Deep Reinforcement Learning for Solving Combinatorial Games with Exponential Action Spaces

Wei Hu wei.hu@uottawa.ca

Ben Burk ben.burk@uottawa.ca

Contents

Abstract

An ability to perform in environments with large action spaces is essential to bringing reinforcement learning to a larger class of problems. In this paper, we present a new policy architecture that operates efficiently with a large number of actions. We apply this approach to a family of games described by Erdös and Selfridge to reduce the action space from exponential to linear. The architecture achieved strong performance and demonstrated generalization over a large range of instances.

We also explore a generalized, *score-keeping* variant of the game, present theoretical results for optimal play against sub-optimal agents, and apply the approaches we developed in our theoretical results to open number theory problems.

1 Introduction

In a paper by Erdös and Selfridge [1], they described a family of games in which two players take turns selecting objects from a combinatorial structure, with the feature that optimal strategies can be defined by potential functions derived from conditional expectations over random future play. These games have a simple characterization of optimal play and the difficulty can be tuned by a simple set of parameters. We focus in particular on one of the best-known games in this genre, Spencer's Attacker-Defender Game (also known as the "Tenure Game") [2].

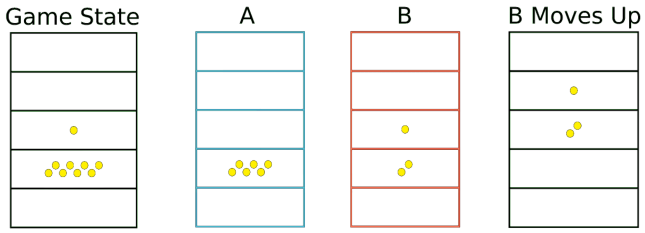


Figure 1: One turn in an ESS Tenure Game. The attacker proposes a partition A, B of the current game state, and the defender chooses one set to destroy (in this case A). Pieces in the remaining set (B) then move up a level to form the next game state.

The game consists of an attacker A, a defender D, and a game state represented by an array S of size k, where $S[i] \geq 0$ is the number of pieces on the board that is i levels away from the top of the board.

At the beginning of each turn, the attacker partitions S into two arrays S_1 and S_2 such that for all $0 \leq i < k$, we have $S[i] = S_1[i] + S_2[i]$, with $S_1[i] \geq 0$ and $S_2[i] \geq 0$.

Definition 1.1. Let v be a value function, which maps a level i to a real number representing the value of a piece at level i. A value function can be represented by an array $(w_0, w_1, w_2, ..., w_k)$ where:

$$v(i) = w_i$$

Definition 1.2. Let S be a board state with k levels. The value function applied to S is defined as:

$$v(S) = \sum_{i=0}^{k} S[i]v(i)$$

A defender would apply its value function to each of S_1, S_2. It would then select the set deemed more valuable by the value function and destroy it. The surviving array is assigned to the main board S and it is moved one level up towards position 0. Any pieces in position 0 after this shift are said to have gained *tenure* and are removed from the board. An example of a turn is illustrated in 1.

The game ends when S has no pieces remaining and will take at most k turns. The $score$ is defined to be the total number of pieces that have gained tenure when the game ends.

Note that the board is zero-indexed. Thus, a piece on level 0 will get tenure and yield a point for the attacker (unless it is destroyed by the defender this turn).

Theorem 1.1. *The optimal value function v_* for the defender, which minimizes the score, is:*

$$v_*(i) = \frac{1}{2^{i+1}}$$

Thus a piece at position 0 is worth $\frac{1}{2}$, and a piece at position 1 is worth $\frac{1}{4}$. The proof is found in [3]. The proof relies on the *splitting lemma*, which we will prove in the next section.

The original ESS game declares the attacker the winner if $score > 0$ at the end of the game. We define a *score-keeping* version of the game where, with S being the initial state: the attacker wins if $score > v_*(S)$, the defender wins if $score < v_*(S)$, and we declare a draw otherwise (when $score = v_*(S)$)

1.1 Challenges

In the original paper [3], the focus was on training defenders. Defenders have an action space of size two; attackers, however, have an action space that is exponential on both the number of pieces and the number of levels.

Although [3] presents a way to train attackers, it forces the attacker's action space to be made linear. This reduction increases the proportion of optimal moves in the search space (making the game easier for the attacker). Furthermore, we only have the guarantee that the reduced space contains the optimal move for the optimal defender – and might not contain the optimal move for playing a suboptimal defender).

We only get feedback whenever a piece gets tenure. Rewards are thus sparse, and since most of the attacker's actions are bad and lead to no reward, it is difficult to get good training feedback.

1.2 Biased Defenders

We observe the optimal value function satisfies $v(i) = 2v(i+1)$. By deviating from this equality, we can create sub-optimal defenders. These defenders either overvalue pieces that are close to the top of the board, or those that are far away. This bias is dependent on the direction of the inequality.

Definition 1.3. A farsighted defender is one whose value function satisfies:

$$v(i) < 2v(i+1)$$

for all $0 \leq i < k$, where k is the length of the board.

Definition 1.4. A nearsighted (myopic) defender is one whose value function satisfies:

$$v(i) > 2v(i+1)$$

for all $0 \leq i < k$, where k is the length of the board.

Intuitively, we can think of a farsighted defender as one which overvalues pieces that are close to the bottom of the board, while a nearsighted one would overvalue pieces that are near to the top of the board.

2 Proof of Theorems

Lemma 2.1 and 2.2 can be used to give an alternate proof of the splitting lemma (of which [3] makes use for proving Theorem 1.1). Furthermore, these lemmas will be the foundation in proving the optimal strategy for playing biased defenders.

2.1 General Theorems

Lemma 2.1. *Given a finite multiset S consisting strictly of powers of $\frac{1}{n}$, for some $n \in Z^+$. If $\sum_{e \in S} e \geq 1$, then $\exists S' \subset S$ such that $\sum_{e \in S'} e = 1$.*

Proof. We define the concept of a *promotion*. If there exists $A \subset S$ satisfying $|A| > 1$ and $\sum_{e \in A} e = (\frac{1}{n})^k$ for some $k \in Z^*$. Then we would consider $T = (S - A) \cup \{\frac{1}{n}^k\}$ to be a promotion of S. We would call S the *parent* of T. We note $\sum_{i \in T} i = \sum_{j \in S} j$.

Furthermore, we claim $\forall B \subset T, \exists C \subset S$ such that $\sum_{i \in B} i = \sum_{j \in C} j$. Indeed, every element in T that is not $(\frac{1}{n})^k$ has greater or equal multiplicity in S than in T. Thus, unless B contains all copies of $(\frac{1}{n})^k$ in T, we are able to choose $C \subset S$ made up of the same elements. Suppose B contains all copies of $(\frac{1}{n})^k$ in T, then we would use $A \subset S$ to account for one copy of $(\frac{1}{n})^k$. For the remaining elements in B, there will be a corresponding copy in S. This concludes the proof of the claim.

Every promotion has fewer elements than its parent. Let $S >_p S_1 >_p S_2 >_p ... >_p S_q$ denote a chain of promotions, where there exists no more promotions in S_q. Since S is finite, the length of the chain is finite. We observe that $\forall k \geq 1$, the multiplicity of $\frac{1}{n}^k$ in S_q is less than or equal to $n-1$ (If it were n or greater, then we have a promotion by letting A equal n copies of $\frac{1}{n}^k$). Thus, the sum of the elements that are less than 1 is strictly upper bounded by $\sum_{i=1}^{\infty} \frac{n-1}{n^i} = (n-1)\sum_{i=1}^{\infty} \frac{1}{n^i} = 1$. Yet we have $s = \sum_{e \in S} e = \sum_{e \in S_q} e \geq 1$, thus there must be at least one element in S_q in the form $\frac{1}{n}^k$ where $k = 0$, that is to say $1 \in S_q$.

Thus, there is a subset of S_q that adds to 1; per our previous observation, we conclude there must also be a subset of S which adds to 1 □

Corollary. *Given a finite multiset S consisting strictly of elements in the form $(\frac{1}{n})^r$, for some $n \in Z^+, r \geq k$. We have $\forall s \leq k$, if $\sum_{e \in S} e \geq \frac{1}{n}^s$, then $\exists S' \subset S$ such that $\sum_{e \in S'} e = \frac{1}{n}^s$.*

The *splitting lemma* is the special case where $n = 2$ and $k = 1$.

Lemma 2.2. *Let S be a board state containing only pieces on levels greater than i. Suppose $v_*(S) \geq v_*(i)$, then $\exists S^{'} \subset S$ such that $v_*(S') = v_*(i)$.*

Proof. Note, we have $v_*(S) = \sum_{j=i+1}^{k} S[j]\frac{1}{2^{j+1}} \geq v_*(i) = \frac{1}{2^{i+1}}$. From there, dividing $\frac{1}{2^{i+1}}$ from both sides we get: $\sum_{j=i+1}^{k} S[j]\frac{1}{2^{j-i}} \geq 1$.

Create a multiset M by including $S[j]$ copies of $\frac{1}{2^{j-i}}$ for each $i+1 \leq j < k$. We can apply lemma 2.1 to M and conclude there is a subset M' of M which adds to 1.

Based on how M is defined, we know every element e in M corresponds to a piece on the $log_2(\frac{e}{2^{i+1}}) - 1$ level of S. We can form a bijection between M and S

Thus, we know there exists a subset S' of S corresponding to $M^{'}$ satisfying:

$v_*(S') = \sum_{e \in M'} \frac{e}{2^{i+1}} = \frac{1}{2^{i+1}} \sum_{e \in M'} e = \frac{1}{2^{i+1}} = v_*(i)$. □

Theorem 2.3. *When playing against an optimal defender, one optimal approach for the attacker is to make the value according to v_* of the two partitioned sets as close as possible.*

Proof. We sketch a proof:

We can show by modifying the proof from [3] that when playing against an optimal defender, the best score that can be achieved from a starting state S is $\lfloor v_*(S) \rfloor$.

We know the max value of any piece according to v_* is $\frac{1}{2}$. By repeatedly applying the corollary of Lemma 2.1, we can partition S into $2\lfloor v_*(S) \rfloor$ subsets each with value of at least $\frac{1}{2}$ according to v^*.

By evenly distributing those subsets, we can partition S into 2 subsets each with with value of at least $\frac{1}{2}\lfloor v_*(S) \rfloor$ according to v^*.

We can guarantee the value of the surviving subset is greater than or equal to $\frac{1}{2}\lfloor v_*(S) \rfloor$. After the left shift, the value of the surviving set according to v^* is doubled and the tenured pieces are removed from the board.

We notice the following invariant $current\_score + \lfloor v_*(current\_state) \rfloor = \lfloor v_*(start\_state) \rfloor$.

In at most k steps we will have, $v_*(current\_state) = 0$. We thus have $current\_score = \lfloor v_*(start\_state) \rfloor$, which means we achieve the best possible score.

We complete the proof by noting that when we make the value according to v_* of the two partitioned sets as close as possible, we will achieve the necessary condition which is to produce two sets each with a value of at least $\frac{1}{2}\lfloor v_*(S) \rfloor$ according to v^*. □

2.2 Properties of Suboptimal Defenders

We begin by looking into farsighted defenders, and prove an algorithm for creating a partition that will maximize the value according to v_* of the surviving set.

Lemma 2.4. *Let A, B be two states, and let v be the value function of a farsighted defender. If $max\{x : B[x] > 0\} < min\{x : A[x] > 0\}$ and $v(B) > v(A)$ then $v_*(B) > v_*(A)$*

Proof. By contradiction, $v_*(A) \geq v_*(B)$. We can apply Lemma 2.2 and map each piece in B to a subset of A with equal value. Note, we can make those subsets of A disjoint. Every time we make a mapping we can "remove from consideration" the pieces used in this mapping to produce an A' and B' satisfying $v_*(A') \geq v_*(B')$ to which we can reapply Lemma 2.2. This process is repeated until every piece in B is mapped.

While v_* deems the pieces on both sides of the mapping to be equal, v will not.

WLOG, suppose a piece at level i in B is mapped to $M \subset A$. Consider any piece in M and suppose it is on level j, we note: $\frac{v(j)}{v(i)} > \frac{v_*(j)}{v_*(i)}$, since $v(i) < 2v(i+1)$ for any farsighted defender.

Thus, we have $\frac{v(M)}{v(i)} > \frac{v*(M)}{v*(i)} = 1$. We've mapped every piece in B to a disjoint subset of A that is considered more valuable by v. It must be that $v(A) > v(B)$. □

Theorem 2.5. *The following algorithm maximizes the real value of the surviving pieces at the end of each turn against any farsighted defender.*

Algorithm 1: `Minimizing` v_* `Value of Destroyed Set Against Farsighted Defenders`

Input: a board position S with k levels, and the value function v of a nearsighted defender

Result: a partition of the board (S_1, S_2) which guarantees the subset the farsighted defender will destroy has the lowest possible value according to v_*

// Beginning from the level farthest from tenure, and going from left to right on that level,

// we consecutively label each piece on the board with a natural number.

$required\_potential \leftarrow \frac{v(S)}{2}$;

// The set defined below is the set to be destroyed by the defender.

$bad\_set \leftarrow$ an array of zeroes of length k;

// In the array below, the tuple at the i-th index stores two values: the cd value is equal to

// $\sum_{n=0}^{i} v(p_n)$ where p_n is the n-th piece according to our labelling system above, and the $level$

// value is the number of levels that piece is away from tenure.

$cumulative \leftarrow$ an array of tuples $(cd, level)$;

$index \leftarrow cumulative.length$;

// Since we are iterating through *cumulative* backwards, we are going through the board starting

// from the pieces closest to tenure! We want the defender to destroy the pieces it most overvalues.

while $required\_potential > 0$ *and* $index \geq 0$ **do**

  if $cumulative(index - 1).cd < required\_potential$ **then**

    // We only add a piece to the bad_set when we have to

    // (so that the bad_set has enough value to be destroyed).

    $bad\_set[cumulative[index].level]{+}{+}$;

    $required\_potential -= v(cumulative[index].level)$;

  end

  $index--$;

end

// We output the two partitions. The defender will destroy the bad_set.

return $(bad\_set, (S - bad\_set))$;

Proof. When the algorithm adds a piece of level l to *bad_set*, we can be sure bad_set must include at least another piece whose level $\leq l$; otherwise, it will not be sufficiently valuable according to v to be destroyed.

The reason we can be sure is because we check that $cumulative(index - 1).cd < required\_potential$. If this check passes, it means even if we were to include every piece of a greater index than the current piece in the *bad_set*, the *bad_set* would still not be valuable enough according to v to be worth destroying.

From there, we make an argument based on *Lemma 2.2* and *Lemma 2.4* that there is no reason to add to bad_set a piece closer to tenure than the current piece. Suppose by contradiction we must add a piece p closer to tenure on level $k < l$ for the partition to be optimal. Then, it would mean the inclusion of p in bad_set allowed us to keep some set S with $v_*(S) > v_*(k)$, consisting only of pieces on levels greater than k, from being destroyed.

By *Lemma 2.2*, there exists a subset S' of S that satisfies $v_*(S') = v_*(k)$. Furthermore, by *Lemma 2.4*, we know $v(S') > v(k)$.

We have a contradiction because we can create a more optimal partition by replacing p with S'. □

We will now look into nearsighted defenders. Similarly, we will give an algorithm for creating a surviving partition of maximal value according to v_*.

Lemma 2.6. *Let A, B be two states, and let v be the value function of a nearsighted defender. If $min\{x : B[x] > 0\} > max\{x : A[x] > 0\}$ and $v(B) > v(A)$ then $v_*(B) > v_*(A)$*

Proof. The argument follows the same principle as *Lemma 2.4*. □

Lemma 2.7. *Let S be a game state. Let the defender be nearsighted. Set $U = min\{x : S[x] > 0\}$. Let $\gamma = \frac{v(S)}{2}$. Suppose $v(U) \leq \gamma$, then there exists S' that is a surviving set (a set not destroyed by the defender) in an optimal partition, with $S'[U] < S[U]$.*

Proof. Let S, T form an optimal partition against a near-sighted defender, where S survives and T is destroyed. In other words, $v(T) \geq \gamma \geq v(S)$.

Since $v(T) \geq \gamma > v(U)$, if T did not contain a piece at level U, then by *Lemma 2.6*, we know $v_*(T) > v_*(U)$. (Note: if T contains a piece at level U, the proof is complete).

By *Lemma 2.2*, we know there exist a subset T' of T such that $v_*(T') = v_*(U)$.

If T' consists of a single piece at the level U, then the proof is complete.

If T' contains more than one piece and there is not a piece at level U also in T, then we know for sure there is at least one piece at level U in S.

By Theorem 2.6, we can replace T' with U, and our surviving set would still be at least equally optimal.

□

Theorem 2.8. *The following algorithm maximizes the real value of the surviving pieces at the end of each turn against any nearsighted defender.*

Algorithm 2: `Minimizing` v_* `Value of Destroyed Set Against Nearsighted Defenders`

Input: a board position S with k levels, and the value function of a nearsighted defender v

Result: a partition of the board (S_1, S_2) which guarantees the subset that the defender will destroy has the lowest possible value according to v_*

$required\_potential \leftarrow \frac{v(S)}{2}$;

$bad\_set \leftarrow S$;

$minimal\_value \leftarrow v_*(S)$;

// When we add a piece to the $current\_set$, it either bad_set is already optimal

// or that it is guaranteed an optimal bad_set can be built by adding to $current\_set$.

$current\_set \leftarrow$ an array of zeroes of length k;

$current\_value \leftarrow 0$;

for $i \leftarrow 0$ *to* k **do**

  for $piece \leftarrow 0$ *to* $S[i]$ **do**

    if $v(i) < required\_potential$ **then**

      $current\_set[i]{+}{+}$;

      $required\_potential{-}{=}v(i)$;

      $current\_value{+}{=}v_*(i)$;

    end

    else

      // One of these sets built from $current\_set$ will be optimal.

      if $current\_value + v_*(i) < minimal\_value$ **then**

        $minimal\_value \leftarrow current\_value + v_*(i)$;

        $bad\_set \leftarrow current\_set$;

        $current\_set[i]{+}{+}$;

      end

      $break$;

    end

  end

end

// We output the two partitions. The defender will destroy the bad_set.

return $(bad\_set, (S - bad\_set))$;

Proof. We are iterating through the board starting with the pieces that v_* deems most valuable.

Whenever the condition $v(i) < required\_potential$ is met, we are guaranteed by *Lemma 2.7* that an

optimal set can be formed by adding in a piece at the i-th level.

From there, we recursively include pieces in accordance with *Lemma 2.7*, while keeping track of the bad_set with a minimal value according to v_* out of all the valid bad_set that can be formed.

To do so, every time we come across a piece whose value according to v exceeds the remaining $required\_potential$, we would calculate the value according to v_* of the set formed by adding that piece to the $current\_set$. If that value is lower than that of our current $minimal\_value$, we update our $minimal\_value$ and make the $current\_set$ our tentative output.

Since the way we are greedily narrowing down our search space is in accordance with *Lemma 2.7*, we know at least one of the optimal partitions will not be eliminated. □

3 Deep Reinforcement Learning

In reinforcement learning, a computer agent is tasked with interacting with a virtual environment (in our case, an interface of the Tenure Game), and attempting to maximize some reward function. At each turn, the agent is expected to choose an action from a set of valid actions (known as the agent's *action space*). The goal is to have the agent learn a *policy*, or a function that maps a state to an action in the action space.

In the Tenure Game, the Attacker's action space is much more complicated than that of the Defender. We attempted approaches, such as Table-Based Q-Learning, Deep-Q Learning, Actor-Critic, which all had disappointing results.

We designed our own approach which linearizes the action space at the cost of an enlarged state space. Our implementation enables agents to learn through *self-play*. Agents start off knowing nothing about the game, except for the game definition and they are not given a skilled opponent to play against.

3.1 Micro-actions

In the Tenure Game, the attacker splits the board into two partitions and the defender chooses which one to remove. A challenge with this is that the action space of partitions to choose from grows exponentially with the number of levels K and with the number of pieces on the board. A naive implementation of the attacker would be to have the policy output how many pieces should be allocated to A for each of the K levels as in the construction from [2].

Previous attempts to train attackers involved a parameterization of the action space where the optimal policy is representable and is linear instead of exponential in K [3]. The attacker outputs a level l. The environment assigns all pieces before level l to A, all pieces after level l to B, and splits level l among A and B to keep the potentials of A and B as close as possible.

In our approach, we break the process of selecting a partition into choosing one piece after another for each iteration. The action space is augmented with an additional "done" action. The attacker repeatedly selects a level l and moves a piece from that level to the new partition. When it is done creating the partitions, it selects the "done" action.

In [3], the agent struggled when the number of pieces on the board is large, whereas our micro-action approach generalizes well. The trained agents perform well even when the number of pieces is meaningfully increased (see Section 4.2).

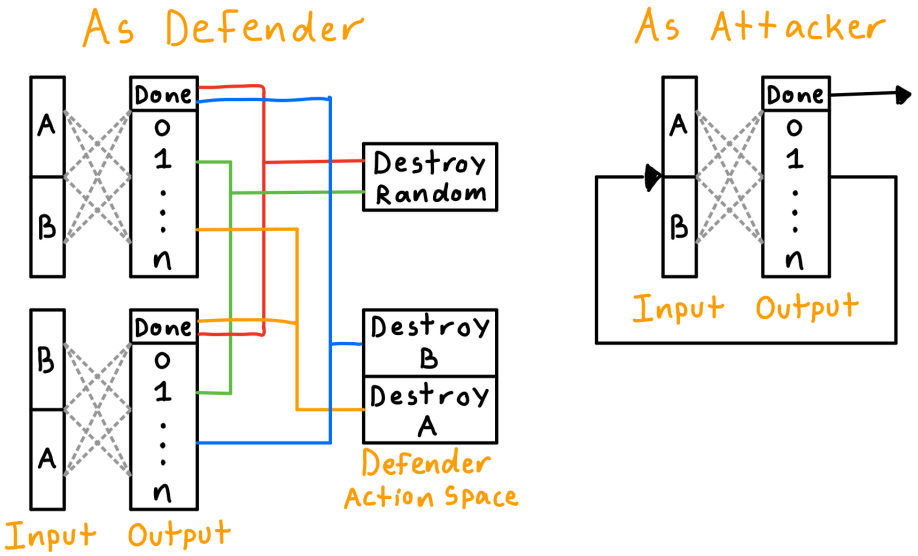
3.2 Unifying Attacker and Defender model

This parameterization of the action space allows for many benefits. Firstly, it allows the model for the attacker and defender to be the same. The defender can be simulated using the same interface. If the

defender moves a piece from one partition to the other, it sees the partition being taken from as more valuable, so that partition should be destroyed. If it does not move a piece, it sees the partitions as having equal value.

This parameterization also allows the agents to be trained via self-play. We implemented the AlphaZero algorithm to train an agent. We trained a neural network to output a micro-action policy while also estimating the expected payout of the current state.

We implemented the AlphaZero algorithm to train the Tenure Game agent via self-play. AlphaZero is a reinforcement learning algorithm developed by DeepMind to master the games of Chess, Shogi, and Go. The program, called AlphaZero, descends from AlphaGo, an A.I. that became known for defeating Lee Sedol, the world's best Go player, in March of 2016. It also achieved mastery of Chess and Shogi through pure self-play without any human knowledge beyond the rules of the game. It combines a neural network and Monte Carlo Tree search to achieve stable learning.



3.3 Monte Carlo Tree Search

Using depth-first-search, we can explore the tree of all possible game states and return the one with the highest score. For more complicated games such as Chess or Go, the game tree is too massive to explore efficiently. Monte Carlo tree search (MCTS) allows us to estimate the value of a state by making random moves to explore the game tree. If you play 1000 games by randomly making moves from a starting position and you win 75% of the time, then it is likely that the starting position is better for you than your opponent.

As we explore the tree, the following values are computed: $Q(s, a)$, the expected reward for taking action a from state s and $N(s, a)$, the number of times action a was taken from state s.

When we reach an unvisited node (can't calculate its value) we randomly simulate a game to its completion. One way to do so would be simply to uniformly sample valid moves until the game is finished. These simulated games are called roll-outs. In general, this process can be summarized in four steps:

1. Selection: Starting from the current state, select child nodes, maximizing the UCB until we reach unvisited states with no payout estimate.
2. Expansion: For each unvisited node, expand N of its child nodes.
3. Roll-out: For each of those child nodes, run M Monte Carlo simulations.
4. Update: At the end of each roll-out (i.e. we reached a terminal win/loss/draw state), we update the payout score of all intermediary states.

One of the main issues with MCTS is that the algorithm may favor a losing move with only one or a few forced refutations, but due to the vast majority of other moves, it provides a better random playout score than other better moves. Upper confidence trees (UCT) attempt to resolve this by using a heuristic approximation that balances exploration and exploitation. In each node of the game tree the move for which the expression below has the highest value:

$$UCT_i = w_i + C\sqrt{\frac{\ln N}{n_i}}$$

where w_i is the win ratio of the child node, N is the number of times the parent node was visited, n_i is the number of times the child node was visited, and C is a constant to adjust how much exploration should occur.

The first term in the heuristic encourages exploitation by playing known moves with high win rates. The second term encourages exploration by trying out moves that have a low visit count and updating the statistics to more accurately model the value of those unexplored states.

When the UCT algorithm first starts, it more heavily weights exploring the game tree. As it collects more information, exploiting it's known strategy becomes more heavily favored. If given infinite time and memory, UCT theoretically converges to Minimax.

One of the best features of upper-confidence trees is that the algorithm generates weighted exploration trees. In games with a large branching factor such as Chess, there is an intractable number of states. Most of them are unimportant since they can only be reached if one or both players play poorly. Using UCT, one can avoid exploring these useless states and focus most of the computational energy on simulating games in the interesting portion of the state space.

3.4 Deep Learning Guided MCTS

In our project inspired by Alpha Go, we train a neural network to guide our expansion. Let f_θ be a neural network parameterized by θ, which takes as input a game state s and outputs $\pi_\theta(s)$, and $v_\theta(s)$. $v_\theta(s)$ represents a policy. A policy is a discrete probability distribution, which assigns a probability to each available move.

The move with the highest probability represents the move that is recommended by the policy. $v_\theta(s)$ is a real number in the range $[-1, 1]$ which estimates the value of the s. If $v_\theta(s)$ close to 1, the policy believes the current player is winning, while a $v_\theta(s)$ close to - 1 suggests the opponent's chances are favorable.

Instead of a roll-out, we take $v_\theta(s)$ to be an estimate of the payout of an unvisited non-terminal node. For terminal nodes, we use + 1 for a win, - 1 for a loss, 0 for a draw. After a sufficient number of simulations (in Alpha Go's case, 1600 Monte Carlo simulations are made per move), we can produce an estimation of the policy by taking $n_i(a)$ as the probability for each action a. We call this new policy π_t.

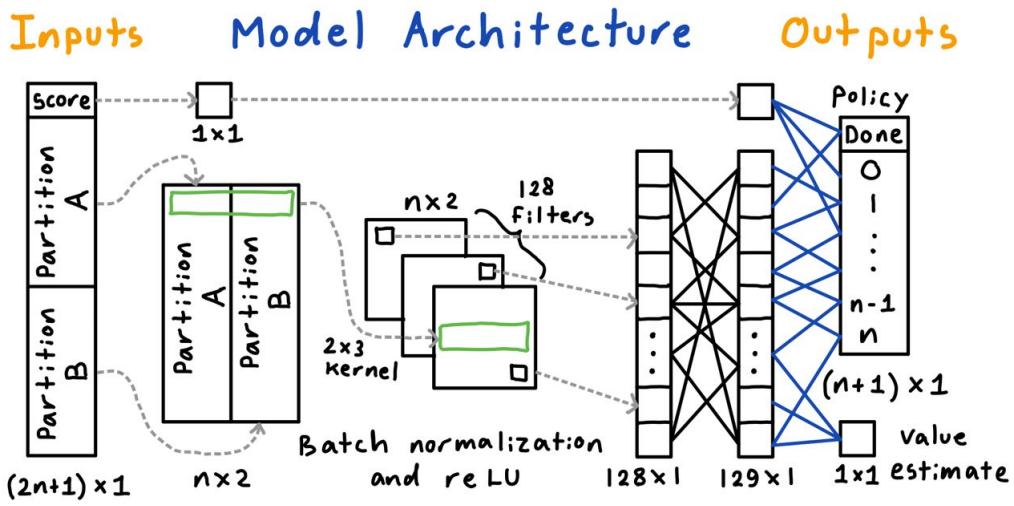


Figure 2: The neural network architecture

To train the neural network, we use gradient descent. First, we uniformly sample n positions from the last N games.

For each of those positions s_i, we take the squared difference between the predicted value $r_\theta(s_i)$ and the actual outcome of the game, which we will call r. We take the sum of these differences. We represent these difference in matrix notation as:

$$[r - r_\theta(s)]^2$$

The MCTS policy π_t is expected to be stronger than $\pi_\theta(s)$. We thus take the cross-entropy loss between the two. Here, π_t^T is the transpose of π_t:

$$\pi_t^T[log\pi_\theta(s)]$$

Lastly, to prevent overfitting, we add $L2$-regularization term on θ. Combining these terms, we have the loss function used to train f_θ:

$$l = [r - r_\theta(s)]^2 + \pi_t^T[log\pi_\theta(s)] + c||\theta||_2$$

At the end of each training step, we use the updated network to play the old network. We only keep the new network if it manages to defeat the old network more than 55% of the time.

3.5 Self-play

We can now select the best move according to π_t. By playing the best move according to π_t and then repeating the aforementioned process, we can play entire games. These games are considered to have been self-played. The goal is to have the program discover the game through many rounds of self-play, without needing any human knowledge (e.g. opening books). For each of these self-played games, we will be storing the intermediate states and policy estimations (the π_t's), as well as the final outcome. These values will be used to improve f_θ (discussed in the next section).

Since f_θ can be trained to continually improve, we do not need to use the search tree as our policy. Once f_θ is sufficiently trained, it is expected to produce a sensible policy for any state that it is fed. Thus, we will rely solely on f_θ and discard the search tree.

It is also possible to rebuild the search tree for each new game. By using a continually improving f_θ instead of performing roll-outs, we could get a quick value estimate for each move even when starting with an empty tree. In this case, we should think of the search tree as being a policy improvement coach instead of being the policy itself.

4 Results

In [3], models for defenders were trained against attackers that played according to Theorem 2.3 but occasionally played randomly. The goal for them was to allow for exploration. We note the sensibility of this approach by observing that our self-play agent performs worst against these slightly sub-optimal agents.

In evaluating our self-play model, we test attacking and defending capabilities separately. We test against agents that: play optimally in accordance with Theorem 2.3 (by splitting the two partitions evenly), play completely randomly, sometimes play randomly and sometimes play according to Theorem 2.3. In other words, it is harder to play optimally against a slightly sub-optimal agent than it is to play against an optimal agent.

4.1 Model Performance

The difficulty of an Attacker-Defender game S for a Defender is in large part determined by how small $\lceil v_*(S) \rceil - v(S)$ is (as long as it is not 0). The smaller this value, the more optimally it has to play to draw an optimal attacker. Similarly, for an attacker, the difficulty is determined by the size of $v(S) - \lceil v_*(S) \rceil$

Furthermore, by using the score-keeping version of the game, we are adding an additional dimension of parameterization. We can vary the difficulty of the game as shown in Figure 3.

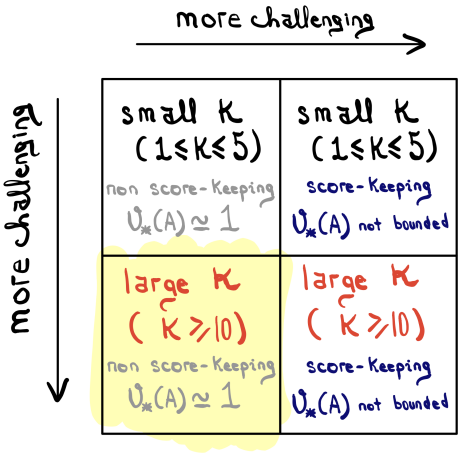


Figure 3: Varying the difficulty of the game

The top-left quadrant is not worth exploring, as even simple strategies such as table-based Q-learning converges to optimal play easily. Meanwhile, [3] focused on the bottom left quadrant (highlighted in yellow).

Our agents trained in that quadrant, as well as the top two quadrants. We look into how agents trained in the environment from the left column generalizes into the environments in the right column. In particular, for the fourth quadrant (bottom-right), although our model is designed to be able to interface

with it, it is infeasible to train for this project. The large quantity of pieces that are involved in the states makes them too computationally expensive for our local machines. Given more time, we would like to explore training on the cloud and setting up parallelization (e.g. playing multiple games simultaneously and combining gradients).

Meanwhile, we test generalization by observing how well agents trained on the third quadrant generalize to the fourth quadrant. It is worth noting that as $v_*(S)$ becomes large, the number of actions increase meaningfully.

For the bottom left quadrant, our results are shown in Figure 4.

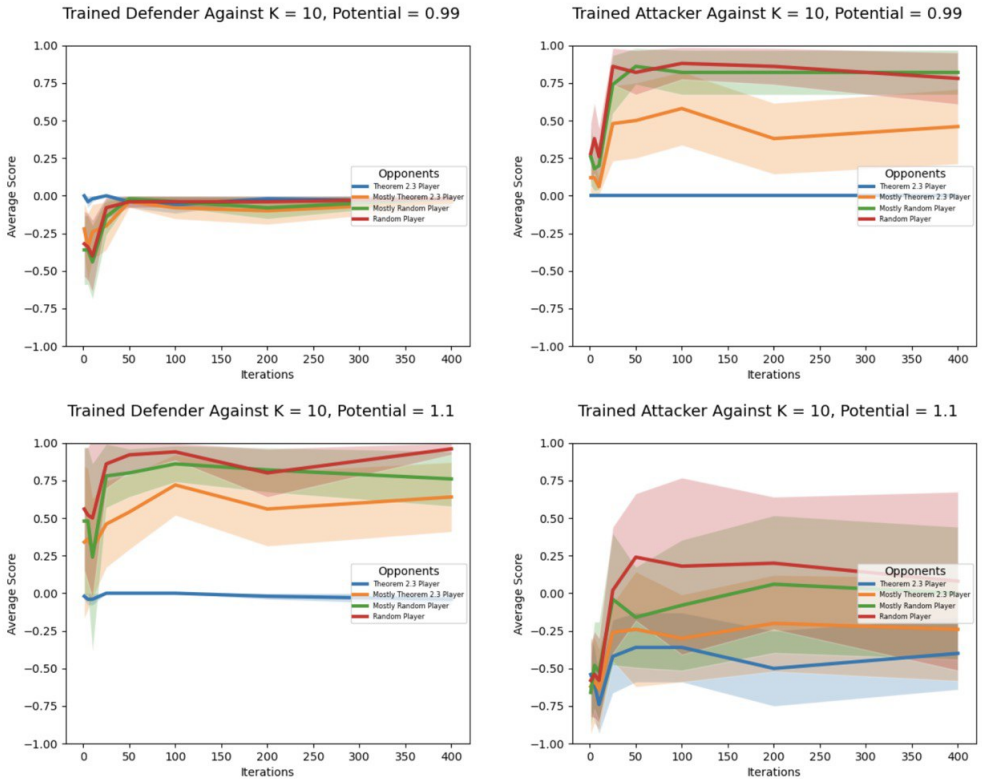


Figure 4: The results training on large K, with a value close to 1

Note, in the top row, it is impossible for the Defender to win the game (because the expected number of promotions is 0), thus the best score to aim for is 0. Meanwhile, a minor mistake would allow the attacker to score a point (so it is relatively easy for the attacker to win). Our trained agent was mostly able to defend against all the opponents, and as an attacker was able to score.

Conversely, in the bottom row, it is easy for the Defender to win because the Attacker is expected to exploit a small advantage and score a point.

4.2 Model Generalization

When training our agents, we exposed expose them only to a very specific set of starting states, initialized according to the algorithm below.

Algorithm 3: `Generating Game States`

Input: a board size K and a potential upperbound p

Result: a game state S of size K with $v_*(S) \leq p < v_*(S) + \frac{1}{2^K}$

$result \leftarrow empty\_board$;

while $p > 0$ **do**

  $S \leftarrow$ {all l such that $v_*(l) < p$ }

  $level \leftarrow$ random sample from S

  add a piece of level $level$ to $result$

  $p-=v_*(level)$

end

return $result$;

To generate training states, we use this algorithm with a p taken from $\mathcal{N}(0.95, 0.75)$. Thus, we can explore our model's generalization by testing them on games of much higher potential.

The results in Figure 5 and those in the bottom row of Figure 4 are significant for three reasons: (1) at a potential of 1.1 or 2.1, it is very hard for an attacker to capitalize on the advantage and get sufficiently many tenures as to tie the game (2) as shown on the left panes, even a small deviation from optimality results in opponent agents losing to our defender, (3) our attacker can get a sufficient number of tenures in most cases, and in the case of random defenders, even get two tenures with a potential of 1.1 (resulting in a positive score).

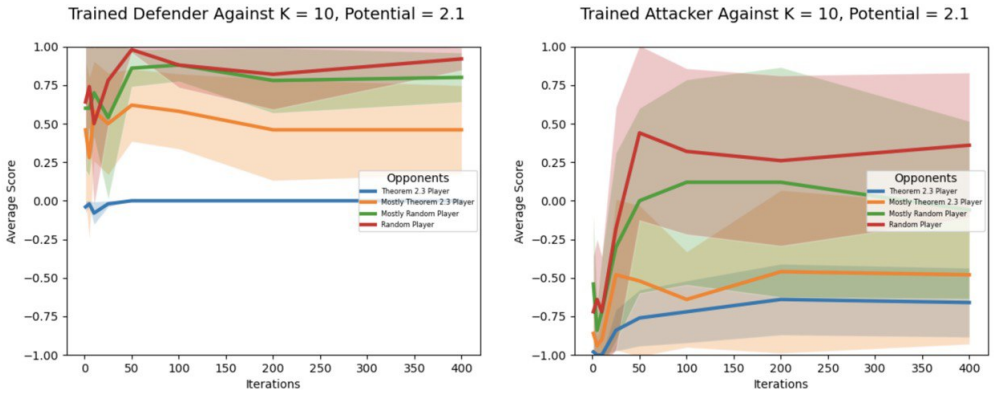


Figure 5: Generalizing to games with double the potential than those that the model was trained on.

Figure 6 shows that the same network architecture works on a game with many more levels. However,

it takes much longer to train and benchmark.

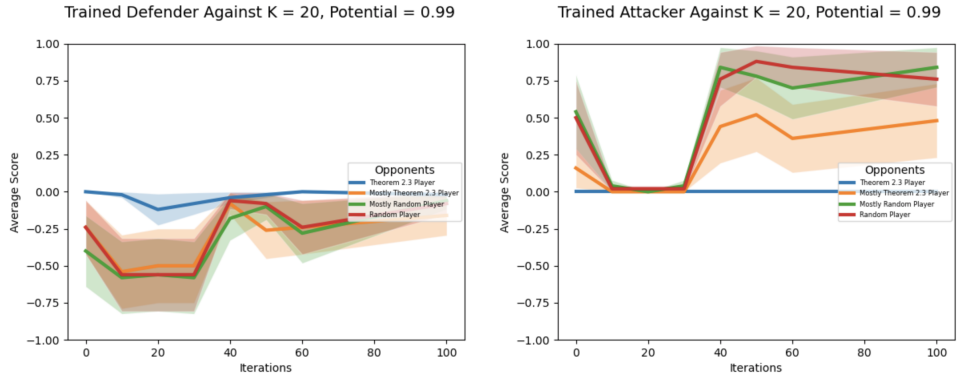


Figure 6: Obtaining similar results on games with a larger number of levels.

Figure 7 examines generalizing to games with much larger potentials (entering the bottom right quadrant of Figure 3). These Defenders were only ever exposed to games sampled with potential from $\mathcal{N}(0.95, 0.75)$. As shown there, Defender agents can effectively adapt to states with 10 times the expected potential, which is evidence of good generalization.

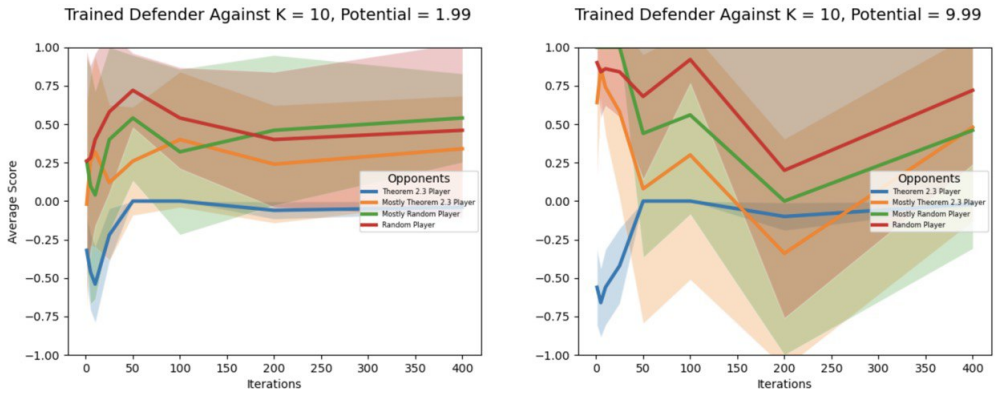


Figure 7: Defender generalizing to games with significantly higher potential

While it is impossible to win against an opponent attacking according to Theorem 2.3, it is also very hard to lose to such an opponent. By always splitting the two sets to be of even potential, it is usually optimal for the Defender to choose to destroy either one. There are unbalanced partitions that are optimal, even against an optimal Defender. Proposing these unbalanced partitions may be the best way to play against potentially sub-optimal opponents (because the Defender needs to make the right decision).

In Figure 8, the aggregated score (which is the expected number of wins) is calculated. Here, every agent is matched with every other agent and made to play simulated games. As shown here, despite never losing any games, the Theorem 2.3 "optimal" player achieves a low score, as it never manages to

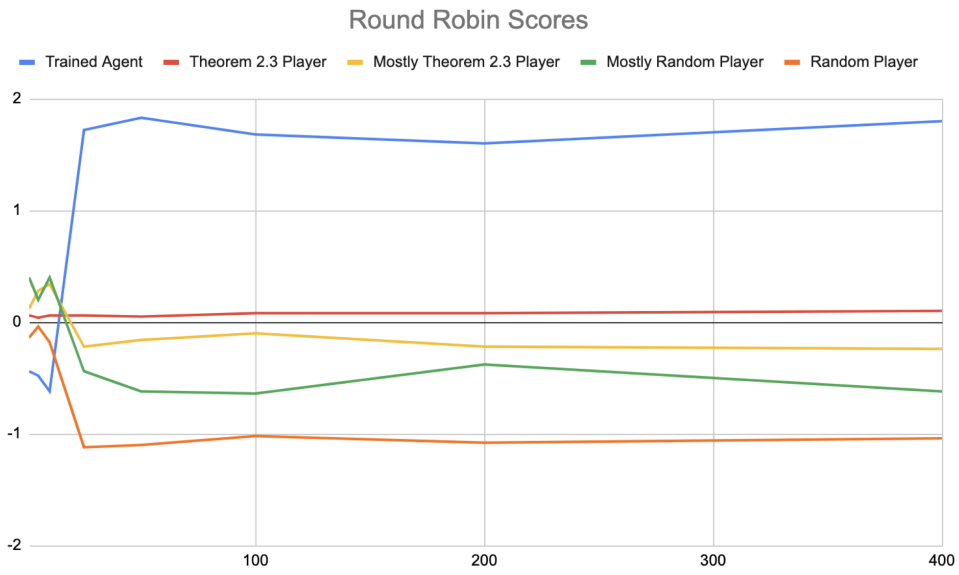


Figure 8: Generalizing model to games with higher potential

win. Throughout the entire process, the players with some degree of randomness were always the more difficult ones to play against.

Our hypothesis is that by pitting the old network against the new one and keeping the one that has the most wins, we are selecting networks that are able to exploit suboptimality, and produce optimal unbalanced partitions.

5 Promotions and Pseudopromotions

Here we look at some properties of the concept of a *promotion* as discussed in Section 2. We also generalize to the idea of *pseudopromotion*.

Definition 5.1. An Egyptian Fraction is a real number that can be represented by $\frac{1}{k}$ where $k \in Z^+$

In particular, we care about the properties of promotions in Egyptian Fraction multisets (EFMs). Let S be an EFM, we generalize our definition in Section 2:

Definition 5.2. If there exists $A \subset S$ satisfying $|A| > 1$ and $\sum_{e \in A} e = \frac{1}{n}$ for some $n \in Z^*$. Then we would consider $T = (S - A) \cup (\frac{1}{n})^k$ to be a **promotion** of S. We would call S the **parent** of T.

Let S be a multiset of rational numbers. We generalize by defining:

Definition 5.3. Suppose $A \subset S$ satisfying $|A| > 1$. Let g be the smallest element in A. If and $\sum_{e \in A} e = (\frac{m}{n})$ for some $m, n \in Z^*$ with $(m, n) = 1$. If $\frac{1}{n} > g$, then we would consider $T = (S - A) \cup (\frac{m}{n})$ to be a **pseudopromotion** of S. We would call S the **pseudoparent** of T.

5.1 Observations

Like we have shown for chains of promotions, chains of pseudopromotions must also be finite. Furthermore, the multisets in a chain of pseudopromotions must also all have the same sum.

Definition 5.4. A simple EFM is one for which there does not exist a parent. A pseudosimple EFM is one for which there does not exist a pseudoparent.

Definition 5.5. Let $p, q, n \in$, we define $D_{p,q}(n) = \text{argmin}_{x>0}\ px \equiv -n \mod q$

Lemma 5.1. *Let p, q be primes. Let S be an EFM containing n instances $\frac{1}{p}$, m instances of $\frac{1}{q}$ and at least one instance of $\frac{1}{pq}$. If $\frac{m}{p} + \frac{1}{q} \geq 1$ then S is not pseudosimple.*

Proof. Let $r = D_{p,q}(1)$ and $s = D_{q,p}(1)$. It is sufficient to show $\frac{s}{p} + \frac{r}{q} < 1$.

Since $\exists x, y \in Z$ with $rp = xq + 1$ and $sq = yp + 1$. We have $\frac{s}{p} + \frac{r}{q} = \frac{y}{q} - \frac{1}{pq} + \frac{x}{p} - \frac{1}{pq} = \frac{py+qx-2}{pq}$

We know $\exists k \in Z$ with $0 < k < p$ such that $k \equiv q^{-1} \mod p$. Similarly, $\exists l \in Z$ with $0 < l < q$ such that $l \equiv p^{-1} \mod q$.

Bearing in mind that r, s are minimal, we thus note $0 < py + qx < 2pq$ and $py + qx \equiv 1 \mod pq$. Thus, $py + qx = pq + 1$

Thus, $\frac{s}{p} + \frac{r}{q} = \frac{py+qx-2}{pq} = \frac{pq-1}{pq} < 1$ □

5.2 Simple EFMs

Definition 5.6. Let n be the product of all the denominators in an EFM. The **index** of the EFM is the number of primes in the prime factorization of n

Theorem 5.2. *Let S be an EFM with index 2. If $\sum_{e \in S} e > 2$, then either $1 \notin S$ or S is not simple.*

Proof. This proof is still in progress, but the idea here is that if $\sum_{e \in S} e > 2$ and $1 \notin S$, then we can find a chain of pseudopromotions eventually leading to an integral sum. □

6 Conclusion

In this paper, we presented a new policy architecture that operates efficiently with a large number of actions. We demonstrated a successful application to the Tenure Game described by Erdös and Selfridge to reduce the action space from exponential to linear.

We show that the model trained using this approach achieves strong performance and generalizes over a large range of instances. In particular:

- We extended on the evaluation/comparison environment proposed in [3] by considering a *score-keeping* variant of the game. Moreover, we consider training against suboptimal defenders and offer a theoretical analysis of optimal play against them.
- We proved a unification of the attacker and defender model, allowing for Monte Carlo Tree Search *self-play* (without simplifying the attacker's action space as in [3]) and achieving good generalization both the original and our modified variant of the Tenure Game.
- We supplied empirical evidence for the effectiveness of a micro-action approach for tackling reinforcement learning environments with exponential discrete action spaces.

We believe there are meaningful applications for the approaches we developed here, and for the Tenure Game, as a combinatorial evaluation environment.

6.1 Future Work

There is a great deal of motivation to approximate combinatorial optimization problems [4] and [7]:

"For NP-hard graph problems, solvers use carefully handcrafted heuristics to find approximate solutions. However, [these solvers] are problem-specific; their development for new problems requires significant time. An alternate approach by the Machine Learning community is to develop generic learning algorithms which can be trained to solve any combinatorial problem from problem instances themselves."

There is growing interest in this area [6]. In particular, one approach involves transforming these problems into sequential decision-making processes, and to learn heuristics implicitly using reinforcement learning (RL) techniques. A process known as *Zermelo Gamification* can be used to transform problems into two-player games. Finding an optimal strategy for these Zermelo games is shown to be equivalent to solving the original problem [5]. Traditional RL algorithms like Monte Carlo Tree Search can thus be used to solve these games through self-play, without needing expert knowledge.

References

[1] Paul Erdos and John. Selfridge. “On a combinatorial game.” In: *Journal of Combinatorial Theory.* (1973).

[2] Joel Spencer. “Randomization, derandomization and antirandomization: Three games.” In: *Theoretical Computer Science.* (1994).

[3] Maithra Raghu et al. “Can deep reinforcement learning solve Erdös-selfridge-spencer games?.” In: *International Conference on Machine Learning* (2018).

[4] Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. *On Learning Paradigms for the Travelling Salesman Problem*. 2019. arXiv: 1910.07210 [cs.LG].

[5] Ruiyang Xu and Karl Lieberherr. *Learning Self-Game-Play Agents for Combinatorial Optimization Problems*. 2019. arXiv: 1903.03674 [cs.AI].

[6] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. “Machine learning for combinatorial optimization: a methodological tour d’horizon.” In: *European Journal of Operational Research* (2020).

[7] Iddo Drori et al. *Learning to Solve Combinatorial Optimization Problems on Real-World Graphs in Linear Time*. 2020. arXiv: 2006.03750 [cs.LG].