Fine-Grained Evaluation of Neural Monte Carlo Tree Search Algorithms for Combinatorial Optimization Problems

Camellia Debnath
Khoury College of Computer Sciences
Northeastern University
Boston, United States
debnath.c@husky.neu.edu

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

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

***Abstract*—Google Deepmind achieved surprising success in playing Go and Chess at a super-human level using the Neural Monte Carlo Tree Search (Neural MCTS) algorithm. Although primarily aimed at playing two-player, zero-sum games, we can extend it to solve combinatorial optimization problems. Combinatorial optimization problems provide a unique way of evaluating the performance of an algorithm as complex as Neural MCTS. If we choose a problem in this class for which the ground truth is easily computable, then we can use this knowledge to measure the correctness of moves with ease, and determine exactly at what point the players become perfect not just from win-loss statistics, but also from the kind of moves they make. We choose the Highest Safe Rung (HSR) problem for our experiments, transform it into a two-player Zermelo game for the purpose of training the Neural MCTS and show experimentally that if a player is sure to win according to the ground truth, then Neural MCTS will always find a winning strategy for the player. We also show the bottom-up nature of learning of Neural MCTS, and illustrate how a fully trained Neural MCTS will also be able to make moves which can be considered "strong" by a human.**

***Index Terms*—Neural MCTS, HSR, Zermelo Gamification**

I. Introduction

In recent years we have seen ample research that has been able to substantially improve game-winning strategy by combining Reinforcement Learning algorithms with deep learning methods. Particularly, the Neural MCTS as used by AlphaGo Zero Algorithm (Silver et al., 2017 [1]) among them was able to achieve stunning success in Go, Chess, Shogi and Atari games. The primary improvement of this algorithm upon the existing RL algorithms is its tabula rasa learning, which is learning the winning strategy purely through self-play, without any knowledge of what moves are considered good or bad by human players. The algorithm only needs to know the game rules. Although much more recently, world models (Schmidhuber, 2018 [2]) and MuZero (Silver et al., 2019 [3]) have shown how to learn effective winning strategy without even knowing the game rules. But in this paper, we focus our experiments on Neural MCTS and plan to extend to MuZero in future work.

The goal of this paper is to explore the winning strategy of Neural MCTS, building on the work of Xu and Lieberherr, 2019 (Xu et al. [4]), by applying this algorithm to combinatorial optimization problems instead of the traditional 2-player games such as chess, go, shogi etc on which AlphaGo Zero was originally trained on (Silver et al. [1]). Combinatorial optimisation is the task of searching for a solution from a finite collection of possible candidate solutions that maximises the objective function (Caldwell et al [10]). In an attempt to apply the Neural MCTS techniques to general problems in other domains, we attempt to solve such a problem, the Highest Safe Rung (HSR) (Xu et al.[4]). Since Neural MCTS was originally designed to solve two-player, zero-sum games, adapting it to problems in other fields requires gamification of those problems, and also customizing the Neural MCTS algorithm. We use a technique called Zermelo gamification (introduced by Xu et al.[4]), which is a method to transform combinatorial problems to Zermelo games using a variant of Hintikka's Game-Theoretical Semantics (Hintikka [5]) and also implement a Neural MCTS designed specifically for the HSR problem. Then we conduct several experiments to understand how Neural MCTS approaches its winning strategy, and whether it is able to grasp any finer aspect of the game, such as identifying any strong move.

II. Preliminaries

A. The Highest Safe Rung Problem (HSR)

For the purposes of experiments illustrated in the later part of this paper, we focus on the combinatorial optimization problem of the highest safe rung (HSR) (similar to the egg-dropping problem (Sniedovich [12]). The HSR problem can be described as follows: consider throwing jars from a specific rung of a ladder. The jars could either break or not. If a jar is unbroken during a test, it can be used next time. A highest safe rung is a rung that for any test performed above it, the jar will break. Given k identical jars and q test chances to throw those jars, what is the largest number of rungs a ladder can have so that there is always a strategy to locate the highest safe rung with at most k jars and q tests? We need to convert this problem into a finite, perfect information game with only one winner and one loser, and during the game, players move alternately (i.e., no simultaneous moves), such that is fit for being used by a specially designed Neural MCTS algorithm to find a winning strategy. The winning strategy of the Zermelo game (Amir et al. [21], Xu et al [4]) can be used to find a solution to the original combinatorial problem.

B. Stanford AlphaZero Framework

The Stanford Alpha(Go)Zero framework is implemented based on the AlphaGoZero paper by DeepMind [1]. The framework consists of a Neural Network and a Monte Carlo Tree Search for policy improvement.

The neural network f_θ is parameterized by θ and takes as input the state s of the board. It has two outputs: a continuous value of the board state $v_\theta(s) \in$ [-1, 1] from the perspective of the current player, and a policy $p(s)$ that is a probability vector over all possible actions. When training the network, at the end of each game of self-play, the neural network is provided training examples of the form (s_t, $\overrightarrow{\pi_t}$, z_t). π_t is an estimate of the policy from state s_t, and $z_t \in [-1, 1]$ is the final outcome of the game from the perspective of the player at s_t (+1 if the player wins, -1 if the player loses). The neural network is then trained to minimize the following loss function (excluding regularisation terms):

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \overrightarrow{\pi_t}.\log(\overrightarrow{p_\theta} s_t) \tag{1}$$

The underlying idea is that over time, the network will learn what states eventually lead to wins (or losses). In addition, learning the policy would give a good estimate of what the best action is from a given state. The neural network architecture in general would depend on the game. Most board games such as Go can use a multi-layer CNN architecture. In the paper by DeepMind, they use 20 residual blocks, each with two convolutional layers.

Given a state s, the neural network provides an estimate of the policy $\overrightarrow{p_\theta}(s)$. During the training phase, the policy estimates are improved using a Monte Carlo Tree Search (MCTS). Each node in the tree represents a board configuration. Starting with an empty search tree, we expand the search tree one node (state) at a time. When a new node is encountered, instead of performing a roll-out, the value of the new node is obtained from the neural network itself. This value is propagated up the search path.

For the tree search, we maintain the following:

- $Q(s, a)$: the expected reward for taking action a from state s
- $N(s, a)$: The number of times action a was taken from state s across simulations
- $P(s, .) = p(s)$: the initial estimate of taking an action from the state s according to the policy returned by the current neural network.

The upper confidence bound $U(s, a)$ on the Q-values is calculated as:

$$U(s, a) = Q(s, a) + c_{puct}.P(s, a).\frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \tag{2}$$

c_{puct} is a hyperparameter that controls the degree of exploration. The complete Neural MCTS (AlphaZero) algorithm is as follows: The neural network is initialized with random weights, thus starting with a random policy and value network. In each iteration of the algorithm, we play a number of games of self-play. In each turn of a game, we perform a fixed number of MCTS simulations starting from the current state s_t. We pick a move by sampling from the improved policy t. This gives us a training example (s_t, t, r). The reward r is filled in at the end of the game: +1 if the current player eventually wins the game, else -1. The search tree is preserved during a game. At the end of the iteration, the neural network is trained with the obtained training examples. The old and the new networks are pitted against each other. If the new network wins more than a set threshold fraction of games (55% in the DeepMind paper, but only 10% in our case since it was observed during game play that during the initial training phase, the new network loses most of the games, so if we keep the threshold low, there will not be any updates to the existing Neural Networks), the network is updated to the new network. Otherwise, we conduct another iteration to augment the training examples.

C. Zermelo Gamification of Combinatorial Optimization Problems

We define the combinatorial optimization problem in first order logic (Hodgson [22]) as follows (Xu et al. [4]):

$$\exists n : \{G(n) \wedge (\forall n' \neg G(n'))\} \tag{3}$$

$$G(n) := \forall x \; \exists y : \{F(x, y; n)\} \tag{4}$$

In this statement n is a natural number and x, y can be any instances depending on the concrete problem. F is a predicate on n, x, y. Hence the logic statement above essentially means that there is a maximum number n such that for all x, some y can be found so that the predicate $F(x, y; n)$ is true.

The Zermelo Gamification (ZG) introduced by Xu et al[4] is an essential step towards customizing Neural MCTS for combinatorial optimization problems, since it converts such problems into a two player, zero sum game, as understood by the Neural MCTS algorithm (Silver et al[1], Anthony et al [8]) to learn optimal moves through self-play between a proponent and an opponent. It is defined to be a two-player, finite and perfect information game with only one winner and loser, and during the game, the players move alternately (i.e. no simultaneous moves, as is typical for two-player board games such as chess, go etc). Leveraging the logical statements in equations (3),(4) and (5), the Zermelo game is built on the Game Theoretical Semantic Approach (Hintikka [5]). Two roles are introduced to make it a two-player game: The Proponent (P), who claims 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 (Xu et al. [4]).

D. HSR Gamification

To formulate the HSR problem as defined in section II. A, Xu et al[4] introduce the predicate $G_{k,q}(n)$ which means there is a strategy to find the highest safe rung on an n-level ladder with at most k jars and q tests, using the following recursive strategy: After performing a test, depending on whether the jar is broken or not, the highest safe rung should only be located either above the current testing level , or below or equal to the current testing level. This fact means that the next testing level should only be located in the upper partial ladder or the lower partial ladder. Hence $G_{k,q}(n)$ can be written as:

$$G_{k,q}(n) = \exists\ 0 < m \leq n : \{G_{k-1,q-1}(m-1) \wedge G_{k,q-1}(n-m)\} \tag{5}$$

$$G_{k,q}(0) = True, G_{0,q}(n) = False, G_{k,0}(n) = False, G_{0,0}(n) = False, n > 0, k > 0, q > 0 \tag{6}$$

The interpretation of this formula is as follows: if there is a strategy to locate the highest safe rung on an n-level ladder, then it must tell a testing level m so that, no matter the jar breaks or not, the strategy can still lead to find the highest safe rung in the following sub-problems. More specifically, for sub-problems, we have $G_{k-1,q-1}(m-1)$ if the jar breaks, that means we only have $k-1$ jars and $q-1$ tests left to locate the highest safe rung in the lower partial ladder (which has $m-1$ levels). Similarly, $G_{k,q-1}(n-m)$ for upper partial ladder. Therefore, the problem is solved recursively, and until there is no ladder left, which means the highest safe rung has been located, or there is no jars/tests left, which means one has failed to locate the highest safe rung. With the notation of $G_{k,q}(n)$, the HSR problem now can be formulated as [4]:

$$HSR_{k,q} = \exists n : \{G_{k,q}(n) \wedge (\forall n' > n \neg G_{k,q}(n'))\} \tag{7}$$

Introducing the universal quantifier, we regard the environment as another player who plays against the tester so that, after each testing being performed, the environment will tell the tester whether the jar is broken or not. In this way, locating the highest safe can be formulated as following [4] :

$$G_{k,q}(n) = \begin{cases} True, \text{if } n = 1 \\ False, \text{if } n > 1 \wedge (k = 0 \vee q = 0) \\ \exists m\ \in [1..n-1] : \{G_{k-1,q-1}(m) \wedge \\ \qquad\qquad G_{k,q-1}(n-m)\} \end{cases} \tag{8}$$

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

HSR being a trivial problem, the solution can be computed and also represented easily using a Bernoulli's Triangle (Fig. 1). We use the notation $N(k,q)$ to represent the solution for the HSR problem given k jars and q tests. In other words, $G_{k,q}(N(k,q)) \wedge (\forall n' > N(k,q) \neg G_{k,q}(n')$ (Xu et al.[4]).

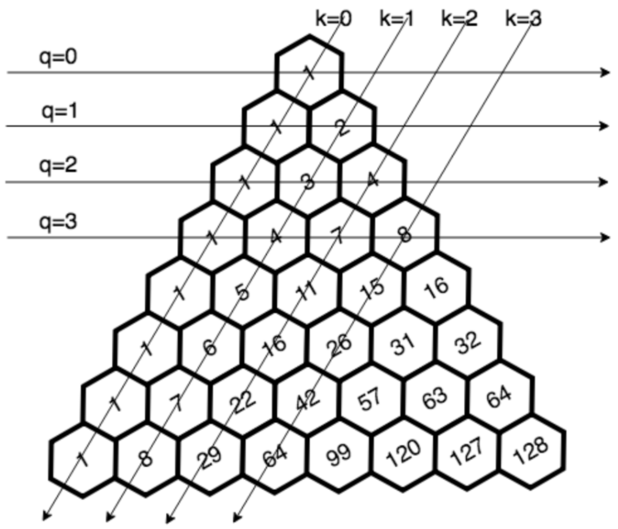


Fig. 1: Theoretical values for maximum n in HSR problem with given k, q which can be represented as a Bernoulli's Triangle.

III. Experiments

A. Correctness Measurement

After defining the HSR problem suitable to adaptation for the Neural MCTS algorithm, we conducted two sets of experiments to track the learning pace of the algorithm, as well as correctness. Before we explain those methods, it's important to have a primer on the correctness of the moves by Proponent and Opponent, given as follows (Xu et al. [4])

- 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.

- 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 $m < n - N(k-1,q-1)$ and take the action "break" if $m > N(k-1,q-1)$. Otherwise there is no correct action.

1) Switch Experiment: Our first experiment is concerned with counting the number of "Switches" per iteration of the training process as a measure of how close the Proponent or Opponent is to finding the winning strategy. A switch is defined as follows:

Given that it is a two-player game where players move alternately, a switch takes place inside one game when the previous player makes a wrong move and the current player captures that mistake (i.e. it makes a correct move). A wrong move can only take place if and only if there is at least one potential correct move to choose, but the player did not make any correct move. In a situation where the player is forced to make an incorrect move, i.e. the player is in a losing position and has no correct move to choose from, then that incorrect move is not considered as a mistake. A perfect Opponent will always take advantage of the mistakes made by the Proponent, i.e., if the Proponent makes a mistake, the perfect Opponent will be in a winning position, and it will want to make sure that it stays in the winning position till the end of the game. The same can be said for the other way round. Section III D provides more detailed analysis of switch statistics.

2) Opponent Policy Tree: For our next set of experiments, we generate a version of a Policy Tree. Zermelo games are in a space of games where there are multiple starting positions. For example, the HSR game can have starting position HSR($k = 3, q = 5, n = 50$) or HSR($k = 2, q = 3, n = 7$). A full game tree for such a problem will therefore grow too large with the size of the problem, and will add to visual confusion and increased difficulty in forming decisions about the nature of the game states. It is useful to introduce policy trees in this context. A policy tree is a sub-tree of the game tree. It is a decision tree which shows the chosen moves of one player P and all moves and their probabilities learned by Neural MCTS for the other player OP. We call such policy trees Opponent Policy Trees because only the policy for OP is shown in the tree. The moves selected by P are explicitly represented. If the starting position is:

- winning for P, the Opponent policy tree shows how P should select its moves based on the behavior of OP. If P is perfect, the policy probabilities for OP don't matter because OP will lose at all leaves of the policy tree.
- winning for OP, the opponent policy tree shows how OP should select its moves based on the probabilities shown in the policy tree. If OP is perfect, the moves of P don't matter because OP will find a root-to-leaf path where OP will win.

Our policy trees contain additional information besides the game state and the OP policy probabilities. Another added advantage to such policy trees is that by showing only the actual actions taken by Proponent, we can determine easily how perfect the player is with ease, and by showing all the possible moves of the Opponent (which are only 2), we can also easily determine if the Opponent player has achieved perfection. This will be clearer as we discuss our Opponent Policy Tree trees in detail.

In a game, the alternate moves by the Proponent and the Opponent are as follows: The Proponent proposes a a number (m) according to how it "thinks" should be the value of n for the given value of k and q. The next turn is of the Opponent, in which the Opponent gives a verdict as either "break" or "not break". To generate our Opponent Policy Tree, we first train our Neural MCTS for a certain number of iterations. Then for one game, we obstruct decisions from the Opponent, and force the Proponent to play out both the cases for "break" and "not break" scenarios, simultaneously, we also print the probability with which the Opponent would have predicted "break" or "not break". Similar to the switch statistics, we can say that the learning is assumed to be converged, when both the Proponent and the Opponent can make the right moves with high probability (close to 1). More of it will be clear when we illustrate the experiment results in subsection C.

B. Experiment Design

The bigger the values of k, q, n, the bigger the action space and the Opponent Policy Tree, it is highly time consuming in our machines to train the Neural MCTS for the Zermelo Games. 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, n parameters.

C. Experiment Implementation

Since the Zermelo game has two phases, namely proposal and refutation, we implement two independent Neural Networks to learn the two games respectively. The Neural MCTS accessed the first Neural Network during the proposal game and the second one during the refutation game. The neural networks consist of four 1-D convolution Neural Networks and two dense layers. The input is a tuple (k, q, n, m, r), where k, q are resources, n is the number of rungs on the current ladder, m is the testing point and r indicates the current player. The Neural Network outputs two vectors A_P, A_{OP} which are the probabilities of the action space for Proponent and Opponent respectively, and a scalar v as the game result evaluation. The loss-value that is minimized during the training is then defined as:

$$L = (v - v_0)^2 + H(A_P, \pi_P) + H(A_{OP}, \pi_{OP}) \qquad (9)$$

H here is the cross-entropy function and v_0, π_P, π_{OP} are target values (section II B). The algorithm continues iterating till convergence (i.e., till the loss value is lesser than a target value), or till a pre-defined number of maximum iterations. During each iteration, the "number of self-plays" episodes of self-plays will be executed through the Neural MCTS using the current Neural Network, and data generated using self-play will be stored in the replay buffer. The newly trained neural network and the previous old neural network are pitted against each other. During the competition phase, the new neural network will first play as the OP for half the number of self-play matches, and then it will play as the P for the other half. 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 $\frac{W_{new}}{W_{new}+W_{old}}$ is more significant than a given threshold ("update threshold").

D. Results

1) Switch Statistics: We generated switch statistics () for 2 sets of HSR problems, namely for (k, q, n)= (3,4,15) (Fig. 2) and (k, q, n)=(4,6,56) (Fig. 3). We use the following run configurations: no. of self-plays: 100, No. of MCTS simulations:

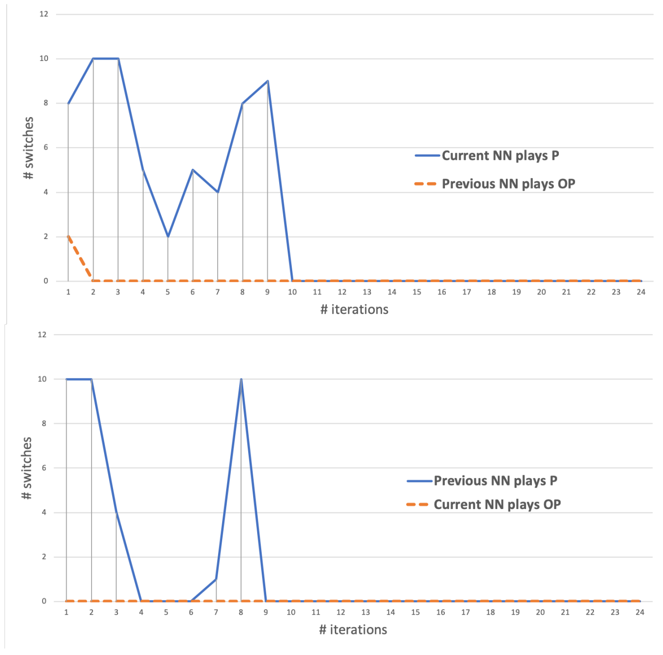


Fig. 2: Number of win-loss switches for an execution of the algorithm (trained for 24 iterations, k, q, n=(3,4,15). In the top figure, we can see that the learning converges for the previous NN that plays OP just after 2 iterations, but the current NN that plays P takes much longer time to converge (after 10 iterations). In the bottom image there are even lesser switches for the previous NN that playes P, and zero switches for OP. This illustrates the learning imbalance for the two players

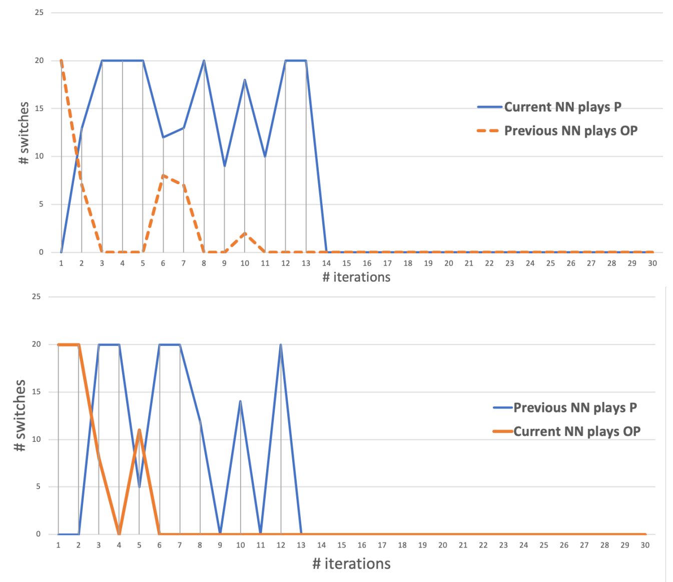


Fig. 3: Number of win-loss switches for an execution of the algorithm (trained for 30 iterations, k, q, n=(4,6,56)). This case being more complex than the one in Fig. 2, the convergence takes higher number of iterations as expected. But in both the top and bottom cases, the NN that plays P takes longer time to converge than the NN that plays OP, once again illustrating the asymmetric learning nature.

128, and update threshold: 0.1. Both set of experiments use the current NN for playing P (Proponent) and previous NN for playing OP (Opponent) in the first half of the self-plays, and reverse for the second half (section III C).

A small number of switches means most likely that both players are close to perfect and do not make a lot of mistakes. Even when at least one of the players is close to perfect, we should expect a lower number of switches since if the imperfect player makes a mistake, it perfect player will catch the mistake and then onward never relinquish its winning position. Hence, it is possible that there is no switch if one of the players is in a winning position to start with. A high number of switches means that both players show weaknesses. However, they show some good skills too because they catch the mistakes of the other player for at least one move. There is a small probability that a small number of switches means that both the players are imperfect so that even if a player makes a mistake, the others cannot catch it. But, since we can safely assume that during the initial stages of the training neither the proponent nor the opponent has learned to play perfectly, hence if we observe a higher number of switches during this period, we should train the Neural MCTS further till we can observe if the number of switches decreases. We observed in all our experiments that after a certain number of iterations for a given problem, the number of switches always drop to zero for all further iterations, which can mean that now both the players have achieved perfection. However, Opponent Policy Trees [Section III D(2)] are a more efficient way of verifying this truth.

However, Opponent Policy Trees [Section III D(2)] are a more efficient way of verifying this truth.

2) Opponent Policy Trees: We produced the Opponent Policy Trees from a Neural MCTS trained with no. of iterations: x (x varies with the size of the problem, since for larger values of k, q, n, x is also large), no. of episodes: 100, update threshold: 0.1, no. of MCTS Simulations: 128, arena compare (No.of games per iteration): 40, c_{puct}: 1, TensorFlow version: 1.13.0rc2. Each node is comprised of 7 values: the first 3 values represent k, q, and n respectively; m represents the value proposed by the Proponent Neural Network; W represents whose winning position this particular combination of k, q, and n is; and lastly, V and p represent the probability of win for the Proponent, and with which probability the Opponent will choose this node. The node which is a winning position for Opponent is printed in bold. At each node, instead of allowing the Opponent to make the decision, we force the Neural MCTS to play out both the scenarios for when the jar breaks, and when it does not break, that is the reason each node has two children, the first results when the jar breaks, the second otherwise (hence the value p, which indicates which node is likelier to be chosen by the Opponent).

The trees are printed in an indented pre-order traversal format as in Figure 4.

```
root
├→left
│  ├→lleft
│  └→rleft
└→right
   ├→lright
   └→rright
```

Fig. 4: Tree structure in indented pre-order traversal format.

```
(k,q,n)=3,4,15,  m=12,  W=P,  V=[0.99441797],  p=-
├→(k,q,n)=2,3,12,  m=3,  W=O,  V=[0.9383285],  p=0.9776839
│  ├→(k,q,n)=1,2,3,  m=1,  W=P,  V=[0.9397475],  p=0.6169514
│  │  ├→(k,q,n)=0,1,1,  m=-,  W=P,  V=[0.99907607],  p=0.3698314
│  │  └→(k,q,n)=1,1,2,  m=1,  W=P,  V=[0.95362586],  p=0.6301685
│  │     ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[0.95059437],  p=0.5063509
│  │     └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.9625776],  p=0.4936491
│  └→(k,q,n)=2,2,9,  m=2,  W=O,  V=[0.02084012],  p=0.38304856
│     ├→(k,q,n)=1,1,2,  m=1,  W=P,  V=[0.95362586],  p=0.10611043
│     │  ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[0.95059437],  p=0.5063509
│     │  └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.9625776],  p=0.4936491
│     └→(k,q,n)=2,1,7,  m=1,  W=O,  V=[-0.9865037],  p=0.89388955
│        ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.9625776],  p=0.016640522
│        └→(k,q,n)=2,0,6,  m=-,  W=O,  V=[-0.9979049],  p=0.98335946
└→(k,q,n)=3,3,3,  m=1,  W=P,  V=[0.9983739],  p=0.022316067
   ├→(k,q,n)=2,2,1,  m=1,  W=P,  V=[0.9999318],  p=0.6883524
   └→(k,q,n)=3,2,2,  m=1,  W=P,  V=[0.9993113],  p=0.31164762
      ├→(k,q,n)=2,1,1,  m=1,  W=P,  V=[0.9992142],  p=0.7881197
      └→(k,q,n)=3,1,1,  m=1,  W=P,  V=[0.99899304],  p=0.2118803
```

Fig. 5: Opponent Policy Tree for $k = 3$, $q = 4$, $n = 15$, produced after training Neural MCTS for 1 iterations. [m: value proposed by Proponent Neural Network; W: winning position (either O for Opponent, or P for Proponent); V: probability of win for the Proponent; p: probability with which Opponent chooses this node]. The proponent still has not learnt optimum use of resources, leading to a winning path for Opponent outlined by the boldfaced nodes.

We get a good idea about the number of iterations we need for the learning for Neural MCTS to converge by looking at the Opponent Policy Trees. Opponent Policy Trees help to visualize if the P and OP have indeed become perfect players after the so called convergence. For example, let us consider the case $k = 3$, $q = 4$ and $n = 15$. From Bernoulli's triangle (Figure 1), we can see that this is a winning case for the Proponent. So we produce the trees after training the Neural MCTS for 1 iteration (Fig. 5), 5 iterations (Fig. 6) and 30 iterations (Fig. 7).

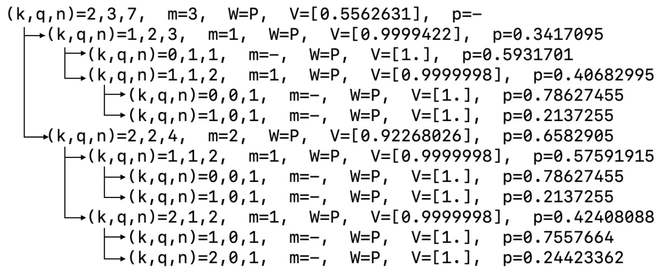
The tree produced after only 1 iteration (Fig. 5) has a few bold nodes, meaning they are winning positions for the Opponent. This can be directly verified from Bernoulli's Triangle (Fig. 1) as $N(k, q) < n$, meaning Proponent can never determine the highest safe rung with the given resources. Now, this is clearly an under-trained network, since for $k = 3$, $q = 4$ and $n = 15$, Neural MCTS should always find the winning strategy. So we train Neural MCTS further, and we can see that after 5 iterations (Fig. 6), the Proponent has learnt to choose m wisely enough so that it never relinquishes its winning position to the Opponent. However, if we look at the V values of the nodes, we can see that there is still more room to improve, because although the Proponent has learnt not to give up winning position, it still does so with relatively low probability. For example in Fig. 6, the V value of the Proponent is only ≈ 0.14, but it should be ideally close to 1.

```
(k,q,n)=3,4,15,  m=7,  W=P,  V=[0.1391548],  p=-
├→(k,q,n)=2,3,7,  m=3,  W=P,  V=[0.964941],  p=0.37479645
│  ├→(k,q,n)=1,2,3,  m=1,  W=P,  V=[0.9992331],  p=0.38520908
│  │  ├→(k,q,n)=0,1,1,  m=-,  W=P,  V=[0.9999989],  p=0.4095275
│  │  └→(k,q,n)=1,1,2,  m=1,  W=P,  V=[0.9996943],  p=0.59047246
│  │     ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[0.9987794],  p=0.49645165
│  │     └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.99968207],  p=0.50354826
│  └→(k,q,n)=2,2,4,  m=2,  W=P,  V=[0.9787988],  p=0.6147909
│     ├→(k,q,n)=1,1,2,  m=1,  W=P,  V=[0.9996943],  p=0.48430195
│     │  ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[0.9987794],  p=0.49645165
│     │  └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.99968207],  p=0.50354826
│     └→(k,q,n)=2,1,2,  m=1,  W=P,  V=[0.99990445],  p=0.5156981
│        ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.99968207],  p=0.5487849
│        └→(k,q,n)=2,0,1,  m=-,  W=P,  V=[0.9998163],  p=0.45121503
└→(k,q,n)=3,3,8,  m=4,  W=P,  V=[0.9653546],  p=0.6252036
   ├→(k,q,n)=2,2,4,  m=2,  W=P,  V=[0.9787988],  p=0.5379635
   │  ├→(k,q,n)=1,1,2,  m=1,  W=P,  V=[0.9996943],  p=0.48430195
   │  │  ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[0.9987794],  p=0.49645165
   │  │  └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.99968207],  p=0.50354826
   │  └→(k,q,n)=2,1,2,  m=1,  W=P,  V=[0.99990445],  p=0.5156981
   │     ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.99968207],  p=0.5487849
   │     └→(k,q,n)=2,0,1,  m=-,  W=P,  V=[0.9998163],  p=0.45121503
   └→(k,q,n)=3,2,4,  m=2,  W=P,  V=[0.9887683],  p=0.46203652
      ├→(k,q,n)=2,1,2,  m=1,  W=P,  V=[0.99990445],  p=0.5416288
      │  ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.99968207],  p=0.5487849
      │  └→(k,q,n)=2,0,1,  m=-,  W=P,  V=[0.9998163],  p=0.45121503
      └→(k,q,n)=3,1,2,  m=1,  W=P,  V=[0.99990636],  p=0.45837125
         ├→(k,q,n)=2,0,1,  m=-,  W=P,  V=[0.9998163],  p=0.58849365
         └→(k,q,n)=3,0,1,  m=-,  W=P,  V=[0.99986887],  p=0.41150635
```

Fig. 6: Opponent Policy Tree for $k = 3$, $q = 4$, $n = 15$, produced after training Neural MCTS for 5 iterations. [m: value proposed by Proponent Neural Network; W: winning position (either O for Opponent, or P for Proponent); V: probability of win for the Proponent; p: probability with which Opponent chooses this node]. The proponent has learnt to optimize resources now but training is further required to increase Proponent's chance of winning (V values).

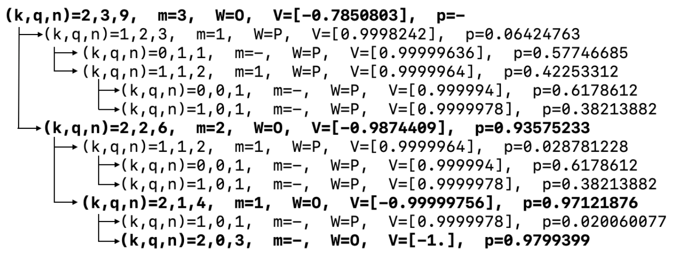
```
(k,q,n)=3,4,15,  m=7,  W=P,  V=[0.9623042],  p=-
├→(k,q,n)=2,3,7,  m=3,  W=P,  V=[0.994187],  p=0.28154776
│  ├→(k,q,n)=1,2,3,  m=1,  W=P,  V=[0.9999977],  p=0.32826352
│  │  ├→(k,q,n)=0,1,1,  m=-,  W=P,  V=[1.],  p=0.43060043
│  │  └→(k,q,n)=1,1,2,  m=1,  W=P,  V=[1.],  p=0.56939954
│  │     ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[1.],  p=0.54451966
│  │     └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.45548034
│  └→(k,q,n)=2,2,4,  m=2,  W=P,  V=[0.99934417],  p=0.6717365
│     ├→(k,q,n)=1,1,2,  m=1,  W=P,  V=[1.],  p=0.5201886
│     │  ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[1.],  p=0.54451966
│     │  └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.45548034
│     └→(k,q,n)=2,1,2,  m=1,  W=P,  V=[1.],  p=0.4798113
│        ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.5731275
│        └→(k,q,n)=2,0,1,  m=-,  W=P,  V=[1.],  p=0.4268725
└→(k,q,n)=3,3,8,  m=4,  W=P,  V=[0.9961866],  p=0.7184522
   ├→(k,q,n)=2,2,4,  m=2,  W=P,  V=[0.99934417],  p=0.51437205
   │  ├→(k,q,n)=1,1,2,  m=1,  W=P,  V=[1.],  p=0.5201886
   │  │  ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[1.],  p=0.54451966
   │  │  └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.45548034
   │  └→(k,q,n)=2,1,2,  m=1,  W=P,  V=[1.],  p=0.4798113
   │     ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.5731275
   │     └→(k,q,n)=2,0,1,  m=-,  W=P,  V=[1.],  p=0.4268725
   └→(k,q,n)=3,2,4,  m=2,  W=P,  V=[0.99974245],  p=0.48562795
      ├→(k,q,n)=2,1,2,  m=1,  W=P,  V=[1.],  p=0.54238373
      │  ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.5731275
      │  └→(k,q,n)=2,0,1,  m=-,  W=P,  V=[1.],  p=0.4268725
      └→(k,q,n)=3,1,2,  m=1,  W=P,  V=[1.],  p=0.45761627
         ├→(k,q,n)=2,0,1,  m=-,  W=P,  V=[1.],  p=0.57770675
         └→(k,q,n)=3,0,1,  m=-,  W=P,  V=[1.],  p=0.42229325
```

Fig. 7: Opponent Policy Tree for $k = 3$, $q = 4$, $n = 15$, produced after training Neural MCTS for 30 iterations. [m: value proposed by Proponent Neural Network; W: winning position (either O for Opponent, or P for Proponent); V: probability of win for the Proponent; p: probability with which Opponent chooses this node]. Learning can be considered complete.



```
(k,q,n)=2,3,7,  m=3,  W=P,  V=[0.5562631],  p=-
 ├→(k,q,n)=1,2,3,  m=1,  W=P,  V=[0.9999422],  p=0.3417095
 │   ├→(k,q,n)=0,1,1,  m=-,  W=P,  V=[1.],  p=0.5931701
 │   └→(k,q,n)=1,1,2,  m=1,  W=P,  V=[0.9999998],  p=0.40682995
 │       ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[1.],  p=0.78627455
 │       └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.2137255
 └→(k,q,n)=2,2,4,  m=2,  W=P,  V=[0.92268026],  p=0.6582905
     ├→(k,q,n)=1,1,2,  m=1,  W=P,  V=[0.9999998],  p=0.57591915
     │   ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[1.],  p=0.78627455
     │   └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.2137255
     └→(k,q,n)=2,1,2,  m=1,  W=P,  V=[0.9999998],  p=0.42408088
         ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[1.],  p=0.7557664
         └→(k,q,n)=2,0,1,  m=-,  W=P,  V=[1.],  p=0.24423362
```

Fig. 8: Opponent Policy Tree for $k = 2$, $q = 3$, $n = 7$, produced after training Neural MCTS for 25 iterations. [m: value proposed by Proponent Neural Network; W: winning position (either O for Opponent, or P for Proponent); V: probability of win for the Proponent; p: probability with which Opponent chooses this node].



```
(k,q,n)=2,3,9,  m=3,  W=O,  V=[-0.7850803],  p=-
 ├→(k,q,n)=1,2,3,  m=1,  W=P,  V=[0.9998242],  p=0.06424763
 │   ├→ (k,q,n)=0,1,1,  m=-,  W=P,  V=[0.99999636],  p=0.57746685
 │   └→ (k,q,n)=1,1,2,  m=1,  W=P,  V=[0.9999964],  p=0.42253312
 │       ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[0.999994],  p=0.6178612
 │       └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.9999978],  p=0.38213882
 └→(k,q,n)=2,2,6,  m=2,  W=O,  V=[-0.9874409],  p=0.93575233
     ├→ (k,q,n)=1,1,2,  m=1,  W=P,  V=[0.9999964],  p=0.028781228
     │   ├→(k,q,n)=0,0,1,  m=-,  W=P,  V=[0.999994],  p=0.6178612
     │   └→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.9999978],  p=0.38213882
     └→ (k,q,n)=2,1,4,  m=1,  W=O,  V=[-0.99999756],  p=0.97121876
         ├→(k,q,n)=1,0,1,  m=-,  W=P,  V=[0.9999978],  p=0.020060077
         └→(k,q,n)=2,0,3,  m=-,  W=O,  V=[-1.],  p=0.9799399
```

Fig. 9: Opponent Policy Tree for $k = 2$, $q = 3$, $n = 9$, produced after training Neural MCTS for 25 iterations. [m: value proposed by Proponent Neural Network; W: winning position (either O for Opponent, or P for Proponent); V: probability of win for the Proponent; p: probability with which Opponent chooses this node].

Proceeding this way, training Neural MCTS up to 30 iterations gives us a close to ideal outcome (Fig. 7).

Similarly, the values $k = 2$, $q = 3$ and $n = 7$ are winning positions for the Proponent. Since $N(2,3) = 7$ from Bernoulli's triangle, Proponent should find a winning strategy for this game. The Opponent Policy Tree in Fig. 8 confirms this fact. None of the nodes are in bold, which means Proponent always maintains its winning position after learning converges (25 iterations), and does so with very high probability ($V \approx 1$). Conversely, for the Opponent there is no winning strategy, and when the Proponent is at the winning position, it is confused as to what verdict to give, which can be seen from the probability values p. For almost all the nodes except the first, none of the probability values are high enough to take definitive action, i.e. the Opponent is confused as to what action to take when the Proponent is perfect.

In Fig. 9, we deal with the case $k = 2$, $q = 3$ and $n = 9$. Since $n > N(2,3)$, Proponent cannot win in this case. Hence for all the boldfaced nodes, that is, the nodes for which the Proponent is in the losing position, the Opponent decides "not break" with probability ≈ 1 so that it can maintain its winning position. That is in a game, the perfect Opponent will always catch a wrong move by the Proponent and maintain its winning position.

IV. Limitations

The primary limitation we faced while conducting our experiments was in terms of the computing power of our machines. Even for small cases such as (k, q, n) = (3,4,15), it takes around 3-4 hours for 40 iterations of the algorithm. Also, every time we want to train the Neural MCTS, we have to start from ground up, meaning, the training for (k, q, n) = (3,4,15) cannot be built upon training of smaller cases, such as (k, q, n) = (2,3,7).

V. Future Work

Since HSR is just one particular example of a combinatorial optimization problem, we can extend Neural MCTS into other domains in the future. With the publication of MuZero (Silver et al, 2019 [3]), another future focus of our experiments is evaluating MuZero for domains other than games such as Go, Chess or Shogi. The HSR problem is a good starting candidate since its ground truth can easily be calculated from Bernoulli's triangle, and for given values of k, q, n, we can decide whether the Proponent or Opponent should be the winner. Another future focus can be designing balanced learning for both the proponent and opponent, which in the present case is imbalanced (Fig 2,3))

VI. Related Work

In our work we use Bernoulli's Triangle [4] as a ground-truth measure for verifying the winning/losing positions of the Proponent (hence also Opponent). This knowledge is essential in determining the correctness of a move made by the player. For example, if Proponent is at a winning position according to Bernoulli Triangle (Section III. A), and chooses to take an action such that the next state is a losing position, we can definitely consider it to be a bad move (provided the Proponent had the option to make a move which would have resulted into a winning position in the next state).

In their work, Xu and Lieberherr [4] use correctness ratio as a measurement for tracking the learning progress of the Neural MCTS. They propose a novel method of adapting combinatorial problems to Zermelo Games, which was used in this paper for the gamification of the HSR problem.

Soemers et al [7] propose a novel objective function known as the Tree-Search Policy Gradient (TSPG) objective for policies in Markov Decision Processes. Policies can be trained to optimize this objective using self-play, similar to cross-entropy- based policies in AlphaGo Zero. The goal of this modified objective function is to extract interpretable strategies, rather than prioritizing state of the art game-play. Such an object function, hence, does not need such extent of exploration, and trains policies that are not exploratory. They claim that this policy function extracts stronger strategies since does not have an incentive to explore. In our work we keep the policy function for Neural MCTS unchanged, and try to understand if Neural MCTS approaches winning strategy by figuring out the strong moves. For example, one of the "strong" strategy that it clearly learns for the case of the HSR problem is that the Proponent Neural Network always proposes a rung level $m \leq n/2$, where n is the total number of rungs, to throw the jar from in any test case.

Alhough there have been a lot of work on the area of solving combinatorial optimization problems with Reinforcement Learning (Barrett et al [11], Miagkikh et al. [14], Bello et al. [15], Khalil et al. [16]), our work tries to analyse the learning procedure of Neural MCTS with respect to ground-truth, which makes it easier to observe the gradual improvement of the algorithm in terms of human-interpretable game states. Lanzi [19] proposes evaluating the strength of a player in terms of the minimum number of iterations required by the MCTS to become equivalent to a target player. They use a vanilla MCTS for this purpose and use score difference (win-loss statistics for the target player and the candidate player) and True Skill measurements (Moser [20]). If we try to extend this to Neural MCTS, we can only find the comparative strength of a player and how many iterations we need to train the Neural MCTS to reach this strength, but it will remain unknown as to why the player is strong, that is what changes in its decision making to grant it its strength. In our work, we show why the player becomes stronger, for the Proponent it consists of learning to choose a computationally sensible value of $m \leq n$, and maintaining a winning position (for both Proponent and Opponent). Lan [9] shows a way to improve the performance of neural MCTS by running two separate MCTS on two different sized neural networks. The small neural network allows more MCTS simulation while the large one has more learning capacity. In our work, we also use two neural networks, but they shared the same MCTS. It is interesting to test this idea in our future work, though.

VII. Conclusions

We gained exciting insights into how Neural MCTS works for the semantic game associated with the HSR problem. After sufficient training, the tabula rasa trial-and-error approach of Neural MCTS always found the correct solution (winning strategy) for the examples we tried. We introduced switch statistics and the concept of opponent policy trees to analyze the learning of game playing. We conjecture that the Neural MCTS approach is quite general, and it solves many instances of combinatorial problems defined by interpreted first-order predicate logic, such as the HSR problem.

We provide evidence that on useful games, Neural MCTS not only approximates a winning strategy, but it finds a perfect winning strategy, an impressive skill for a problem-solving approach which is automatic.

We note the bottom-up learning of the algorithm. After training our network for a particular set of (k, q, n), We were able to produce correct Opponent Policy Trees for other sets of (k, q, n) in cases where the values of k, q, n all were lesser than the one the network was originally trained on. Alternatively speaking, if we train our network for a set of (k, q, n), it can perfectly play games where the new values $k' \geq k$ and $q' \geq q$, but $n' \leq n$, because when we keep the number of rungs non-increasing, but increase the number of tests and jars, the problem becomes easier to solve, since we have more tests and jars at our disposal to determine the highest safe rungs. But when we increase the value of n, complete relearning is required as expected.

Neural MCTS also learns "good moves" for the game. In Fig. 5, only after 1 training iteration, Neural MCTS has not learned optimum use of resources yet, which is evident from the imbalanced split of resources between the two sub-trees at the root $(k, q, n) = 3, 4, 15$. In one of the sub-problems, it tries to solve HSR(2,3,12), which is a losing position for the Proponent, and the Opponent catches this mistake and Proponent loses the game. But after 5 iterations (Fig. 6), it wisely chooses m=7 at the root, thus evenly dividing up the resources and maintaining Proponent's winning position.

References

[1] Silver, D., Schrittwieser, J., Simonyan, K. et al. "Mastering the game of Go without human knowledge". Nature 550, 354–359 (2017) doi:10.1038/nature24270 .

[2] Schmidhuber, World Models (2018) .

[3] Silver, D., Schrittwieser, J., Simonyan, K. et al. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model". arXiv:1911.08265 (2019)

[4] Xu, R., Lieberherr, K. (2020). "Learning self-play agents for combinatorial optimization problems", The Knowledge Engineering Review, 35, E11. doi:10.1017/S026988892000020X

[5] Hintikka, J., " Game-theoretical semantics: insights and prospects." Notre Dame J. Formal Logic 23, 2 (04 1982), 219–241 (1982.)

[6] Xing, Z., Tu, S., "A Graph Neural Network Assisted Monte Carlo Tree Search Approach to Traveling Salesman Problem" ICLR 2020 Conference Blind Submission.

[7] Soemers, D., Piette, E., Stephenson, M., Browne, C., "Learning Policies from Self-Play with Policy Gradients and MCTS Value Estimates", arXiv:1905.05809, 2019

[8] Anthony, T., Tian, Z., Barber, D., "Thinking Fast and Slow with Deep Learning and Tree Search", arXiv:1705.08439, 2017

[9] Lan, L., Li W., Wei T., Wu I., "Multiple Policy Value Monte Carlo Tree Search", arXiv:1905.13521, 2019

[10] Caldwell, J., Watson, R., Thies, C., Knowles, J., "Deep Optimisation: Solving Combinatorial Optimisation Problems using Deep Neural Networks", arXiv:1811.00784 , 2018

[11] Barrett, T., Clements, W., Jakob N. Foerster, Lvovsky, A., "Exploratory Combinatorial Optimization with Reinforcement Learning", arXiv:1909.04063, 2020

[12] Sniedovich, M., "OR/MS games: 4. the joy of egg-dropping in Braunschweig and Hong Kong", INFORMS Trans. Edu. 4(1), 48–64., 2003

[13] Francois, A., Cappart, Q.,Rousseau, L., "How to Evaluate Machine Learning Approaches for Combinatorial Optimization: Application to the Travelling Salesman Problem.", 2019

[14] Miagkikh, V., Punch, William. "An Approach to Solving Combinatorial Optimization Problems Using a Population of Reinforcement Learning Agents.", 1999

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

[16] Khalil, E., Dai, H., Zhang, Y., Dilkina, B. Song, L. , Learning combinatorial optimization algorithms over graphs, in 'Advances in Neural Information Processing Systems', pp. 6348–6358., 2017

[17] Lynce, I. and Ouaknine, J., "Sudoku as a SAT Problem. 9th International Symposium on Artificial Intelligence and Mathematics.", 2006

[18] Weber, T., "A SAT-based Sudoku Solver. TU Munich", 2005

[19] Lanzi, P., "Evaluating the Complexity of Players' Strategies using MCTS Iterations", 2019 IEEE Conference on Games (CoG), London, United Kingdom, 2019.

[20] Moser, J., "Computing your skill", http://www.moserware.com/2010/03/computing-your-skill.html, 2010

[21] Amir, R., Evstigneev, I., "On Zermelo's theorem", arXiv:1610.07160, 2016.

[22] Hodgson, J., "First Order Logic", Saint Joseph's University, Philadelphia, 1995.