Learning Self-Game-Play Agents for Combinatorial Optimization Problems

Ruiyang Xu and **Karl Lieberherr**
CCIS/PRL
Northeastern University, Boston, MA 02115
Email: ruiyang@ccs.neu.edu, lieber@ccs.neu.edu

Abstract

Recent progress in self-game-play reinforcement learning (RL) 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 time of training. In this paper, we do research on neural Monte-Carlo-Tree-Search (neural-MCTS), an RL self-play learning algorithm which has been applied successfully by DeepMind to play Go and Chess at the super-human level. We try to leverage the computational power of neural-MCTS to solve a class of combinatorial optimization problems. We propose a procedure to transform certain combinatorial optimization problems into games. A specifically designed neural-MCTS is then being applied to those games. The winning strategies in those games correspond to the solution of the original optimization problem. We choose a specific problem (HSR, in which the ground-truth policy can be computed efficiently) to carry out the experiment so that, along with the ground truth, a correctness measurement can be applied with a fine granularity during the experiment. Our experimental result indicates that, though the algorithm works soundly on problems with limited size, solving pure math problems of any size via self-game-play with neural-MCTS is still inefficient, regardless of its superhuman performance in AlphaZero. Nevertheless, transforming a math problem into game-play and using self-game-play agents to figure out a winning strategy can still be a promising approach to tackle hard problems in the future.

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 a stunning performance on both traditional simple Atari video games ((Mnih et al. 2015)) and modern complex RTS games (like Dota2 (Schulman et al. 2017)), and even certain hard board games like Go and Chess. 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 and learn to play the game with zero knowledge in the beginning. Such tabula-rasa learning can be regarded as an approach towards a general artificial intelligence.

In this paper, we transform a certain form of combinatorial optimization problems (e.g., the Product Stress Testing Problem $HSR_{k,q}$) into games so that a game-play agent can be leveraged to play the transformed game and solve the original problem on a specific instance (e.g., $HSR_{3,7}$). In Fig. 9 you can see how two competitive agents called P and OP gradually, but with setbacks (as in AlphaZero for Chess), improve and jointly arrive at the winning strategy. The tabula-rasa learning converges and solves a non-trivial problem although the underlying game is fundamentally different from Go and Chess . If a human finds the solution for $HSR_{3,7}$ she must have a good understanding of how to solve the problem for general n, k, q. Unfortunately, the neural networks don't give us an understanding of the general solution (yet).

One critical issue with those transformed games is that they usually have a large state space as well as a sparse reward function. Therefore, REINFORCE (Williams 1992) style algorithms like DQN (Mnih et al. 2015) and even its modern-day advanced variation PPO (Schulman et al. 2017) can't be applied to such kind of problems, because they all require a 'dense' reward function so that a feedback signal is generated after each action. However, one can hardly define a reward function on games like Sokoban or Sudoku where an agent has no idea whether the current action is correct or not until a much later phase or the end of the game. Imagination-Augmented Agents (I2As (Weber et al. 2017)), an algorithm invented by DeepMind, is used to handle such complex games. Although the algorithm is well performed, it is not model-free. Namely, one has to supervised train 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, it can hardly be applied to games where even the training data is limited and hard to generate or label. In other words, a model-free learning algorithm is needed to play generally on different kinds of games. So we turn to the neural-MCTS method used in AlphaZero by DeepMind. To our knowledge, this kind of neural-MCTS is the only tabula-rasa style algorithm which can handle games with both large state spaces and sparse rewards as well.

We make three main contributions: 1. We introduce a way to transform certain combinatorial problems into two-phase Zermelo games inspired by semantic games; 2. we implemented a variant of the neural-MCTS algorithm specifically designed for such kind of games; 3. we evaluate our algorithm on a specifically chosen problem (i.e., HSR) for which the winning strategy can be computed efficiently. Our result shows that, for problems under a certain size, neural-MCTS 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 certain combinatorial optimization problems. Section 3 introduces a general way to transform certain combinatorial optimization problems into two-phase games, where we specifically discuss the transforming of the HSR game. Section 4 gives our correctness measurement and presents experimental results. 5 and 6 made a discussion and conclusions.

2 Preliminaries

2.1 Monte Carlo Tree Search

The PUCT algorithm implemented in AlphaZero (Silver et al. 2017b; 2017a) is essentially a neural-MCTS algorithm which uses PUCB (Rosin 2011; Auger, Couetoux, and Teytaud 2013) as its confidence upper bound (Kocsis and Szepesvri 2006; Auer, Cesa-Bianchi, and Fischer 2002) and uses the neural prediction $P_\phi(a|s)$ as the predictor. The algorithm usually proceeds through 4 phases (S.E.R.B.) during each iteration:

Select At the beginning of each iteration, the algorithm select a path from the root (current game state) to a leaf (either a terminal state or a unvisited state) in the tree according to the PUCB. Specifically, suppose the root is s_0, we have:

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

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

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

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.

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 (Browne et al. 2012). 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 make a prediction of the result evaluation so that the algorithm saves the time on rolling out.

Backup Backup 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 alternatively until the game ends, this process is called MCTS simulation (self-play).

An MCTS algorithm often consists of several simulations (self-play). In each simulation. the algorithm will play one complete game step by step. Each step was chosen by running several iterations of S.E.R.B. as mentioned above. And for a neural-MCTS structure, the MCTS usually needs hundreds of simulations to generate enough training data.

2.2 Combinatorial Optimization Problems

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

$$\exists n \left((\forall x \, \exists y \, F(n, x, y)) \wedge (\forall n' > n \, \exists x \, \forall y \, \neg F(n', x, y)) \right)$$

In this statement, n is a 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 $F(n, x, y)$ is true. Formulating those problems as logic statements is crucial to transforming them into (semantic) games. Next, we will introduce two specific examples of problems which can be transformed into the given logic form.

Silver Ratio Our first example is a classical function optimization problem which involves to figure out the saddle point of a 2-variable function: $f(x, y) = xy + (1 - x)(1 - y^2); x, y \in [0, 1]$. The problem can be defined as:

$$z = \min_{x \in [0,1]} \max_{y \in [0,1]} f(x, y)$$

The solution of this problem can be derived through partial derivatives, namely $\frac{\partial}{\partial x} \frac{\partial}{\partial y} f(x, y) = 0$. The result is

$(x, y) = (\frac{5-\sqrt{5}}{5}, \frac{5-\sqrt{5}}{2\sqrt{5}})$ and the value $z = \frac{\sqrt{5}-1}{2}$, known as the silver ratio. Although this problem can be solved effectively with calculus, we are interested in solving it via a game-play. Hence we rewrite it as the following logic statement:

$$\exists z\ ((\forall x\ \exists y\ f(x, y) \geq z) \wedge (\forall z' > z\ \exists x\ \forall y f(x, y) < z'))$$

Highest Safe Rung Our second example (which is used as our experimental subject in this paper) is called "The Highest Safe Rung (HSR) Problem" (Kleinberg and Tardos 2006) (also known as "The Product Stress Testing Problem"): consider throwing jars from a specific rung of a ladder, the jars could either break or not. One has k jars and q test chances to throw those jars, can one locate the highest safe rung on the ladder?

The problem actually contains two nested subproblems:

1 Given limited resources (k,q), what is the highest ladder (i.e., the maximum number of rungs) one can handle? It could be the case that the ladder has hundreds of rungs while there are only a few jars and test chances so that the resources are obviously insufficient. Now we define a predicate $H^1_{k,q}(n)$ as "using k jars and q tests, one can locate the highest safe rung on a ladder with n rungs", and the subproblem can be rewritten as:

$$\exists n\ ((H^1_{k,q}(n)) \wedge (\forall n' > n\ \neg H^1_{k,q}(n)))$$

2 The predicate $H^1_{k,q}(n)$ contains another problem: given sufficient resources, how to figure out the optimal strategy of testing so that the highest safe rung can be located. Likewise, we define a predicate $H^2_{k,q}(x, \pi; n)$ as "using resources k, q, one can find a policy π to locate any highest safe rung x on a ladder with n rungs". Then $H^1_{k,q}(n)$ can be rewritten as:

$$H^1_{k,q}(n) := \forall x\ \exists \pi\ H^2_{k,q}(x, \pi; n)$$

Therefore, once given the resources k, q, a whole HSR problem can be represented as:

$$HSR_{k,q} :=$$

$$\exists n\ ((\forall x\ \exists \pi\ H_{k,q}(x, \pi, n)) \wedge (\forall n' > n\ \exists x\ \forall \pi \neg H_{k,q}(x, \pi, n')))$$

where $H_{k,q}(x, \pi, n) := H^1_{k,q}(n) \wedge H^2_{k,q}(x, \pi; n)$.

The solution to a problem $HSR_{k,q}$ is a number $N(k, q)$ such that $HSR_{k,q} = True$. To derive the solution theoretically, we first look at two extreme cases: (i) there is no jar to throw (i.e. $k = 0$). In this case, we assume the highest safe rung is the first rung (the ground) so that even if there is no jar, we know that a jar would not break on the ground. In other words, we have $N(0, q) = 1$. (ii) there is the same number of jars as the number of chances (i.e. $k = q$). In this case, using the Pigeon Hole Principle, there are at most 2^q possible testing sequences for all possible highest safe rungs, or we can say $N(q, q) = 2^q$. Also, notice that there are only q chances of the test, therefore for any $k > q$ there are $k - q$ jars one have no chance to use them, which also means $N(k, q) = N(q, q), k > q$.

Given the two extreme conditions above, the general case can be solved recursively by dividing the problem into two subproblems. The reason behind the recursion is that if one knows the highest ladder which one can be handled by $(k - 1, q - 1)$ and $(k, q - 1)$, then one can just concatenate those two ladders together to get a higher ladder which can exactly be handled by (k, q). On the other hand, if one has been given a ladder of $N(k, q)$ rungs, one can divide it into $N(k-1, q-1)$ and $N(k, q-1)$ by testing the first jar on the concatenating point. If the jar broke, then we got $k - 1$ jars and $q - 1$ chances left, otherwise, we still got k jars left but $q - 1$ chances.

Summarizing the analysis above, the solution for the HSR problem can be represented efficiently with a Bernoulli's Triangle (Fig. 1).

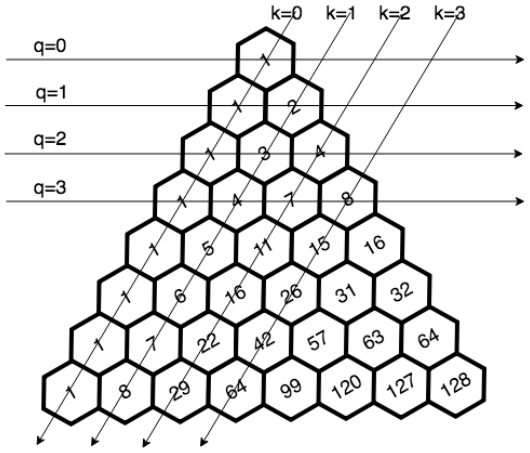

Figure 1: Theoretical values for $N(k, q)$, which can be represented as a Bernoulli's Triangle. In general, we have $N(k, q) = N(k - 1, q - 1) + N(k, q - 1), 0 < k < q$ and $N(0, q) = 1, N(k > q, q) = N(q, q) = 2^q$.

3 Transforming to Zermelo Games

3.1 General Formation

In this section, we will introduce our method to transform combinatorial optimization problems into two-phase Zermelo games. It is called Zermelo because it is a two-player finite zero-sum game with perfect information and no ties. Leveraging the logic statement (see section 2.2) of the problem, the Zermelo game is based on a finite debate between the two players. For the purpose of later contents, we introduce two roles: the Proponent (P), who claims that the statement is true, and the Opponent (OP), who argue that the statement is false. The original problem can be solved if and only if P is able to 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:

$$\exists n\ ((\forall x\ \exists y\ F(n, x, y)) \wedge (\forall n' > n\ \exists x\ \forall y\ \neg F(n', x, y)))$$

This statement implies a two phase debate game (Fig. 2):

Proposal Game This in the initial phase of the debate game, in this phase, 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:

$$(\forall x\ \exists y\ F(n,x,y)) \wedge (\forall n' > n\ \exists x\ \forall y\ \neg F(n',x,y))$$

Or we can rewrite it in a separate form:

$$A \wedge B$$

$$A := \forall x\ \exists y\ F(n,x,y)$$
$$B := \forall n' > n\ \exists x\ \forall y\ \neg F(n',x,y)$$

Specifically, OP tries to refute P by attacking either on the statement A or B. OP will accept n proposed by P if OP confirms $A = False$. OP will reject n if one is unable to confirm $A = False$. In this case, OP treat n as non-optimal, and propose 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 OP claims $\forall x\ \exists y\ F(n',x,y)$ holds. Therefore, the rejection can be regarded as a role-flip between the two players, and in order to make the debate non-trivial, in the following game, we require that P has to accept the new n' and tries to figure out the corresponding y to defeat OP.

Refutation Game This is the phase where the two players actually 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: one player (Proponent) claims $\forall x\ \exists y\ F(n,x,y)$ holds for some n, the other player (Opponent) will refute this claim by giving some instances of x so that $\forall y\ \neg F(n,x,y)$ holds. If the Proponent successfully figures out the exceptional y which makes $F(n,x,y)$ holds, the Opponent loses the game, otherwise, the Proponent loses. Also notice that, in an extreme case where $\forall x\ \exists y\ F(n,x,y)$ is theoretically true and both of the players are perfect, the Proponent can always win the game while the Opponent always loses the game.

Notice that the general form of the debate game can already handle most cases of combinatorial optimization problem which has simple parameters (i.e. x, y). It is obvious that for the Silver Ratio example, it is trivial to transform it into this form of game-play. However, for problems which have complex parameters (e.g. HSR where we have a policy π under the existential quantifier), a further transform and refinement of the refutation game have to be applied, which we will discuss in the next section.

3.2 HSR Game

In this section we introduce the HSR Game, our experiment subject, which comes from the HSR problem (section 2.2). By simply following the transforming method in section 3.1, one can derive the game immediately (Fig. 3). However, it is an abstract game because the refutation phase requires a sample of π from some policy space, which is complex. Therefore, a further transformation and refinement of the HSR refutation game are needed.

Since the crux is on π the policy, it is reasonable to understand the form of the policy at first. Recalling the definition of the policy: a policy is a function which maps a game state

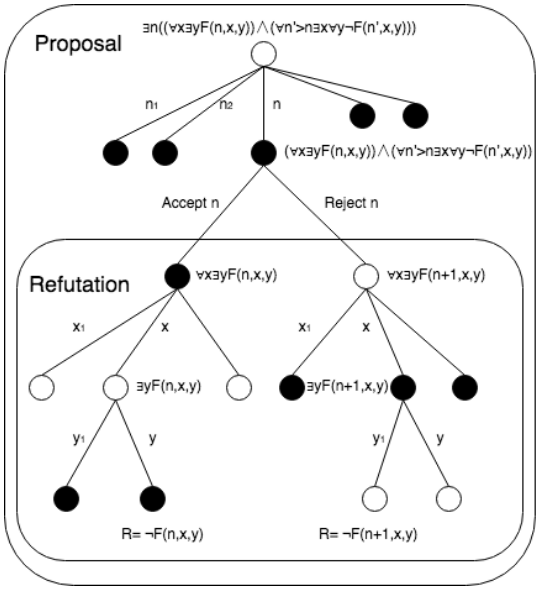

Figure 2: A general debate game where white nodes stand for the Proponent and black nodes stand for the Opponent. A role-flip happened after Opponent's rejecting of n. The refutation game can be treated uniformly where the first player is the Opponent and the second one is the Proponent. The Opponent wins if and only if the Proponent fails to find any y to make $F(n,x,y)$ holds, hence the game result $R = \neg F(n,x,y)$.

into an action, in this problem, an action was taken by a Proponent is actually a specific testing point (assume it is m) on a ladder. Also, notice that all the information a Proponent need to make a decision is (k,q,n,H), where H is a history of testing results, k,q stand for the number of jars and the number of chances to perform the test, and n is the total number of rungs on the ladder under test. Therefore, the policy we are looking for has the following form:

$$\pi(k,q,n,H) \longrightarrow m$$

After the Proponent taking each action, the result for whether a jar is broken or not can be seen immediately base on the highest safe rung chosen by the Opponent. Specifically, suppose the Opponent choose some x as the highest safe rung, then the jar will break if and only if $m > x$. Assuming the Opponent uses some meta-policy Π to find out x (it is meta because it's been used only in the abstract game tree (Fig. 3)), then the meta-policy has the following form:

$$\Pi(k,q,n) \longrightarrow x$$

Now we define a new policy

$$\hat{\pi}(k,q,n,m) = \begin{cases} \text{break at } m, & \text{if } m > \Pi(k,q,n) \\ \text{not break at } m, & \text{if } m \leq \Pi(k,q,n) \end{cases}$$

This new policy gives an recursive definition of the history H (assume t tests has been performed):

$$H^{t+1} = \hat{\pi}(k^t, q^t, n, \pi(k^t, q^t, n, H^t)) + H^t$$

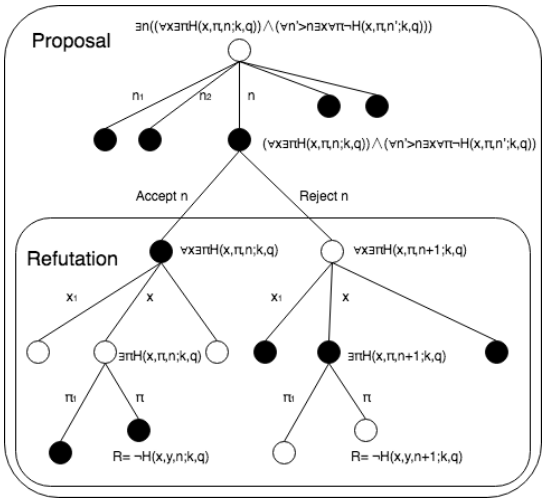

Figure 3: An abstract HSR game where the parameter π is complex since it is sampled from a policy space.

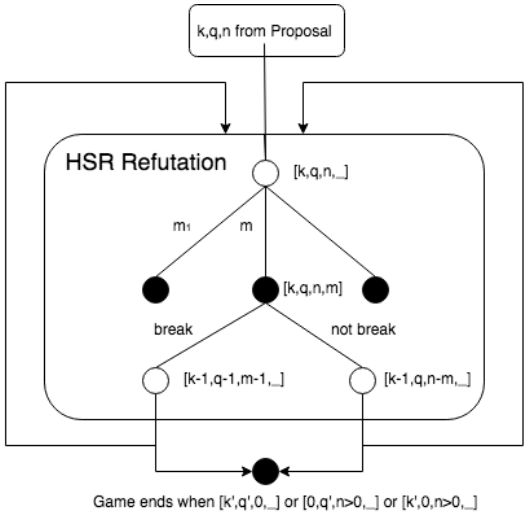

Figure 4: Refined design of the HSR refutation game, which is the one used in our experiment.

Since $\pi(k^t, q^t, n, H^t) = m^t$, we have:

$$\pi(k^{t+1}, q^{t+1}, n, H^{t+1}) = \pi(k^t, q^t, n, \hat{\pi}(k, q, n, m^t) + H^t)$$

Which then implies:

$$m^{t+1} = \pi(k^t, q^t, n, \hat{\pi}(k^t, q^t, n, m^t) + H^t)$$

The inference above actually inspired a new way to carry out the game without searching in a policy space explicitly. Using $\hat{\pi}$ and π, we refine and transform the abstract refutation game into a concrete game where Proponent and Opponent move alternatively. In this new refutation game, the Proponent will pick a testing point m based on current policy π, then the Opponent will reply "break" or "not break" based on the current policy $\hat{\pi}$. The state information changes according to the following rule:

$$(k^{t+1}, q^{t+1}) = \begin{cases} (k^t - 1, q^t - 1), & \text{if break} \\ (k^t, q^t - 1), & \text{if not break} \end{cases}$$

We define a function F which takes in an history H and output a boolean on whether the highest safe rung has been located. it can be seen that $F(H) = True$ if and only there are two consecutive testing points m and $m+1$ in the history such that a jar breaks at $m+1$ but not breaks at m. The game ends (at round T) when one of the following cases happen after an Opponent's move:

1. $k^T \times q^T = 0$ but $F(H^t) = False$, in this case, the Proponent fails to locate the highest safe rung and lose the game.

2. $F(H^t) = True$, in this case, the Opponent fails to hide the highest safe rung so that Proponent can't find it with given resources, hence the Opponent lose the game.

In practice (Fig. 4), one can assume certain physic laws as common sense knowledge to an agent and build those laws into the game mechanism. In our implementation of HSR game, since it is useless to test on a point m if one already knows that the jar will break at some $m' < m$; similarly, it is pointless to test on m if one already knows that the jar will not break at some $m' > m$. Therefore, the game state can be efficiently represent as $[k, q, n, m]$ without recording the whole history. Specifically, we apply the following state transition after an Opponent's move:

$$[k^{t+1}, q^{t+1}, n^{t+1}, \_] = \begin{cases} [k^t - 1, q^t - 1, m - 1, \_], & \text{break} \\ [k^t, q^t - 1, n^t - m, \_], & \text{not break} \end{cases}$$

The game ends in the following cases:

$$\begin{cases} [0, q^T, n > 0, \_] \vee [k^T, 0, n > 0, \_], & \text{Opponent wins} \\ [k^T, q^T, 1, \_], & \text{Proponent wins} \end{cases}$$

4 Experiment

4.1 neural-MCTS implementation

In this section, we will discuss our neural-MCTS implementation on the HSR game. Since the debate 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 is also two independent replay buffer which stores the self-play information generated from each phase respectively.

Our neural network consists of four layers of 1-D convolution neural network 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 is an indicator of the current player. The output of the neural network consists of two vectors of probability 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 putting 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, 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.

We shall mention that since it is highly time-consuming to run a complete debate game on our machines, to save time and as a proof of concept, we only run the complete game for $k = 3, q = 3$ and $n \in [1...10]$ for the time being. 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 larger parameters.

4.2 Correctness Measurement

Informally, an action is correct if it preserves a winning position. It is straightforward to derive the correctness measurement using the Bernoulli Triangle (section 2.2).

Proponent'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.

Opponent's correctness Given (q, k, n, m), When $n > N(k, q)$, any action is regarded as correct; when $n \leq N(k, q)$, Opponent 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.

4.3 Complete Game

In this experiment, we run a full debate game under the given resources $k = 3, q = 3$. Since there are two neural networks which learn the proposal game and the refutation game respectively. We measure the correctness separately: Fig. 5 shows the ratio of correctness for each player during the proposal game. And Fig. 6 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 very soon after 10 iterations. It is because, for $k = 3, q = 3$ the game is relatively simple so the neural-MCTS can quickly find the optimal policy. Even though it takes several hours to get this result, the experimental result still serves as a proof-of-concept.

4.4 Refutation Game

In order to test our method on larger cases, 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. 1), we know that $N(7, 7) = 2^7$. We set $n = N(k, q)$ so that the learning process will converge when the Proponent 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. 7 verified that the result is as expected. Then we run the same

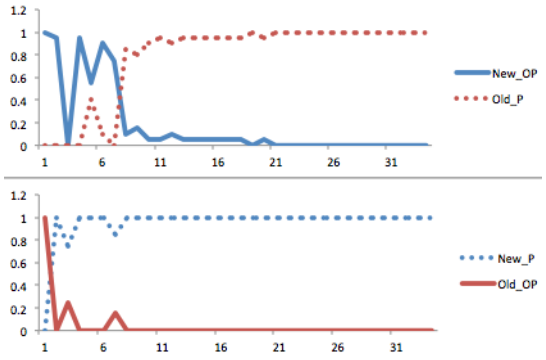

Figure 5: Correctness ratio measured for the proposal game on $k = 3, q = 3$. The legend "New_OP" means that the newly trained neural network plays as an Opponent; "Old_P" means that the previously trained neural network plays as an Proponent. The same for the following graphs.

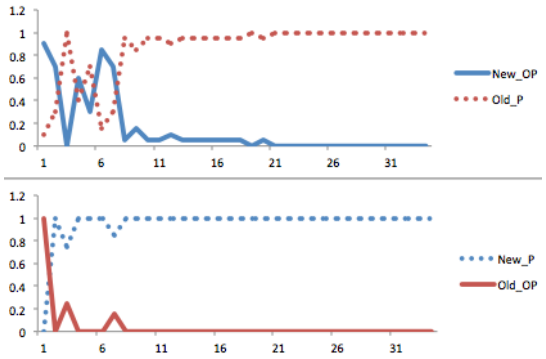

Figure 6: Correctness ratio measured for the refutation game on $k = 3, q = 3$.

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. 8, again, verified our expectation and one can see that the Proponent 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. 9) and $n = N(3, 7) - 1$ (Fig. 10). All experimental results are conforming to the ground-truth as expected.

The $HSR_{k,q}$ game is also intrinsically asymmetric in terms of training/learning because the Opponent always takes the last step before the end of the game. This fact makes the game harder to learn for the Proponent. Specifically, considering all possible consequences (in the view of the Proponent) of the last action, there are only three cases: win-win, win-lose, and lose-lose. The Opponent will lose the game if and only if the consequence is win-win. If the portion of such type of consequence is very small, then the Opponent could only focus on learning the last step while ignoring other steps. However, the Proponent has to learn every step to avoid possible paths which lead him to either win-

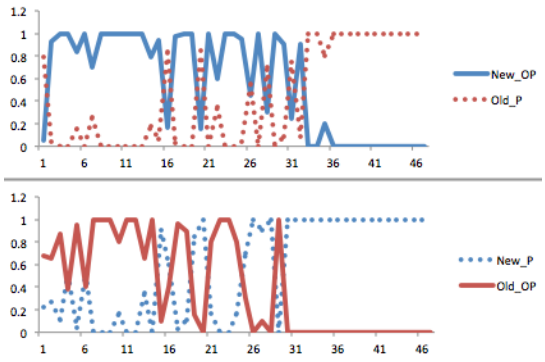

Figure 7: Refutation game on $k = 7, q = 7, n = 128$

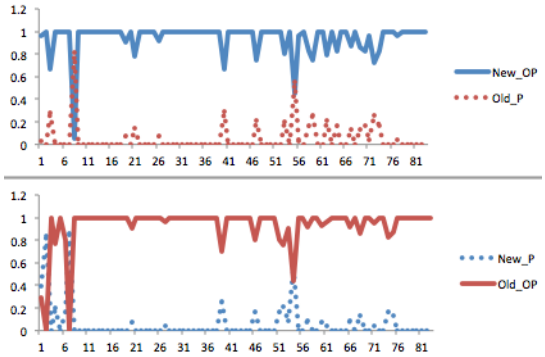

Figure 8: Refutation game on $k = 7, q = 7, n = 129$

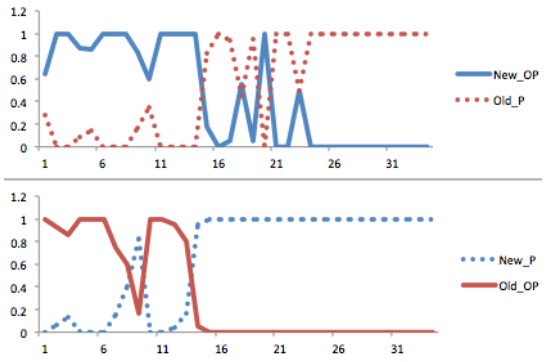

Figure 9: Refutation game on $k = 3, q = 7, n = 64$

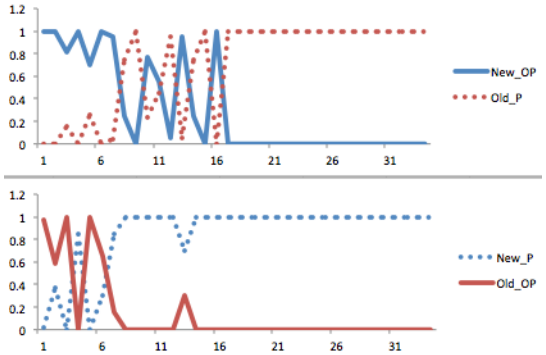

Figure 10: Refutation game on $k = 3, q = 7, n = 63$

lose or lose-lose, which, theoretically, are more frequently encountered in the end game.

5 Discussion

5.1 Asymmetry

One can observe some asymmetry emerged from the charts we presented above (section 4), and notice that it is always the case that during the beginning iterations the Opponent is always dominating until the Proponent has gained enough experience and learned enough knowledge. This asymmetry is caused by two facts: 1. the action space of the Proponent is quite different from the one of the Opponent. 2. the Opponent always takes the last step before the end of the game. These two facts make the game harder to learn for the Proponent but easier for the Opponent.

5.2 Limitations

The neural-MCTS algorithm is known to be time-consuming. It usually takes a large amount of time to converge. In order to make the algorithm run faster, we have to use resources (more CPUs, distributed parallel computing) to trade for time. 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, in order 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 Conclusion

Our original question was: 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 a generic pattern of how to translate such 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 as the Zermelo game which we give to the adapted Neural-MCTS algorithm. For our proof-of-concept example, HSR, we notice that the corresponding 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. 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. In the future, we hope to shed more light on why the AlphaZero algorithm works so well.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time Analysis of The Multiarmed Bandit Problem. *Machine learning* 47(2):235–256.

Auger, D.; Couetoux, A.; and Teytaud, O. 2013. Continuous Upper Confidence Trees with Polynomial Exploration - Consistency. In *ECML/PKDD (1)*, volume 8188 of *Lecture Notes in Computer Science*, 194–209. Springer.

Browne, C.; Powley, E. J.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Liebana, D. P.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intellig. and AI in Games* 4(1):1–43.

Kleinberg, J., and Tardos, E. 2006. *Algorithm Design*. Addison-Wesley.

Kocsis, L., and Szepesvri, C. 2006. Bandit Based Monte-Carlo Planning. In Frnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *ECML*, volume 4212 of *Lecture Notes in Computer Science*, 282–293. Springer.

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.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Rosin, C. D. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61(3):203–230.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms.

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.; and Hassabis, D. 2017a. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR* abs/1712.01815.

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.; and Hassabis, D. 2017b. Mastering the game of Go without human knowledge. *Nature* 550:354.

Weber, T.; Racanire, S.; Reichert, D. P.; Buesing, L.; Guez, A.; Rezende, D. J.; Badia, A. P.; Vinyals, O.; Heess, N.; Li, Y.; Pascanu, R.; Battaglia, P.; Hassabis, D.; Silver, D.; and Wierstra, D. 2017. Imagination-augmented agents for deep reinforcement learning.

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