The Knowledge Engineering Review, Vol. 35, e11, 1 of 18.
doi:10.1017/S026988892000020X

Learning self-play agents for combinatorial optimization problems

RUIYANG XU and KARL LIEBERHERR

Northeastern University Khoury College of Computer Sciences, Boston, MA, USA
e-mails: ruiyang@ccs.neu.edu, lieber@ccs.neu.edu

Abstract

Recent progress in reinforcement learning (RL) using self-play has shown remarkable performance with several board games (e.g., Chess and Go) and video games (e.g., Atari games and Dota2). It is plausible to hypothesize that RL, starting from zero knowledge, might be able to gradually approach a winning strategy after a certain amount of training. In this paper, we explore neural Monte Carlo Tree Search (neural MCTS), an RL algorithm that has been applied successfully by DeepMind to play Go and Chess at a superhuman 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 to transform specific combinatorial optimization problems into Zermelo games whose winning strategies correspond to the solutions of the original optimization problems. A specially designed neural MCTS algorithm is then introduced to train Zermelo game agents. We use a prototype problem for which the ground-truth policy is efficiently computable to demonstrate that neural MCTS is promising.

1 Introduction

The past several years have witnessed the progress and success of reinforcement learning (RL) with self-play and deep learning. The combination of classical RL algorithms with newly developed deep learning techniques delivers stunning performance on both simple Atari video games (Mnih *et al.,* 2015), as well as classic board games (Silver *et al.,* 2017a). One characteristic feature of these learning algorithms is tabula rasa, which means that the learning process starts from zero experience and knowledge of the winning strategy, and there is no pretrained model or human player data to inform the agent as to how to win the game. The agent must rely only on itself to learn every tactic and strategy through self-play, which can be regarded as an approach toward general artificial intelligence (AI).

AI and, in particular, RL have been applied successfully to several strategic multiplayer games. However, the literature falls short when it comes to applying these techniques to general problems in other domains. It is intriguing to consider whether those self-play agents' superhuman capabilities can be used to solve problems in other realms. In this work, as a proof of concept, we transform specific combinatorial optimization problems into games via a process called Zermelo Gamification (ZG), so that a neural Monte Carlo Tree Search (neural MCTS) (Anthony *et al.,* 2017; Silver *et al.,* 2017b, 2018) self-play agent can be leveraged to play the transformed games and solve the original problems. Our experiments show that the two competitive agents gradually, although with setbacks, improve and jointly arrive at the optimal strategy. Although the transformed game is quite different from Go and Chess, the neural MCTS algorithm can still converge upon correct solutions. The trained self-play agent can be

used to reveal the solution (or show the nonexistence of any solution) of the original problem through competitions against itself based on the learned strategy.

As a note, the prototype problem (viz., the highest safe rung (HSR)) chosen in this study is not a computationally complex problem, for two main reasons: (1) It is efficient to compute the ground-truth solution for correctness measurements. (2) It is easy for us to debug and further understand how the algorithm works. We should mention here that the primary purpose of this study is not to invent a new algorithm that can achieve the state of the art. Instead, we give a proof of concept to apply an existing self-play algorithm to another domain.

Moreover, although it has been proven that an MCTS algorithm converges asymptotically to a Minimax algorithm (Kocsis & Szepesvári, 2006), it remains an open problem whether a neural MCTS algorithm also has this property. Therefore, we also plan to use this proof of concept to reveal the undiscovered properties of the neural MCTS algorithm in future studies.

To summarize, in this paper, we make three main contributions: (1) We introduce the ZG a way to transform combinatorial problems to Zermelo games using a variant of Hintikka's Game-Theoretical Semantics (Hintikka, 1982). (2) We implement a modification of the neural MCTS algorithm[1] designed explicitly for those Zermelo games. (3) We run experiments on a prototype problem and show that it is promising to apply neural MCTS to another domain. Our result shows that, for problems within a specific input size, the trained agent does converge upon the optimal strategy and find the optimal solution, hence solving the original optimization problem in a tabula rasa way.

The remainder of this paper is organized as follows. Section 2 explores some works related to this research. Section 3 presents essential preliminaries on neural MCTS and the combinatorial optimization problems studied. Section 4 introduces the Zermelo game and a general method for transforming the given type of combinatorial optimization problems into Zermelo games, where we specifically discuss our prototype problem HSR. Section 5 gives our method and correctness measurements, and Section 6 presents experimental results. Sections 7 and 8 provide discussion and conclusions.

2 Related work

Solving combinatorial problems through gamification has been broadly studied in the literature. For instance, Novikov and Katsman (2018) propose an approach to verify the solution of logical–combinatorial problems. They interpret the application of the logic rules as moves in an intellectual game. In this way, the solution of the problem maps to a game process, and the correct solution corresponds to a winning strategy.

Similarly, in Fujikawa and Min (2013), a game environment was developed to obtain research data from general users. The authors use this gamification to help solve the Minimum Power Broadcast Tree in the wireless *ad hoc* network problem. Although those studies propose ways to solve combinatorial problems via gamification, they focus solely on human learners or players. However, unlike in their studies, our learners/players are AI agents.

Imagination-Augmented Agents (Racanière *et al.,* 2017), an algorithm invented by DeepMind, is used to handle complex games with sparse rewards. The algorithm has acceptable performance. However, it is not tabula rasa. Namely, one has to train, in a supervised way, an imperfect but adequate model first and then use that model to boost the learning process of a model-free agent. Even though I2As, along with a trained model, can solve games like Sokoban to some level, they can hardly be applied to games where even the training data are limited and difficult to generate and label.

By formulating a combinatorial problem as an Markov Decision Process (MDP), Ranked Reward (Laterre *et al.,* 2018) binarizes 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.

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

However, this method cannot be applied to problems that already have a binary reward (such as a zero-sum game with reward $-1, 1$). It should be mentioned that the idea of improving the performance threshold after each learning iteration has also been used in our implementation.

Pointer networks (Vinyals *et al.,* 2015) have been shown to solve specific combinatorial Nondeterministic Polynomial Time (NP) problems of a limited size. The algorithm is based on supervised attention learning on a sequence-to-sequence Recurrent Neural Network. However, due to its high dependency on the quality of data labels (which can be very expensive to obtain), Bello *et al.* (2016) improved the method of Vinyals *et al.* (2015) 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 (Williams, 1992) style policy gradient. Their result shows a significant improvement in performance. However, this approach can only be applied to sequence decision problems (namely, what is the optimal sequence to finish a task?). Also, scalability remains a challenge.

Graph neural networks (GNNs) (Battaglia *et al.,* 2018) 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 (Gilmer *et al.,* 2017). Based on message passing and GNNs, Selsam *et al.* (2018) developed a supervised SAT solver: neuroSAT. neuroSAT has been shown to perform very well on NP-complete problems within a specific size range. Combining such GNNs with RL (Khalil *et al.,* 2017) could also be a potential future work direction for us.

Our work is also closely related to General Game Play (GGP) (Genesereth *et al.,* 2005). In terms of using MCTS as a strategy searching technique, CADIAPLAYER (Bjornsson & Finnsson, 2009) has been proven to be an efficient and competitive GGP player. However, CADIAPLAYER only uses traditional MCTS without a neural network. It turns out that there are only a few papers on GGP which combine MCTS with a neural network. We only found a work by Rezende and Chaimowicz (2017), which is quite similar to ours. Nevertheless, the neural network used in their research has to be dynamically generated during the training, which is hard to train and quite different from most neural network architectures we see nowadays.

3 Preliminaries

3.1 Monte Carlo tree search

In this section, we will briefly introduce the MCTS algorithm – mainly, the Polynomial Upper Confidence Tree algorithm used in AlphaZero (Silver *et al.,* 2017b). MCTS is known to be efficient in searching for a near-optimal strategy for games with large state spaces (Kocsis & Szepesvári, 2006). It leverages a Monte Carlo process to collect statistical information from the simulated self-play. An MCTS algorithm, which uses an Upper Confidence Bound (UCB) to balance exploration and exploitation, is also called an Upper Confidence Tree algorithm. One widely used bound is the UCB1 bound (Auer *et al.,* 2002):

$$\frac{W(s,a)}{N(s,a)} + c\sqrt{\frac{\ln \sum_{a'} N(s,a')}{N(s,a)}}$$

In the expression above, $W(s, a)$ stands for the number of winning outcomes for taking the action a under state s; $N(s, a)$ stands for the number of times for taking action a under state s; and $\sum_a N(s, a)$ stands for the total number of simulations under state s. The exploration parameter c is conventionally set to $\sqrt{2}$. The first component of UCB1 controls the exploitation (it weighs more on children with a high winning ratio), while the second component controls the exploration (it weighs more on children with few visits).

In practice, a UCB1-based vanilla MCTS runs given number of iterations (as a hyperparameter, called 'simulation times') on a game tree until it finds an optimal decision. Each iteration passes through four phases:

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) according to UCB1. Specifically, suppose the root is s_0. UCB1 selects a serial of states $\{s_0, s_1, \ldots, s_l\}$ by the following process:

$$a_i = \arg\max_a \left[\frac{W(s_i, a)}{N(s_i, a)} + c\sqrt{\frac{\log \sum_{a'} N(s_i, a')}{N(s_i, a)}} \right]$$

$$s_{i+1} = \text{move}(s_i, a_i)$$

2. EXPAND: Once the selected phase ends at an unvisited state s_l, the state will be fully expanded and marked as visited. All its child nodes will be considered as leaf nodes during next iteration of selection.
3. ROLLOUT: The rollout is carried out for every child of the expanded leaf node s_l. Starting from any child of s_l, the algorithm will randomly roll out the rest of the game (Browne *et al.,* 2012): it 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) is then recorded and used to update the statistics in the next phase.
4. BACKUP: This is the last phase of an iteration in which the algorithm updates the statistics for each node in the selected states $\{s_0, s_1, \ldots, s_l\}$ from the first phase. To illustrate this process, suppose the selected states and corresponding actions are $\{(s_0, a_0), (s_1, a_1), \ldots, (s_{l-1}, a_{l-1}), (s_l, \_)\}$. Let $V(s_r)$ be the outcome of the rollout from child s_r. Then, for each (s_i, a_i) pair, we update the statistics as:

$$W^{\text{new}}(s_i, a_i) = W^{\text{old}}(s_i, a_i) + V(s_r)$$
$$N^{\text{new}}(s_i, a_i) = N^{\text{old}}(s_i, a_i) + 1$$

Such a process will iterate through all rollouts from the last phase.

Once the given number of iterations has been reached, the algorithm returns a vector of action probabilities of the current state (root s_0). Each action probability is computed as $\pi(a|s_0) = \frac{N(s_0,a)}{\sum_{a'} N(s_0,a')}$. The final decision 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 also referred to as self-play in the literature.

3.2 Neural MCTS

In the AlphaZero implementation (Silver *et al.,* 2017b), a variant of UCB is applied[2]:

$$\frac{W(s, a)}{N(s, a) + 1} + cP_\phi(a|s)\sqrt{\frac{\sum_{a'} N(s, a')}{N(s, a) + 1}}$$

The new bound is called the Polynomial Upper Confidence Bound (PUCB), which is different from the UCB1 in several aspects:

1. A polynomial exploration term $\left(\sqrt{\sum_{a'} N(s, a')}\right)$ is used in place of a logarithmic one $\left(\sqrt{ln \sum_{a'} N(s, a')}\right)$. This modification makes the exploration polynomial, which proves to be consistent in a large (or even infinite) state and action space (Auger *et al.,* 2013).

[2] Theoretically, the exploratory term should be $\sqrt{\frac{\sum_{a'} N(s_{i-1},a')}{N(s_{i-1},a)+1}}$; however, 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 only the AlphaZero one works, while the other one quickly converges to an incorrect strategy.

2. A prediction $P_\phi(a|s)$ generated from the neural network ϕ is used to bias the exploration. This modification causes the algorithm to apply a penalty on actions that have low exploration value.
3. The $+1$ term in the denominator is used to smooth the case $N(s, a) = 0$. Specifically, when $N(s, a) = 0$, the bound becomes

$$cP_\phi(a|s)\sqrt{\sum_{a'} N(s, a')}$$

Smoothing is necessary for pruning a search tree with a large action space. Otherwise, every unvisited action would be accessed at least once, because of an infinitely large value evaluated from the formula.

A neural MCTS is a combination of PUCB and neural networks, and it goes through two main phases during each iteration: self-play and training. For self-play, similar to a vanilla MCTS, it also goes through four subphases, but with several changes:

1. SELECT: The selection rule is changed from UCB1 to PUCB:

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

2. ROLLOUT: The rollout is replaced by an evaluation neural network. Hence, instead of playing the game randomly, a neural network V_ϕ is used to evaluate the outcome. In this way, the algorithm saves time on rolling out the whole game.
3. BACKUP: The update from each child node s_r is changed to

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

The algorithm will run for a given number of self-plays (a.k.a. the hyperparameter 'number of self-plays'). Each iteration of self-play i will generate a sequence of action vectors $(\pi_i(s_0), \pi_i(s_1), \ldots, \pi_i(s_t))$ and a game outcome R_i with a trajectory $(s_0, s_1, \ldots, s_t)$. These action vectors and outcomes will then be used in the training phase: the action vectors are used as targets for P_ϕ, and the outcomes are used as targets for V_ϕ. To be more specific, given a self-play i and a state s in the trajectory of i, the neural network P_ϕ is trained to predict $P_\phi(s) = \pi_i(s)$, while the neural network V_ϕ is trained to predict $V_\phi(s) = R_i$. The neural MCTS iterates over the self-play and training phases until the algorithm converges.

3.3 Game-theoretical semantics

Stepwise evaluation of a first-order logic statement can be regarded as a two-player game. This idea originated from Hintikka (1982), in which a player 'Verifier' claims that the statement is true in the given context, while a player 'Falsifier' claims the opposite[3]. Hintikka noticed that, with properly defined game operations, predicate-logical semantics are equivalent to a system of games of object picking and fact testing. Furthermore, for each predicate logic formula, one can map it into a game given the

[3] In our paper, we call Verifier the Proponent, and Falsifier the OP.

Table 1 The mapping between $\langle \Phi, M \rangle$ and game $(\langle \Phi, M \rangle, \text{V}, \text{F})$

Proposition Φ	**Operation**	**Subgame**
$\forall x : \Psi(x)$	F picks x_0	game $(\langle \Psi[x/x_0],\ M \rangle,\ \text{V},\ \text{F})$
$\Psi \wedge \chi$	F picks $\theta \in \{\Psi, \chi\}$	game$(\langle \theta,\ M \rangle,\ \text{V},\ \text{F})$
$\exists x : \Psi(x)$	V picks x_0	game$(\langle \Psi[x/x_0],\ M \rangle,\ \text{V},\ \text{F})$
$\Psi \vee \chi$	V picks $\theta \in \{\Psi, \chi\}$	game$(\langle \theta,\ M \rangle,\ \text{V},\ \text{F})$
$\neg\Psi$	N/A	game $(\langle \Psi,\ M \rangle,\ \text{F},\ \text{V})$
φ	N/A	N/A

In this table, F stands for Falsifier, and V stands for Verifier. M is the model of the formula, which defines all non-logical symbols in the formula. The game ends at an atomic proposition φ. It is to be noted that the negation switches the role of the two players, namely, strategies for V in a game for $\neg\Psi$ are strategies for F in the game for Ψ.

underlying model, as shown in Table 1. A winning strategy for the Verifier/Falsifier exists if and only if the underlying claim is true/false.

3.4 Combinatorial optimization problems

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

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

In this statement, n is a natural number, and x, y can be any instance 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 these problems as interpreted logic statements is crucial to transforming them into games (Hintikka, 1982). In the next section, we will introduce our gamification method in detail.

4 Zermelo gamification

4.1 General formulation

We introduce 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 ZG by deriving a prototype game: the HSR.

A Zermelo game is defined as a two-player, finite, and perfect information game with only one winner and loser. During the game, players move alternately (viz., no simultaneous move). Leveraging the logic statement (see Section 3.4) of the problem, the Zermelo game is built on the Game-Theoretical Semantic approach (Hintikka, 1982). 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, one can recall the logic statement in Section 3.4, which implies the following Zermelo game (Figure 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

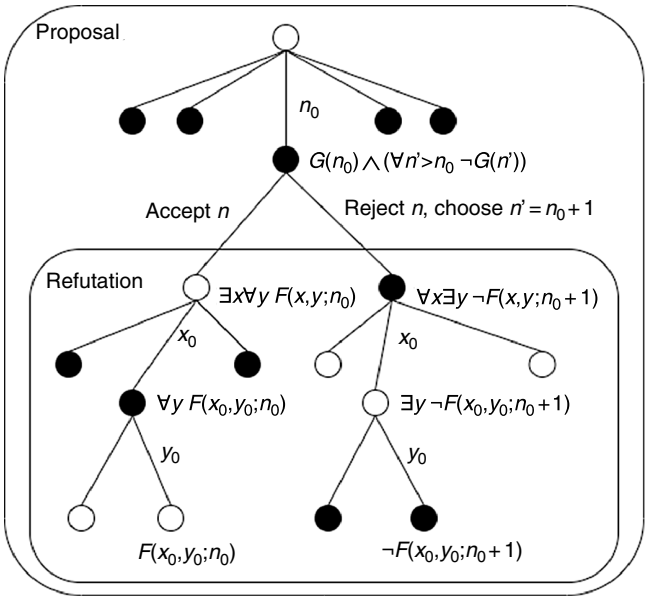

Figure 1 A Zermelo game where white nodes stand for the P's turn, and black nodes stand for the OP's turn. A role-flip happened after rejecting n by OP. The refutation game can be treated uniformly where, depending on the order of the quantifiers, the OP takes the first move. The OP wins if and only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

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 $A = \text{False}$. The OP will reject n if she is unable to confirm $A = \text{False}$. 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 $B = \text{False}$. To put it in another way, $B = \text{False}$ implies $\neg B = \text{True}$, 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 next 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 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, that is, the P chooses the optimal n.

2. Refutation phase: In this phase, 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 makes 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 $F(x, y; n)$ does not hold. It should be noticed that the OP is in a dilemma when the P is perfect.

4.2 HSR problem

In this section, we first introduce our prototype, the HSR problem (see also Sniedovich, 2003). Then, we will see how to perform ZG 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. The HSR is the rung that, for any tests performed above it, the jar will break. Given k identical jars and q test chances to throw those jars, what is the maximum number of rungs a ladder can have so that there is always a strategy to locate the HSR with at most k jars and q tests?

To formulate the HSR problem in predicate logic, we utilize the recursive property. Notice that, after performing a test, the HSR can only be located either above or below the current testing level. This 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 subproblems. We introduce the predicate $G_{k,q}(n)$, which means there is a strategy to find the HSR on a n-level ladder with at most k jars and q tests. One thing to notice is that the ground level is always included in these n levels, and a jar is always safe on the ground, which also means that a testing point can only be chosen from the rest of $n-1$ rungs. As a result, using the recursive property we have mentioned, $G_{k,q}(n)$ can be written as

$$G_{k,q}(n) = \begin{cases} \text{True, if } n = 1 \\ \text{False, if } n > 1 \wedge (k = 0 \vee q = 0) \\ \exists m \in [1 \ldots n-1] : \{G_{k-1,q-1}(m) \wedge G_{k,q-1}(n-m)\} \end{cases}$$

This formula can be interpreted in the following way: if there is a strategy to locate the HSR on a n-level ladder, then it must provide the tester a testing level m so that, no matter whether the jar breaks or not, the strategy still leads the tester to finding the HSR in the following subproblems. More specifically, for subproblems, we have $G_{k-1,q-1}(m)$ if the jar breaks, that means we only have $k-1$ jars and $q-1$ tests left to locate the HSR in the lower part of the ladder (which has m levels). Similarly, $G_{k,q-1}(n-m)$ for the upper part of the ladder. Therefore, the problem is solved recursively until there is only one level left, which either means that the HSR has been located, or there are no jars/tests left meaning the tester has failed to locate the HSR. With the notation of $G_{k,q}(n)$, the HSR problem can be further 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 ZG on the HSR problem. Notice that the expression of $G_{k,q}(n)$ is slightly different from the ones used in Section 3.4: the difference being that there is no universal quantifier in the expression. To introduce the universal quantifier, we think of the environment as another player who plays against the tester so that, after each test performed, the environment will tell the tester whether the jar is broken or not. In this way, locating the HSR can be formulated as follows:

$$G_{k,q}(n) = \begin{cases} \text{True, if } n = 1 \\ \text{False, if } n > 1 \wedge (k = 0 \vee q = 0) \\ \exists m \in [1 \ldots n-1] \; \forall a \in \text{Bool} : \\ \big\{(a \rightarrow G_{k-1,q-1}(m)) \wedge (\neg a \rightarrow G_{k,q-1}(n-m))\big\} \end{cases}$$

Now, with the formula above, one can perform the standard ZG (Section 4.1) to get the corresponding Zermelo game (Figure 2). 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 HSR 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 against each other and adjust their strategies based on the reaction provided.

It should be mentioned that the HSR problem, as a prototype to test our idea, is itself not a hard problem. The solution for the HSR problem can be computed and represented efficiently with a Bernoulli's

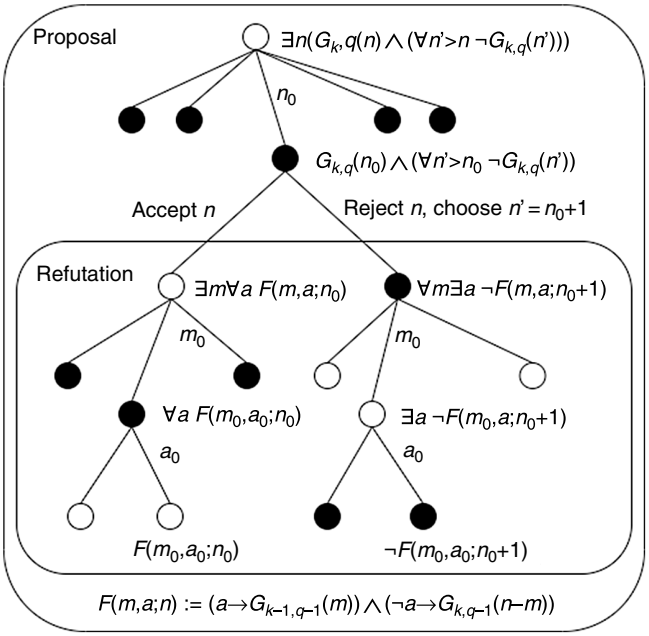

Figure 2 The ZG of the HSR problem. The game is recursively played between two players until the HSR is located or all resources have been used up. It is to be noted that the figure here does not show a formal complete game but a general illustration of the game process

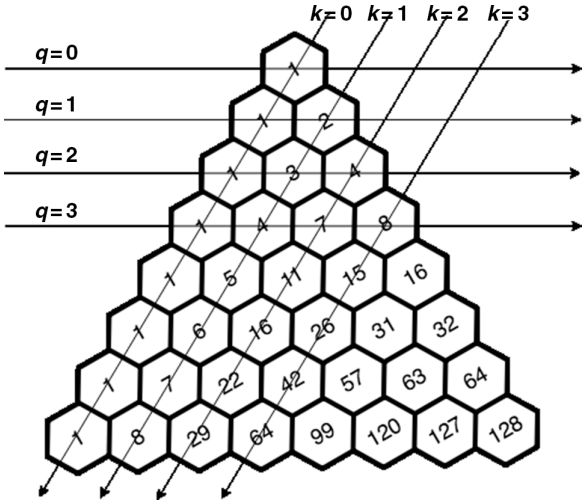

Figure 3 Theoretical values for maximum n in the HSR problem with given k, q, which can be represented as a Bernoulli's triangle. It is to be noted that $N(k, q) = N(q, q)$ for $k > q$

Triangle (Figure 3). We use the notation $N(k, q)$ to represent the solution for the 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 under k, q.

5 Method

5.1 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.

Our neural network consists of four layers of 1-D convolutional neural networks and two dense layers. The input is a tuple (k, q, n, m, r) where k and 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 $A_{\mathrm{P}}, A_{\mathrm{OP}}$ of probabilities on the action space for each player as well as a scalar v as the game result evaluation. The loss value is defined as

$$L = (v - v_0)^2 + H(A_{\mathrm{P}}, \pi_{\mathrm{P}}) + H(A_{\mathrm{OP}}, \pi_{\mathrm{OP}})$$

where H is the cross-entropy function, and v_0, π_{P}, and π_{OP} are the target values (see Section 3 for details). Notice that when it is P's turn, π_{OP} is set to be 0, and vice versa.

The algorithm will continue iterating either until it converges, which means the loss value is less than a given threshold, or it reaches a given maximum number of iterations, which means the problem is too complex to be solved with the current setting. During each iteration of the learning process, there are three phases: (1) The 'number of self-play' 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. (3) The current 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 matches, and then it will play as the P for another 20 matches. Then we collect the winning count of W_{new}, W_{old} for each neural network. The new neural network will be accepted when the winning ratio $\frac{W_{\mathrm{new}}}{W_{\mathrm{new}}+W_{\mathrm{old}}}$ is more significant than a given threshold (a.k.a. 'accepting threshold').

5.2 Correctness measurement

An action is correct if it preserves a winning position. It is straightforward to define the correct actions using the ground truth in the Bernoulli triangle (Figure 3).

5.2.1 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.

5.2.2 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 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.

6 Experiment

6.1 Experiment setup

All experiments were carried out on a computer with a Core i7-9750H 4.5 GHz CPU, 16 GB Memory, and a GTX 1650 Graphics Processing Unit (GPU)[4]. We measure the correctness during the competition phase, where each player's correctness number is counted after every single game. Then, we divide the

[4] The experiments reported in this paper use TensorFlow 1.15 with Graphics Processing Unit (GPU) support for training the neural networks. To test reproducibility, we also ran the experiments on TensorFlow 2.0, which does not have GPU support yet. To our surprise, our algorithms stopped converging to the winning strategy on TensorFlow 2.0. We remark this lack of reproducibility of our results for researchers who plan to build on our work. We are still trying to find the root cause behind the reproducibility problem, which is most likely caused by an implementation bug in TensorFlow 2.0.

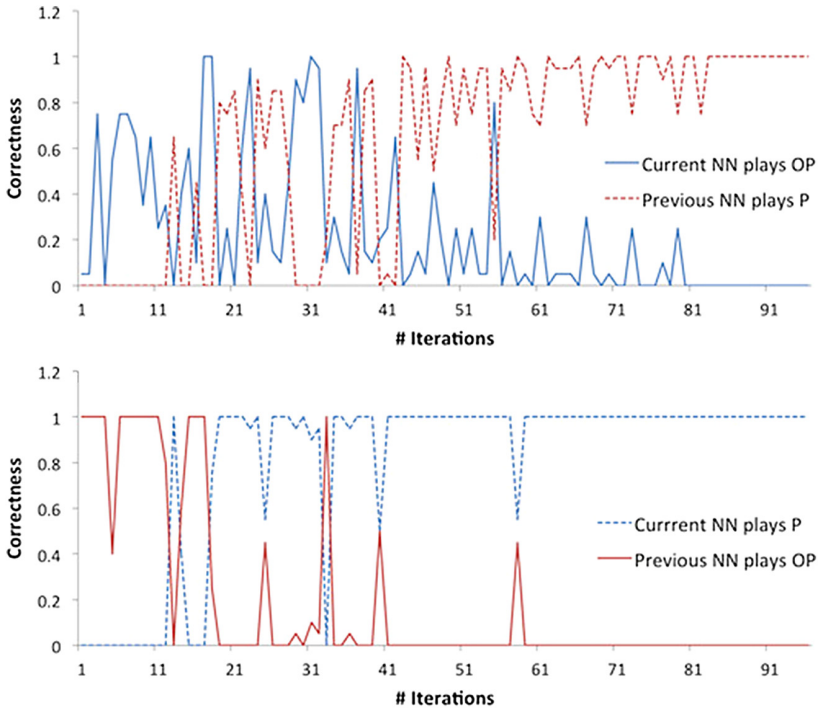

Figure 4 Average correctness measured during the competition phase of each iteration for the proposal game on $k = 7$, $q = 7$. The hyperparameters are set as: 'simulation times' = 128, 'number of self-plays' = 100, 'accepting threshold' = 0.2, and exploration parameter $c = 1$. The first image shows the measurements on matches that the current neural network plays OP, while the previous neural network plays the P. The bottom image shows the opposite. It can be seen that the current neural network is a tougher OP when it plays OP: the previous neural network (who plays P) takes a much longer time (25 min per iteration) to defeat the OP finally and make the algorithm converge

correctness count by the total number of moves in the game to get the correctness ratio for each game. Finally, we average the correctness ratio on all games to get the final evaluation for the current iteration. As a remark, since it is relatively time-consuming to run a complete Zermelo game on our machines, to save time, we only ran the entire game for $k = 7$, $q = 7$, and $n \in [1, \ldots, 130]$. Nevertheless, since the refutation game can be treated independently (when n is given) from the proposal game, we ran multiple experiments on the refutation games for various k, q, and n parameters.

6.2 Complete game

In this experiment, we run a full Zermelo game under the given resources $k = 7$, $q = 7$. Since two neural networks learn the proposal game and the refutation game separately, we measure the correctness separately: Figure 4 shows the ratio of correctness for each player during the proposal game. And Figure 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 slowly (80 iterations, and it takes 25 min per iteration). 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.

6.3 Refutation-only game

To test our method further, we focus our experiment only on refutation games with a given n. In this way, we can skip the proposal game and save a significant amount of time. We first run the experiment with the same hyperparameter on the same case where $k = 7$, $q = 7$. 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). Figure 6 verifies that the result is as expected. Then,

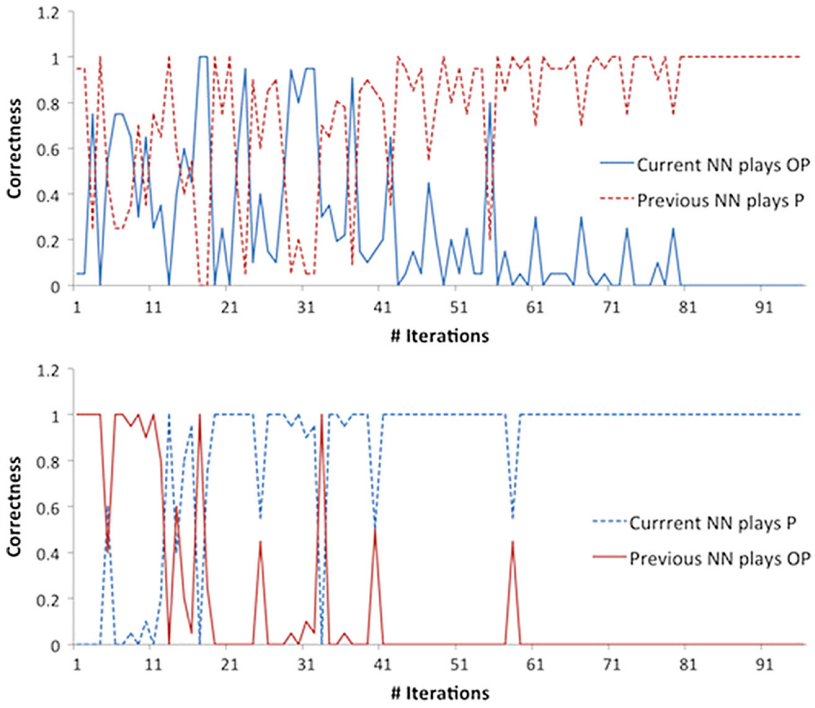

Figure 5 Average correctness measured during the competition phase of each iteration for the refutation game on $k = 7$, $q = 7$. Notice that all refutation games are continuations of the same proposal games in Figure 4

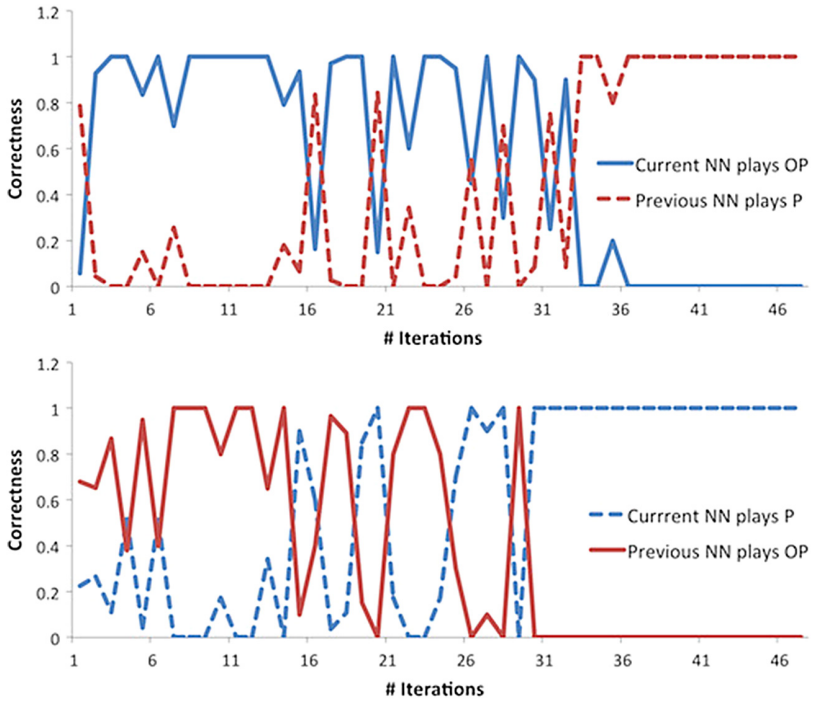

Figure 6 Average correctness measured during the competition phase of each iteration only for the refutation game on $k = 7$, $q = 7$, where $n = 128$ is given. The hyperparameters are set as: 'simulation times' $= 128$, 'number of self-plays' $= 100$, 'accepting threshold' $= 0.2$, and exploration parameter $c = 1$. The first image shows the measurements on matches where the current neural network plays OP, while the previous neural network plays the P. The bottom image shows the opposite. Comparing to the complete game in Figures 4 and 5, refutation-only games converge much faster: it only takes 37 iterations (10 min per iteration) to converge. This is because the proposal game needs to locate the optimal n, which is time-consuming

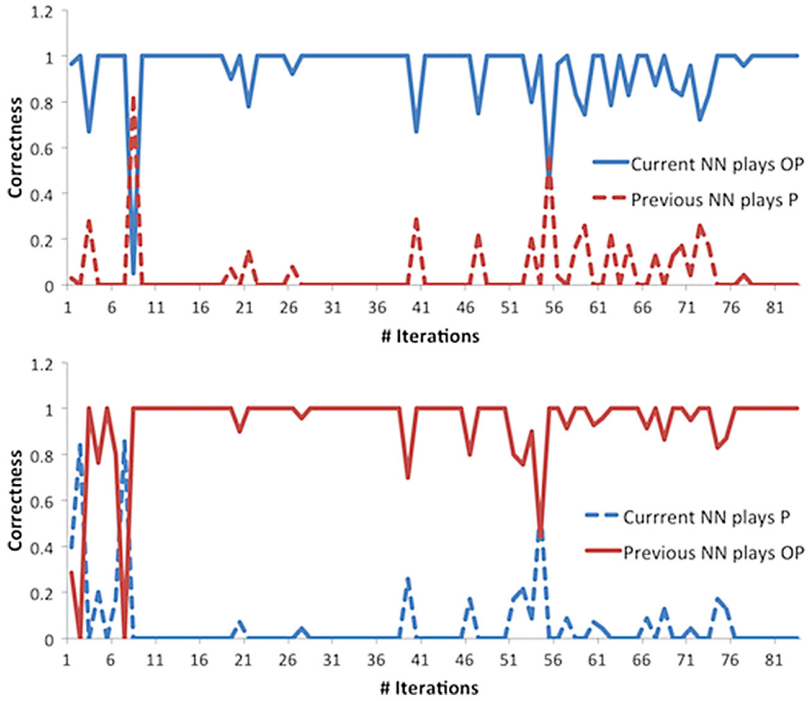

Figure 7 Average correctness measured during the competition phase of each iteration for the refutation-only game on $k = 7$, $q = 7$, where $n = 129$ is given. The hyperparameters are set as: 'simulation times' = 128, 'number of self-plays' = 100, 'accepting threshold' = 0.2, and exploration parameter $c = 1$. The top image shows the measurements on games that the current neural network plays OP, while the previous neural network plays the P. The bottom image shows the opposite. It is to be noted that, in this game, player P is doomed because there exists no winning strategy. It turns out that our algorithm takes a significant amount of time to converge

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. Figure 7, again, verifies our expectation, and one can see that the P can never find a winning strategy no matter how many iterations it has worked through.

It should be noted that our algorithm takes a significant amount of time to converge in this case. This is because the OP corresponds with a universal quantifier (see Section 4.2), and it is much harder to prove a universally quantified formula than an existentially quantified one. In other words, finding a winning strategy for the OP is more challenge than finding a winning strategy for the P.

In later experiments, we also tested our method in two more general cases where $k = 3$, $q = 7$ for $n = N(3, 7)$ (Figure 8) and $n = N(3, 7) - 1$ (Figure 9). All experimental results were consistent with the ground truth as expected (we put the details in the description under those figures).

7 Discussion

7.1 State-space coverage

Neural MCTS is capable of handling a large state space (Silver *et al.,* 2017b). Such an algorithm must 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 Figure 10 that the maximum number of states 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 Figure 10, at the beginning of the learning, neural MCTS accessed a large number of states; however, once the learning converged, it looked at a smaller number of states and pruned all other irrelevant states. It can

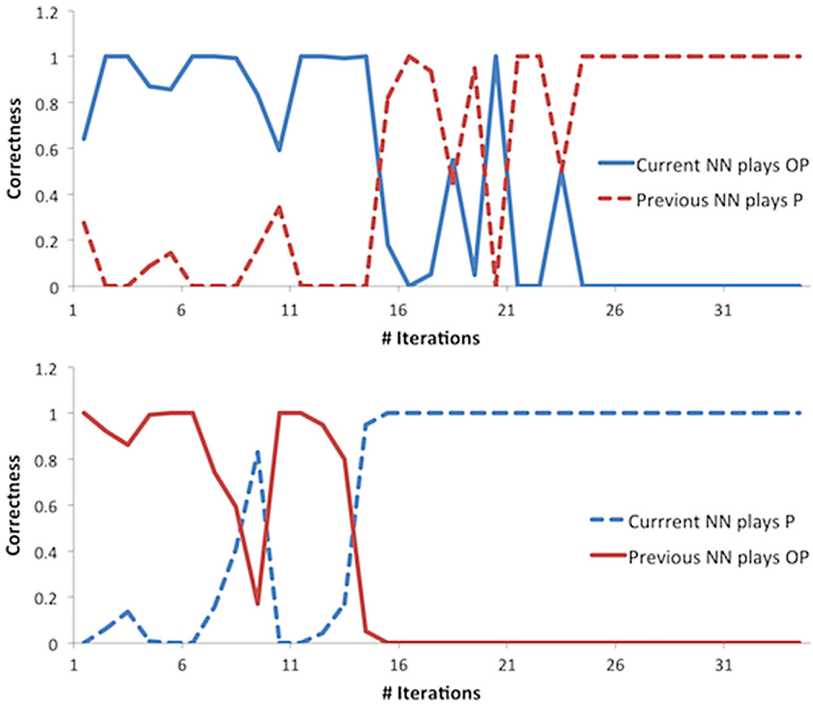

Figure 8 Average correctness measured during the competition phase of each iteration for the refutation-only game on $k = 3$, $q = 7$, where $n = 64$ is given. The hyperparameters are set as: 'simulation times' $= 64$, 'number of self-plays' $= 100$, 'accepting threshold' $= 0.2$, and exploration parameter $c = 1$. The top image shows the measurements on games that the current neural network plays OP, while the previous neural network plays the P. The bottom image shows the opposite. Since this game is much simpler than the previous ones, it takes less time to converge (8 min per iteration)

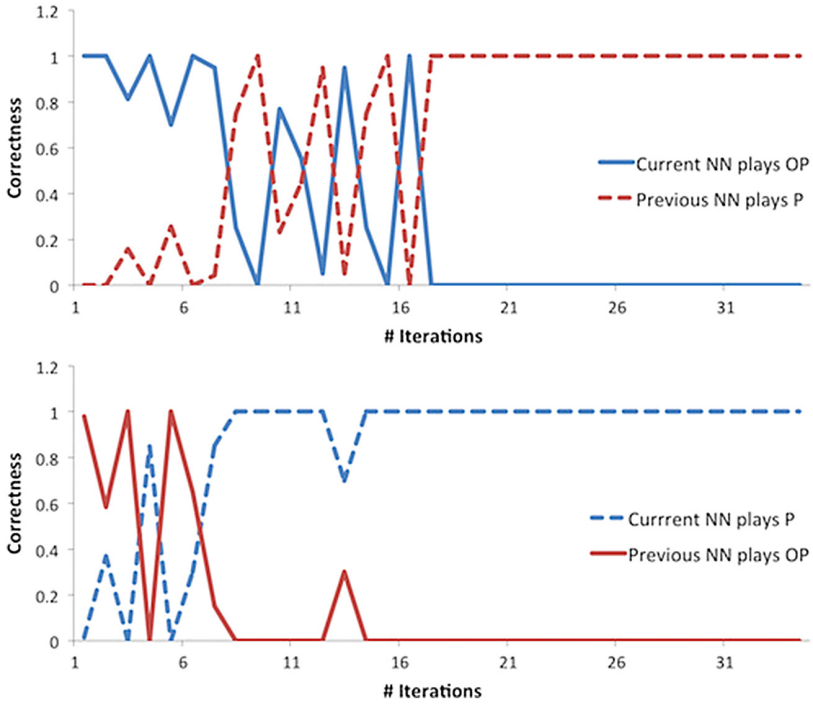

Figure 9 Average correctness measured during the competition phase of each iteration for the refutation-only game on $k = 3$, $q = 7$, where $n = 63$ is given. The hyperparameters are set as: 'simulation times' $= 64$, 'number of self-plays' $= 100$, 'accepting threshold' $= 0.2$, and exploration parameter $c = 1$. The top one image shows the measurements on games that the current neural network plays OP, while the previous neural network plays the P. The bottom image shows the opposite. It also takes 8 min per iteration but converges faster than the $n = 64$ case

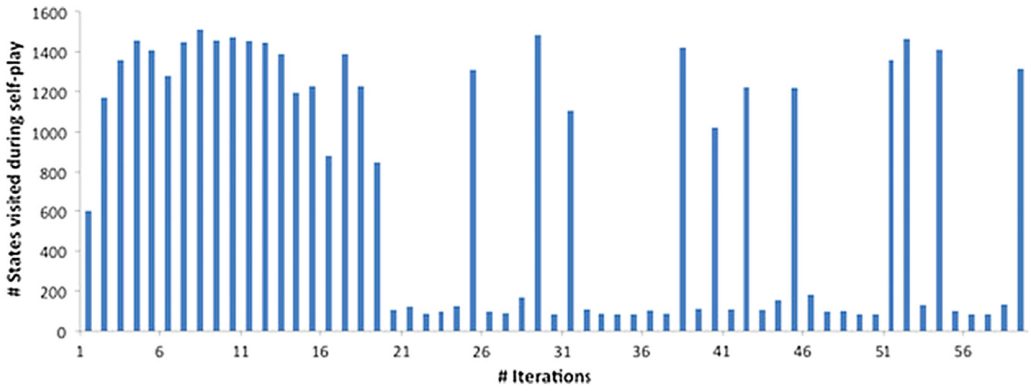

Figure 10 States accessed during self-play for each iteration, in refutation-only game $k = 7$, $q = 7$, $n = 128$. The hyperparameters are set as: 'simulation times' = 128, 'number of self-plays' = 100, 'accepting threshold' = 0.2, and exploration parameter $c = 1$

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 for its capability of handling large state spaces.

7.2 Perfectness

This discussion is designed to be had in a context where the truth is already known. Since the correct solution is derived from the optimal policy, it is essential to question whether the players are perfect after the training has converged (viz., the correctness of each player becomes flat without further changes). The experimental result shows that, after convergence, for a problem that 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 as 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. This is because after one player becomes 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. From our observation on these experiments, the doomed loser is still a robust suboptimal player after being competitively trained from tabula rasa. However, it is to be noted that, for an HSR refutation problem, since the problem itself is discrete, there is no such thing as an approximate solution because any solution is either entirely correct or incorrect. That means one has to let the algorithm run to convergence before getting any meaningful result.

7.3 Asymmetry

One can observe some asymmetry in the charts we presented in Section 6 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 acquired enough knowledge. Two facts cause this asymmetry: (1) The action space of the P is entirely different from that 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. Another way to see such an asymmetry is through the measurement of the 'fault', where a 'fault' of a player means the player makes a mistake (i.e., there is a move to keep the winning position, but the player chooses an incorrect one), meanwhile the OP player catches this fault (which means the OP player takes a move and switches to a winning position). We count the number of 'faults' for each 20 games in the competition phase (see Section 5 for details) and generate the visualization in Figure 11.

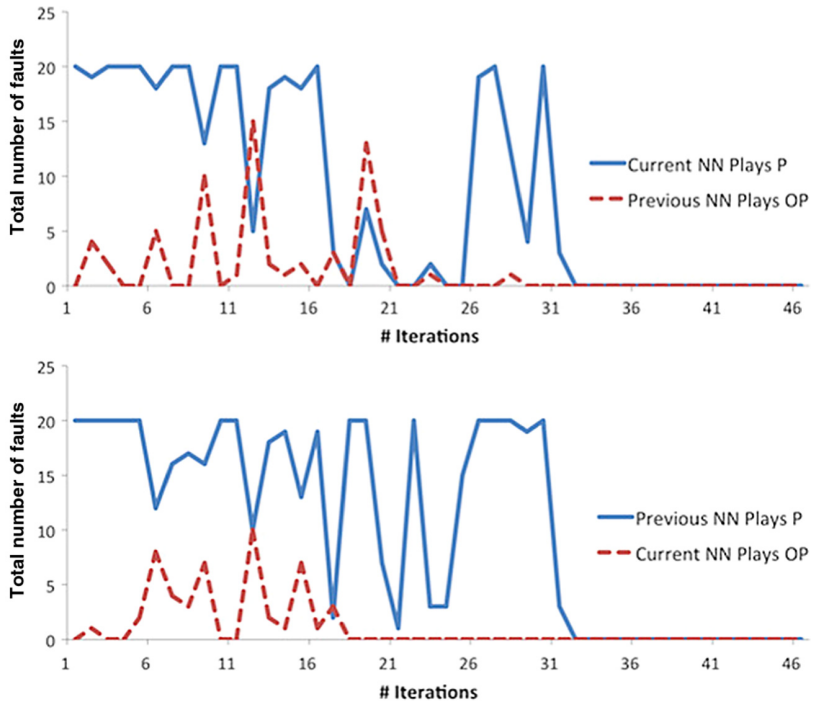

Figure 11 Number of faults measured during the competition phase of each iteration for the refutation-only game on $k = 7$, $q = 7$, where $n = 128$ is given. The hyperparameters are set as: 'simulation times' = 128, 'number of self-plays' = 100, 'accepting threshold' = 0.2, and exploration parameter c = 1. The top image shows the measurements on games that the current neural network plays P, while the previous neural network plays the OP. The bottom one shows the opposite. It can be seen that the OP player makes fewer faults and learns much faster than the P player, which is evidence of the asymmetry

7.4 Disadvantages versus advantages

It is to be noted that neural MCTS performs well on a deep tree but not on a wide tree (by wide, we mean a tree has a relatively high branching factor). We notice that neural MCTS algorithm is time-consuming on the HSR game with large k, q: as one can see, it takes a significant amount of time to converge even on a small case where $k = q = 7$, while the Minimax Algorithm might only need a few seconds to solve the same problem. The reason is that, during the MCTS simulation, the algorithm has to at least sample all possible actions once before learning the correct action. This fact becomes an issue when the action space is increasing (the HSR game has a special structure that increases to the value of k, q will also increase the size of the action space). As a result, we see that the larger the action space, the lower the efficiency of the algorithm.

On the other hand, for games like Chess or Go, even though they have large game trees, the action space in those games is bounded and relatively small. Therefore, no matter how deep the game tree is, the neural MCTS can always deal with it by using a neural network to fast 'roll-out' the game. However, the neural MCTS cannot circumvent the enumeration of the action space. This is the reason that we do not experience the exceptional performance of AlphaZero like Chess and Go on huge game trees. This fact also suggests that the problem and game should be carefully designed to avoid unbounded action spaces.

8 Conclusion

Can the tabula rasa learning capability of the neural MCTS algorithm used in AlphaZero be applied to problems in other domains? In this paper, we provide a plausible answer to this question for a class of combinatorial optimization problems. To be specific, we use ZG to transform specific combinatorial optimization problems into Zermelo games and solve the original problems by applying the neural

MCTS self-play algorithm. Our experiments show that, for instances that can be handled by our hardware, the algorithm converges and finds the winning strategy exactly (viz., not just an approximation). Additionally, we discuss the features and limitations of the algorithm. Primarily, we notice the following: (1) Even if a game is asymmetric (i.e., the players have different learning objects and action spaces), neural MCTS can still find the optimal strategy. (2) Neural MCTS has limitations on handling games with large action spaces, which also means that when transforming a combinatorial problem into a Zermelo game, one must always consider how to conflate the action space. Nevertheless, we hope our research sheds some light on why the neural MCTS works so well on certain games and how to apply this algorithm to another domain.

References

Anthony, T., Tian, Z. & Barber, D. 2017. Thinking fast and slow with deep learning and tree search. In *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS '17*, 5366–5376.

Auer, P., Cesa-Bianchi, N. & Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* **47**(2), 235–256.

Auger, D., Couetoux, A. & Teytaud, O. 2013. Continuous upper confidence trees with polynomial exploration - consistency. In *ECML/PKDD (1)*, Lecture Notes in *Computer Science* **8188**, 194–209. Springer.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gulcehre, C., Song, F., Ballard, A, Gilmer, J., Dahl, G., Vaswani, A., Allen, K., Nash, C., Langston, V., Dyer, C., Heess, N, Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y. & Pascanu, R. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint* arXiv:1806.01261.

Bello, I., Pham, H., Le, Q. V., Norouzi, M. & Bengio, S. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint* arXiv:1611.09940.

Bjornsson, Y. & Finnsson, H. 2009. Cadiaplayer: a simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games* **1**(1), 4–15.

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

Fujikawa, Y. & Min, M. 2013. A new environment for algorithm research using gamification. *IEEE International Conference on Electro-Information Technology , EIT 2013*, Rapid City, SD, 1–6.

Genesereth, M., Love, N. & Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* **26**(2), 62.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O. & Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning* **70**, 1263–1272. JMLR. org.

Hintikka, J. 1982. Game-theoretical semantics: insights and prospects. *Notre Dame Journal of Formal Logic* **23**(2), 219–241.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B. & Song, L. 2017. Learning combinatorial optimization algorithms over graphs. *In Advances in Neural Information Processing Systems*, 6348–6358.

Kocsis, L. & Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *Proceedings of the 17th European Conference on Machine Learning, ECML '06*, 282–293. Springer-Verlag.

Laterre, A., Fu, Y., Jabri, M. K., Cohen, A.-S., Kas, D., Hajjar, K., Dahl, T. S., Kerkeni, A. & Beguir, K. 2018. Ranked reward: enabling self-play reinforcement learning for combinatorial optimization. *arXiv preprint* arXiv:1807.01672.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* **518**(7540), 529–533.

Novikov, F. & Katsman, V. 2018. Gamification of problem solving process based on logical rules. In *Informatics in Schools. Fundamentals of Computer Science and Software Engineering*, Pozdniakov, S. N. & Dagienė, V. (eds). Springer International Publishing, 369–380.

Racanière, S., Weber, T., Reichert, D. P., Buesing, L., Guez, A., Rezende, D., Badia, A. P., Vinyals, O., Heess, N., Li, Y., Pascanu, R., Battaglia, P., Hassabis, D., Silver, D. & Wierstra, D. 2017. Imagination-augmented agents for deep reinforcement learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS '17*, 5694–5705. Curran Associates Inc.

Rezende, M. & Chaimowicz, L. 2017. A methodology for creating generic game playing agents for board games. In *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, 19–28. IEEE.

Selsam, D., Lamm, M., Bunz, B., Liang, P., de Moura, L. & Dill, D. L. 2018. Learning a sat solver from single-bit supervision. *arXiv preprint* arXiv:1802.03685.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K. & Hassabis, D. 2017a. Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. *CoRR* abs/1712.01815.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. & Hassabis, D. 2018. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science* **362**(6419), 1140–1144.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. & Hassabis, D. 2017b. Mastering the game of Go without human knowledge. *Nature* **550**, 354.

Sniedovich, M. 2003. OR/MS games: 4. The joy of egg-dropping in Braunschweig and Hong Kong. *INFORMS Transactions on Education* **4**(1), 48–64.

Vinyals, O., Fortunato, M. & Jaitly, N. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M. & Garnett, R. (eds), **28**. Curran Associates, Inc., 2692–2700.

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