Alpha-beta Pruning in Chess Engines

Otto Marckel
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
marck018@morris.umn.edu

ABSTRACT

Alpha-beta pruning is an adversarial search algorithm that uses tree pruning to improve the minimax search of data tree structures. This method of searching allows two opponents to each attempt to get the best result when analyzing a search tree. Some of the best examples of this approach are modern chess engines, which use Alpha-beta as the primary method to calculate their moves. Throughout this paper we discuss the steps of the Alpha-beta pruning and how it is implemented in chess engines. In particular we analyze how the 60 year old Alpha-Beta method has been refined been improved and optimized to better utilize current hardware.

Keywords

ACM proceedings, Alpha-Beta, Minimax, Chess, Artificial intelligence

1. INTRODUCTION

Since the creation of computers, people have used the game of chess as a means to test the power and ability of these machines. Each chess game has an estimated 10^{120} potential moves [11]. To put this number into perspective; if every atom of the universe calculated one possible move every nanosecond since the big bang, we would be more then 10 orders of magnitude short of looking at all possible moves in chess. Therefore, with current technology we cannot explore every possible option to find the best move, so search algorithms are used to automate and optimize the process. These algorithms and the code they use are known as chess engines.

The technique used since nearly the beginning of autonomous chess engines is the Alpha-Beta Search Algorithm [6]. Alpha-Beta allows computers to exhaustively search all potential options for moves and choose the best one. It is currently the most efficient method of doing so.

This paper contains several sections:

Section 1 is an overview of the history of chess engines and how they have been developed to make decisions.

Section 2, Implementing Alpha-Beta, describes the algorithm used to make decisions as quickly and effectively as possible, and the different parts necessary for it to work.

Section 3, Implementing Alpha-Beta in Chess Engines, discusses implementing the algorithm into chess engines and enhancements that cant be used to make it more efficient.

Section 4, Hidden Parts of Chess, examines the application of Alpha-Beta Pruning in specific stages of a chess game.

1.1 History and Today

1912 marked the creation of the first chess device, which could automatically play a partial game - a king and rook endgame against king from any position, without any human intervention. In the 1950's, Alan Turing published a complete program capable of playing chess. In 1956 John McCarthy is credited with inventing the Alpha-Beta Search Algorithm. In 1957 the first programs were widely released and in 1961 the first program that is capable of credible play is created using the Alpha-Beta Algorithm as its base [6].

In 1997 the supercomputer Deep Blue beat the world-champion, Gary Kasparov, in an official tournament setting. This was the first time that the world champion was defeated by a computer.

Modern chess software and hardware leaves Deep Blue in the dust. On an average computer, any top consumer chess engine could defeat Deep Blue today. This is due to improvements in hardware, better implementation of software, including Alpha-Beta, and its use.

1.2 How Chess Engines Think

How is it possible to make a computer that can play chess? There are several potential ways.

One approach is combining analysis, strategy and tactics to determine the next move. This mimics the way humans play in theory, this approach would produce chess player that, with constant advancement, would become the greatest player in the world. In practice the weakness of this approach is a lack of ability to program the complex combinations that humans perform intuitively.

A second option is to look through every possible move, using brute force computing power to calculate the entire game. We showed earlier that there is no way to do this with current technology [11].

A third option is to look forward a limited number of moves and evaluate positions, choosing the move that is most likely to result in a win. This option avoids the problems inherent in the previous options and is the option that is taken. This is what the Alpha-Beta algorithm is used to do.

In the following section, we look at how Alpha-Beta works and methods that can be used to both improve and speed up the searches.

UMM CSci Senior Seminar Conference, Spring 2017 Morris, MN.

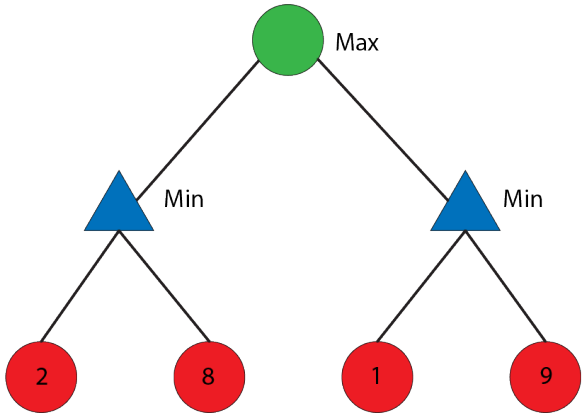

Figure 1: An simple Minimax tree

2. ALPHA-BETA

2.1 Tree Structure

We will use a search tree to look at the board and evaluate specific board states. Our search algorithm will look at this data structure and encode the potential nodes into the tree. Each connecting branch represents a legal move that can be made from one game state to another. Each node can have as many children, or branching nodes, as there are legal moves with, with the single root node represents the beginning of the game [8].

The specific type of search tree in use is the adversarial search tree. We can use a adversarial tree because chess is a two-player, zero-sum, perfect information game. Zero-sum means that any loss to one player is advantage to the other and perfect information means that nothing is hidden or left to chance [9].

A search tree has two defining parts; the branching factor and the depth. The branching factor is the number of possible options that can be taken from a node. The depth is the number of moves that will be made in a specific line.

For example the game of chess has 20 possible opening moves, or child nodes. The next player also has 20 possible moves, making 20 the branching factor. Therefore, after each player has moved once there are 400 potential game states, or nodes. The result of this for the tree is a root node that has 20 child nodes which, in turn, each have 20 child nodes. This means that with a depth of 2 the branching factor of 20 at each stage will leave us with 400 possible nodes.

The nodes in the tree are evaluated according to a linear scoring polynomial. This is what Alpha-Beta will use to find its choice of move. The polynomial puts the value of the node into the following function example:

$$V = c_1 f_1 + c_2 f_2 + ... + c_n f_n$$

With the node value V being the sum of evaluations of different features of the game c_n multiplied by some function giving it weight f_n. For chess engines features include things such as remaining pieces, position of pawns, and captureable pieces among others This polynomial is what we use to fill our nodes and evaluate with the search algorithm [4].

2.2 Minimax

We need a method to search the tree of game states. That is our search algorithm Minimax. The Minimax algorithm is a decision rule used to minimize the loss for the worst possible case [3]. It returns the branch choice with the highest guaranteed value, without knowing the moves of the other players. Minimax takes node values of a certain depth to evaluate a path to take. There are two types of decisions that alternate, a maximum player and minimum player. Each will choose the best end result for themselves at every choice [8].

Minimax is a bottom up algorithm, the search begins evaluating at the bottom of the tree and works its way back to the root node for the result [6].

The nodes are each given a value for their position with higher being better for the current player. The value that they are given represents the position evaluated to represent the strength of each position for the player. Higher numbers indicate better position.

Minimax must search through the entire tree in order to work, or the entire tree up to a certain depth. In order to find the best move the whole tree must be searched, there are no shortcuts with this method.

The "maximin" value the algorithm is named for is the value the current player can be sure to get without knowing the actions of the other players. Each node of our adversarial search tree has a value that corresponds to the current player. The result of this is that the player choosing will always choose the branch that can force the highest minimum value.

Minimax has a search time of $O(b^m)$ which is the sum of all levels of the search tree, $b^0+b^1+b^2+...+b^m$, b equaling the maximum branching factor of the search tree and m is the maximum depth of the tree.

Figure 1 shows us a tree that we can use to demonstrate the decision process that the engine would take with a search depth and branching factor of 2. Each player is attempting to get the best result for themselves. This means that the Max player will always choose the higher number and the Min player will always choose the lower number

The algorithm starts at the root (green) node and goes down the left branch. It goes to the bottom and compares bottom (red) layers the 2 and the 8. Since the blue layer is a Min player they choose the lower number the 2. Next the tree goes down the right branch and compares the bottom (red) nodes the 1 and the 9. Since it is a Min player again it chooses the lower, the 1. Now the root (green) node chooses between the 2 blue nodes which currently have 2 and 1. The blue node is the Max player and so wants a higher number, choosing the 2. This is the final calculation and results in choosing the left branch.

This example is good at showing the basics but the system becomes much more complicated as you add layers. Going up to a depth of 3 will allow both players to make decisions effecting the game tree. This is where the Min and Max values come into play more. Each decision will be based on choosing the tree that has the guaranteed highest minimum. This is a result of each player wanting different results, and needing to force as good a result for the choosing player as possible.

The original Minimax algorithm, when first conceived, was based on exact values from game-terminal positions, or finding the fastest way to win the game. Later methods used by chess engines would focus on the heuristic analysis

of position and given the large search space, a more realistic method of playing the game [6].

2.3 Alpha-Beta

Alpha-Beta is an improvement over naive Minimax. It eliminates, or prunes, branches that are guaranteed to be worse then what has already been considered.

The name of Alpha-Beta comes from the two variables in the algorithm [6]. These are the same basic ideas as the Max and Min from the Minimax algorithm with one being the best guaranteed option for the alpha player in the current branch and the other the best guaranteed option for the beta player in the current branch. The two values begin at -∞ for alpha and ∞ for beta. This is used in order to guarantee that each value will be replaced. The function uses these in the same way with one exception, storing them to be used for pruning.

In the algorithm Alpha is used to show the best value that the Max or alpha player currently can force along the current branch. Beta is used to show the best value that the Min or beta player can force along the current branch.

Each node keeps its alpha and beta value that it has found along the branch and when it reaches a branch that guarantees it can't beat it, prunes the remaining nodes, saving valuable calculation time. This allows more searching of a more promising subtree in the same time.

We can use Figure 2 to see a simple example of Alpha-Beta pruning with the result, keeping the same search depth of 2 as the Minimax example.

Evaluating the tree begins the same as Minimax, from left to right. This also uses that adversarial search tree with Min and Max players. Searching through the left tree will yield 2 as the Min value by default as the first node to be evaluated setting the beta value to 2. Next we encounter 7 and, while it is better then 2, we know that the next player will choose the best move and so choose the 2 node making it irrelevant. Brining the 2 back up to the root node sets sets alpha to 2 as it is the best option so far. Going down to 1 as the first choice in the second branch sets this branches beta to 1 since it the lowest option it has seen so far. The program then can cut off all remaining analysis of the branch as it knows that the beta is less then the alpha or $1 < 2$, making it unnecessary calculation time to look at the remaining nodes in the branch. This pruning is what makes Alpha-Beta an improvement over Minimax.

This simple example is good at showing how the system works but doesn't truly show what it has the ability to do. Figure 3 shows this better with entire branches of nodes being cut off. Alpha-Beta will not just trim single nodes but entire branches without needing to even generate the positions. In exhaustive search methods this is extremely useful.

Modern chess engines will go to a depth of up to 35 making trimming even more valuable then the examples. For example, with 288 billion different possible positions after four moves each, we can see that being able to trim off billions of calculations is extremely advantageous. By the time a depth of 35 is reached it is impossible to evaluate the entire tree directly [11].

This ability to cut off unnecessary calculation speeds up the process by an average of 25% per level. Alpha-beta pruning has a worst case search space of $O(b^m)$ with a best case time of $O(\sqrt{b^m})$ [6].

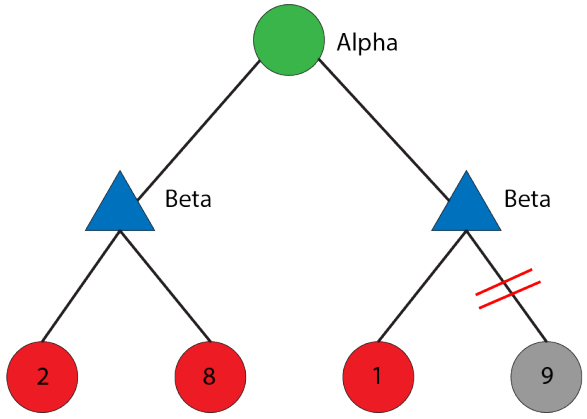

Figure 2: An Alpha-beta pruned simple tree

2.4 Enhancements

Given a proper evaluation function it seems that a chess engine should have the ability to play a perfect game. The trouble arises when we look at the number of moves, and by extension, decisions needed to be made.

We looked at how the tree is put together and say that there are 400 different positions after each player makes one move apiece. The faster tree traversals, with the fewer nodes visited and generated allows us to reach higher depths.

The first of the enhancements is to limit the search space. The first chess engine to beat the world champion, Deep Blue, had a depth of 6-12 on average. A engine on modern hardware will typically be limited to around a depth of 35 [11].

The next is the method called iterative deepening. This is a search method for trees that can be used on any search tree data-structure. It is a time-management strategy that is good for depth-first searches such as Minimax. This is similar to limiting the search space as in iterative deepening the program will first evaluate one node, then all at level 2, followed by level 3 and so on. This will guarantee a move by the end of alloted time as, even if the full search is not completed, it can return the most recent best result. This is used in the time sensitive tournament settings with limited calculation time per move [4].

Combining iterative deepening with Alpha-Beta, eliminating bad branches explores deeply only the good nodes. There can be problems with this, of course, where seemingly bad options can result in a better choice in the long run. Only deeper searches can fix this problem.

The last part to understand is that there is uneven tree development. Plenty of branches will end sooner then others through the chess game completing. This makes the search time even less as it will trim more of the tree so only unfinished, promising branches will need to be explored even close to all of the way [4].

2.5 New pruning techniques

In *Tree pruning for new search techniques in computer games* K. Greer looks at other methods for searching the best move. The method proposed in the paper evaluates and proposes different search techniques to evaluate a position and choose the best move.

The first method, The Chessmaps Heuristic, uses neural

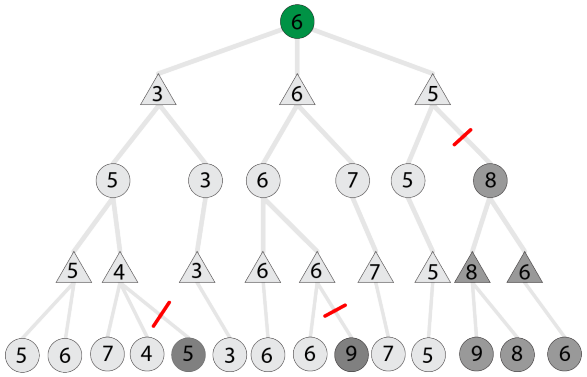

Figure 3: A large Alpha-Beta Pruned Tree

networks to evaluate the potion and choose the move. A neural network is a collection of artificial neurons that is linked by different weighted connections. Using these neurons and changing values of connections the neural network is able to keep improving through use to accomplish its purpose. This networks is put to use as a positional evaluator.

This method runs many iterations of tests trying each possibility and ranking the results. Based on this it re-weights the connections between the neurons making it more accurate each time.

The network is given a position to work with and ranks using the following criteria: safe capture moves, safe forced moves, safe forcing moves, safe other moves, unsafe capture moves, unsafe forced moves, unsafe forcing moves, unsafe other moves. Using these parameters the possible moves are ordered.

This was then combined with Alpha-Beta by using the neural network to organize the order in which nodes were sorted. This neural network is an addition to standard Alpha-Beta and is combined with it as an enhancement. It was lightweight enough that it could be used in this method but proved difficult to code in a way that provided any improvement over other methods.

The new method this paper proposes is Dynamic Move Sequences. This method creates chains of moves from the root node instead of a branching tree like Alpha-Beta. When used with Heuristics to choose the most likely move it can be used to search deeper in several branches without needing to look at other options. The downside to this is the possibility of skipping over moves that Alpha-Beta would see.

This method is proven not to return bad moves. It can often out perform Brute force algorithms in low depth searches, usually depths of 3-4. Due to improved technology and therefor depth brute force algorithms, such as Alpha-Beta, are still more successful.

The test results suggest that the potential of the move chains might be the fact that it can provide a basis for new ways to search a game tree and even under different circumstances. The coding and algorithm to implement these does not exist yet and is one of the topics that can be looked at in the future [4].

3. ALPHA-BETA IN CHESS ENGINES

Now that we know how the algorithm works we must see how it is implemented in the actual engine.

3.1 Board Representation

Computers cannot look at a chess board the same way that humans do. Where we see the chess board the computer must be able to read and analyze each of the pieces and as separate objects. First we must put the board in a format that the computer can understand. This can be done using lists, arrays, or similar sets of data. It should be noted that the way a program stores and accesses this information can greatly impact its performance as millions upon millions of moves are being processed and by extension data-structures being accessed. The pieces are placed in one of two ways, piece-centric and board-centric. Piece representation is done by storing the remaining pieces (the ones not captured at this point of the game) on the board in the data structure. One method of storing is with a bitboard approach, with one 64-bit word for each piece type, with bits to associate their occupancy, or board position. The other method, board-centric is done by looking at each square and determining what is there, empty or full of what piece, and storing it in a similar fashion as piece-centric [10].

3.2 4-Bit Piece Coding

In *An Alternative Efficient Chessboard Representation Based On 4-Bit Piece Coding* the author V. Vuckovik shows a new method of compact chessboard representation. This is based on bitboards as opposed to arrays of data. The proposal this paper makes is to encode the board in a more efficient format with each square needing 4 bits instead of 8 [10].

Bitboards are the chessboard representation based on the idea that a chessboard has 64 squares that is exactly the capacity of one long integer. One 64-bit register is able to represent the Boolean condition for each square of the chessboard, whether or not it is occupied . Since each bit in a bitboard indicates the absence or presence of some state about each place on the board, a board position can then be represented using a series of bitboards.

Bitboards have some flaws that limit their performance in certain conditions. They are substantially slower on 32-bit machines than on 64-bit. This limitation is impossible to overcome because the bitboards require compact 64-bit CPU registers to operate with maximal efficiency. This paper proposes a better method of encoding that will allow a computer to get rid of this problem.

Between the 6 types of pieces, both colors, and the empty square there are 13 different entities to represent. We need 4 bits to represent them, we can assume that this is the minimal uncompressed form of square coding. There are 64 squares on chessboard, so we need 32 bytes to define it completely. 4 bits * 64 squares gives us the 32 bytes for the board.

In tests this method of encoding was used to create the Achilles engine which was tested against 3 others: Fruit 21, Shredder 9 UCI, and Aristarch 4.50. It out performed them all with a high rate of victory, with at least a 67 percent win rate. These games were played usually with 15 minutes of time per player. Tests against chess grandmasters were also done with similar results. The time in these tests is less then normal tournament conditions, this is mainly done to show the speed of the engines is noticeable even with less time to work from. It is estimated that longer games would improve results even more.

Coding in this way is efficient and this method of storage can be easily converted to 64-Bit classic bitboards. There is

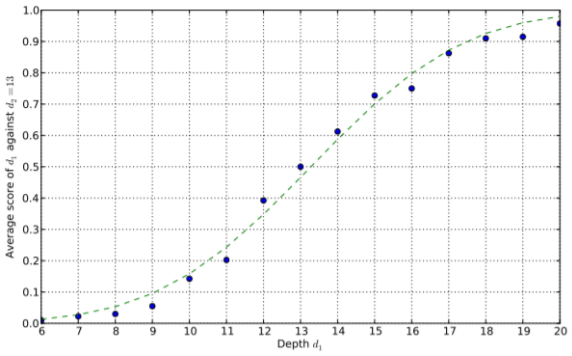

Figure 4: Differences in Search Depth Strength

therefore no disadvantage in adding this method of encoding to a chess engine as it can easily be used by both 32 and 64 bit systems.

3.3 Evaluating Position

Once the computer has the ability to represent the board the next stage is evaluating the position. Each node that is evaluated through Alpha-Beta is ranked through this function. Remember though that each evaluation the algorithm computes is not the current state but upcoming states of the game. In the same vein as the representation the method used to store and read will greatly effect the time spent on analysis. In subsection 4.1 we went over the formula used for it, but what are examples of variables that will go into the formula?

The first thing looked at and most important is obviously if the game has been won, if the king has been checkmated. The next, and more important for looking at the Alpha-Beta algorithm, is how many pieces remain and their values. Each piece has a defined worth and having more pieces, or points, will almost always result in a win if two players are equal. Once that has been tallied there are as many other methods as the programmer desires. Common additions are: pawn structure, ability to castle, development of pieces (their locations), and King safety. It's important to note that there are many others that are included with more or less value for each chess engine [6].

These evaluations are what Alpha-Beta uses to search with, what number is put into the nodes. Figure 1 has only 4 nodes, with the engine seeing those as 4 values that represent the board state after 2 moves. The engine will then use those values to move according the the algorithm as discussed before.

The main danger in other methods, such as the neural networks and the dynamic move sequences mentioned earlier, is the loss in evaluation quality because of the reduction in the search space. The potential for these methods seems to lie in potential new options for searching a game tree.

3.4 Search Depth

One of the largest reasons that chess engines are improving over time is increased computing power. This allows Alpha-Beta to search more deeply which in turn increases its playing ability [2]. According to Ferreria we can see the increase in depth and playing ability in action.

There is a significant difference in ability even at 1 deeper depth search. Figure 5 shows a search depth of 13 versus depths of 6-20. The winning percentage is displayed for 200 games in the Figure.

Ferreria's research in the paper *The Impact of Search depth on Chess Playing Strength* is the first demonstration of the specific ratings compared to depth. The purpose of this paper being finding the increase in ability that each move provides. Played on the Houdini Chess Engine this was affected by both the software and hardware available.

This paper found that the ELO difference between each level was approximately 86.5 points. [2] The Elo rating system is a method for calculating the relative skill levels of players. It is named after the inventor Arpad Elo. In chess ratings a rating above 2300 are usually associated with the *Master title* and a rating above 2500 being a *Grandmaster*. It should be noted that rating alone is not how one acquires these titles. 5 Grandmasters have gone above the rating of 2800. Current computers compete around the level of 3300. These numbers should help to give an estimate for how much improvement 86.5 elo points is.

As software and technology improves there may be changes in the specific numbers found here. Even using different ones currently could give different results. Even accounting for these changes the basic premise that a engine improves with search depth should remain the same [2].

4. HIDDEN PARTS OF CHESS

There are many things about a position a computer can read and evaluate. With perfect play these are harder and harder to achieve. Features like tempo (in chess meaning gaining a extra move), mobility of pieces, control of the center, and the value of these verses the more standard and easily quantifiable parts of chess that are difficult to put into a algorithm.

When making a proper evaluation function these become increasingly important the stronger a computer is. For example: is it worth sacrificing a pawn to gain 3 tempos? How much should king safety matter? These things must be accounted for in some way as they are what can make the difference. Traps and Gambits are both things that computers have struggled with for a long time as the standard evaluation methods do not work. Traps are setting up a obvious move that should result in a better position but long term will often be a mistake for the player that falls into it. Gambits are trading one obvious advantage for another possible advantage, for example in chess trading a bishop for a attack on the king. If we want Alpha-Beta to work to its fullest then they must be accounted for [4].

4.1 Parts of the Game

The next component that is not apparent on the surface is how to deal with different parts of the chess game. There are classically 3 separate sections of the game: the opening, the middle-game, and the end-game.

The Opening is a prepared set of moves that is "from book" meaning that it has been played and studied before. Most openings are only a few moves, though some can go up to 10 moves for each player. The middle game is what takes place as soon as the "book" moves are no longer in action, when new options are taken that have not been played before. Lastly the end-game is where few pieces are left. Exactly when the middle-game transitions to the end-game can differ according to who is analyzing. Common methods of determining this is when there is 13 or less points on

the board (the value of the remaining pieces) or less then 5 pieces that are not the pawns or King.

Each of these sections of the game gives power to different pieces and positions. The best evaluation functions can take advantage of these differences to adjust their play accordingly.

4.2 Opening books

In the hundreds of years of the existence of chess, many different ways of playing the game have been tried. [4] The beginning moves of the game, the opening, has resulted in the creation of many strategies and styles of game. In order to limit the search space at the beginning of the game, most modern chess engines will use a book of openings to play up to the point where it leaves standard ideas [1].

This does a few helpful things. It allows for different play. If you gave the computer a static state and told it to search through the options, the result would always, barring changes to the algorithm, return the same move. The programs can be enhanced with a certain degree of randomness, for example changing what openings are used and keeping things fresh. Without an element of randomness at the beginning only one game would every be played over and over with two unchanging computers.

4.3 Middle Game

The true power of Alpha-Beta comes into play in the middle game of chess. People have been trying to solve chess from both ends of the game for years but the middle game remains that part that is most elusive. With the exhaustive use of Alpha-Beta we can see the best we have ever been able to into how it should be played. There are no special tricks for this part that haven't already been mentioned. Only raw computing power can really improve this dramatically.

4.4 Endgame tables

One weakness of Chess engines has historically been endgames. When there is little change over a huge number of moves the Alpha-Beta function does not do well as nearly all of the positions are evaluated to the same value. The answer to this has been to use a massive amount of time and computing power to analyze every possible set of moves for a select number of pieces.

Chess has not been solved but there is progress of a sort. Going backwards from checkmate with few pieces limits your search space to the point where we can still calculate every possibility and solve the best move for each case. This has been done, at the current date, up to every combination of 6 pieces. Some 7 pieces have been solved as well [5].

New research by Haworth and Rusz has been looking into improving endgame by analyzing positions in order to realize if there are winning positions. This is mainly done by finding time wasting moves and eliminating them. By not moving the same pieces back and forth the endgame simplifies and is easier to search with Alpha-Beta [5].

The problem with endgame tables is their storage space. The number of moves stored for each option will exponentially increase with more pieces. 5-piece end-games take 7.05 GB of hard disk space for all five-piece endings. Storing all the six-piece endings requires approximately 1.2 TB. It is estimated that all of the seven-piece end-game table-bases will require between 50 and 200 TB of storage space. Similar to the reason that chess can't be solved from the front end, there is only so far we can go from the other side before space constraints are met [7].

5. CONCLUSION

The Alpha-Beta Pruning Algorithm has been extremely successful in improving that ability of chess playing engines through its optimality. Used in the chess engine Alpha-Beta has been advancing the ability of programs for the last 60 years [4].

The applications of the algorithm can be expanded very easily to any zero-sum game and used to solve what otherwise would still be guessed. Until computers have the processing power to solve chess entirely, Alpha-Beta will be the top method for playing the game of chess. There can, and have been, advancements in recent years through additions and enhancements to the algorithm as well as better utilization of current and improved hardware.

Acknowledgments

I'd like the thank Elena Machkasova for advising me. Thanks to Emma, Jake, and Dan who gave me feedback on this paper.

6. REFERENCES

[1] P. Audouard, G. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud. Grid coevolution for adaptive simulations: Application to the building of opening books in the game of go. *Applications of Evolutionary Computing*, pages 323–332, 2009.

[2] D. R. Ferreira. The impact of the search depth on chess playing strength. *ICGA Journal*, 36(2):67–80, 2013.

[3] A. Godescu. Information and search in computer chess. *arXiv preprint arXiv:1112.2149*, 2011.

[4] K. Greer. Tree pruning for new search techniques in computer games. *Advances in Artificial Intelligence*, 2013:2, 2013.

[5] G. M. Haworth and Á. Rusz. Position criticality in chess endgames. In *Advances in Computer Games*, pages 244–257. Springer, 2011.

[6] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[7] E. V. Nalimov, G. M. Haworth, and E. A. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, 2000.

[8] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. A new paradigm for minimax search. *arXiv preprint arXiv:1404.1515*, 2014.

[9] A. Saffidine, H. Finnsson, and M. Buro. Alpha-beta pruning for games with simultaneous moves. In *AAAI*, 2012.

[10] V. Vučković. An alternative efficient chessboard representation based on 4-bit piece coding. *Yugoslav Journal of Operations Research*, 22(2):265–284, 2012.

[11] P. Winston. 6. search: Games, minimax, and alpha-betaf.