Minimax Chess AI

Blaise Page and Phillip (Armen) Arsenian
CSCI 3202: Introduction to Artificial Intelligence
Professor: Christoffer Heckman
December 13, 2018

Research

Programming a machine to play chess has always been a huge challenge in the world of AI. For the longest time, a thoroughly trained chess AI struggled to beat a chess grandmaster even with all the computing power that computers were boasting at the time. It wasn't until a few year after the 21st century that computers were able to take the lead in beating the greatest chess players on a consistent level. This is because the number of possible board moves in a chess game is incredibly huge, which according to Claude Shannon can be given a lower bound of 10^{120}.

The first chess AIs began to be developed in the 1950s when researches were studying the capabilities of artificial intelligence as a whole. In 1958, the duo of Allen Newell and Herbert Simon developed the first minimax agent for playing chess. Their implementation included alpha-beta pruning as well as strategy heuristics that are still in use (www.computerhistory.com). By using alpha-beta pruning and various heuristics, Newell and Simon were able to reduce the number of actions that the minimax agent had to search through, which greatly improved the performance of their algorithm.

Creating a good chess AI relies almost entirely on a good evaluation function. The evaluation function chooses the best move to be made based on the current state of the board. One article that we read mentions that creating an evaluation function relies on balancing between how long the program's search-depth is and how vast the program's chess knowledge is when evaluating moves. This is because an evaluation function with a great deal of chess knowledge takes longer to have a single state evaluated (Mannen, 38). Regardless, each piece is then given a value in points where a pawn will have the minimum value of 1 point and a queen will be awarded either 9 or 10 points. Assigning values to pieces makes it easier to attain a score for a minimax tree to choose the best move that returns the maximum score. In order to avoid the issue of a minimax tree becoming overly complex, different techniques are applied to reduce the tree itself. These techniques include Alpha-Beta pruning, move ordering, and transposition tables.

In order to win a chess match, both strategic patience and cognitive planning are necessary. However, often times one move can be so disastrous that it can quickly cost someone the game regardless of all their strategic planning. Quiescence searching is one technique that could help prevent this from occurring. This technique is used mostly for chess AI systems that are not capable of creating minimax trees of considerable depth. Quiescence Searching requires that a move be chosen that does not have a great effect on the current position of the board, or in other words a move that has a very positive or very negative score. For example, suppose the leaf of the AI's minimax tree evaluates a move that could capture the opponent's knight. This move on the surface seems very rewarding, however because the minimax tree has met its full depth, the

chess AI is unaware of potential consequences that can occur (such as the loss of its Queen in the next move). For this reason, the quiescence search technique would ignore this move because it understands that potential, undetected consequences may not be worth the reward (www.cs.cornell.edu).

Another suggestion that has been published, is using a combination of a Monte-Carlo tree search and a minimax implementation to create an AI in games such as chess. The minimax tree would be created in a shallow fashion typically reaching only 2-ply searches where each ply is one move taken by one player (Baier, 2). After the minimax tree is formed, the Monte-Carlo portion of the tree would be executed where random moves are explored in the hopes that a winning move can be found. This is known as simulation. Once a winning move is found, a series of steps in back-propagation could identify the necessary moves for reaching that win (3). One reason as to why a hybrid Monte-Carlo tree search approach would be more beneficial than solely a minimax tree is that if programmed correctly, the Monte-Carlo approach could execute much quicker than the minimax approach. The minimax search is often described as a brute force approach because every potential move must be analyzed in order to make an informed decision. In the game of chess, this number becomes exponentially large adding complexity to the minimax search as shown in Figure 4. The Monte-Carlo tree search algorithm is not a brute force approach but instead uses reinforcement learning when simulating random moves to create a policy that can make informed moves (Sharma). However, because the minimax portion of this hybrid Monte-Carlo approach tree is very shallow, without examining every potential move, there is always the risk that one move can result in such consequences that it can cost the AI the match. For this reason we do not wish to use a Monte-Carlo Tree search to implement our AI.

In our search for a relatively simple chess AI implementation, we decided to develop an agent that used the minimax search algorithm to determine semi-optimal chess moves during each turn. The premise of the minimax search is as follows: There exists two different sets of nodes - maximizer nodes and minimizer nodes. At each state in the game, the maximizer nodes select the maximum scores returned by their child nodes (either terminal nodes or minimizer nodes), while the minimizer nodes select the minimum score returned by their child nodes (either terminal nodes or maximizer nodes). By assigning the turn player a root maximizer node, it will select the move that maximizes its score assuming that each minimizer node will select the move that minimizes the turn player's score. Since each move is assigned either a maximizer or a minimizer node, the minimax tree can become extremely complex. By fault, the minimax algorithm has a complexity of $O(m^d)$, where m is the number of possible moves and d is the maximum depth of the tree. Due to minimax's complexity, we decided to implement alpha-beta pruning to remove tree branches that we knew either the maximizer or minimizer nodes would not choose. Our implementation also used an evaluation function that scored each piece as follows: pawn = 1 point, knight = 3 points, bishop = 3 points, rook = 5 points, queen = 9 points, the king is infinitely valuable, king in check = 10 points, and king in checkmate, 1000.

Results

Minimax Tree Depth	Average Search Time for Single Move (hh:mm:ss)	Difficulty of Game
1	00:00:00.76	Easy
2	00:01:26	Hard
3	01:17:05	Really Hard

Figure 1

Note that in Figure 1 the *difficulty* column is subjective to our skill levels for chess, and the benchmarks in the *search time* column were calculated on a 2012 macbook pro with a 2.6 GHz Intel Core i5 processor and 8 GBs of 1600 MHz DDR3 RAM (in other words not a very powerful machine).

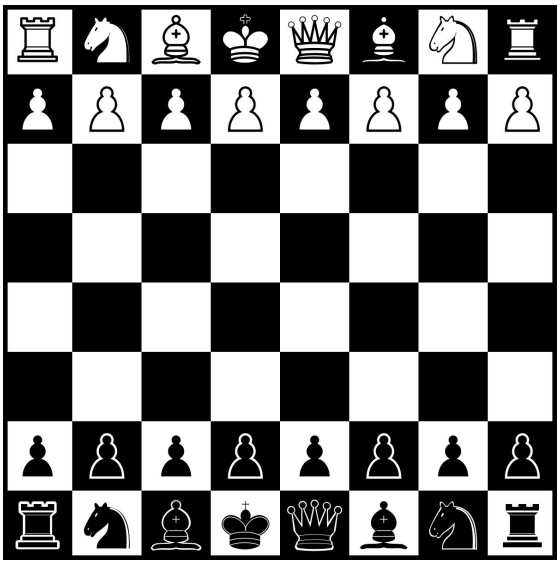

Figure 2

Figure 2 shows the initial state of a chess board at the start of the game.

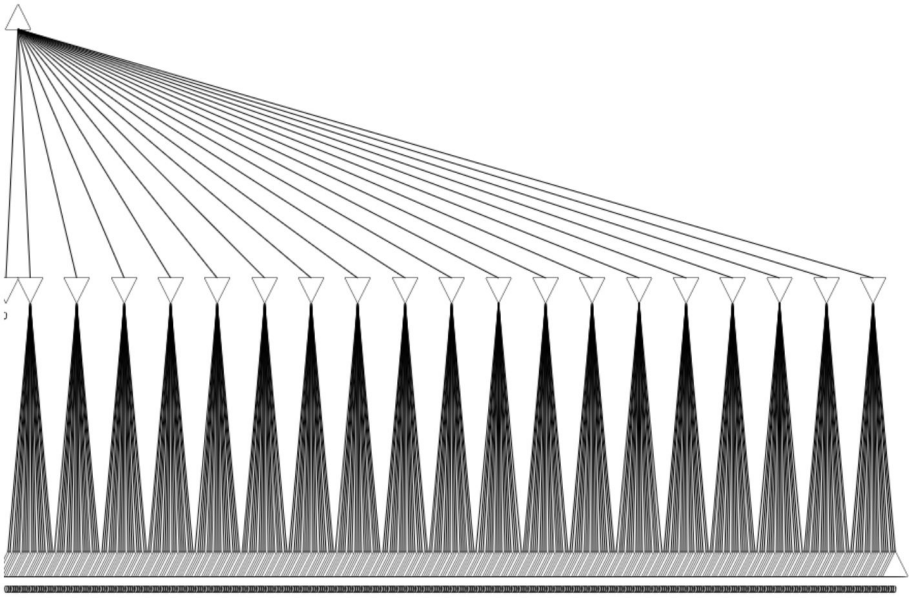

Figure 3

Figure 3 shows the minimax tree abstraction (with a depth of 2) of all the possible moves white can make at the start of the game (Figure 2). Note that white has a total of 20 options for moves to make, and for each of these moves, black also has 20 possible counter moves. If we were to increase the depth of the tree further, it would become apparently obvious that the complexity of the minimax tree would grow exponentially.

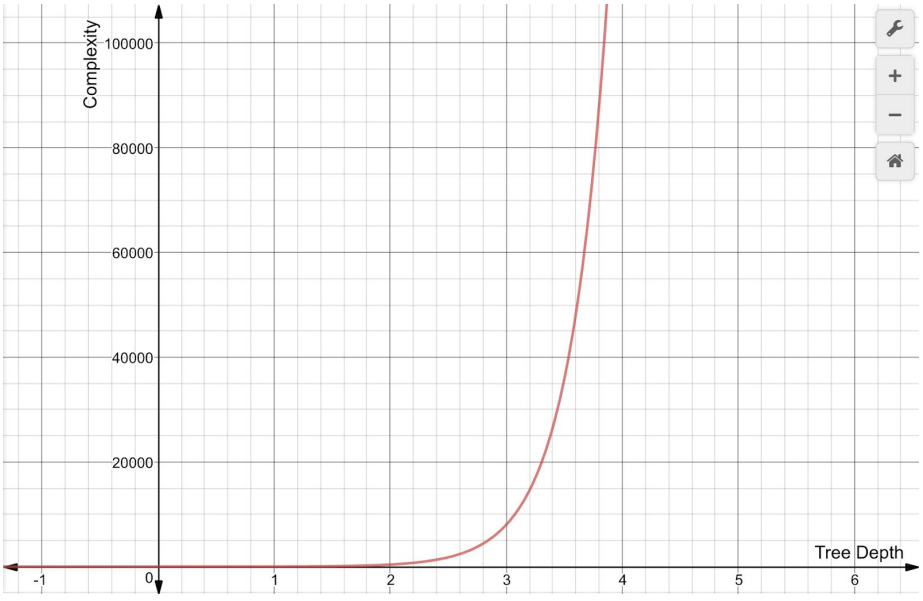

Figure 4

Figure 4 depicts the relationship between the depth of the tree and the complexity of the minimax search itself. This figure uses the the equation $y = m^x$, where y is the complexity of the minimax search, m is the number of possible moves for a given state, and x is the depth of the minimax

tree. Since the number of possible moves on a given turn is variable based on the state of the board, we have set the value to 20 in this figure. Based on this figure and the previous figures (namely Figure 1 and Figure 3), it is evident that performing a minimax search with a tree depth greater than 3 would cause the chess match to last days in computation time alone.

Discussion and Conclusion

It is evident that our minimax chess agent is far from perfect. The main downfall of our implementation is the relationship between move performance and computation time. That is, the easiest way to improve the move performance of our minimax agent is to increase the depth of the minimax tree; however, as seen in Figure 4, increasing the tree depth comes with the cost of increased computation time. For example, when we increased the tree depth from a depth of 2 to a depth of the 3, the average computation time for each move increased from 1 minute and 26 seconds to 1 hour 17 minutes and 5 seconds. Obviously this behavior is not optimal. One possible way that we could improve the performance of our algorithm is to add historical chess strategies that permitted us to select optimal moves without the need to completely search the minimax tree. The disadvantage of this strategy is that it would require us to hardcode specific strategies for each of the state spaces that they applied to, which would prove to be both time consuming and tedious.

A better tactic to avoid our performance issues, would be to scrap most of our program, and instead use a reinforcement learning agent. The benefit of reinforcement learning agents is that they do not need to construct trees for all the possible actions given a state. Instead reinforcement learning agents determine optimal moves by looking up a (state, action) pair in some data structure that already has the corresponding score in it. These scores are generated by training the agents through thousands (or millions) of iterations/simulations.

In conclusion, our implementation of the minimax agent expatiates the capabilities of such chess AIs. Although the moves generated by our agent are effective at creating a formidable adversary for players, the expensive nature of our implementation is not realistic to play against in a live setting.

Works Cited

"AI Chess Algorithms." *Object Recognition*, 2004, www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html.

Baier, Hendrik, and Mark H.M. Winands. "Monte-Carlo Tree Search and Minimax Hybrids." *Maastrichtuniversity.nl*, 2013, dke.maastrichtuniversity.nl/m.winands/documents/paper%2049.pdf.

Mannen, Henk. "LEARNING TO PLAY CHESS USING REINFORCEMENT LEARNING WITH DATABASE GAMES." *Psu*, 2003, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.810&rep=rep1&type=pdf.

"The Minimax Algorithm and Alpha-Beta Pruning." *What Was The First PC?*, www.computerhistory.org/chess/the-minimax-algorithm-and-alphabeta-pruning/.
Sharma, Sagar. "Monte Carlo Tree Search – Towards Data Science." *Towards Data Science*, Towards Data Science, 1 Aug. 2018, towardsdatascience.com/monte-carlo-tree-search-158a917a8baa.