Exploring High-Dimensional Tic-Tac-Toe

Daniel Mendelevitch, Niket Patel,
Elaine Fan, Camden Weber, Thomas Park

UCLA
Mathematics Department

Code: https://github.com/danielmend/math42-gameSolving
Presentation: https://tinyurl.com/math42presentation

11/30/2022

Contents

1 Abstract

This paper looks into the game theory behind Tic-Tac-Toe, as well as how the game changes in higher dimensions. We showed that we can use MiniMax to create an optimal agent for the 3×3 case. We then examined the use of evaluation functions that allow us to shorten our trees and enable us to explore Tic-Tac-Toe in higher dimensions. We found that for higher N, $N \times N$ boards will tend to draw more, and we provided some analytic reasoning behind this phenomena. We propose the longest-chain board evaluation function, which can generalize well to $N \times N$ sized boards. We also evaluate a special case of this function for $N \times N \times N$ boards which performs well by our evaluation metrics. As a consequence of the curse of dimensionality, we needed to investigate possible speedups and alternatives to the MiniMax algorithm, and found that Alpha-Beta pruning dramatically improves search time for MiniMax, while MCTS under-performs in win rate relative to MiniMax.

2 Problem Statement

Optimal game-playing has historically been challenging for humans. We can gain intuition and build some heuristics around a game which allow us to mentally search their state spaces efficiently, but for complicated games with a sufficiently large number of states it becomes intractable for humans to find optimal play with our minds alone. For example, the game of Tic-Tac-Toe is solved for the 3×3 board, but if we extend this game to $3 \times 3 \times 3$ boards it is not obvious how to play optimally. The number of possible states in 3×3 Tic-Tac-Toe is $9! = 362,000$, whereas in 5×5 Tic-Tac-Toe the number of possible states balloons to $25! \approx 1.5 \times 10^{25}$. In order to figure out how to play games better, we cannot rely on our brains alone to figure out higher dimensional Tic-Tac-Toe. Instead, we can rely on computers to do calculations for us, as computers are much faster at evaluating a large number of states.

In this report, we will be investigating various game-playing algorithms and their effectiveness in solving variants of Tic-Tac-Toe. These variants come in two flavors: larger board (e.g. 4×4 board, 5×5 board) and higher dimension (e.g. $3 \times 3 \times 3$ board). This is important because algorithms which solve general game-playing tasks can be adapted to other domains. For example, Fawzi et al. investigated treating the task of discovering matrix multiplication algorithms[1] as a game which they dub call TensorGame, and use game-playing techniques like AlphaZero[2] in order to discover faster matrix multiplication algorithms (which have resulted in a 10-20 percent speed up over older matrix multiplication algorithms such as Strassens algorithm). So while solving games in isolation might seem disconnected from reality, if we can frame real-world problems in terms of games, then these algorithms become incredibly useful.

Some questions that naturally arise from higher-dimensional variants of Tic-Tac-Toe are: which algorithms work best? Which ones work the fastest? Which inductive biases/evaluation functions will allow our agents to make best use of their state space search? And how do these algorithms scale (in computational complexity and winrate) as we scale up the board size and dimension? In our analysis, we will explore all of these questions.

3 Simplifications

We represent the game state of a Tic-Tac-Toe board as an N by N matrix, with the number 1 denoting where the Xs are placed as well as the number -1 placed where the Os are. We then fill the rest of the matrix with 0s. This makes it easy for us to visualize the game as well as for us to compute our evaluation functions on the game. We can then use a recursive definition of the MiniMax function to create and evaluate our agents over a high depth and many simulations.

Another simplification we make in this project is the simplification of our objective: to determine if an agent plays optimally, we have to assess its performance in every possible state. While this may work for small Tic-Tac-Toe variants, verifying an agent is optimal becomes increasingly difficult the more complex our Tic-Tac-Toe game becomes.

To alleviate this, we instead opt to benchmark agents using relative comparisons. This includes benchmarking our agents against a random baseline as well as benchmarking them against each other. The benefit of this is that it is very computationally cheap to benchmark our agents like this: on a $N \times N$ board, a random agent will encounter at most N^2 positions, and evaluating an agent against itself many times is far cheaper computationally than evaluating it on every board state. The downside is that we are only afforded relative comparison between different agents, as an agent which beats a random agent every time still might not be optimal. Still, this lets us build good intuition as to which heuristics work well and lets us build competitive algorithms for higher-dimensional Tic-Tac-Toe.

4 Mathematical Model

4.1 MiniMax

MiniMax is an algorithm proposed by John Von Neumann for general game-playing. The high-level idea is that in a two-player game, one player will try to maximize their reward while the other will try to minimize their opponent's reward.

More formally, we can name the two agents (players) in our game p_0 and p_1. We can call the set of all possible states in our game S. And we can call the set of all possible actions A. Each agent uses a policy $\pi : S \to A$ to decide which move to make in a given state. In this project, we will treat MiniMax as an algorithm trying to approximate some optimal game-playing policy π^*. We can define a function $P : S \times A \to S$ which determines the state transitions given a state-action pair, i.e. $P(s, a) = s'$ for some states $s, s' \in S$ and some action $a \in A$, where s' is the state achieved after using action a on state s. We can also define an evaluation function $f : S \to \mathbb{R}$ which tells the agent how good it is to be in a certain state.

One possible way to use of f to build a policy is to look ahead one move and pick the action a that maximizes the value of the subsequent state $P(s, a)$, i.e.

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, f(P(s, a))$$

However, using the function f to determine the value of a position alone is ignoring what the opponent can do. In the context of Tic Tac Toe, we might make a move which will let us win two turns from now, but if that move gives the opponent the ability to win the game the turn before we do, and f might not capture that. Instead of using $f : S \to \mathbb{R}$ it's more useful to consider a state-action evaluation function, $f' : S \times A \to \mathbb{R}$

$$f'(s, a) = f(P(s, a), a^*)$$

with

$$a^* = \operatorname*{argmin}_{a'} f(P(P(s, a), a'))$$

Intuitively, this means that the value of a given state-action pair is the minimum value of all the child nodes of the subsequent state s', which just means that if p_0 wants to evaluate the value of the state-action pair (s, a), it will have to take into account the fact that p_1 will act to minimize the reward in the following game state $P(s', a')$ according to f for some optimal a'.

With these definitions, we can mathematically define the MiniMax policy. In general, we want to maximize the value of f, so if our adversary is always minimizing the value of f, we should look ahead to all possible game states N moves ahead and determine which group of leaf nodes in our search has the highest minimum value. If p_0 and p_1 are playing optimally, then the policy for each agent π_i is going to be

$$\pi(s) = \operatorname*{argmax}_{a} f'(s, a) = \operatorname*{argmax}_{a} \operatorname*{argmin}_{a'} f(P(P(s, a), a'))$$

In other words, the best way to play is to pick the action with the "maximal minimum value." In this project, we find that MiniMax is the best performing general game-playing algorithm for tic-tac-toe and is the most robust to changes in dimension and board size. Note that this definition only applies to a 2-move lookahead (i.e. what state will we land in after both p_0 and p_1 make a move), but we can simply nest these calculations and apply them iteratively to achieve search of arbitrary depth in a game tree.

4.2 MiniMax Time Complexity Analysis

We dedicate a section of this report to analyzing the time complexity of MiniMax, which can be an issue when applied to games with large state spaces, such as higher dimensional Tic-Tac-Toe. First, let us consider the fact that MiniMax is a tree search which looks over every possible state up to a certain depth δ (which is a hyperparameter of the algorithm).

Since tic-tac-toe is such a simple game with a limited and discrete set of spaces S, we can figure out exactly how many positions are searched by π. We can define the number of states seen by a minimax search on a $N \times N$ board with depth δ as $b : \mathbb{Z}^+ \times \mathbb{Z}^+ \to \mathbb{Z}^+$, with $\delta \in \mathbb{Z}^+$ and $N \in \mathbb{Z}^+$. In essence, $b(N, \delta)$ will yield the number of positions seen by the minimax agent at depth δ on a $N \times N$ board.

First, we should consider the case where $N = 1$. In this case there is only one position, and so $b(1, \delta) = 1$. If we take $N = 2$, we now have four squares on our game board. At depth 1 (i.e. only look at the current game board), we have 4 possible states, so $b(2, 1) = 4$. If we scale this up to $\delta = 2$, it's trivial to see that the number of possible states is 4×3, as after each first move there are 3 possible follow-up moves, so $b(2,2) = 12$. A pattern is starting to emerge here: if k moves were available at $\delta = t-1$, then $k-1$ moves will be available at $\delta = t$, as one piece is placed each turn, taking up a square for future piece placement. The total number of states we observe in this case is $k(k-1)$, as for each of our k moves to consider from a root position s, the following state has $k-1$ children states to consider in our MiniMax search.

Intuitively, the longest possible game on a $N \times N$ board would take N^2 moves (which corresponds to a draw or a win on the last move). Therefore, a MiniMax agent at $\delta = N^2$ must look at every possible game position on the Tic-Tac-Toe board. From this, we can deduce an upper bound on the number of states encountered by a MiniMax agent at depth N^2:

$$b(N, N^2) = N^2!$$

At the start there are N^2 possible moves, then $N^2 - 1$, then $N^2 - 2$, and so on, which corresponds to looking at $(N^2)(N^2-1)(N^2-2)...(1) = N^2!$ moves.

Putting all of this together, we can consider the case where $\delta < N^2$. In this case, we look first at N^2 moves, then at $N^2 - 1$ moves, all the way until we have $N^2 - \delta$ options. The number of positions searched in this case is $(N^2)(N^2-1)...(N^2-\delta)$, which is equivalent to $\frac{N^2!}{(N^2-\delta)!}$ as we consider only the first $N^2 - \delta$ elements of the $N^2!$ factorial expression, so the denominator cancels out the rest. In other words,

$$b(N, \delta) = \frac{N^2!}{(N^2-\delta)!}$$

This is problematic. As our board dimension increases (i.e. as N increases), the number of possible states grows intractably fast. For example, a MiniMax agent playing 3×3 Tic-Tac-Toe with $\delta = 5$ will have to look at about $15,000$ positions, whereas the same agent on a 10×10 Tic-Tac-Toe board will have to look at about $9,000,000,000$ positions.

As a small aside, this analysis is assuming our MiniMax tree never runs into early terminal states (which isn't true). For example, our function $b(N, \delta)$ includes positions where both players are winning simulatenously (which is impossible). That being said, this is still a useful and representative upper bound.

One question that naturally arises is: how do we reconcile the effectiveness of MiniMax with its time complexity? On the one hand, if we search with $\delta = N^2$, we will have optimal play as we can consider each possible game position of our Tic-Tac-Toe board. Yet, this will take our computer forever to do! So what's the right δ? And we can we craft an evaluation function $f(s)$ that introduces enough inductive bias (like the fact that a 2-in-a-row is advantageous to have) that our agent performs well even on low depth?

5 Mathematical Solution

Given enough compute resources, MiniMax can play optimally in any Tic-Tac-Toe variant. Consider the scenario of 3×3 Tic-Tac-Toe: the maximum game length is 9 moves, so if we set our MiniMax agent to $\delta = 9$ we can define an evaluation function that returns +100 if we are winning in a given state, -100 if we are losing, and 0 otherwise. This will allow us to essentially search every position to determine if we have a forced win or draw. Generally speaking, if we have a $N \times N$ board, then a MiniMax agent with $\delta = d^2$ will be able to search every possible state. And for $N \times N \times N$, $\delta = N^3$ yields the same result. In theory, the optimal model should always beat or draw a random agent, and it should always draw against itself. We confirm this experimentally for 3×3 in Section 6.

For larger boards, this becomes computationally intractable, as discussed previously in the time-complexity analysis of MiniMax. What we can do to alleviate this is to equip our MiniMax agents with evaluation functions which introduce useful heuristics to make better use of the information in a given game state. For example, having 2-in-a-row chains is necessary in order to get 3-in-a-row for 3×3 Tic-Tac-Toe, so perhaps counting the number of 2-in-a-row connections a given player has will allow our agent to gain more information from each position (and thus make a more informed decision), allowing us to potentially lower the depth of our search to obtain similar performance. Our method of dealing with intractable size of generalized Tic-Tac-Toe state spaces is to use clever heuristics to determine which positions are good or not thus relieving the need to look as deep into our game tree. In practice, we evaluate most of our methods with these clever evaluation functions and $\delta \leq 3$, as any δ larger than that is slow to experiment with.

This is a common technique used for MiniMax agents in other games with large state spaces. For example, early versions of the chess engine Stockfish used a hand-crafted evaluation function to determine the value of each state. We will now evaluate various heuristic functions and show which ones are the most performant, comparing and contrasting them to random agents acting on our various Tic-Tac-Toe variants to measure performance.

6 Results

6.1 $N \times N$ Tic-Tac-Toe

6.1.1 Random Agent Behavior in $N \times N$ Tic-Tac-Toe

Before analyzing MiniMax behavior, it is useful to analyze the behavior of random agents at varying board sizes to gain intuition for the geometry of Tic-Tac-Toe in $N \times N$ boards. We start by evaluating random agents against each other at various board sizes: N=3, N=4, etc, for 100 simulations at a time:

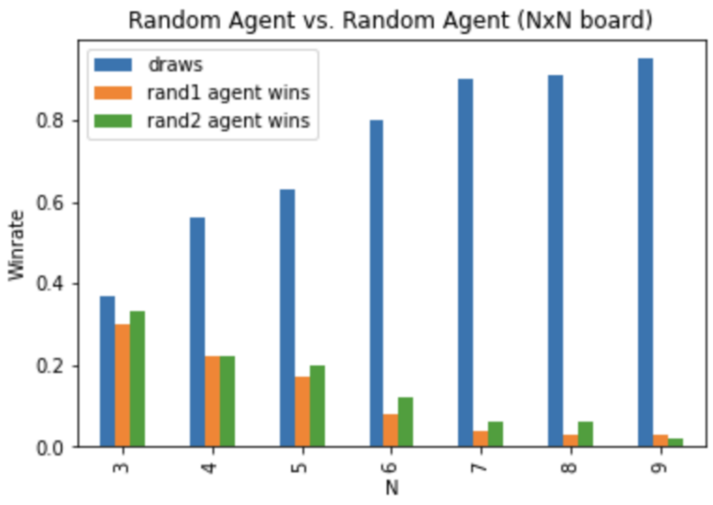

Figure 6.1: Random Agent vs. Random Agent performance on varying $N \times N$ Tic-Tac-Toe boards. Results are aggregated over 100 simulations for each board size.

An interesting pattern emerges here: as N increases, draws become the vast majority of the results. One intuitive reason for this is, as the board size increases, a random opponent will get more opportunities to put their piece in front of a random agents n-in-a-row chain, decreasing the amount of possible wins.

We attempted to prove this phenomenon rigorously, but it proved to be too complex of an issue to address in this report (and it is quite tangential to our main findings). However, we can provide a stronger mathematical intuition that might help explain this phenomenon. Let $T(N)$ be the number of terminal states in an $N \times N$ Tic-Tac-Toe position, let $W(N)$ be the number of winning states in $N \times N$ Tic-Tac-Toe board, and let $D(N) = T(N) - W(N)$ be the number of possible draws in an $N \times N$ Tic-Tac-Toe board. One way to prove that the number of draws converges to 100% of the outcome of random play for arbitrarily large N is to prove the following expression is true:

$$\lim_{N\to\infty} \frac{D(N)}{T(N)} = 1 \implies \lim_{N\to\infty} \frac{T(N) - W(N)}{T(N)} = 1 \implies \lim_{N\to\infty} \frac{W(N)}{T(N)} = 0$$

In other words, if you can show that the number of possible winning states grows more slowly than the total number of terminal states for arbitrarily large N, then this statement will be true. Since the number of terminal states is on the order $O(n^2!)$, it suffices to show that $O(W(N)) < O(n^2!)$ The problem of finding a suitable upper bound for $W(N)$ is out of the scope of this analysis and hence we do not include a formal proof of this, yet we conjecture it to be true given our empirical results.

The insight this gives us is that we should expect our agents to be drawing more often in higher dimensions, independently of the agents themselves. So if we see many draws for larger $N \times N$ boards, this is partially responsible.

6.1.2 MiniMax for $N \times N$ Tic-Tac-Toe

For the 3×3 case, we were able to compute the full MiniMax tree and compute a full solution to this Tic-Tac-Toe case. Although this took about 6 hours to compute, we were able to complete this search tree. Intuitively, we know that for this simple Tic-Tac-Toe case, two ideal players will always draw. We were able to verify this result experimentally: against a random player, our MiniMax agent is able to win 88 times and draw 12 times. And against itself, it always draws.

Now we focus our analysis to lower-depth and larger board dimensions, which involves creating some heuristic functions f. In order to figure out the best heuristic function for $N \times N$ boards, we first define a few candidate evaluation functions and then run simulations to determine the best heuristic function. Below are definitions of the various evaluation functions we used:

1. Edge Eval
   The edge eval evaluation function determines which player has more pieces on the edges of the board, and assigns reward according to how many pieces a given player has placed on any edge.

2. Simple Eval
   simple eval gives a reward of 100 for a winning position, -100 for a losing position, and 0 if it neither wins or loses.

3. Center Eval
   The function center dist eval assigns a value to each square of the board related to how far it is from the center of the board. The further away, the lower the associated reward. Then, each piece under a given players control is scored by this evaluation and the sum for all of a given players pieces is the evaluation of that state.

4. Neighbors Eval
   The neighbors eval function assigns reward based on how many clusters of pieces a player controls. More formally, for each piece belonging to a player, a score is assigned equal to the number of neighboring squares also occupied by another piece controlled by that player. This has the effect of rewarding a player for creating clusters of pieces.

5. Two-in-a-Row Eval
   The evaluation function two in a row eval rewards the player with two pieces in a row, but subtracts when the opponent has two in a row.

In order to find which of these evaluation functions might perform best, we first evaluate them against a random agent on a 3×3 board with variable depth.

Eval Function Name	Draw	MiniMaxAgent	rand
edge	15	55	30
simple	14	78	8
center dist	22	60	18
neighbors	12	84	4
two in a row	14	85	1

Table 6.1: Heuristic Function Performance Against Random Agent for 3x3 board, $\delta = 1$

Eval Function Name	Draw	MiniMaxAgent	rand
edge	24	62	14
simple	19	77	4
center dist	16	68	4
neighbors	8	88	4
two in a row	8	91	1

Table 6.2: Heuristic Function Performance Against Random Agent for 3x3 board, $\delta = 2$

Against a random baseline with $\delta = 1$, the two-in-a-row evaluation function performs the best with the highest MiniMaxAgent, while the edge evaluation function performs the worst. Against Random with $\delta = 2$, not much changes, as the two-in-a-row evaluation function still has the highest MiniMax Agent and the edge evaluation function has the lowest, but overall, all MiniMaxAgent numbers increase. Both the simple eval and neighbors eval perform very well against a random baseline.

We also show heatmaps of agent performance. One heatmap displays the draw percentage of the agent on the Y axis with the agent on the X axis.

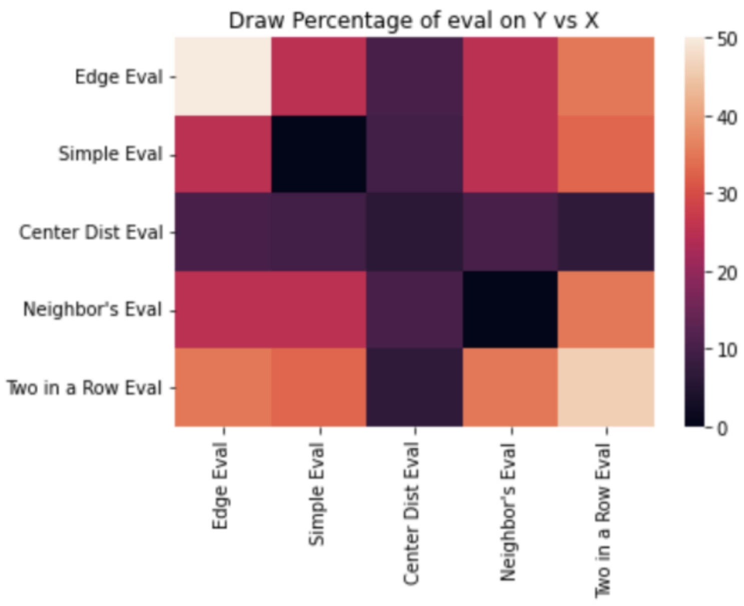

Figure 6.2: A heatmap depicting the draw rate of various agents when pitted against each other. The value of each cell is the relative frequency of draws over 100 simulations of each agent playing each other.

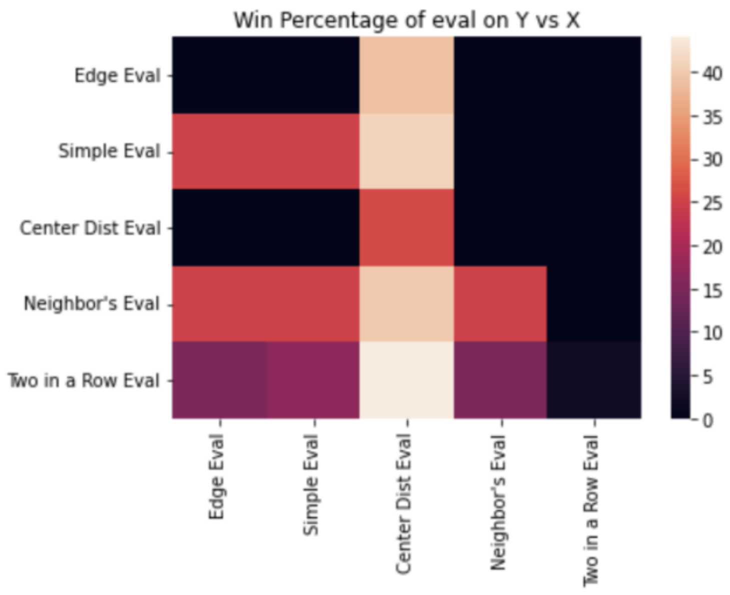

Figure 6.3: A heatmap depicting the win rate of various agents when pitted against each other. Note that order matters, and that the value at a cell (X, Y) is the winrate of agent Y vs. agent X. This means that this heatmap will not be symetrical, as the winrate of Y vs. X is not the same as the winrate of X vs. Y due to draws.

Given these results, it seems promising to explore the two-in-a-row evaluation function further. In order to show that this algorithm is good for the general

$N \times N$ case (not just the 3×3 case), we simulate the agents performance as we vary N. We chose to use the center evaluation function as a baseline comparison, as we believe that, intuitively, the center evaluation function should be robust to changes in N.

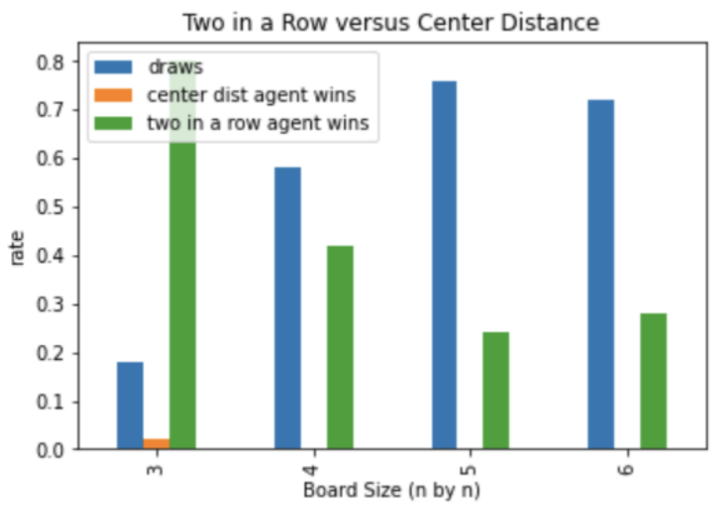

Figure 6.4: A graph depicting the winrate of each agent and their draw percentage relative to the board size N. As N increases, we observe that the performance of the 2-in-a-row evaluation decreases dramatically.

The above results show that for 3×3 and 4×4 boards, the two in a row evaluation worked better. However, the center evaluation method worked better for 5×5, which leads us to hypothesize that for higher dimension boards the two in a row function wont work as well. This makes sense because at higher dimension boards, trying to get 2 pieces in a row is only a fraction of the N pieces in a row you need to win. This evaluation function does not scale.

We can devise a slight adaptation of 2-in-a-row to be more robust at scale: longest row evaluation. Essentially, this function would reward us for the length of the longest chain of pieces in-a-row we have, which should scale better with the dimensions of the board.

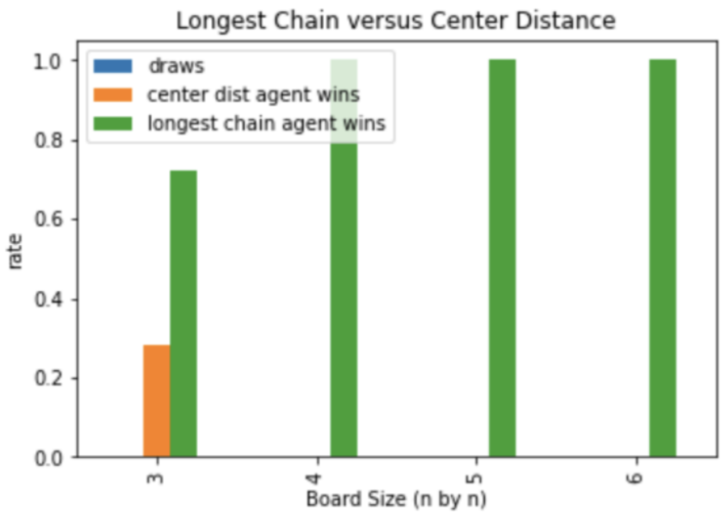

Figure 6.5: A graph depicting the win rate and draw rate of each agent relative to the board size N. Contrasted with the graph above, we see that the longest-chain eval generalizes to higher dimensions much better than 2-in-a-row.

The longest-chain evaluation seems to be much more robust to larger N. Since 2-in-a-row performed the best of all the evaluation functions we proposed, our conclusion is that its generalization (longest-chain evaluation) is the best evaluation function of the ones proposed for the general NxN case.

6.2 $N \times N \times N$ Tic-Tac-Toe

6.2.1 Random Agent Behavior in $N \times N \times N$ Tic-Tac-Toe

We first observed the behavior of the random agents in a $N \times N \times N$ Tic-Tac-Toe game. The simulations for 3D Tic-Tac-Toe games with two random agents were run in the same way as the random agent simulations with 2D Tic-Tac-Toe save for the difference in board dimensions. Observations were noted for 1000 simulations of 3D board with sizes (N) of 3, 4, 5, 10, 20, 50, and 100:

N	Draws	Random 1 Wins	Random 2 Wins
3	0	471	529
4	0	491	509
5	0	524	476
10	0	497	503
20	0	489	511
50	0	510	490
100	0	517	483

Table 6.3: Random Agent Performance in $N \times N \times N$ Tic-Tac-Toe (1000 simulations)

As we would expect with two random agents, the proportion of random agent one wins and the proportion of agent two wins remains roughly equal in the

3-dimensional case. However, unlike what we saw in the simulations of the two dimensional $N \times N$ boards, we observed 0 cases in which there was a draw in any of the simulations of $N \times N \times N$ games. This is an interesting result because in the 2D boards, we noted that as N increases, the proportion of matches ending in a draw increases. However, for the 3D boards the proportion of matches ending in a draw stubbornly stays at a constant 0. Therefore, in our analysis of $N \times N \times N$, we might expect there to be no draws among any of our simulations.

6.2.2 MiniMax for $N \times N \times N$ Tic-Tac-Toe

Due to the state-space complexity of this case, we chose to limit our evaluations to just $3 \times 3 \times 3$. During these experiments, we used 3 different agents: random, 2-in-a-row, and simple eval. We ran 100 simulations of each combination of agents.

Against random agents, the 2 in a row and simple one both won with a 94% win rate. When 2 in a row played against the simple agent it won 64% of the time while the simple agent only won 58% of the time when it started. This leads us to believe that 2-in-a-row is the best heuristic function for the $3 \times 3 \times 3$ case, and by extension, longest-chain for the $N \times N \times N$ case. However, because these evaluation results are so close to each other, it is hard to determine whether or not these results are statistically significant or the result of random noise. One potential avenue of future exploration is to test longest-chain evaluation in the $N \times N \times N$ case.

We also chose to evaluate the center distance function in the $3 \times 3 \times 3$ case. Its definition is similar to in the 3×3 case, and simply sums up the negative distances of every piece placed from the center. While this was very effective against a random agent, winning 99 out of 100 games, when placed against the two in a row agent, it lost 100 out of 100 games.

7 Improvements

7.1 Monte Carlo Tree Search

One alternative to Minimax search is Monte Carlo Tree Search, or MCTS[4]. The goal of MCTS is to assign a value to each position in the tree of possible moves. Instead of searching every position breadth-first with an evaluation function up until a certain depth, monte carlo tree search explores the game tree over a series of simulations and uses something called a random rollout, which essentially boils down to random self-play, to assign value to each node in the tree. Starting from some root node (e.g. the current game position), MCTS will simulate multiple traversals down the game tree.

During each traversal, the exploration of already-seen positions brings up an interesting dilemma: how do we balance exploring new positions with exploiting the knowledge we already have about positions we've encountered more often? There's a balance to strike here: on the one hand, if you explore more positions you might learn something new and valuable which improves your tree's representation of the best possible moves. Yet on the other hand, exploring new positions might reinforce what you already know about the position, and it might be better to get a better understanding of the few good candidate moves by doing more rollouts on positions you already know are likely to be good. In order to solve this, MCTS uses a formula called UCB (upper confidence bound) to determine which already-seen node to traverse to:

$$S_i = x_i + C\sqrt{\frac{\ln(t)}{n_i}}$$

where S_i is the value of node i, x_i is the mean performance of all the rollouts stemming from that position, C is a hyperparameter, t is the total number of MCTS simulations, and n_i is the number of times node i has been visited across all simulations. The hyperparameter C helps the user specify how much they want MCTS to explore new positions and how much it wants MCTS to exploit its initial knowledge when choosing which child node to traverse to from the root. If C is 0, then MCTS will only traverse to the nodes it sees with the highest win% (maximal x_i). However, if C is nonzero, the $\sqrt{\frac{\ln(t)}{n_i}}$ will influence MCTS to explore positions it hasn't visited often (low $n_i \implies$ higher $\sqrt{\frac{\ln(t)}{n_i}}$ since t is a constant). For our experiments, we set $C = 0.1$

In a given traversal, once MCTS has encountered a new position it hasn't seen before, it will compute random rollouts (games played until completion with random moves) and keep track of the result. This result will then be backpropagated to all the parent nodes to update their corresponding values. For example, if a node has two children, one with 5 wins and 5 losses from rollouts and another with 10 wins and 20 losses from rollouts, the value of the parent node will be $\frac{5+10}{5+20} = \frac{15}{25} = \frac{3}{5}$ (assuming a win has value 1).

The reason this method might be useful to us is because it's more computationally efficient than MiniMax. That is to say, for a fixed runtime budget, MCTS will look at more positions than MiniMax, as most of the positions it sees are the result of random rollouts, which lets it look at more positions much faster. It's able to gather information more efficiently than minimax is (on a fixed time budget). Because of this, it might be interesting to compare MCTS to Minimax and see if its efficient use of the state information in the game tree will allow it to perform better than Minimax.

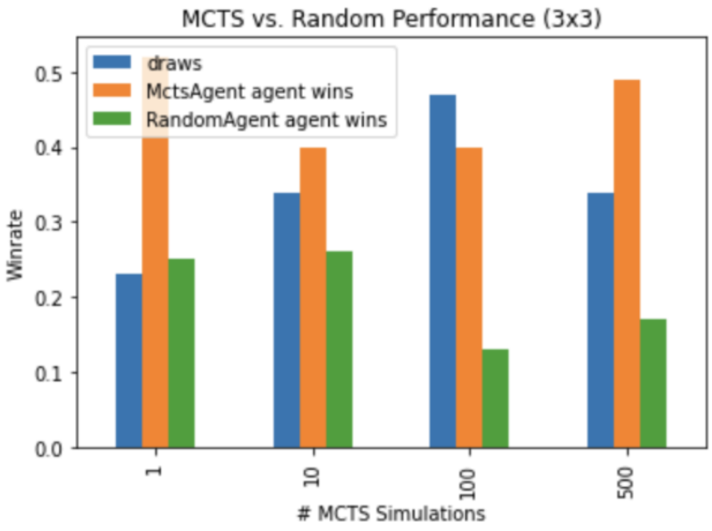

Figure 7.1: A bar graph depicting how the win rate and draw rate of Monte Carlo Tree Search simulations against random performance change based on the number of Monte Carlo Tree Search Simulations run.

From the plot, we can see that increasing the simulation depth seems to greatly affect performance against a random agent, as the random win rate drops dramatically as we increase the number of MCTS simulations, which makes intuitive sense.

Next we evaluate MCTS against various low-depth minimax agents on a traditional 3×3 board using simple eval. We will use 500 MCTS simulations as a baseline comparison.

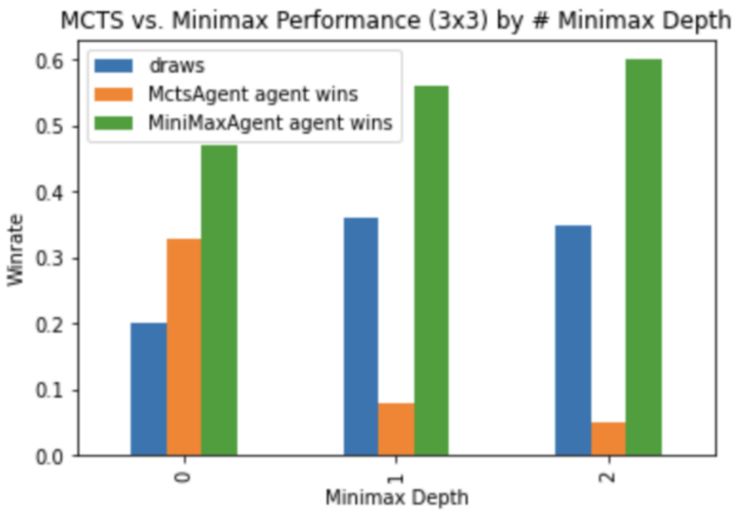

Figure 7.2: A bar graph comparing the performance of Monte Carlo Tree Search and MiniMax as a function of MiniMax depth. As MiniMax depth increases, it greatly outperforms Monte Carlo Tree Search on a 3×3 board.

It seems that giving the MiniMax agent even a little bit of depth lets it dominate the MCTS agent. While the MCTS agent might be using less computational power by stochastically rolling out each state it visits, it probably ascertains a lot less information per state and therefore might not be using all the states it views as effectively as the MiniMax agent.

We also do the same comparison on a 3-dimensional board with the same parameters to see if this result generalizes to 3 dimensions.

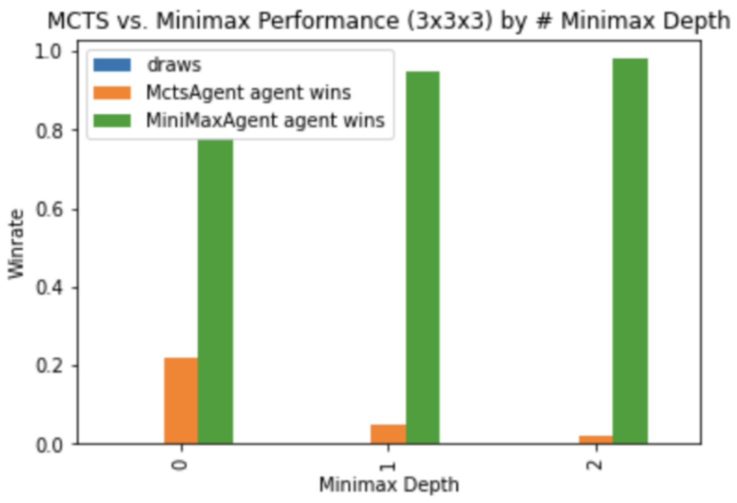

Figure 7.3: MCTS vs. MiniMax on a $3 \times 3 \times 3$ board at various MiniMax depths. MiniMax very clearly dominates MCTS in this situation.

Even on low depth, it seems that MiniMax performs significantly better than MCTS. So while MCTS might be more efficient in its search, it empirically performs worse than MiniMax.

7.2 Alpha-Beta Pruning

An alternative to improving our algorithms performance is to improve its computational efficiency. For higher depth levels, we can see that the MiniMax algorithm can get very slow, this is because, for Tic-Tac-Toe the computation complexity of MiniMax increases factorially as we increase depth. This has the effect of making it difficult for us to create better agents that search with a higher depth. One way to remedy this issue, without changing the mathematical efficiency of MiniMax is called alpha-beta pruning. This algorithm operates on the fact that there are some branches of our MiniMax tree that we compute, when we already know that there is a better solution elsewhere, so we should remove, or prune, these branches of our tree.

More formally, as we traverse our search tree, we can allow α to be the maximum value already guaranteed to the maximizing player p_0. We can also define β to be the minimum value already guaranteed to the minimizing player p_1. If $\beta < \alpha$, then we can say that further descendants of this node should not be considered as they would never be reached in ideal play, and we can stop computing further down this branch. This means that we can potentially consider less positions, lowering the computational complexity of this problem. In the worst case, we can never prune a branch and we get the same complexity as a standard minimax, whereas in the best case we can have a computational complexity that is a square root of the standard model.

After implementing this in code, we demonstrated empirically that alpha-beta pruning improves the search efficiency of MiniMax.

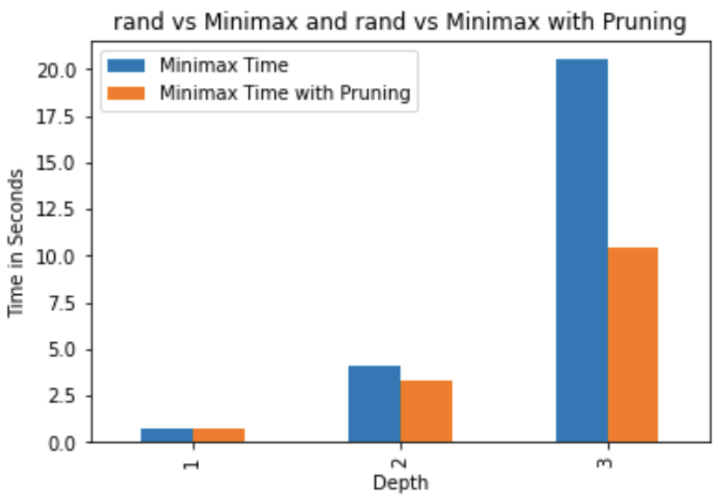

Figure 7.4: A bar graph depicting the change in time of Minimax with and without pruning. As depth increases, both Minimax and Minimax with pruning increases, but Minimax without pruning increases much faster than Minimax with pruning.

8 Conclusion

In $N \times N$ Tic-Tac-Toe, we concluded that longest-chain evaluation is a robust and effective heuristic which allows MiniMax agents to play Tic-Tac-Toe effectively at low-depth despite high dimensional state spaces. We also begin a cursory analysis on MiniMax in $N \times N \times N$. Most notably, while our data is more limited for this case due to the leap in resources required to run $N \times N \times N$ MiniMax, we find that longest-chain eval is also robust in the $N \times N \times N$ case. We also evaluate the performance of Monte Carlo Tree Search, an alternative tree-search algorithm to MiniMax, and find that MCTS performs considerably worse than MiniMax in high dimensional $N \times N$ and $N \times N \times N$ Tic-Tac-Toe. Finally, we discuss alpha-beta pruning, a computational improvement to the MiniMax algorithm, and show empirically that it saves a lot of computational cost.

9 References

1. Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R. Ruiz, F. J., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., & Kohli, P. (2022, October 5). Discovering faster matrix multiplication algorithms with reinforcement learning. Nature News. Retrieved November 29, 2022, from https://www.nature.com/articles/s41586-022-05172-4

2. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2017, December 5). Mastering chess and shogi by

self-play with a general reinforcement learning algorithm. arXiv.org. Retrieved November 29, 2022, from https://arxiv.org/abs/1712.01815

3. How many tic-tac-toe (noughts and crosses) games are possible? How many Tic-Tac-Toe (noughts and crosses) games? (n.d.). Retrieved November 29, 2022, from http://www.se16.info/hgb/tictactoe.htm

4. Libraries, C. U. (1987, January 1). The expected-outcome model of two-player games. Academic Commons. Retrieved November 29, 2022, from https://academiccommons.columbia.edu/doi/10.7916/D8GH9S2C