Brandon Lewis (bcl62)

Learning to Play Chess with Tree Search

Introduction:

Chess is the ultimate board game, a battle of strategy and mental computation between two opposing competitors. As such, the archetypal game of wit should be an adequate challenge for computer learning and artificial intelligence. The history of chess and computation is long and quite colorful. The modern incarnation of chess originated in around the year 1200 CE, but the first chess computers arose more than half a millennium later. Between the years of 1770 and 1854, a supposed chess automaton known as the Mechanical Turk was making waves in Europe, defeating human opponents. However, this machine was actually a fake, and a human hidden inside the automaton's box was secretly playing the game with a set of controls. While this computer was fraudulent, the first true chess computer arose a century later in 1957, when an IBM engineer ran a chess program that calculated a 4-ply search in around 8 minutes. Only 20 years later IBM's own Deep Blue managed to defeat the world champion Korolev, and in even less time after that the top chess engines were unbeatable. Because of this strong connection between computing and chess, I chose to focus my artificial intelligence project on learning to play chess.

My goal in this project is to construct a chess engine that is capable of playing the game to completion. Beyond this, I would like it to play well and be competitive with untrained humans and relatively weak chess engines. This first challenge involves constructing the game from scratch by keeping track of the board and resources in code. Using this information, the engine must generate a set of moves that are in line with the placement pattern of the pieces on the board. This type of move is known as pseudo-legal; to become fully legal, a move must not leave the player's king in check. Additionally, there are special moves such as castling, en-passant, and promotion which do not follow the traditional move model. After each move, the game can conclude due to several end conditions. If the player has no legal moves and is in check, then they lose. If they are not in check, then the game is a stalemate. Additionally, if there is no progress in 50 moves or the same position appears 3 times, the game is a draw. Otherwise, the game will continue as a back and forth between opposing sides. This type of back and forth move selection is ideal for a tree representation, where nodes are board states and moves are the edges between them. The chess engine will navigate this tree to reach a favorable terminal state.

Brandon Lewis (bcl62)

In order to create a passably good chess engine, I will use multiple techniques for tree search and evaluate them by playing them against each other and off-the-shelf reference engines. The first approach will be a traditional minimax tree search. This setup is relatively simple to understand, as it will alternate between playing the maximum scoring and minimum scoring moves at each turn, since it assumes the opposing player will always play what is worst for the engine. At the leaf node, it will use a handcrafted evaluation function to quantify the board state numerically. A simple evaluation function can lead to complex behavior when combined with a deep search tree and domain specific optimizations based on chess knowledge. Humans play the game in a similar way by looking at the likely sequences of moves and determining how much they are a winning or losing position, so this technique is an intuitive approach to playing the game. The second approach will be a Monte Carlo tree search. This technique uses a high number of simulations to gauge which moves lead to a positive outcome the highest percent of the time. Since it plays the game out randomly, it does not use an evaluation function to quantify the board, instead using win-loss ratios. This technique uses no domain specific knowledge and follows the paradigm of learning by doing. Ultimately, I will compare how these two techniques stack up to each other and how chess specific practices improve artificial intelligence versus a general learning approach. Using this playout to gauge the algorithms' success, I will make iterative changes to each engine to improve the win-rate in order to make the programs as intelligent and successful as possible.

Prior work:

Tying back into the long relationship between chess and computing, an incredible amount of progression has occurred in the field of playing chess algorithmically. Nowadays, top tier chess engines will never lose to other extremely good chess engines who are only slightly worse. The best humans on the planet are incapable of defeating an engine in fair games. Even with a substantial piece advantage, the chess engine will win 100 times out of 100 because it can see dozens of turns in the future and evaluate the board on a level far beyond human intuition. In this highly advanced realm of computing there are two primary paradigms in the modern era of chess computing: minimax with a handcrafted evaluation and Monte Carlo tree search with a neural network for evaluation. The first uses a significant amount of pre-programmed chess knowledge and tactics, while the second relies on playing itself to learn how to approach the game intelligently.

Stockfish is the ultimate example of the minimax tree search. It is the most well-known and popularized chess algorithm due to its extremely high performance and open-source nature. It augments its tree search with numerous optimizations to maximize the number of positions it can evaluate. Firstly, it uses alpha-beta pruning to get an easy lossless increase in performance. Beyond this obvious pruning, it aggressively uses forward pruning to eliminate positions and moves which are unlikely to appear in normal play. While it might overlook possible positions, the significant increase in performance is more beneficial. Next, it uses iterative deepening with a variable depth to find the best terminal positions at a variety of search depths. Certain promising lines are followed to an extreme depth, calculating more than 80 turns in the future. For evaluation, Stockfish originally used an extremely complicated linear evaluation heuristic that varied the weights of different traits over the course of the game to play well in the opening, midgame, and endgame. Following the rise of neural network based chess engines, Stockfish implemented a form of neural network known as a NNUE (efficiently updateable neural network) for evaluation. The network is trained to predict the original Stockfish evaluation function at a moderate depth; this essentially increases the depth of the search even more by adding the pretrained search evaluation on to the minimax search. Overall, Stockfish uses these chess and computing tricks to evaluate lots of nodes to a high depth very quickly.

AlphaZero represents the other paradigm, the Monte Carlo tree search. Despite only playing less than a handful of public games, Google's AlphaZero became notorious in the field of chess computation, as it was the first game engine to learn the game from scratch and rapidly become a top-level engine competitor. Central to its algorithm is the Monte Carlo tree search. Instead of using random playout, it uses a probabilistic policy network that outputs the probability of playing each move from a certain position. In addition to this neural network, it uses a value network that evaluates board positions. Both networks are trained by self-play through reinforcement learning. It uses these networks to determine the random playout and in a complex formula for choosing the root child to play. While the specifics of AlphaZero's code is kept hidden, an open-source program named Leela Chess Zero uses the same tactics and has improved upon AlphaZero's level of play to continue remaining competitive at the top level of chess engine play.

These two engines and their successors have battled for the position of best chess program. Since 2008, Stockfish has consistently been the dominant engine in the space. It has the best domain-specific optimizations and calculates positions faster than every other engine, so other engines struggle to compete against it. In 2017, AlphaZero challenged Stockfish for the throne, as it managed to

consistently defeat the former champion program after only a few days of self-play. Despite assessing an order of magnitude less nodes than Stockfish, AlphaZero was still able to defeat it due to its neural networks. Due to its open-source nature, Stockfish quickly adapted as its original linear evaluation function was replaced by a neural network to augment its search depth. This allowed Stockfish to continue its reign of dominance as Google never publicized any more games from AlphaZero. The open-source successors to AlphaZero have not had the same level of success versus Stockfish as their predecessor, as the minimax model of computing remains on top.

Methods:

The first step in creating a chess engine is creating the game representation. I created multiple layers of objects to manage the different steps of the game. In the outermost layer is the game manager, which keeps track of the score and handles calling on the 2 player agents to make their moves. It directly controls the board user interface, which updates itself on command from the game manager. Beneath this level is the board representation. The board is represented in multiple ways for efficiency; there is a 2D array of pieces and multiple hashsets for each type of piece for each side of the game. Hence, I can find the piece in a given location or location of a given piece in constant time. Additionally, the board can make moves or unmake moves to change its state. Moves are represented as a starting location, end location, captured piece, and a flag for special moves. This data structure is used by the move generator, which creates a list of legal moves when given the board representation. Constructing the move list is complicated and requires numerous checks for check, double-check, pins, discovered check, and other instances. At the lowest level is the piece representation. Pieces are stored as integers where each bit signifies a certain type of piece with the leading two bits representing color. Hence bit operations and bitmaps can be used to acquire the data stored in each piece integer. These layers coalesce and interact with each other in order to create a complete representation of the game of chess.

The player interface interacts with these data structures to pick a move from the list of legal moves. Each player is given a notification from the game manager when it is their turn to begin computing. Then the player computes their choice until they find a move. After the move is found, a flag is tripped, and the game manager plays the move. There are four implementations of the player class: human, minimax, Monte Carlo, and Stockfish. While minimax and Monte Carlo are the primary focus of

this project, the other two are used for debugging and reference. The human player must interact with the interface in a simple drag-and-drop method for playing moves. When a piece is clicked, it cross references the list of legal moves and highlights all possible end locations. Furthermore, in debug mode, more information will be highlighted in the user interface. On the other hand, the Stockfish player runs an executable and passes the game state into it. This process is polled to find the best move to play each turn, and the player parses its output to play. Since Stockfish is a top-level engine, I implemented this parser into my project so that I could use it to compare the effectiveness of my engines against a strong enemy.

The minimax search algorithm uses a significant number of chess-specific and general optimizations to process a large number of positions and choose the move which leads to the best evaluated outcome. The evaluation function is linear and judges positions primarily using the pieces on the board. Each type of piece has a value assigned to it based on its usefulness, and the evaluation function sums up the value of each piece for each side. Additionally, pieces have value based on their location on the board using precomputed position value tables. The evaluator adds the location points to the sum for each side if the pieces are in the locations. Finally, it subtracts the sum for the enemy side from the player's sum to get the final evaluation. The score is centered on 0 with positive scores being good and negative scores being bad with a range in the thousands in each direction. This function is simple and easy to quickly calculate, but it is still the bottleneck for performance in the early and midgame.

To minimize the number of necessary evaluations, the tree search uses alpha-beta pruning to get an easy performance increase. Alpha-beta pruning gives the same result just in a shorter time span. Furthermore, it adjusts the values for alpha and beta based on the assumption that the best possible move is mating the opponent in 1 step from the root and the worst is being mated in 1 turn. Another way to use alpha-beta pruning more effectively is through move ordering. The tree nodes which are high value are traversed first so that there will be more pruning early on. The value heuristic for moves is simple and scores based on the piece captured and whether the move is a promotion. All other moves are assigned equal value. Ideally it would also deprioritize moves that will result in being captured, but this caused problems during testing, so the functionality was removed. The combination of these techniques allows for significant pruning of the search space at very little or no downside to the gameplay.

Another method for optimizing the search is the use of a transposition table. Around 10% of the nodes reached in the tree have been seen by previous searches, so it is optimal to reuse the previous score instead of recalculating it. This should occur if the remaining depth of the previous search is greater than or equal to that of the current position, so that only information is gained. This transposition table is a hashmap containing a hash of the board position and the depth as an integer. The hashing works through a technique known as Zobrist hashing, which generates a set of randomized long integers for each possible piece in each possible position and for every other state value. These longs are XORed together to give the hash, and as long as the numbers are large enough it is virtually impossible for different positions to give the same hash value. This hashing is also used to check for a draw by repetition.

Since this tree search is expected to run in real time, it is imperative that it is able to stop mid-search. To do this, I created an implementation of iterative deepening for the search. This allows it to stop early and check for terminal nodes at a range of depths in the tree space. The depth of the search varies with a minimum of 4 and maximum of 50. It halts after it has checked either 100,000 evaluations or 1,000,000 nodes total. Evaluations are the limiting factor in the early and midgame, while the total node count limits the endgame where evaluations become sparse. These numbers were set as high as possible while maintaining a reasonable compute time. On average, the program checks to a depth of 5, but in the endgame, it will check to a depth of between 10 and 20 depending on how close to checkmate it is since there are less evaluations of leaf nodes in the endgame due to the lesser number of pieces.

The final and most important chess-specific search technique is the quiescence search, which is meant to address the horizon effect. In a normal search, a high-value capture at the leaf node might be considered very good, but the unseen next move might result in that piece being captured in a much less valued trade. This occurs very often and leads to blunders and bad piece trades where important pieces like the queen are sacrificed for less important pieces. To address this, leaf nodes are not actually scored using the evaluation function. Instead, a secondary search that only looks at captures begins from the leaf node. It continues indefinitely until it reaches a quiet position where no more captures are possible. These quiet positions are evaluated and propagate into the regular search as the scores for positions. The quiescence search resulted in the most improvement to the minimax search of any added technique.

On the other hand, my Monte Carlo tree search implementation uses very little actual chess information in its search process. This method simulates 100 different games traversing the tree to get a probability of success for the child nodes of the root. To achieve this, I created a tree node structure where nodes are associated with a board state and the move that led to that state. Each node has statistics for wins, losses, and game count, and the children for each node can be manually added or generated for traversal. These nodes can also gauge game result when requested. The Monte Carlo tree search picks the best of these nodes based on its win/loss ratio until it reaches a leaf node. In addition to the game count, an exploration weight is used to select which node to pick. The exploration weight favors trying different nodes early on in the first few simulations and disfavors commonly tried nodes. As the number of simulations increases, it will decrease in value so that the most winning nodes are chosen. This method allows for a wider range of valid choices to be selected instead of zeroing in on one specific line of moves from early on.

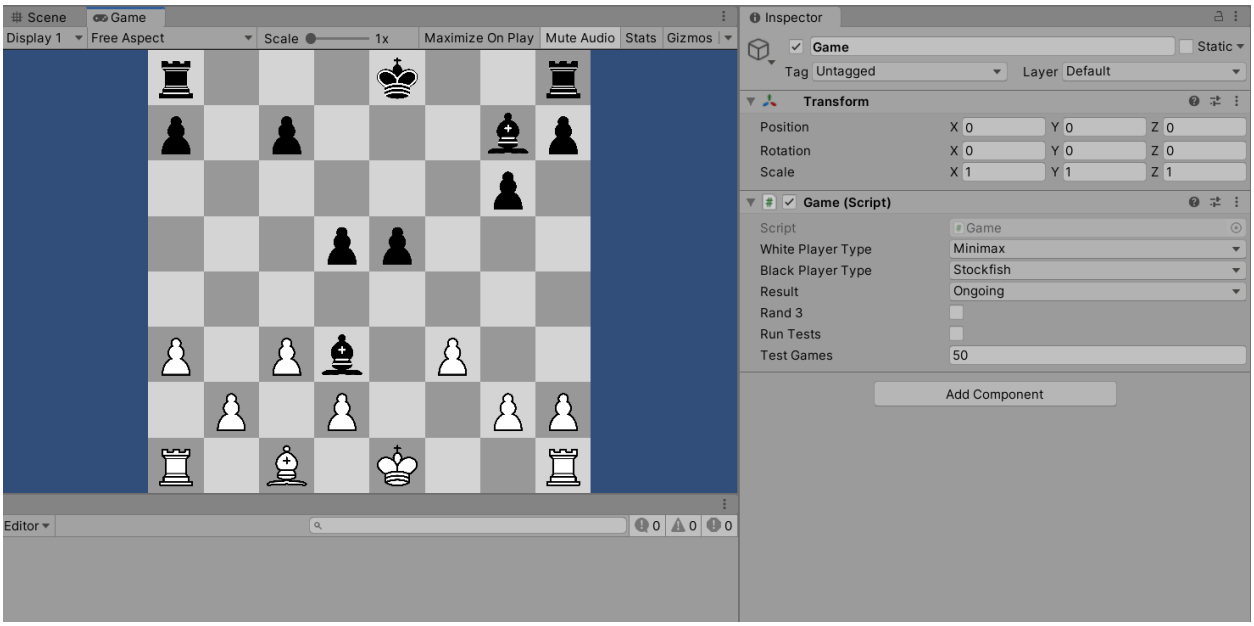
Once the search reaches a terminal node, the rollout stage begins. A random child is selected, played, and added to the main tree structure. Then at each step, another random child node is picked and played. If it reaches a terminal state, the outcome is evaluated based on traditional chess scoring where wins are 1 point, losses are 0, and draws are 0.5. This is the only piece of chess knowledge used in the search, as it is necessary to somehow evaluate game results. If the rollout exceeds 50 moves it will automatically end as a draw in order to improve performance and not proceed on a several hundred turn game. Once a game result is found, the result propagates up the main tree updating the score statistics for every node until it reaches the root node, where it stops. Only the first node in the rollout is added to the main tree structure and given a score, so that each simulation only adds one node to the tree.

After the simulation completes, the search chooses the best move based on the max-robust child criteria, which picks the most played child node of the root. This choice is the most stable selection criteria since good nodes will be played the most. The nodes it chooses have the most consistent results as opposed to relatively unplayed nodes which might have a good score but a low game count. These nodes are outliers and not representative of the potential of the board position following the move. After the move is selected, it is passed up to the game manager to be played, and the tree structure is discarded.

Brandon Lewis (bcl62)

Results:

I created my chess representation and engine in the Unity game development engine, which uses the language C#. I choose this environment because I was familiar with it, and it easily allows for the interactive setup that is useful for games. The GUI uses a 2D board representation where untextured quads are used for the tiles and sprites represent the pieces. In the Unity debug panel, there are enumerator dropdowns for the white player and the black player, so I can switch between the type for each side among the 4 possible player types. Additionally, I can pass in a starting board position in FEN notation. If the board is left unspecified, it will use the traditional starting position as default. Upon game start, the game will play out in real time with the engines taking around 1 second for computation each turn. It runs on a single thread, so it does not allow for premoves or precomputation for simplicity. At the end of each move, it checks the game result and updates the result enumerator. If the game reaches a terminal state, it passes the result to the console. Additionally, diagnostics can be printed for each engine move in the console.



The four players are completed and interact with the game well. The human player's drag and drop functionality moves the sprites in the GUI, and the move visualizer highlights legal moves and diagnostic information if its in debug mode. If two human players are playing, moves can be undone in real time for test purposes. The minimax algorithm takes less than 1 second to play in the early game, but as time goes on it uses more compute time. On the other hand, Monte Carlo tree search uses a fixed

number of simulations, so its compute time is consistent throughout. The Stockfish implementation uses a fixed time constraint as well.

Beyond just the game interface, I also created a test interface. The first test setup is used to ensure the validity of move generation. If a test box is ticked, it will calculate the number of moves at each depth from the starting board position. This mirrors the PERFT (performance test) functionality in other engines, which keeps track of various statistics as it traverses the list of moves. Additionally, there is a divide function which also prints the split of move count given the starting move. This performance test has customizable depth.

Additionally, there is a test interface for comparing players. If player testing is enabled, it will randomize the first 3 moves for each side and then play a pre-defined number of games to completion. After these games complete it prints the win, loss, and draw counts for the tests. This allows me to gauge how effective each player is in comparison to itself and the other players. Ultimately, I found that the Stockfish algorithm is vastly better than the minimax search which is vastly better than the Monte Carlo tree search.

	Wins	Losses	Draws
Minimax vs Monte Carlo	100	0	0
Stockfish vs Minimax	97	0	3
Stockfish vs Monte Carlo	100	0	0

Discussion:

The most surprising result from the self-play tests is the good performance of the minimax search. After watching it play, it was obvious to me that complex behavior emerged from relatively simple guiding rules. It was capable of forcing trades and asserting pressure on individual pieces and areas of the board. Any poor decision making was masked by thinking at least 4 moves ahead. Most impressively, it was able to last consistently more than 60 turns against Stockfish and even managed to bring it to a draw by repetition 3 times.

Despite this good performance, its weaknesses of this approach were still obvious to me. The search depth was too uniform, as it only changed based on how far into the game it was. This caused the problem of bad chess lines being given similar compute time to good lines. Ideally the better a line is, the further depth it would be taken to. Furthermore, the simplistic evaluation function started to perform poorly in the endgame. Tactics should change based on whether it is playing from material

advantage or to force a checkmate, but it focuses on material advantage all the way to the end. Due to this, it struggles in several endgames that should be easy to play. This can be addressed by using a sliding scale between midgame evaluation and a different endgame evaluation that considers different variables. Finally, the poor optimization of the board representation limited the depth of the search. It was able to calculate around 100,000 positions in a second, while better engines can calculate millions of positions in the same timeframe. To address this, the board representation should be rewritten using bit operations and the 2D array should be replaced with a 1D array. However, these weaknesses are not glaring. The engine plays to an adequate level given the time dedicated to it, and the room for improvement is obvious. As of right now, I would estimate it to have around 1500 engine ELO, and it picks the best move according to Stockfish the majority of the time.

On the contrary, Monte Carlo search tree performed very poorly. It was the overall worst engine and lost 100% of the time to the other engines. Analyzing the games played, it picked very bad moves most of the time. This poor move choice is due to the lack of computational power here. The randomness of the rollouts meant that its evaluation wasn't representative of the true sample space. 100 games is not enough to accurately gauge a position, so it needs some sort of extra evaluation function to improve. A neural network like AlphaZero's would be very beneficial to prune the sample space. Additionally, the tree node structure is poorly optimized, as it checks less nodes than minimax but takes more compute time. Generating child moves is slow, so each step in the rollout takes longer than each step in minimax. However, even if it was optimized it would not make a significant difference, as 10,000 simulations might not be enough to judge a game like chess where there are millions of possible moves at higher depths. To improve the Monte Carlo tree search, the techniques from AlphaZero should be implemented and integrated into the framework, which is still functional and effective despite its poor performance.

Conclusion:

Ultimately, I found that minimax search will outperform Monte Carlo tree search in high complexity games like chess. The search space is far too wide for Monte Carlo to run enough simulations to truly learn the game. Any attempt at a learned probabilistic evaluation will fail to represent the true move valuation, as it is far too affected by low sample size. A simplistic hard-coded evaluation performs better with equivalent search time. As such, Monte Carlo tree search needs some method for guiding its

rollout stage instead of using a random decision. A neural network that outputs the probabilities of different moves being played fits this role perfectly, as it allows the search to prune large portions of the sample space that are unlikely to be played. Then the lack of calculations will matter less due to the high-quality learned evaluation provided by a neural network which exceeds the effectiveness of hardcoded evaluation functions.

Because of this difference, the minimax search will have a huge head start on learning the game thanks to its domain specific knowledge. Pre-programmed information allows it to skip most of the learning step by outsourcing it to the human author. Reinforcement learning techniques will lose until they reach an inflection point where they can effectively compute the search space of the game and outperform the traditional methods. This mirrors the battle between Stockfish and AlphaZero, but in my project the Monte Carlo tree search implementation has not reached the breakpoint where it becomes competitive as it is too simplistic. Despite this challenge, it still has room to improve and eventually beat minimax by following the techniques shown with AlphaZero.

However, even with these techniques it will not completely outclass minimax tree search. As more domain specific knowledge is added to optimize the minimax search, it will remain competitive thanks to improved evaluation and pruning. The path of improvement for Monte Carlo is vaguer, as it cannot just stack more and more self-play to reach higher levels. Reinforcement learning tends to suffer from diminishing returns, and learning to play versus yourself is different from playing others. For example, these algorithms perform worse if they are playing worse opponents since they assume the opponent will pick optimal moves. As such, the minimax tree search will remain the dominant technique in the field of chess engines until some new project like AlphaZero combines lesser used and novel techniques to achieve higher levels of intelligent gameplay.

Works Cited:

Maharaj, Shiva, Nick Polson, and Alex Turk. "Chess AI: competing paradigms for machine intelligence." *Entropy* 24.4 (2022): 550.

Silver, David, et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." *arXiv preprint arXiv:1712.01815* (2017).

Plaat, Aske, et al. "A new paradigm for minimax search." *arXiv preprint arXiv:1404.1515* (2014)

Chou, Cheng-Wei, et al. "Strategic choices: Small budgets and simple regret." *2012 Conference on Technologies and Applications of Artificial Intelligence*. IEEE, 2012.

Luštrek, Mitja, Ivan Bratko, and Matjaz Gams. "Why minimax works: An alternative explanation." *Proceedings of IJCAI*. 2005.