Final Report

Noah Hendrickson

May 3, 2022

Abstract

There are many approaches to creating a chess AI. I studied chess AIs in a variety of iterations. The first iteration was an AI that chose moves at complete random. The second iteration used a basic evaluation function to determine moves. The third AI utilized Alpha-Beta pruning combined with a more robust evaluation function in order to decide moves. The fourth AI utilized a neural network and Monte Carlo tree search. The results showed that the first two AI iterations lost nearly every game whereas the third and fourth iterations did the opposite and won nearly every game. From these results we can see just how powerful tree search and neural networks are in making more human like decisions.

Introduction

"Intelligent is as intelligent does", but what does it mean to be *intelligent*? [Ash61] What defines an *intelligent machine*? Is it simply the ability to solve problems, or does that machine need to solve problems on the same level as us humans? How do we define what our intelligence is? Some might say intelligence requires free thought, others might say that intelligence is simply making decisions based on input data.

These questions and many like them are core questions in the field of Artificial Intelligence–hereafter referred to as "AI". "How can we use 'intelligent' machines to solve problems" is one that leads to many innovations within computing. Many problems have been solved using AI–path-finding and image recognition to name a few–but the focus of this report is on an incredibly popular game: chess.

Chess: Why Its *NOT* Boring

Chess is, quite possibly, one of *the* most recognizable games. The basic premise of the game is that there is an 8x8 board of tiles that alternate black and white (or some other combination of two colors). No two tiles of the same color will have edges directly touching each other. Two players can play, each getting a set of pieces consisting of 8 pawns, 2 rooks, 2 knights, 2 bishops, 1 queen, and 1 king. Each piece has restricted movement to some degree. Pawns can only move one, or two in some cases, tile in the direction of the other players side, rooks can only move horizontally or vertically but as many spaces as they wish, knights move in an L shape, bishops can only move diagonally, queens can move as many spaces as they wish both horizontally and vertically, and the king can only move a single space in any direction. A player can capture a piece by moving one of their pieces onto the enemy piece.

I won't go too much deeper into the rules of chess for the sake of brevity, however, these somewhat complex, yet still simple and understandable, rules make chess a perfect target for AI. This is reflected by the number of strategies that have been used to create incredible chess AIs. From El Ajedrecista, to Deep Blue, to AlphaZero, to the Leela Chess Zero program utilized in this report, chess AIs have continuously grown and evolved over time and have utilized many different strategies to outwit even the best, most skilled, chess players.

That is why chess is also the perfect game to test an AI against because it is where the human brain shines. The game requires thought and strategy. It requires thinking ahead, deciding moves based on what your opponent could do and what you could do after your opponent moves. It is the definition of a mind game, and that's why we are so fascinated with making chess AIs. If we want to

model something after our own brain, it will have to do what we can do. It will have to be able to play chess, and it will have to play chess on the same level as a human could, and even beyond. Luckily, computers are *really* good at computation and thinking ahead. Even if they might not be the best at making decisions, they have many tools up their sleeves.

Chess AI Through The Decades

Chess AIs have history dating back as early as the 1770's with Wolfgang von Kempelen's "Mechanical Turk" which was capable of playing a "convincing" game of chess against human opponents. However, despite being a marvel, it was, in fact, a fake being operated by a human inside. [Sch99] In order to find one of the first *actual* chess automatons, we must look to the early 1900s at Leonardo Torres y Quevedo. Torres, among other achievements such as creating a radio controlled boat in 1906 and becoming, both the president of the Academy of Sciences of Madrid, and a member of the French Academy of Sciences, created what would be one of the first autonomous chess players. [Ran82]

The El Ajedrecista played, without any human interaction, games of chess already in endgame. The automaton controlled a king and rook, while the opposite player controlled a single king. Although it did reach checkmate each time, it was by no means efficient. The algorithm depended on the location of the automaton's king in relation to its rook. Based on their locations, it would move the pieces accordingly. I will not write out the whole algorithm for the sake of brevity once again. [Rob82] This was obviously a very rudimentary AI, however, for the time it was a marvel. From then on, chess playing AIs only became better. The next focus is one of the most famous chess playing AI: IBM's Deep Blue.

Deep Blue is possibly the most famous chess playing AI due to its win over world champion Garry Kasparov in 1997. It was a marvel of computer science for its time. Hundreds, if not, *thousands*, of people worked on Deep Blue, from chess pros, to skilled names in computer science. Algorithms proposed by Claude Shannon, which will likely be talked about in the next section, were used alongside parallel searching and a database of of possible endgames to create a beast that could beat even the best chess player. [New00] Deep Blue revolutionized chess AIs and led to an expansion of AI in general, introducing iterative deepening depth-first search and leading to new thinking about what it means to be intelligent. [New00] Was it that Deep Blue was being intelligent? Was it thinking? That depends on what you think intelligent means. The creator of Deep Blue's VLSI chips characterized the matches it played as such: "no as man versus machine, but rather as man as performer–Kasparov–versus man as toolmaker–Hsu and the IBM team." [Eva07] Deep Blue was a marvel no doubt, but that was more than two decades ago. Since then, new strategies for creating chess playing AIs have arisen. One that makes use of some of those new strategies is AlphaZero.

AlphaZero is an incredible feat of human ingenuity. Whereas previous chess playing AI utilized only tree searching algorithms and such like that, AlphaZero makes use of machine learning and deep neural networks to play games against itself and progressively get better and better. Now I am not at all qualified to talk about how AlphaZero does what it does, considering I know very little about neural networks and machine learning in general, so I will let the creators say it themselves. "AlphaZero is an adaptive learning system that improves through many rounds of self-play. It consists of a deep neural network f_θ with weights θ that compute $(p, v) = f_\theta(s)$ for a given position of state s. The network outputs a vector of move probabilities p with elements $p(s' \mid s)$ as prior probabilities for considering each move and, hence, each next state s'." [TPHK22] AlphaZero also makes use of a Monte Carlo Tree Search up to a certain depth in order to select moves to play. [TPHK22] Both MCTS and neural networks will be gone into more in the next section.

There have been many other chess playing AIs created since El Ajedrecista. The IBM 704 had a program which could play a complete game of chess against a human, albeit at an amateur level. [BADVRB58] The Greenblatt chess program, from the Artificial Intelligence Laboratory of Project MAC at MIT, could win about 80% of games against non-tournament players, and in tournaments had a highest performance rating of 1640. For reference, the weakest opponent it played against had a rating of 1680 and the mean rating of all US tournament players at the time was 1800. [GEC67] BLITZ V from the University of Southern Mississippi was made to perform a search on fewer than 500 nodes and make a decision based on that. It was called BLITZ because its moves were taken in under 15 seconds. It used "human-like" processes to make its decisions. BLITZ V, when playing against non-chess club players, won roughly 90% of its games and in tournament play, against players rated

1500-1700, it would win about half the games. [Hya77] Belle, developed by Ken Thompson and Joe Condon at Bell Laboratories in 1979, was the first chess program to be awarded the title of Master by the United States Chess Federation and could search nearly 150,000 positions per second. [New03] Clay Blitz, developed by Robert Hyatt, Albert Gower, and Harry Nelson at the University of Southern Mississippi in 1983 went even further beyond, being able to compute 200,000 positions per second, and obtained a USCF rating of 2400. [New03]

These AIs, AlphaZero, and many others are not just feats of engineering, but also raise more questions about what it is to be intelligent. Machines are now learning from doing in much the same way that we learn. As more and more is learned about how to make AIs smarter, it is inevitable that we will be lead to question more about what it means to be intelligent. In the meantime, what is it that makes these AIs learn in the first place? Multiple methods have been mentioned thus far: heuristics, Monte Carlo Search, neural networks, depth first search, but *what are these* and *how are they used to simulate intelligence*?

How Do They Do It?

There are numerous strategies that are implemented by chess playing AIs in order to properly play the game–Minimax, Depth-First Search, Alpha-Beta Pruning, Monte Carlo Tree Search, and neural networks to name a few. A natural starting point is one of the simplest: the heuristic.

The word "heuristic" is derived from the Greek word "heuriskein" which means "find", and that is exactly what computers use a heuristic for. Games, including chess, cannot be played algorithmically or exhaustively due to the incredible amount of computational power that would be required, thus, one must take a heurstic approach. The heuristic determines how "ideal" a position or set of positions is based on some sort of function defined by the programmer. The difference between a good and great player is how good their heuristic is. [Fin03] An example of a crude chess heuristic comes from an article by Claude E. Shannon:

$$f(P) = 200(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') +$$

$$(P - P') - 0.5(D - D' + S - S' + I - I') + 0.1(M - M')$$

where K, Q, R, B, N, and P are the number of white kings, queens, rooks, bishops, knights, and pawns on the board, D, S, and I are doubled, backward, and isolated pawns, and M is white mobility which could be measured by the number of legal white moves. This is, obviously, not a great heuristic, however, the heuristics involved with chess AIs have naturally evolved over the decades to become more and more accurate. [Sha88] Heuristics provide a good starting point for the making of a chess AI, however, a heuristic alone could not even beat a decent human player. More is needed, and that more comes in the form of the minimax algorithm. Although the minimax algorithm can be used in the context of single player games, chess is two player, thus that is the context it shall be discussed under.

The minimax algorithm was proved by John von Neumann in 1982, and can be used in two-player, zero-sum, games, where one person wins, and the other loses. Say that a move gives a player a certain score. The minimax algorithm works to find a move for the current player that minimizes the maximum value move that the opposing player can make on the immediate following turn. A minimax algorithm can look multiple turns ahead and analyze the moves to determine which move would be the best, however, this can get incredibly computationally intense if there are a lot of possible moves. [Sha88] [Mar91] The space that the minimax algorithm searches is very often represented as a tree, with each new game state being a node in the tree. The computation gets expensive because the algorithm must visit every node in the tree. "For a uniform tree with exactly W moves at each node, there are W^D nodes at the layer of the tree that is D ply from the root." [Mar91] The minimax algorithm is a key algorithm for the search of game state spaces, however, its base implementation is not at all viable for non-trivial games like chess. Luckily, other variations have been created since the inception of the minimax algorithm, namely, Alpha-Beta Pruning.

The idea of Alpha-Beta Pruning withing a minimax search goes back to either Newell, Shaw, and Simon in 1958 or John McCarthy and his group at MIT. Alpha-Beta Pruning is an extension of the minimax algorithm, in which lower (α) and upper (β) bounds on the expected value of the tree are implemented. [Mar91] This is important because it allows the algorithm to deduce that some moves

in the tree will lead to board states that do not matter in the final calculation of the value, thus those nodes, or even entire branches, can be pruned from the tree and do not have to be expanded. The effectiveness of this algorithm can vary. "If every position on levels $0, 1, ..., l-1$ of a game tree has exactly d successors for some fixed constant d, then the alpha-beta procedure examines exactly $d^{\lfloor l/2 \rfloor} + d^{\lceil l/2 \rceil} - 1$ positions on level l." [KM75] Early versions of this algorithm had a flaw such that deep-cutoffs were not achieved, but a recursive algorithm was introduced by Knuth and Moore in 1975 such that the flaw was corrected. Additionally, they implemented another algorithm called "negamax" which ensured that only maximizing was needed. [Mar91] [KM75] [CM83] Another variation of alpha-beta pruning is aspiration search. This sets initial bounds on the values of α and β such that if the minimax value is within those bounds, the search completes. [Mar91]

Quiescence is another important factor to consider within search algorithms for chess. Quiescence is the idea that a search should only evaluate positions whose value would not be ope to substantial change if searched one turn deeper which excludes moves such as checks, promotions, or complex captures. [Mar91] The idea is that we want the outcome to be predictable, and not to miss something that could happen that is incredibly important. This is a key idea, but can potentially lead to even more computation time depending on the move.

Alpha-Beta pruning and minimax together can be a powerful combination. Many variations of minimax further build on its vision such as palphabeta and scout. [CM83] Additionally, improvements have been made to the base Alpha-Beta Pruning algorithm. Principal Variation Search in a negamax framework is one of those improvements. [Mar91] Additionally, forward pruning, in which the programmer assumes the opponent will be playing such that their move will make a situation worse, can be implemented, however, it relies on those assumptions and thus can fail in certain circumstances. [Mar91] A more recent implementation of the minimax algorithm with alpha-beta pruning makes use of hardware parallelization to speed up the computations for the game of Loop Trax. [MAS15] This algorithm also uses the technique of Path Based Memory and Parallel Path Analysis to both speed up computation, and cut down on memory consumption. [MAS15] This does not directly apply to chess, however, it is a demonstration of the strength and usability, even still today, for these algorithms and techniques. Many other techniques have been tried to improve the search that an AI performs, however, attempting to go through all of them would be a futile task, thus, we shall now take a look at two much more modern methods of chess AI building: The Monte Carlo Tree Search and Neural Networks/Machine Learning.

The Monte Carlo Tree Search was created in 2006 and was first used by Remi Coulom in his Go-playing program, Crazy Stone. [Fu18] Later, MCTS would be used by both AlphaGo and AlphaZero in order to train the neural networks that ran the AIs. MCTS combines precision of tree search while also implementing random sampling. It takes random samples in the decision space and builds a search tree according to the results. [BPW+12] The tree search balances looking down paths that have not been explored yet with looking down paths that seem to have good values, and progressively updates the nodes of the tree based on the paths that were taken and the outcome that was observed from the taken path. It improves upon the previously mentioned tree searches because it does not have to evaluate each individual node, rather only the value of the terminal node matters. [BPW+12] [Fu18] MCTS combines Markov Decision Processes with Game Theory to produce an algorithm that can be applied to many situations, although generally the base algorithm has to be tweaked to fit the specific implementation. The basic steps of the MCTS are as such: The first step is to select children from the root node to descend through the tree until the most urgent node is reached, then a child or children are expanded and a simulation is run from that or those child node/s, and when the simulation finishes, it is backed up through the node path to update the nodes. [BPW+12] A notable variation of MCTS is the Upper Confidence Bounds for Trees algorithm. UCT can allow MCTS to converge to minimax and thus be optimal. [BPW+12] Akin to minimax, iterative deepening can be applied to MCTS to make it more effective. Additionally, as with AlphaGo and AlphaZero, MCTS can be used to train neural networks, which are, coincidentally, the next topic.

Machine Learning is a field of computer science in which software analyzes examples of desired behavior and comes to conclusions based off of them. Reinforced learning specifically applies to chess, which is where a program must "make a sequence of decisions before receiving a reward signal." [Die03] The program takes actions, makes plays, in a certain sequence and then at the end receives a "reward" which may be a point evaluation of the outcome. The board state is also updated with the outcome of the sequence of actions. The goal of this process is to construct an optimal policy for choosing

actions. [Die03] Now, the astute observer may realize that this seems quite similar to the process of a Monte Carlo Tree Search, and they would be right. MCTS and the process of reinforced learning go hand in hand. Neural networks can take the chess MCTS and run it multiple times to build up a database of information about probabilities of certain moves being more beneficial than others. The previously talked about AlphaZero does just this. This sort of machine learning can be especially useful in the chess beginning game and endgame. In both situations, there are a limited number of moves that generally follow a structure or strategy. Machine learning paired with MCTS or other strategies can very quickly deduce the best course of action for a large variety of situations, even not limited to beginning game or endgame. [Li20] Neural Networks can be paired with minimax as well to greatly improve the performance over base minimax. [dP09] Although machine learning is a relatively new concept compared to much else of what has been talked about here, it is quite possibly the most powerful tool that we have at our disposal for chess AI creation.

In the end, a great variety of algorithms and processes have been created to continually refine the creation of chess AIs. Evolution is the staple of computer science, and that can be clearly seen with chess AIs. From the very rudimentary El Ajedrecista that used sensors and an incredibly basic algorithm, to AlphaZero which makes use of cutting edge Neural Networks and MCTS, chess has pushed the boundaries for AI creation and thought on the original question: what does it mean to be intelligent. But instead of just saying all of this, let's get into actual implementations and data!

The Approach

I want to start this section with a simple statement: Chess AIs are *cool*! I think they are so cool, in fact, that I decided I would make my own for this project, and that I kind of did! The general structure of my testing is that I obtained 4 different chess AIs. One made moves at complete random, one made moves based on an evaluation of the current board state, one made moves based on a variable depth minimax algorithm with alpha-beta pruning, and the final combine neural networks and Monte Carlo search trees to make a formidable opponent. I will cover the first two AIs first because they are my own creations, and because it is generally best to start small and build up the the big league.

A Fool's Gambit

About a month ago, I set out to create an chess program and a chess AI. My original idea was to use a turtle program for C++ so that I did not have to learn a whole new graphics system while making use of the speed of a lower level programming language. Unfortunately, that proved to be a futile task and would come back to bite me in the ass. CTurtle, the library used, did not have the functionality needed, so I ended up wasting about two weeks. Fortunately, python is *incredibly* easy and the turtle functionality is quite robust, so thus began the next leg of my journey. I put together the basic functionality of the chess game fairly quick. First the board, then the pieces, then selecting pieces, then moving pieces, and, eventually, move generation. There were hiccups along the way and my code progressively got messier and messier, but it worked. Eventually, two people could play a full game against each other with minimal bugs, *and* I had a much better understanding of the intricacies of chess. The next task on my docket was to make an AI player. This was a quite easy task at first. Some funky things with turtle mouse-input had to be figured out, but, without too much effort, and AI was made that could play moves at random. I gathered up a group of friends and set to work gathering data.

The Weakling

The first AI, nicknamed "The Weakling", played moves at complete random. It generated the valid moves it could play and utilized python's 'random' module in order to generate a random move. Needless to say, this AI did not perform well. Six games total were played against it. The results of the games are pictured in the table below.

The Weakling					
Game	White Moves	White Captures	Black Moves	Black Captures	Time
1 - W	N/A	N/A	N/A	N/A	N/A
2 - W	28	11	27	2	527
3 - W	60	15	59	11	172
4 - W	42	15	41	5	188
5 - W	15	8	14	0	262
6 - W	7	2	6	0	42

The above graph depicts six games and the stats for five of them. The color that won is stated after the game number. W or White was the human player and B or Black was the AI. As the graph shows, it lost every game that was played. The games were played against people of various skill levels, however, that generally didn't matter. Because of the random moves, the stats are kind of all over the place. Some games were won very quickly, others took longer. Sometimes the AI captured a lot of pieces, other times, none. Randomness is an important idea in AI with some of the most powerful algorithms making use of it, however, as these results show, relying completely upon randomness is a death sentence in almost every circumstance. Luckily, we have many more tools up our sleeves.

Pain

It was at this point that I was starting to regret using turtle. Pro tip: plan things out before doing them–this will come up more later. Anyways though, at this point I had started on my second iteration of the AI. I was aiming to have some sort of evaluation function to determine how much a certain board state was "worth", and I ended up with something that worked, but, again, not very well. The process of making this AI led me to fix multiple bugs in the original program–a lot to do with castling–and made my code *even messier*. Once I was done, I gathered up that same group of friends and took to playing games.

The Somehow-Worse

The functionality of this AI is essentially a one deep minimax. The AI generates its possible moves and, for every move that is available, it performs that move on a test board. From there, it iterates through the entire board and counts up the values of the pieces. Kings are worth +Infinity, Queens are worth 900, Rooks are worth 500, Bishops and Knights are worth 300, and Pawns are worth 100. If the piece's color is white, it subtracts that piece's value from the total rather than adding. Additionally, the function generates the opponents responding moves and keeps count of the total number. That total number is then multiplied by 10 and subtracted from the evaluation total. The basic premise is that Kings are priceless, Queens are worth a lot, rooks are very useful, knights and bishops are roughly equal, pawns are on the lower end, and the fewer moves the opponent can play, the better. This time, nine games were played and the results are shown in the table below.

The Somehow-Worse					
Game	White Moves	White Captures	Black Moves	Black Captures	Time
1 - W	41	15	40	5	196
2 - W	28	13	27	4	354
3 - W	29	12	28	7	531
4 - W	41	15	40	7	116
5 - W	34	15	33	5	171
6 - W	5	2	4	0	9
7 - W	8	3	7	1	8
8 - W	5	2	4	0	7
9 - W	5	2	4	0	5

Once again, the game number is listed with the color that won (W for man, B for machine) to the right of it, as well as the stats. These results show that the AI still lost every game, but had more consistently bad stats. This may be a good thing, however, this AI doesn't even have the chance of getting a good play in by complete luck. Additionally, as one can probably see from looking at the last 4 games, it can be tricked *really* easily. Those last 4 games were from the players learning how the AI played and utilizing that in speed running checkmate. The evolution is clear, but it isn't enough of an evolution to really give the AI the boost it needs. Very reminiscent of the first iteration, the evaluation of the board is used in many algorithms, however, on its own, it is not enough to allow the AI to make human-like decisions. In order to do that, the AI needs to be able to think a couple steps ahead. As mentioned before it also had alpha and beta variables in order to make use of alpha beta pruning and not do unnecessary computations.

Admitting Defeat

I do not claim to be a great programmer. I managed to get chess working, however, I did not start with a solid plan and I did not plan for the future, and that tripped me up when I went to implement a minimax search with alpha-beta pruning. I initially thought that it would be a simple task. The code seemed to come together nicely, and I couldn't see any logical errors in it, however, when attempting to run the algorithm, bug after bug occurred. I eventually deduced that the way that I had written my move generation would not work with the search, and so I gave up.

What I had coded out was an algorithm that was given a depth, alpha, and beta (and a couple other things that aren't important to mention). The algorithm then generated every move for the specified piece color and played each move individually. After playing the move, the algorithm recursively called with a decremented depth. When depth reached zero, the algorithm would evaluate the board state and would return the value. It was a minimax algorithm with alpha-beta pruning...or at least was supposed to be.

However, I still had backups. I wanted to code out my own AI because I wanted to both get better at coding and learn what it takes to make an AI. Having been partially successful at done that, I was satisfied with my attempt and fell back on my backup. The third AI is basically what I wanted my third iteration to be like, but before I get into the details, a much more thorough and detailed description of the chess program and AI that I am using for the third iteration can be found in this YouTube video which is also where I got the idea for this project!

The Backup

The third AI iteration is a chess program and AI created by Sabastian Lague in the YouTube video titled "Coding Adventure: Chess AI". [lag] The link to this video is at the end of the last section. This is not my own work, but it is what inspired me to do chess as my project and to make my own chess AI.

Beyond just having a nice and fancy display and feel, this AI is much more powerful than the previous two and has many more components. Starting with the basics, the AI implements a piece valuing system similar, but very different, from the second iteration. Each piece is given a value and those values are the same as the second iteration values. In addition to the piece values, the AI favors positions where the opponents king will be forced away from the center. The total value is updated based on that distance. Additionally, the king is incentivized to move close to the opponents king in order to cut off routes and help with checkmate. [lag] This is again implemented by updating the total evaluation with the distance between the kings. Both of these help with endgame situations and without them the AI would flail aimlessly. Akin to adding to the total based on distance from edge, another feature of the evaluate function implements giving more value for certain regions of the board based on the piece moving. The following maps are reminiscent to what is used in the program, although likely not the exact values:

Knight

−50	−40	−30	−30	−30	−30	−40	−50
−40	−20	0	0	0	0	−20	−40
−30	0	10	15	15	10	0	−30
−30	5	15	20	20	15	5	−30
−30	0	15	20	20	15	0	−30
−30	5	10	15	15	10	5	−30
−40	−20	0	5	5	0	−20	−40
−50	−40	−30	−30	−30	−30	−40	−50

Pawn

0	0	0	0	0	0	0	0
50	50	50	50	50	50	50	50
10	10	20	30	30	20	10	10
5	5	10	25	25	10	5	5
0	0	0	20	20	0	0	0
5	−5	−10	0	0	−10	−5	5
5	10	10	−20	−20	10	10	5
0	0	0	0	0	0	0	0

Rook

0	0	0	0	0	0	0	0
5	10	10	10	10	10	10	5
−5	0	0	0	0	0	0	−5
−5	0	0	0	0	0	0	−5
−5	0	0	0	0	0	0	−5
−5	0	0	0	0	0	0	−5
−5	0	0	0	0	0	0	−5
0	0	0	5	5	0	0	0

[wik] These serve to mix up the positions that the AI will move to, especially during the early game, as without them it just aimlessly moves back and forth at the beginning rather than taking control of the center.

The next major boost to performance is the implemented minimax search. The search accepts depth, alpha, and beta. Each time it is called, it generates the available moves, checking for check and returning -Infinity if so, and for each of those moves it makes the move, calls recursively with decremented depth, and unmakes the move once that returns. Once the depth reaches 0, the evaluate function, described in the previous paragraph, is called. The alpha and beta values are updated accordingly and utilize alpha-beta pruning to drop the search times significantly. An example of this is a move evaluation going from roughly a second, evaluating 33,553,501 positions, to a fifth of a second, evaluating 464,795 positions. This doesn't make the search better, just faster. Additionally, alpha-beta pruning really shines when the moves are ordered by importance of the move. A minimax algorithm with alpha-beta pruning is equivalent to a normal minimax when the moves are ordered from lowest to highest, thus, move ordering makes a large impact on the decision time. The third iteration AI makes use of this idea by ordering the moves based on the importance of the move. It prioritises capturing high value pieces with low value pieces and promoting pawns, and it penalizes moving pieces to a square that being attacked by an opponent's pawn. [lag]

At this point, the AI still misjudges situations. To remedy this, a capture search is initiated when the depth of the minimax search reaches zero, instead of just evaluating. The capture search is very similar to the minimax search except that it is, well, only searching captures. [lag]

The final nail in the coffin for many opponents of this AI is the database of starting sequences that it has in its back pocket. For the first five moves, the AI picks a random move from one of the 8,000 grand master games that are included in the database and plays it. This helps to spice up the early game and leads to less wandering about at the beginning. [lag]

It is needless to say, but the combination of all these features leads to this AI being a formidable opponent. Because of the ease of playing against this AI, there is 10 games worth of data in the below tables.

The Backup: Man vs Machine				
Game	White Moves	White Captures	Black Moves	Black Captures
1 - B	20	4	20	7
2 - B	27	6	27	14
3 - B	18	3	18	7
4 - B	17	3	17	6
5 - B	25	5	25	11
6 - B	28	4	28	13

The Backup: Machine vs Machine				
Game	White Moves	White Captures	Black Moves	Black Captures
1 - W	79	15	70	11
2 - W	69	15	68	12
3 - W	56	13	58	12
4 - W	123	15	122	13

Clicking on the Game Number will bring you to a lichess.org representation of the game

Analyzing the first graph, we can see the dramatic shift in the games. Every game is now being won by the AI, granted, none of the people testing it are *really* good at chess. The total moves are consistently lower than the first two iterations and the captured pieces basically flipped. Additionally, on average, games last fewer moves, showing that the AI is able to quickly get the human into checkmate. The additional move checking and evaluation of this AI are clearly having an impact on the performance. However, the results would not nearly be the same if only one of any of the elements were present. Everything in combination causes this AI to be a formidable opponent.

The second graph depicts four games of AI vs AI, with each game being won by the AI moving the white pieces. This is somewhat expected, considering white gets to move first. Because the two AIs are coded the exact same, it would make sense that the one with the turn advantage is the most likely to win. One can also tell that they are evenly matched by looking at both the captures and the number of moves. The games last significantly longer than any of the previous iterations and both sides capture many more pieces. Many of the games are being decided in the endgame because the AIs are so evenly matched. The matches that are shorter are likely a result of the opening on one side or the other being better or worse.

The third AI iteration is a formidable opponent, no doubt, however, can we get better? The answer is *yes.*

Checkmate

The fourth, and final, AI iteration that I tested was Leela Chess Zero. Leela Chess Zero (hereafter referred to as Lc0) is an open source, neural network based chess engine. If the name sounds familiar, then good on you for remembering things from earlier on in the report! Lc0 is build very similar to the private, non-open-source, AI, AlphaZero. Lc0 borrows much of what makes AlphaZero so good and improves upon it even further. Currently, on lichess.org, it has a ELO of 2308 in Classic, 2538 in Rapid, and 2465 in Blitz. What makes Lc0 so good?

The King

The documentation on the Lc0 website is not very difficult to follow, moreso to find in the first place, but after drudging through it, Lc0 is a marvel. Lc0 consists of two main parts (that you can download if you fancy): the binary and the net. The binary is what actually communicates with the pieces and the boards. The net is what makes the deciding of which move to play. The binary communicates with the net in order to create a well oiled machine. "[The binary] is a robot body with eyes to see the pieces and hands to move them, and [the net] is the brain of the robot that chooses the best move." [lc0] I do not know enough about neural networks to confidently say what Lc0's net does, however, the main neural networks used for play, the T Nets, are strong neural networks that are created entirely from playing against themselves for some number of games, then learning. [lc0] Additionally, Lc0 makes use of a Monte Carlo Tree Search, just like AlphaZero, in order to train those neural networks. Lc0 has numerous bots up on lichess.org, one of which was used for this project, that further train the networks by playing against users.

The strength of Lc0 is incredibly evident. It can beat the best neural networks out there such as Stockfish, Rybka, and Komodo. But can it beat a couple college students? Unfortunately, we won't find out. lichess.org requires you to sign up for an account in order to play, thus, nobody seemed to want to test it. Additionally, having only one person, myself, play against an AI would not give sufficient data. A conundrum arises. Luckily, one is able to download game data over a time period from the bot's lichess.org page and load it. In order to gather relevant and reliable data, I did just that. I took all the games played from March 1st, 2022 to May 2nd, 2022, got rid of the early finishing games, and put the relevant data in the graph below. Unfortunately, a lot of games finished too early to be relevant.

The Backup: Man vs Machine				
Game	White Moves	White Captures	Black Moves	Black Captures
1 - W	17	4	16	4
2 - B	20	5	20	6
3 - B	61	12	61	13
4 - B	27	8	27	9
5 - B	62	9	61	9
6 - W	28	9	27	8
7 - B	29	9	29	9

Clicking on the Game Number will bring you to a lichess.org representation of the game

This table represents 7 games played by Lc0. Nearly every game was forfeit by the human player, however, that is how almost all of the games had gone that I could find, so I took the ones that were either complete, or nearly complete, and included them. Every game was won by Lc0, playing as the piece color that won. In nearly almost every case, the game was very even. Additionally, most games were relatively short, similar to the third iteration. This indicates that there is a much wider skill range of players present on lichess.org, however, Lc0 still wins against them every time. It is good at either forcing checkmate or getting the player to give up without taking a majority of the pieces. This shows that its skills at trapping the king in unwinnable positions are very good.

Lc0, however, is not infallible. It is good, but other AIs such as Stockfish and AllieStein–which is a combination of two spinoffs of Lc0–edge it out in some tournaments. [wik] Regardless, it is clear that it is top of its game in regards to the chess AI scene. Comparing Lc0 to the rest of the AI iterations presented, it is clear that it is **The King**.

Wrapping Up

At this beginning of this journey, we set out to answer a question and solve a problem: how do we define intelligence, and how do we make a chess AI? Chess is a complicated game, and the field of chess AI creation is even more complicated. We discussed the history of chess and AI, summarized the most important concepts and algorithms used, and experienced some historic moments. Over the four iterations of chess AI discussed, we covered evaluation functions, minimax algorithms, alpha-beta pruning, neural networks, MCTS, general randomness, and much more.

The conclusions we can draw from these tests are, I think, obvious. There is no one-shoe-fits-all solution to chess AIs. If you just use MCTS or just use randomness or just use minimax, you will be in for a bad time. Even if you combine two, such as minimax and randomness, there is no guarantee that your AI will be what we would call "intelligent". The AI becomes "intelligent" when it makes use of all its resources. Tree searching, evaluation, and randomness of MCTS and the learning of the neural networks combine to create a chess playing machine that can beat even the best humans. But is that intelligence?

Next Time on *Chess and You:*

Just like a neural network, humans are capable of learning from their mistakes...or is it the other way around? Regardless, there is much that I could have done better on this project, from not using turtle, to starting earlier, to maybe making a concrete plan for once. My code may be a mess, but I learned from it. I learned, not just about coding chess, but about what it takes to make an AI. I learned the strategies that one can use to make their own intelligence.

Just like me, chess AI continues to evolve. Neural Networks are a relatively new concept to chess AI and we continue to see improvements upon the design and methods used. Programs like Lc0, Stockfish, Komodo, and numerous others are all pioneers in their field. Its hard to predict the future of chess and AI in general–if I could, I doubt I wouldn't be in so much debt–but I *can* see my own future, and I see myself redoing my code completely to make an AI that actually works.

Until that time, though, the evolution of AI through the last century has proven that we do not know, nor have we ever known, what intelligence is. Even still to this day we have to keep redefining what it means to be intelligent because our own creations keep breaking those definitions. "Intelligent is as intelligent does." We have figured out what intelligent does. Intelligent makes chess, then intelligent makes more intelligence to play chess. We are intelligent and so are our creations. Who is more intelligent is of no concern, nor is the definition of intelligence, rather, the question is such: what can intelligent do with intelligence?

Extra Remarks

If you would like to see the code used for the first two AI iterations, it is located in my personal github. It is a mess and not commented...enjoy! *:)*
Additionally, I had a really fun time in this class! If a TA is reading this, thank you for your hard work! and if Professor Gini is reading this, thank you for the hard work too! This was one of my favorite classes, and I hope that I can continue to pursue AI related topics!

References

[Ash61] W. Ross Ashby. What is an intelligent machine? In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), page 275–280, New York, NY, USA, 1961. Association for Computing Machinery.

[BADVRB58] A. Bernstein, T. Arbuckle, M. De V. Roberts, and M. A. Belsky. A chess playing program for the ibm 704. In *Proceedings of the May 6-8, 1958, Western Joint Computer Conference: Contrasts in Computers*, IRE-ACM-AIEE '58 (Western), page 157–159, New York, NY, USA, 1958. Association for Computing Machinery.

[BPW+12] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[CM83] Murray S. Campbell and T.A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[Die03] Thomas G. Dietterich. *Machine Learning*, page 1056–1059. John Wiley and Sons Ltd., GBR, 2003.

[dP09] Mathys C. du Plessis. A hybrid neural network and minimax algorithm for zero-sum games. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '09, page 54–59, New York, NY, USA, 2009. Association for Computing Machinery.

[Eva07] John M. Evans. Definitions and measures of intelligence in deep blue and the army xuv. In *Proceedings of the 2007 Workshop on Performance Metrics for Intelligent Systems*, PerMIS '07, page 148–151, New York, NY, USA, 2007. Association for Computing Machinery.

[Fin03] Nicholas V. Findler. *Heuristic*, page 781–784. John Wiley and Sons Ltd., GBR, 2003.

[Fu18] Michael C. Fu. Monte carlo tree search: A tutorial. In *Proceedings of the 2018 Winter Simulation Conference*, WSC '18, page 222–236. IEEE Press, 2018.

[GEC67] Richard D. Greenblatt, Donald E. Eastlake, and Stephen D. Crocker. The greenblatt chess program. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), page 801–810, New York, NY, USA, 1967. Association for Computing Machinery.

[Hya77] Robert M. Hyatt. Blitz v, a computer chess program. In *Proceedings of the 15th Annual Southeast Regional Conference*, ACM-SE 15, page 328–342, New York, NY, USA, 1977. Association for Computing Machinery.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[lag] Coding adventure: Chess ai. https://youtu.be/U4ogK0MIzqk. Accessed: 2022-05-02.

[lc0] What is lc0? (for non programmers). https://lczero.org/dev/wiki/what-is-lc0-for-non-programmers/. Accessed: 2022-05-02.

[Li20] Hongyue Li. The application of machine learning in chess endgames prediction. In *Proceedings of the 2020 2nd International Conference on Big Data and Artificial Intelligence*, ISBDAI '20, page 9–14, New York, NY, USA, 2020. Association for Computing Machinery.

[Mar91] T.A. Marsland. Computer Chess and Search. In *Encyclopedia Of Artificial Intelligence*, pages 1–29, 1991.

[MAS15] Sajjad Mozaffari, Bardia Azizian, and Mohammad Hadi Shadmehr. Highly efficient alpha-beta pruning minimax based loop trax solver on fpga. In *2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*, pages 1–4, 2015.

[New00] Monty Newborn. Deep blue's contribution to ai. *Ann. Math. Artif. Intell.*, 28:27–30, 10 2000.

[New03] Monty Newborn. *Computer Chess*, page 336–339. John Wiley and Sons Ltd., GBR, 2003.

[Ran82] Brian Randell. From analytical engine to electronic digital computer: The contributions of ludgate, torres, and bush. *Annals of the History of Computing*, 4(4):327–341, 1982.

[Rob82] H. Vigneron: Robots. English Translation in: David Lecy, Monty Newborn. *Chess and Computers*, pages 13–23. Computer Science Press, 1982.

[Sch99] Simon Schaffer. *Enlightened Automata*, pages 126–165. The University of Chicago Press, 1999.

[Sha88] Claude E. Shannon. *Programming a Computer for Playing Chess*, pages 2–13. Springer New York, New York, NY, 1988.

[TPHK22] Nenad Tomašev, Ulrich Paquet, Demis Hassabis, and Vladimir Kramnik. Reimagining chess with alphazero. *Commun. ACM*, 65(2):60–66, jan 2022.

[wik] Simplified evaluation function. https://www.chessprogramming.org/Simplified_Evaluation_Function. Accessed: 2022-05-02.