performance is forced to increase during each iteration. However, this method can hardly be applied to problems that already have a binary reward (such as a zero-sum game with reward $-1, 1$). Even though, the idea that improves the performance threshold after each learning iteration has also been used in AlphaZero as well as our implementation.

Pointer networks [18] have been shown to solve specific combinatorial NP problems with a limited size. The algorithm is based on supervised attention learning on a sequence to sequence RNN. However, due to its high dependency on the quality of data labels (which could be very expensive to obtain), Bello et al. [3] improved the method of [18] to the RL style. Specifically, they applied actor-critic learning where the actor is the original pointer network, but the critic is a simple REINFORCE [20] style policy gradient. Their result shows a significant improvement in performance. However, this approach can only be applied to sequence decision problem (namely, what is the optimal sequence to finish a task). Also, scalability is still a challenge.

Graph neural networks (GNNs) [2] are a relatively new approach to hard combinatorial problems. Since some NP-complete problems can be reduced to graph problems, GNNs can capture the internal relational structure efficiently through the message passing process [5]. Based on message passing and GNNs, Selsam et al. developed a supervised SAT solver: neuroSAT [13]. It has been shown that neuroSAT performs very well on NP-complete problems within a certain size. Combining such GNNs with RL [7] could also be a potential future work direction for us.

8 CONCLUSION

Can the amazing game playing capabilities of the neural MCTS algorithm used in AlphaZero for Chess and Go be applied to Zermelo games that have practical significance? We provide a partial positive answer to this question for a class of combinatorial optimization problems which includes the *HSR* problem. We show how to use Zermelo Gamification (ZG) to translate certain combinatorial optimization problems into Zermelo games: We formulate the optimization problem using predicate logic (where the types of the variables are not "too" complex) and then we use the corresponding semantic game [6] as the Zermelo game which we give to the adapted neural MCTS algorithm. For our proof-of-concept example, *HSR* Zermelo Gamification, we notice that the Zermelo game is asymmetric.

Nevertheless, the adapted neural MCTS algorithm converges on small instances that can be handled by our hardware and finds the winning strategy (and not just an approximation). Our evaluation counts all correct/incorrect moves of the players, thanks to a formal *HSR* solution we have in the form of the Bernoulli triangle which provides the winning strategy. Besides, we discussed the coverage ratio and transfer learning of our algorithm. We hope our research sheds some light on why the neural MCTS works so well on certain games. While Zermelo Gamification currently is a manual process we hope that many aspects of it can be automated.

Acknowledgements: We would like to thank Tal Puhov for his feedback on our paper.

REFERENCES

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. 2002. Finite-time Analysis of The Multiarmed Bandit Problem. *Machine learning* 47, 2 (2002), 235–256.
[2] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018).
[3] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940* (2016).
[4] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intellig. and AI in Games* 4, 1 (2012), 1–43.
[5] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 1263–1272.
[6] Jaakko Hintikka. 1982. Game-theoretical semantics: insights and prospects. *Notre Dame J. Formal Logic* 23, 2 (04 1982), 219–241.
[7] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*. 6348–6358.
[8] Levente Kocsis and Csaba Szepesvari. 2006. Bandit Based Monte-Carlo Planning.. In *ECML (Lecture Notes in Computer Science)*, Vol. 4212. Springer, 282–293.

[9] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Torbjorn S Dahl, Amine Kerkeni, and Karim Beguir. 2018. Ranked Reward: Enabling Self-Play Reinforcement Learning for Combinatorial Optimization. *arXiv preprint arXiv:1807.01672* (2018).

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533.

[11] Christopher D. Rosin. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61, 3 (mar 2011), 203–230.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. (2017). arXiv:arXiv:1707.06347

[13] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L Dill. 2018. Learning a SAT Solver from Single-Bit Supervision. *arXiv preprint arXiv:1802.03685* (2018).

[14] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (Jan. 2016), 484.

[15] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[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. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR* abs/1712.01815 (2017).

[17] David Silver, Julian Schrittwieser, 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. 2017. Mastering the game of Go without human knowledge. *Nature* 550 (Oct. 2017), 354.

[18] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 2692–2700.

[19] Theophane Weber, Sebastien Racaniere, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. 2017. Imagination-Augmented Agents for Deep Reinforcement Learning. (2017). arXiv:arXiv:1707.06203

[20] R. J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8 (1992), 229–256.