Improving a minimax search algorithm for chess

Kristian Wasastjerna

School of Science

Bachelor's thesis
Espoo 10.07.2022

Supervisor

Prof. Kai Virtanen

Advisor

Prof. Kai Virtanen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the bachelor's thesis

Author	Kristian Wasastjerna				
Title	Improving a minimax search algorithm for chess				
Degree programme	Teknillinen fysiikka ja matematiikka				
Major	Matematiikka ja systeemitieteet			**Code of major**	SCI3029
Teacher in charge	Prof. Kai Virtanen				
Advisor	Prof. Kai Virtanen				
Date	10.07.2022	**Number of pages**	26+12	**Language**	English

Abstract

An intermediate computer chess program can play chess better than most humans. In this thesis we consider the minimax algorithm, used in chess and other two player zero sum games of complete information. This minimax algorithm is then by using different techniques like Alpha-Beta pruning, Quiescence search and more.

By using predefined test positions and versus play of different algorithms, the different versions of the algorithm can be compared. Also a more general approach is taken in comparing the algorithms, by comparing the resulting search tree sizes, instead of speed. This allows for implementation independent results. With these comparisons a modified minimax algorithm can be created, that outperforms a bare bones minimax algorithm.

The resulting algorithm is similar to modern, elite chess playing algorithms, albeit far simpler and weaker. The results from different versions of the algorithm support current sentiments on what is strong and what is weak. The the results are also compared to mathematically optimal outcomes. What improvements can be made and further areas of research are also discussed.

Keywords Computer chess, Minimax, Alpha-Beta, Null-move, Quiescence search, Pruning , Transposition table

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Tekniikan kandidaatintyön tiivistelmä

Tekijä	Kristian Wasastjerna				
Työn nimi	Improving a minimax search algorithm for chess				
Koulutusohjelma	Teknillinen fysiikka ja matematiikka				
Pääaine	Matematiikka ja systeemitieteet			**Pääaineen koodi**	SCI3029
Vastuuopettaja ja ohjaaja	Prof. Kai Virtanen				
Päivämäärä	10.07.2022	**Sivumäärä**	26+12	**Kieli**	Englanti

Tiivistelmä

Keskitason tietokoneshakkiohjelma osaa pelata shakkia paremmin kuin useimmat ihmiset. Tässä opinnäytetyössä tarkastellaan minimax-algoritmia, jota käytetään shakissa ja muissa kahden pelaajan täydellisen tiedon nollasummapeleissä. Perinteistä minimax algoritmia parannellaan käyttämällä erilaisia tekniikoita, kuten Alpha-Beta-karsiminen, Quiescence-hakua ja muuta.

Työssä käytetään ennalta määritettyjä testi asemia ja eri algoritmien vastakkain pelaamista, jotta lopulta saadaan eri algoritmin versioita voidaan vertailla. Vertailuista saadaan paranneltu versio minmax algoritmista, joka suoriutuu paremmin kuin pelkistetty minimax algoritmi.

Tuloksena oleva algoritmi on samanlainen kuin nykyaikaiset parhaat shakki algoritmit, vaikkakin paljon yksinkertaisempi ja heikompi. Algoritmin eri versioiden tulokset tukevat tämänhetkisiä käsityksiä eri tekniikoista. Työssä verrataan myös tuloksia matemaattisesti optimaalisiin tuloksiin ja pohditaan mitä parannuksia voisi tehdä, ja mitkä ovat kiinnostavia kohteita lisätutkimusta varten.

Avainsanat Computer chess, Minimax, Alpha-Beta, Null-move, Quiescence search, Pruning , Transposition table

Contents

1 Introduction

Chess is a two player, zero-sum game of complete information. These characteristics, and the fact that it is not too complex, nor too trivial, make it ideal task for a computer program. Most traditional chess programs utilize a modified variation of the minimax search algorithm. These different variations are generally judged on how well the resulting chess program plays. This means that the implementation of the minimax algorithm plays a crucial role in the strength of the program. Since the faster the software is, the more it can calculate, which results in stronger play.

In general chess playing program can be divided into three parts, the move-generator, search algorithm and evaluation function. A chess program works by taking the current position as input, creating a search tree, and assigning values to the search tree according to some algorithm. The algorithm used in this thesis is minimax. Generally different algorithms are compared by having them play against each other, or comparing the best moves they suggest [Levene et al., 2005].

In this thesis we only focus on the search algorithm part of a chess engine. We take an alternative approach in evaluating the strength of each version of the minimax search algorithm. Since most improvements to the minimax algorithm are pruning methods, e.g., reducing the amount of nodes traversed in the search tree. By running the algorithm on predefined positions and comparing the resulting search tree sizes comparisons of the algorithms can be made. Since the algorithms still have to make good moves, versus play is also conducted to ensure that each version plays good moves. Since some implementations can be more computationally expensive, keeping track of the time is also relevant in evaluating the search algorithms.

We will use multiple different concepts implemented on a bare bones minimax algorithm. Concepts such as Alpha-Beta pruning [Beal, 1999] [Knuth et al., 1975], Null move pruning [Heinz 1999] [David-Tabibi et al., 2002], transposition tables and Quiescence search [Beal, 1999]. With the testing done in this thesis, and sentiments echoed in current literature, we will construct an improved version of a minimax search algorithm for chess. This final version of the algorithm outperforms a bare bones minimax algorithm by a large margin.

Section 2 will briefly cover the concept of a minimax search algorithm, and all improvements that will be tested later on. It will also discuss what are the current sentiments about what are the best versions of each technique. Section 3 will largely cover the implementation of our minimax search algorithm, and its versions, and the performance criterion used to evaluate each version. Sections 4 and 5 will gather the results and present conclusions and compare them to existing sentiments and research.

2 Background

2.1 Structure of a computer chess program

The first complete algorithm for playing chess was devised by Alan Turing in 1950, but there didn't exist suitable hardware to run it. The first chess algorithm that was implemented into hardware was in 1957 by Alex Bernstein for an IBM 704.

These algorithms consisted of the three main parts, the move-generator, the search algorithm, and the evaluation function. The search algorithm is what this thesis focuses on, but surface knowledge of the other parts is required.

The evaluation function is a function that takes a position, or node, and returns a float that represents the state of the game. A positive value being an advantage for the maximizer, white, and negative being an advantage for the minimizer, black. Our evaluation function takes into account two things. The material, with pawns having values 100, knights having values 280, bishops 320, rooks 500 and queens 900. Kings have no material value since both sides always have one king in play. The second thing that it takes into account is positional score for each piece. Each piece receives a small positive or negative score depending on which square the piece occupies. This gives us an extremely rudimentary evaluation for a position, or node, but this is more than sufficient for the purpose of this thesis. Position and node are used interchangeably due out this thesis.

The move-generator takes a position, or node, and returns all possible positions that can be reached by making one move. This allows us to construct a search tree from the starting position, or root node, up to h moves ahead, or depth h. The move-generator also dictates in which order the moves are generated, e.g. first king moves, then queen moves and so on. This is relevant later on in section 2.3.

2.2 Minimax

Minimax was first presented by John von Neumann in 1928 as method for solving two-sided, zero-sum, perfect-information problems [Beal, 1999], such as chess. Minimax is a decision making rule, which aims to maximize the minimum gain. The algorithm gives the necessary rules to fill a search tree, which is generated by the taking a position and letting it be the root node. From here each node has child nodes corresponding to every possible move that the player can make from that position. This is continued until we have looked n moves ahead, n being the desired search depth. This creates a search tree, with n layers, which contains every single possible chess position reachable in n moves from the starting position. The nodes that have no child nodes are called leaf nodes.

A bare bones minimax search algorithm gives each node values by the following principles.

– The value of a leaf node is given by the heuristic evaluation function

– The value of a parent node is equal to the maximum or minimum value of its child nodes, depending if the node is a maximizing or minimizing node

An underlying assumption on the minimax algorithm, is that the backed up evaluation at the root node better when the search depth increases [Beal, 1999]. This means that methods that prune unwanted, or bad moves, are valuable since the more moves that the program can look ahead in the given time, the stronger it will play. Figure 1 shows an search tree to depth 4, filled by a minimax algorithm.

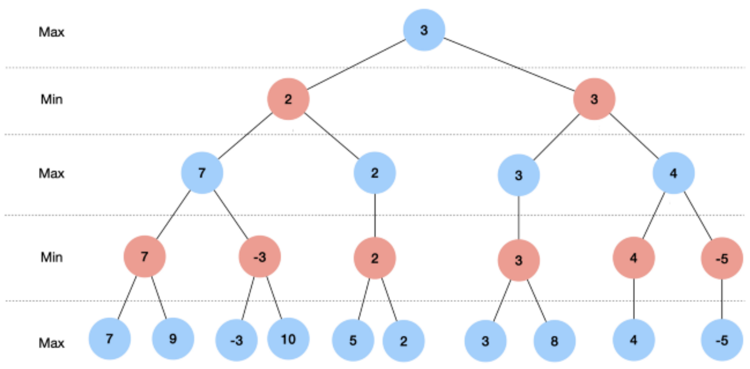

Figure 1: Example of a search tree to depth 4 with node values according to minimax

2.3 Alpha-Beta pruning

Alpha-Beta pruning (henceforth AB-pruning or AB) is an extension to a plain minimax algorithm. AB aims to identify nodes that can be pruned, since a better alternative has already been found, either for the maximizer or minimizer. Such nodes are referred as fail-high cutoff, and fail-low cutoff, for maximizer and minimizer respectfully.

An AB-pruning algorithm works on the following principles, along with the minimax principles

– Two values alpha and beta are passed down to each child node by their respective parent, with initial values at the root node being alpha=-∞ and beta=∞ respectfully

– A maximizing node updates alpha to be the maximum of its child nodes and the alpha passed down by its parent. Respectfully a minimizing node updates beta to be the minimum of its child nodes and the beta passed down

- If a maximizing node encounters a situation were the current evaluation of the node is greater than beta, we identify the node as a fail-high cutoff, and discard the node and its resulting sub-tree. Respectfully if a minimizing node encounters a situation were the current evaluation of the node is smaller than alpha, it is identified as a fail-low cutoff.

An important note of AB is that it will always return the same value for the root node as a plain minimax algorithm, since fail-low and -high cutoffs occur only if there exists a better option for either player. Thus AB-pruning is considered backwards pruning, meaning it doesn't effect the value of the root node. The theoretical minimum amount of nodes searched of a search tree with depth h and d, the average amount of child nodes for each parent node, by AB is [Knuth et al., 1975]

$$d^{[\frac{h}{2}]} + d^{[\frac{h}{2}]} - 1 \tag{1}$$

And AB-pruning can achieve this lower bound in the best case scenario, if the order in which the nodes are searched is optimal [Knuth et al., 1975]. Generally AB-pruning needs to examine $O(d^{[\frac{h}{2}]})$ nodes, as opposed to a plain minimax algorithm, which needs to examine all nodes $0(d^h)$ [Russel et al., 1995]. In the worst case scenario AB will have to examine all nodes, just as minimax [Russel et al., 1995].

This means that the ordering in which we search the nodes is critical to get the maximum amount of gain from AB. Since we cannot know beforehand is a node good for the current player, we have to either guess what kind of moves are generally good in chess, or do some sort of analysis on each move. Section 3.2.2 will cover move ordering in more depth.

2.4 Null-move pruning

Null-move pruning (henceforth NM-pruning or NM) is an extension to AB-pruning. NM attempts to identify fail-low and -high cutoffs by conducting a shallower null move search. The underlying assumption which NM relies on, is that for any position in chess, there exists a move better than doing a null move, or no move. Of course a null move is illegal, and there are positions were making a null move would be optimal (zugzwang position), but we ignore this for the purpose of our NM. This allows us to get a bound on the value of the node by doing a null move search. This bound is used to identify fail-low and -high cutoffs.

Null move pruning works on the same principles as AB and also

- For each node at depth n, $n < h - R$, were h is the search depth and $R \in N$ is the NM depth reduction, we conduct a search to depth $h - R$, but we swap the node to be a minimizing node if it was a maximizing node, and vice versa. This value we get from the search is the NM search value

- If the node is a maximizing node and the NM search value is greater than β, we identify this node as a fail-high cutoff, and if the node is a minimizing node and the search produces a value that is less than α, we identify it as a fail-low cutoff

NM-pruning is a forward pruning method, such that it may alter the value of the root node compared to a plain minimax search algorithm. The null move depth reduction $R \in N$ can be $1 \leq R < h$. Research [Heinz, 1999] shows that $R = 2$ appears to be the best NM depth reduction, with $R = 1$ being too conservative, and $R = 3$ being too aggressive. Conservative cutoffs mean we spend too much time guessing values for it to be worth wile, and aggressive meaning the NM search value isn't accurate enough.

Their are methods to circumvent NM failing in zugzwang positions. Verified null-move pruning [David-Tabibi et al., 2002] was introduced in 2002 to counteract these kinds of positions. When a cutoff is identified in a regular NM search, instead of a cutoff, the search is continued normally to a reduced depth $h - 1$. This thesis will only focus on traditional NM pruning.

2.5 Transposition table

A transposition table (henceforth TT) is a database which contains previously encountered nodes and relevant information about them, like their value. Since in chess there are multiple ways to arrive in the same position, we will encounter identical nodes in different parts of the search tree. If we store the value of each new node we encounter we could just look it up, instead of searching the same sub tree again.

The concept of a TT is simple, and the complexity comes from implementing it fast and reliably, but that is out of the scope of this thesis. Instead we will focus on three different approaches, and their pros and cons.

1- First we require that to lookup a nodes value the node that we are at, and the node we are looking up are at the same depth

2- Second we allow that the node we are at to be at the same depth or deeper in the search tree, such that lookup.node.depth$\leq$current.node.depth

3- Third we disregard the depth all together

The first approach doesn't alter the search in anyway. Instead of searching already searched nodes we just lookup the value previously encountered. The second approach should theoretically give us a more accurate value at the root node, since we may encounter a position for the first time at depth k, and encounter it later at depth l. Since we allow look ups from the TT to occur if $k \leq l$ then for these cases the value that we lookup, is produced by a sub-tree search deeper than what it would have been with no lookup. This gives a more accurate value for the node at depth l. We also encounter more look ups meaning that the second method should be all around better, with more look ups and more accurate values. The third method we disregard the depth all together. This lets us conduct even more look ups, but at the expense of the accuracy of the values we lookup. Since if $k > l$, then the value that we lookup would be more inaccurate than if we had just conducted a regular search for the node at depth l.

2.6 Quiescence search

Quiescence search (henceforth QS) is a narrow selective search at the end if the primary search meant to minimize the horizon effect. Primary search refers to the search tree which is formed by considering all possible moves to a certain depth h. Horizon effect is when the primary search misses something inevitable, which happens at depth $h + z$, $z \in N, z \geq 1$. The narrow selective search, conducted at the end of the primary search tree, is formed by taking a subset of all possible moves. The optimal subset would be all non-quiet moves. Quiet moves are moves that do not alter, or only faintly alter the evaluation of the position. Non-quiet moves are the complement set. Moves like capturing and checks are examples of non-quiet moves. Since capturing an for example an enemy queen would drastically alter the evaluation of the given position. Thus a quiet position is a position were the static evaluation of the node at depth h, is x, and the maximum static evaluation (if maximizing node, minimum if minimizing) of all child nodes at depth $h + 1$ is $x + \epsilon$, with ϵ being small.

Note that just extending the primary search tree to depth $h + 1$ doesn't get rid of the horizon effect, it just pushes it further. Thus the primary goal of a QS is to call the evaluation function only on positions that are considered quiet. The form of QS we will be considering is to take only captures and promotions. Then our QS tree is be formed by only captures and promotions, and the leaf nodes are nodes were no captures or promotions are possible, thus being quiet.

Quiescence search works on the same principals as minimax and also

- When the search tree encounters a node at depth h, with h being the maximum depth for the primary search, instead of getting the value of that node from the evaluation function we create a sub tree from that node consisting of only captures and promotions

- If the a node in the QS tree is a maximizing node, its value is the maximum of its child nodes and the evaluation given by the evaluation function for itself, respectfully for a minimizing node its value is the minimum of its child nodes and the evaluation given by the evaluation function

- If a node in the QS tree has no child nodes, i.e. is a leaf node, its value is given by the evaluation function

The QS we chose is self terminating, that is that we don't have to specify a maximum depth, for when to stop calling the move generator and call the evaluation function, since there is a maximum amount of consecutive captures. Our QS doesn't always terminate on quiet nodes. For example if either king is in check, the position is non-quiet, since the evaluation can drastically change in a single move. But since accounting for checks the QS would become way more complex, this thesis focuses only on a QS that consists of captures and promotions. Note that QS cannot detect checkmates or stalemates, since detecting either requires searching for all possible child nodes for a given node, and having the amount of child nodes be zero. Since QS inherently searches only a subset of all possible child nodes, we cannot tell is the set of all child nodes empty, or only the subset we are searching.

3 Methods

In this section we present the different versions of minimax we will be testing and comparing. The criterion for comparing each version will also be explained. The exact implementations of the algorithms in C++ will be noted in the appendix's and are not in the scope of this thesis. The implementation plays an important part in the actual strength of the algorithm, since a faster, better coded algorithm can search deeper. But since the specific implementation can differ between people, we will primarily focus on the search tree size and versus play when comparing different versions.

Forsyth–Edwards Notation (henceforth FEN) is a notation for describing a chess position. This notation is used when describing a position and it works by the following principles

- A string is divided into six different fields, separated by white space
- Field one contains piece information. Starting from rank 8 to 1, going from A file to H file, with "/" separating ranks. Each piece is denoted by a letter with pawn = "P", knight = "N", bishop = "B", rook = "R", queen = "Q" and king = "K", with uppercase being white, and lowercase being black. Empty squares are denoted by a number from one to eight, depending on the amount of consecutive empty squares
- Field two is either w or b, signifying the turn
- Field three is castling rights, with "K" being king side castling available and "Q" being queen side castling available. Again uppercase for white and lowercase for black. "-" denotes no castling rights.
- Field four denotes the enpassant square. That is the square which a pawn just moved over when moving two squares. If no enpassant square exists, a "-" is here
- Field five has the number of half moves since the last capture
- Field six has the number of full moves, that is when both players have made a move

3.1 Performance criterion

Each version of the search algorithm will be run on 24 unique chess positions. The 24 unique positions are shown in Appendix A in FEN, with their respective primary search depth. The algorithm is ran for each position to their respective search depth, such that we can compare the results of different algorithms. From the formed search trees we collect,

- Total nodes

- Primary search nodes. This is equal to total nodes if there is no QS implemented
- Leaf nodes
- Branching factor, which is the amount of child nodes divided by the amount of parent nodes
- Nodes in the quiescence search, that is nodes with depth greater than the primary search depth h. This is 0 for all algorithms that don't implement quiescence search
- The best recommended move, that is the node with the greatest value at depth 1
- The value of the root node, that is the evaluation the search gives for the root node
- The time it takes for the search algorithm to process the search tree

Minimizing the amount of total nodes is our primary focus. The amount of leaf nodes is also relevant, since the evaluation function is called for each leaf node, making them computationally more expensive than non-leaf nodes. Branching factor gives us the average amount of child nodes for each parent node, with values ranging from two to 40, so visualizing the data with branching factor is more intuitive. The QS tree nodes give us data on what amount of the total nodes are in the QS tree, versus the primary search tree. This tells us how relatively computationally expensive is it to implement our version of the QS.

The value of the root node gives us the backed up evaluation. This evaluation can be compared to more values given by more sophisticated chess playing software to get a error for the root node. Of course the true evaluation for a position isn't known, except for endgame situations. But modern high end chess algorithms are good enough to beat even the best humans with ease, so their evaluation of a position is the most accurate available.

The time it takes for each algorithm to run through each search tree. This is more relevant than total nodes, if ones goal is to create a strong chess algorithm, but since speed is reliant on hardware, and the implementation of our software, we will focus only secondarily on the time it takes.

We will also conduct versus play for algorithms that have different recommended best moves. This allows us to compare the relative strength of each algorithm.

3.2 Implementation

In this section, we will explore the exact concepts of minimax and the improved versions we've used. We cover each concept and the different variations of each one.

3.2.1 Minimax

The minimax algorithm is the core of our chess algorithm. It also works as a benchmark for future algorithms, since a plain minimax algorithm runs through each possible node in a given search tree. Appendix G shows the exact implementation in C++ as used in this thesis. This structure of the basic minimax is present in every version of the algorithm.

3.2.2 Alpha-Beta pruning

The exact implementation of our AB pruning improved minimax algorithm is noted on Appendix B. The only difference from the plain minimax algorithm is that AB passes down alpha and beta for each child node. If the boundaries are broken, they produce a cut-off and we terminate the search of that sub tree and return the evaluation of the child node that caused the cut-off. Alpha can only be updated by the maximizer white, since alpha represents the best option for the maximizer so far, and vice versa for beta and the minimizer. Alpha is initialized as negative infinity and beta as infinity at the start of the search, so we don't start producing cutoffs until we have found a suitable move.

The amount of cutoffs produced is dependant on the order of the moves searched. Move ordering heuristics can be divided into two categories, static and dynamic. Static move ordering is ordering that is done without evaluating the moves, for example generating captures first and then the rest of the moves would be a simple example of a static move ordering heuristic. General consensus is that this move ordering in searching captures first is quiet effective [Thé, 1992]. Dynamic move ordering heuristics are done by evaluating the moves after they have been generated and ordering them by some metric. An example of this would be to do a shallower search for each position in moves, and then order them from best to worst according to the shallower search. Dynamic move ordering heuristics are generally more computationally expensive, but often yield more cutoffs. In this thesis we will only focus on static move ordering heuristics.

We have 4 different move ordering heuristics, with each generating the moves in the following orders

1 - Promotions, K captures, P captures, N captures, B captures, R captures, Q captures, castling, P moves, N moves, B moves, R moves, Q moves, K moves

2 - Castling, P moves, N moves, B moves, R moves, Q moves, K moves, Promotions, K captures, P captures, N captures, B captures, R captures, Q captures

3 - Promotions, K captures, P captures BNRQ, N captures RQ, B captures RQ, R captures Q, P captures P, N captures NBP, B captures NBP, R captures RNBP, Q captures, castling, P moves, N moves, B moves, R moves, Q moves, K moves

4 - Promotions, K captures, R captures Q, B captures RQ, N captures RQ, P captures NBRQ, P captures P, N captures BNP, B captures BNP, R captures RBNP, Q captures, castling, Q moves, R moves, B moves, N moves, P moves, K moves

Here P=pawn, N=knight, B=bishop, R=rook, Q=queen and K=king. Moves in this context refer to moves that do not capture an enemy piece. The search trees produced by each of these AB pruning minimax algorithms with specific move ordering heuristics will be presented and analyzed in section 4 and 5.

3.2.3 Null-move pruning

The exact implementation of our NM pruning version of minimax is found in Appendix C. NM requires AB, since NM pruning tries to guess a lower bound (upper bound if minimizing node) for each node and if the guessed value is out of the bounds set by alpha and beta, we just return the value given by the NM search. Note that our the players turn who it is, can't have his king in check, since then the resulting position would be illegal in chess.

The main variable in NM pruning is the NM depth reduction (hence forth "R") variable. This defines how much more shallow our NM search is compared to our primary search depth. The choice lies in how accurate we want our NM guess to be. With a value of 0, we would just be searching a NM sub tree to the same depth as the original sub tree from that node. This is not a good idea since searching the original node without a NM search is just as fast. Since we are trying to guess these cutoffs with less computational power, we let R> 0. The larger we let R be, the less accurate our guess for the nodes value will be, since we search a shallower NM sub tree, but this is computationally less expensive. Research [Heinz, 1999] showed that R=2 is generally best. We conduct the test positions for cases $R \in (1, 2, 3)$, and compare the resulting search trees. Since NM is a forward pruning method, NM sometimes will alter the result of the search. This is why conducting versus play with these versions of the algorithm is important in seeing if increasing R reduces the level of play, as one would expect.

3.2.4 Transposition table

The implementation of our TT into our minimax algorithm can be found in Appendix D. A transposition table requires that each position is represented by a 64-bit hash key. Note that this hash key might not be unique for two different positions, since a 64-bit number has 2^{64} unique variations, whilst chess has about 10^{120} [Shannon C., 1950]. Since we are running an empty TT for each position for our algorithm, we on average encounter only 10^8 to $5 \cdot 10^9$ positions per search. Thus we make the assumption that key collisions are rare enough to not impact the search, and disregard the threat. This is also shown in [Hyatt, et al. 2005], that key collisions impact the search less than previously thought.

Each key stores the nodes evaluation and the depth that it was evaluated at. Then when the algorithm searches through the search tree, it initially creates the hash key for that node, and then checks does it find the hash key in the TT. If the key is found, it then checks if the depth is acceptable, and if it is, it just returns the evaluation stored in the TT, without searching the nodes sub tree. If no hash key is found, a normal search is conducted, and the TT is updated at the end with the evaluation and depth. The three cases of different acceptable depths considered in this thesis are the ones noted in section 2.5. The first case, handles exactly the same as a minimax algorithm without a TT. The second and third cases can alter the searches, and thus we will conduct versus play in addition to comparing the resulting search trees.

3.2.5 Quiescence search

The QS implementation can be found in Appendix E. Note that the QS implementation has AB pruning and NM pruning implemented in it. This is because the QS search tree grows extremely large, so having these implemented alongside it to be able to search the test positions to the same primary search depth.

The primary interest in a QS is to improve the play by reducing the horizon effect. This means that the focus is on the versus play when comparing the algorithms, since the size of the search tree formed by our QS implementation is larger.

4 Results

In this section we will cover the resulting search trees produced by each version of the algorithm. The search trees produced by our plain minimax algorithm work as a benchmark for future algorithms.

The amount of nodes per search tree with a basic minimax search algorithm can be seen in Figure 2. The amount of time taken can also be seen in Figure 3.

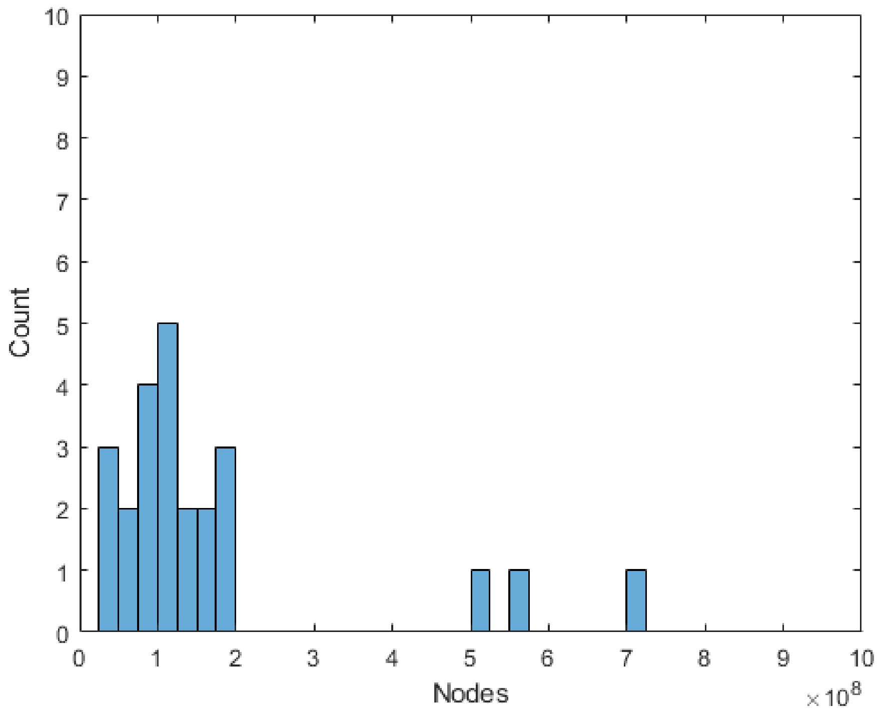

Figure 2: Amount of nodes for the 24 different search trees produced by plain minimax algorithm

The average amount of nodes being $1.716082 \cdot 10^8$ and the search trees have an average branching factor of 37.4773. The average time taken was 39.95845 seconds. The time is only for reference and as previously stated we will be considered only secondarily in the ranking of algorithms. Also noting that 96.6764% of all nodes searched were leaf nodes.

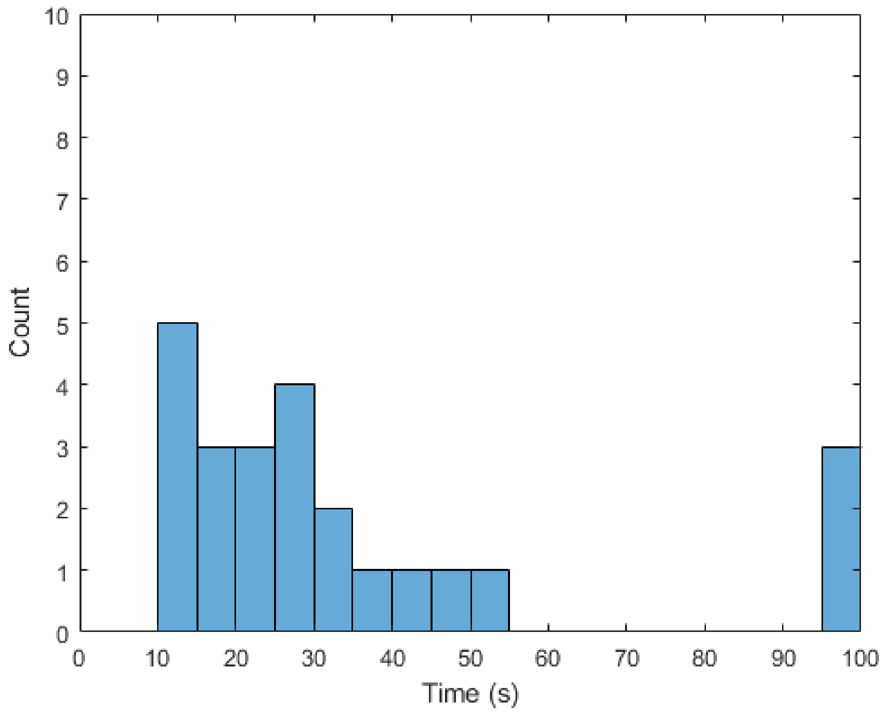

Figure 3: Time taken for each search tree for a plain minimax algorithm

AB pruning algorithms have no difference in best moves, so when comparing them to the basic minimax algorithm and each other, we will not conduct versus play. We had four different move orderings for AB, see section 3.2.2. The best performing one was number 1 with an average of $2.981413 \cdot 10^5$ nodes, and an average time of 0.1275 seconds. The theoretical minimum amount of average nodes for all search trees with AB can be retrieved by applying the equation (1) for each search tree and adding the results up. This gives the minimum amount of nodes for all 24 search trees to be $7.2282 \cdot 10^5$ nodes. Given that the sum of the 24 search tree nodes was $7.155392 \cdot 10^6$, there is a more optimal ordering, but it is highly likely that dynamic move ordering heuristics are required to achieve significant improvement on this.

Since the improvement compared to a regular minimax algorithm was so drastic, we added 2 to all primary search depths when comparing the AB algorithms to each other, to get a better comparison. The averages of each AB version with primary search depth +2 can be seen below in Table 1

	Average Nodes	Total Nodes	Percentage of Leaf Nodes	Average Branching Factor	Average Time
Version 1	$1.089416 \cdot 10^7$	$2.61459855 \cdot 10^8$	88.5215%	8.88412	4.8040 s
Version 2	-	-	-	-	-
Version 3	$1.721357 \cdot 10^7$	$4.13125714 \cdot 10^8$	85.7950%	8.53434	8.9213 s
Version 4	$1.595609 \cdot 10^7$	$3.82946100 \cdot 10^8$	85.4641%	8.54064	8.2946 s

Table 1: Data for each AB version for comparison with primary search depth +2

Here we can clearly see that of the four static move ordering heuristics, version 1 gives us the most cutoffs. Version 2 is by far the worst, given that we couldn't even get results without the testing environment crashing. Versions 3 and 4 worked fairly well, but inferior to version 1. Hence we will be using the move ordering heuristic for our final algorithm from version 1.

For NM pruning, we implement it on top of AB version 1, since it was the best version. The AB version 1 will work as a bench mark, note that all primary search depths are increased by 2 for NM testing. Since NM is selective pruning we also conduct versus play to compare them. Below on Table 2, we can see the results for each NM version, and AB version 1.

	Average Nodes	Total Nodes	Percentage of Leaf Nodes	Average Branching Factor	Average Time
AB Version 1	$1.089416 \cdot 10^7$	$2.61459855 \cdot 10^8$	88.5215%	8.88412	4.8040 s
NM R=1	$5.369111 \cdot 10^6$	$1.28858680 \cdot 10^8$	90.1071%	8.83318	8.5463 s
NM R=2	$5.561990 \cdot 10^5$	$1.3348766 \cdot 10^7$	75.0400%	4.46826	1.0667 s
NM R=3	$5.277818 \cdot 10^6$	$1.26667634 \cdot 10^8$	88.0728%	7.95092	2.5789 s

Table 2: Data for each NM version and AB version 1 with primary search depth +2

On Table 3 we can see cumulative score when playing against each other. The games were best of six, with 3 starting positions listed on Appendix F, with both sides playing white and black for a total of 18 games for each version.

	Points
AB Version 1	9.5/18
NM R=1	8.5/18
NM R=2	9/18
NM R=3	9/18

Table 3: Data for versus play with NM versions and AB version 1

As we can see the versus play results are drawn, one game was drawn by AB version 1 against NM R=1, but otherwise all results were identical, most games being identical in moves. This means that our NM pruning was extremely conservative in all versions, since we nearly never pruned the best move, and if we did, the resulting best move was sufficient for a draw. NM R=2 had by far the best pruning, with the average amount of nodes per search tree being a magnitude smaller than R=1 and R=3 versions. The time also tells us that for NM R=1, the time spent conducting null move searches, even though it pruned a lot of sub trees, was not worth it. Interesting note is that NM R=2 had a lot less leaf nodes comparatively, which also gives it an advantage in speed since the evaluation function is called there. Clearly the best option from this data is NM R=2, which is inline with prior research [Heinz, 1999]. Thus we will implement NM pruning to our final version with R=2.

The TT results can be seen in Table 4 and the versus results in table 5. Since the search results of TT version one are identical to a plain minimax, we conduct versus play with only the 3 versions of TT.

	Average Nodes	Total Nodes	Percentage of Leaf Nodes	Average Branching Factor	Average Time
Minimax	$1.71608186 \cdot 10^8$	$4.118596476 \cdot 10^9$	96.6764%	37.4773	39.9585 s
TT version 1	$4.8383341 \cdot 10^7$	$1.1612002 \cdot 10^9$	97.2613%	37.0026	54.2508 s
TT version 2	$4.8233201 \cdot 10^7$	$1.157596829 \cdot 10^9$	97.2701%	37.0026	52.6646 s
TT version 3	$3.6350633 \cdot 10^7$	$8.72415198 \cdot 10^8$	97.4408%	36.9840	40.9875 s

Table 4: Data for each TT version and plain minimax with primary search depth +0

From table 4 we can see that each TT table reduced the average nodes. The time however was increased, due to our non optimal implementation. Also TT version 3 reduced the search tree size the most, as was expected. From Table 5 we can see that TT version 1 fared the best with version 2 faring the worst. Initial expectations were that TT version 3 would perform the worst, but the sample size may be too small to draw significant conclusions. Since all TT versions displayed only marginal improvements in search tree size, with the time it took growing, and the playing strength wasn't clearly better, we will not implement a TT in the final algorithm.

	Points
TT version 1	7.5/12
TT version 2	4/12
TT version 3	6.5/12

Table 5: Versus play with TT versions, note that version 1 plays same moves as minimax

The final improvement is a QS. Since the QS isn't about reducing the search tree size, but reducing the horizon effect resulting in better play, we will focus on versus play. The amount of nodes can be seen in Table 6, with NM R=2 as a benchmark. We will also tested QS with primary search depth -1, since QS grows the time it takes to go through the search trees, to get a fairer comparison to NM R=2.

	Average Nodes	Total Nodes	Quiescence nodes	Percentage of Leaf Nodes	Average Branching Factor	Average Time
NM R=2	$5.561990 \cdot 10^5$	$1.3348766 \cdot 10^7$	0	75.0400%	4.46826	1.0667 s
QS primary search depth +0	$1.6192559 \cdot 10^7$	$3.886214 \cdot 10^8$	$3.852213 \cdot 10^8$	51.1840%	2.3312	28.5708 s
QS primary search depth -1	$7.471061 \cdot 10^6$	$1.793054 \cdot 10^8$	$1.787808 \cdot 10^8$	50.1020%	2.2469	11.4 s

Table 6: Data for QS with primary search depth +0 and -1, NM R=2 with primary search depth +2

We can see from table 6 that even with the primary search depth being -1, QS still has more nodes. This is mostly because the position, "r4r1k/4bppb/2n1p2p/p1n1P3/1p1p1BNP/3P1NP1/qP2QPB1/2RR2K1 w - -", was a huge outlier, with over 70% of all nodes from the 24 test positions being from this position for the QS algorithm for both depths. This means that for some positions, QS will slow down the search a lot, because it creates a huge search tree.

Also of note that the maximum search depths reached was 32, and that almost all of the nodes in QS search trees were at depths greater than the primary search depth, called quiescence nodes.

	Points
NM R=2 primary search depth +2	2/12
QS primary search depth +0	8.5/12
QS primary search depth -1	7.5/12

Table 7: Versus play with QS and NM R=2

From Table 7 we can clearly see that QS outperforms NM R=2 by a large margin, with NM R=2 winning 0 games and drawing only 4. QS is then good to implement, even though it cuts down our primary search depth, the improvement in playing strength is apparent and worth it.

4.1 Summary

From the results we can create a stronger chess search algorithm by combining the different versions together. The bare bones minimax algorithm is the backbone and we add the best performing versions to it. This will be the final version of the improved minimax algorithm.

First different move ordering heuristics were considered for AB pruning, with AB version 1 being the best. This is added to the minimax algorithm. Then NM pruning was considered, with R=2 producing the best results. So this is added to our final algorithm. TT were tested and while they showed the ability to reduce the search tree sizes, our implementation was inefficient and resulted in slower searches. So TT is not implemented. Finally QS was tested and it produced vastly better results in versus play than a non QS implementation. Thus we implemented it into our final algorithm. The final algorithm can be found in Appendix.

5 Conclusions

In this thesis we improved on a plain minimax algorithm for chess with 4 different techniques. We tested the different versions with 24 test positions, compared the search trees and pitted the different versions against each other in regular chess and compared results. The results were then used to build the best version of a chess playing algorithm possible within our own limitations.

Implementing the original minimax algorithm was simple enough, and was straight forward. The move generator was our own, but could have been taken from any open source chess engine if speed was our number one concern, but since we wanted to try different move ordering heuristics, using our own move generator made it simpler. As far as static move ordering goes, it seemed that searching captures and then the rest of the moves is the best option. This is also supported by research [Knuth et al., 1975] Dynamic move orderings were not researched in this thesis. Implementing the Alpha-Beta pruning was also straightforward and simple.

Null move pruning is one of the most researched forward pruning methods, and letting R=2 is known to give the best results generally, and our researched backed that up clearly. More aggressive NM pruning could also be researched, with instead requiring the null move value to be outside alpha and beta to produce a cutoff, we could produce a cutoff if its even close to the edge of allowed values. More aggressive forward pruning produces smaller search trees allowing us to search deeper, but at the cost of potentially discarding the best moves. More research could be done to evaluate how aggressive can NM pruning be until it produces significantly worse results.

Our transposition table worked as expected, but implementing it is extremely technical, since creating the hash keys and searching them can be slow, as it was in our implementation. Transposition tables are considered necessary for top level chess computers, but we decided against it. The advantage of a TT implementation is also that you can store results of previous searches to use in future searches, cutting down the search tree even more, but this runs into hash collision problems as the table fills up.

The quiescence search we implemented was simple, yet effective. This was the single most effective improvement regarding playing strength. Moves like checks could and should be considered when implementing a QS, but implementing them can be fairly complicated compared to our implementation.

Probably the greatest improvement still to be made, is storing information from previous searches and using it on the current search, through TT and principal variation search (PVS). PVS keeps track of the path of the best route in a search tree, and uses this route in its next search as the first guess for the best move. It could be considered dynamic move ordering to an extent. Since our implementation doesn't retain any information from previous searches, it often could save a lot of computational power by saving some previous results.

Regardless of any implementation problems and small sample sizes, the resulting algorithms, which can be found in Appendix E, can play chess against humans without problems. Future work on this algorithm can be done especially by implementing a good TT, more forward and aggressive pruning methods, more comprehensive QS and dynamic move ordering. The source code for the whole thesis can be found in Appendix H.

References

Beal D.F., The Nature of Minimax Search, 1999

David-Tabibi O., Netanyahu, N.S., Verified Null-Move Pruning, ICGA Journal, Vol. 25, No. 3, pp. 153-161, 2002

Hyatt R., Cozzie A., The Effect of Hash Signature Collisions in a Chess Program. ICGA Journal, Vol. 28, No. 3, 2005

Heinz E.A. Adaptive null-move pruning, ICCA Journal, Vol. 22, No. 3, pp. 123–132, 1999

Knuth D.E., Moore R.W., An Analysis of Alpha-Beta Pruning, 1975

Levene M., Bar-Ilan J., Comparing Move Choices of Chess Search Engines, J. Int. Comput. Games. Assoc., Vol. 28, pp. 67-76, 2005

Russel S., Norvig P., Artificial Intelligence: A Modern Approach, 1995

Shannon C., Programming a Computer for Playing Chess, Philosophical Magazine, Ser.7, Vol. 41, No. 314, 1950

A Appendix

Position in FEN	Primary search depth
r3qb1k/1b4p1/p2pr2p/3n4/Pnp1N1N1/6RP/1B3PP1/1B1QR1K1 w - -	5
r4rk1/pp1n1p1p/1nqP2p1/2b1P1B1/4NQ2/1B3P2/PP2K2P/2R5 w - -	5
r2qk2r/ppp1b1pp/2n1p3/3pP1n1/3P2b1/2PB1NN1/PP4PP/R1BQK2R w KQkq -	5
r1b1kb1r/1p1n1ppp/p2ppn2/6BB/2qNP3/2N5/PPP2PPP/R2Q1RK1 w kq -	5
r2qrb1k/1p1b2p1/p2ppn1p/8/3NP3/1BN5/PPP3QP/1K3RR1 w - -	5
1r1bk2r/2R2ppp/p3p3/1b2P2q/4QP2/4N3/1B4PP/3R2K1 w k -	5
r3rbk1/ppq2ppp/2b1pB2/8/6Q1/1P1B3P/P1P2PP1/R2R2K1 w - -	5
r4r1k/4bppb/2n1p2p/p1n1P3/1p1p1BNP/3P1NP1/qP2QPB1/2RR2K1 w - -	5
r1b2rk1/1p1nbppp/pq1p4/3B4/P2NP3/2N1p3/1PP3PP/R2Q1R1K w - -	5
r1b3k1/p2p1nP1/2pqr1Rp/1p2p2P/2B1PnQ1/1P6/P1PP4/1K4R1 w - -	5
8/kp5p/p4p2/P3p1p1/1Pb1P1P1/2P1b2P/3rN3/4RK2 w - -	7
6kb/4p3/3p2P1/2p2P2/1p2P1P1/3P4/1QPq4/K1R5 w - -	6
r2q1rk1/4bppp/p2p4/2pP4/3pP3/3Q4/PP1B1PPP/R3R1K1 w - -	5
rnb2r1k/pp2p2p/2pp2p1/q2P1p2/8/1Pb2NP1/PB2PPBP/R2Q1RK1 w - -	5
2r3k1/1p2q1pp/2b1pr2/p1pp4/6Q1/1P1PP1R1/P1PN2PP/5RK1 w - -	5
3rn2k/ppb2rpp/2ppqp2/5N2/2P1P3/1P5Q/PB3PPP/3RR1K1 w - -	5
4b3/p3kp2/6p1/3pP2p/2pP1P2/4K1P1/P3N2P/8 w - -	7
2r3k1/pppR1pp1/4p3/4P1P1/5P2/1P4K1/P1P5/8 w - -	6
1nk1r1r1/pp2n1pp/4p3/q2pPp1N/b1pP1P2/B1P2R2/2P1B1PP/R2Q2K1 w - -	5
r1b2rk1/2q1b1pp/p2ppn2/1p6/3QP3/1BN1B3/PPP3PP/R4RK1 w - -	5
r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq -	5
r4rk1/1pp1qppp/p1np1n2/2b1p1B1/2B1P1b1/P1NP1N2/1PP1QPPP/R4RK1 w - -	5
r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq -	6
3r1k2/4npp1/1ppr3p/p6P/P2PPPP1/1NR5/5K2/2R5 w - -	6

B Appendix

```
    int32_t minimax(bitBoard &board, uint8_t depth, bool whiteTurn,
                    uint8_t maxDepth, int32_t alpha, int32_t beta){

    if(depth >= maxDepth){
        return evaluation(board);
    }

    vector<bitBoard> moves;
    int32_t eval;
    moves.reserve(50);

    if(whiteTurn){
        generateWhiteMoves(board, moves);
        if(moves.empty()){
            return whiteKingCheck(board);
        }
        eval = NEGINF;
        for(bitBoard &i : moves){
            int32_t moveEval = minimax(i, depth+1, false, maxDepth, alpha, beta);
            eval = max(eval, moveEval);
            if(beta <= eval){
                break;
            }
            alpha = max(alpha, eval);
        }
        return eval;
    }else{
        generateBlackMoves(board, moves);
        if(moves.empty()){
            return blackKingCheck(board);
        }
        eval = INF;
        for(bitBoard &i : moves){
            int32_t moveEval = minimax(i, depth+1, true, maxDepth, alpha, beta);
            eval = min(eval, moveEval);
            if(alpha >= eval){
                break;
            }
            beta = min(beta, eval);
        }
        return eval;
    }
}
```

C Appendix

```
int32_t minimax(bitBoard &board, uint8_t depth, bool whiteTurn,
                uint8_t maxDepth, int32_t alpha, int32_t beta){

    if(depth >= maxDepth){
        return evaluation(board);
    }

    vector<bitBoard> moves;
    int32_t eval;
    moves.reserve(50);

    if(whiteTurn){
        if((maxDepth > depth + NMDEPTHREDUCTION) && (whiteKingCheck(board) == 0)){
            int32_t nullMoveValue =
            minimaxNoNM(board, depth + NMDEPTHREDUCTION,
                        false, maxDepth, alpha, beta);
            if(nullMoveValue >= beta){
                return nullMoveValue;
            }
        }

        generateWhiteMoves(board, moves);
        if(moves.empty()){
            return whiteKingCheck(board);
        }
        eval = NEGINF;
        for(bitBoard &i : moves){
            int32_t moveEval = minimax(i, depth + 1, false, maxDepth, alpha, beta);
            eval = max(eval, moveEval);
            if(beta <= eval){
                break;
            }
            alpha = max(alpha, eval);
        }
        return eval;
    }else{
        if((maxDepth > depth + NMDEPTHREDUCTION) && (blackKingCheck(board) == 0)){
            int32_t nullMoveValue =
            minimaxNoNM(board, depth + NMDEPTHREDUCTION,
                        true, maxDepth, alpha, beta);
            if(nullMoveValue <= alpha){
                return nullMoveValue;
            }
        }

        generateBlackMoves(board, moves);
        if(moves.empty()){
            return blackKingCheck(board);
        }
        eval = INF;
        for(bitBoard &i : moves){
            int32_t moveEval = minimax(i, depth + 1, true, maxDepth, alpha, beta);
            eval = min(eval, moveEval);
            if(alpha >= eval){
                break;
            }
            beta = min(beta, eval);
        }
        return eval;
    }
}
```

D Appendix

```
    int32_t minimax(bitBoard &board, uint8_t depth, bool whiteTurn,
                    uint8_t maxDepth, int32_t alpha, int32_t beta){

    if(depth >= maxDepth){
        return evaluation(board);
    }

    uint64_t posHash = hashFunction(board, whiteTurn);
    if(table.count(posHash)){
        if((table[posHash].second) == depth){
            return table[posHash].first;
        }
    }

    vector<bitBoard> moves;
    int32_t eval;
    moves.reserve(50);

    if(whiteTurn){
        generateWhiteMoves(board, moves);
        if(moves.empty()){
            return whiteKingCheck(board);
        }
        eval = NEGINF;
        for(bitBoard &i : moves){
            int32_t moveEval = minimax(i, depth + 1, false, maxDepth, alpha, beta);
            eval = max(eval, moveEval);
        }
    }else{
        generateBlackMoves(board, moves);
        if(moves.empty()){
            return blackKingCheck(board);
        }
        eval = INF;
        for(bitBoard &i : moves){
            int32_t moveEval = minimax(i, depth + 1, true, maxDepth, alpha, beta);
            eval = min(eval, moveEval);
        }
    }
    table.emplace(posHash, make_pair(eval, depth));
    return eval;
}
```

E Appendix

```
int32_t qSearch(bitBoard &board, bool whiteTurn, int32_t alpha, int32_t beta){

    int32_t standingEval = evaluation(board);

    if(whiteTurn){
        if(standingEval >= beta){
            return standingEval;
        }
        alpha = max(standingEval, alpha);
    }else{
        if(standingEval <= alpha){
            return standingEval;
        }
        beta = min(standingEval, beta);
    }

    vector<bitBoard> moves;
    int32_t eval = standingEval;
    moves.reserve(10);

    if(whiteTurn){
        generateWhiteNonQuietMoves(board, moves);
        for(bitBoard &i : moves){
            int32_t moveEval = qSearch(i, depth + 1, false, alpha, beta);
            eval = max(eval, moveEval);
            if(beta <= eval){
                break;
            }
            alpha = max(alpha, eval);
        }
    }else{
        generateBlackNonQuietMoves(board, moves);
        for(bitBoard &i : moves){
            int32_t moveEval = qSearch(i, depth + 1, true, alpha, beta);
            eval = min(eval, moveEval);
            if(alpha >= eval){
                break;
            }
            beta = min(beta, eval);
        }
    }
    return eval;
}
```

```
int32_t minimax(bitBoard &board, uint8_t depth, bool whiteTurn,
                uint8_t maxDepth, int32_t alpha, int32_t beta){

    if(depth >= maxDepth){
        return evaluation(board);
    }

    vector<bitBoard> moves;
    int32_t eval;
    moves.reserve(50);

    if(whiteTurn){
        if((maxDepth > depth + NMDEPTHREDUCTION) && (whiteKingCheck(board) == 0)){
            int32_t nullMoveValue =
            minimaxNoNM(board, depth + NMDEPTHREDUCTION,
                        false, maxDepth, alpha, beta);
            if(nullMoveValue >= beta){
                return nullMoveValue;
            }
        }

        generateWhiteMoves(board, moves);
        if(moves.empty()){
            return whiteKingCheck(board);
        }
        eval = NEGINF;
        for(bitBoard &i : moves){
            int32_t moveEval = minimax(i, depth + 1, false, maxDepth, alpha, beta);
            eval = max(eval, moveEval);
            if(beta <= eval){
                break;
            }
            alpha = max(alpha, eval);
        }
        return eval;
    }else{
        if((maxDepth > depth + NMDEPTHREDUCTION) && (blackKingCheck(board) == 0)){
            int32_t nullMoveValue =
            minimaxNoNM(board, depth + NMDEPTHREDUCTION,
                        true, maxDepth, alpha, beta);
            if(nullMoveValue <= alpha){
                return nullMoveValue;
            }
        }
        generateBlackMoves(board, moves);
        if(moves.empty()){
            return blackKingCheck(board);
        }
        eval = INF;
        for(bitBoard &i : moves){
            int32_t moveEval = minimax(i, depth + 1, true, maxDepth, alpha, beta);
            eval = min(eval, moveEval);
            if(alpha >= eval){
                break;
            }
            beta = min(beta, eval);
        }
        return eval;
    }
}
```

F Appendix

`r1bqkbnr/pppp1ppp/2n5/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R b KQkq - 0 3`	Ruy lopez
`r1bqkbnr/pppp1ppp/2n5/4p3/3PP3/5N2/PPP2PPP/RNBQKB1R b KQkq - 0 3`	Scotch
`rnbqkb1r/pppp1ppp/4pn2/8/2PP4/6P1/PP2PP1P/RNBQKBNR b KQkq - 0 3`	Catalan

G Appendix

```
    int minimax(bitBoard &board, uint8_t depth,
                bool whiteTurn, uint8_t maxDepth){

    if(depth == maxDepth){
        return evaluation(board);
    }

    vector<bitBoard> moves;
    int eval;
    moves.reserve(50);

    if(whiteTurn){
        generateWhiteMoves(board, moves);
        if(moves.empty()){
            return whiteKingCheck(board);
        }
        eval = NEGINF;
        for(bitBoard &i : moves){
            int moveEval = minimax(i, depth + 1, false, maxDepth);
            eval = max(eval, moveEval);
        }
        return eval;
    }
    else{
        generateBlackMoves(board, moves);
        if(moves.empty()){
            return blackKingCheck(board);
        }
        eval = INF;
        for(bitBoard &i : moves){
            int moveEval = minimax(i, depth + 1, true, maxDepth);
            eval = min(eval, moveEval);
        }
        return eval;
    }
}
```

H Appendix

Source code: https://github.com/KristianWasas/Kandi
Platform: Windows
Compiler: g++.exe (Rev6, Built by MSYS2 project) 11.2.0