**An investigation into effects of optimizations on the adversary search algorithm Minimax**

Research Question: How does search depth affect the extent to which optimizations on the Minimax search algorithm make the search more efficient, as applied to chess?

Subject: Computer Science [English]

Word Count: 3971

Contents

[1: Introduction 1](#_Toc58804995)

[2: The Minimax Algorithm 3](#_Toc58804996)

[3: Optimizations on Minimax 6](#_Toc58804997)

[3A: Alpha-Beta Pruning 7](#_Toc58804998)

[3B: Transposition Table 11](#_Toc58804999)

[3C: Iterative Deepening 13](#_Toc58805000)

[4: Effect of Depth on Increase in Efficiency 14](#_Toc58805001)

[4A: Experimental Setup 14](#_Toc58805002)

[4B: Projected Unoptimized Performance 15](#_Toc58805003)

[4C: Experimental Data 15](#_Toc58805004)

[4D: Analysis 16](#_Toc58805005)

[5: Conclusion 20](#_Toc58805006)

[5A: Experimental Evaluation 20](#_Toc58805007)

[5B: Conclusion 21](#_Toc58805008)

[6: Works Cited 22](#_Toc58805009)

[7: Appendix 24](#_Toc58805010)

[7A: Project Code 24](#_Toc58805011)

[7B: Raw Data 31](#_Toc58805012)

# 1: Introduction

Chess is a game which has long fascinated mankind. From the hands of combinatorial geniuses in the past such as Mikhail Tal to present-day grandmasters like Magnus Carlsen, chess has repeatedly demonstrated its ability to offer fascinating lines beyond imagination. I am personally an avid fan of chess – I have been playing on my school’s chess team since my freshman year. I also regularly practice on chess.com, having played more than 14,000 games on the website over the years.

With the advent of using computer engines to play chess in the 20th century, chess analysis took a new turn as programmers sought to beat the game with artificial intelligence. The classic approach which soon surfaced is an application of Minimax – a search algorithm which finds the best move by generating a tree with alternating layers of minimum and maximum nodes (Steen 345). Minimax is at the heart of classic algorithms around adversarial games, from tic-tac-toe to checkers to chess.

Due to computational limits imposed by the sheer number of possibilities in chess, a variation of Minimax – depth-limited Minimax, where engines halt at a certain depth – is applied instead. On top of that, many optimizations have been made on the algorithm to prune unpromising lines in order to reduce computational burden, resulting in much faster engines than ever before (Best Schools).

Stockfish, the highest rated open-source chess engine today, uses an extremely enhanced variant of Minimax for decision making (Rin). Stockfish’s use of Minimax demonstrates how powerful and foundational the search algorithm is, especially when optimized correctly.

In this paper, I will investigate the question: How does search depth affect the extent to which optimizations on the Minimax search algorithm make the search more efficient, as applied to chess?

Throughout this paper, the word “ply” will be used to signify one move by one player. “Search depth” indicates how many plies ahead the search will look, while the depth of a position is the number of plies by which the given position is ahead of the root position.

# 2: The Minimax Algorithm

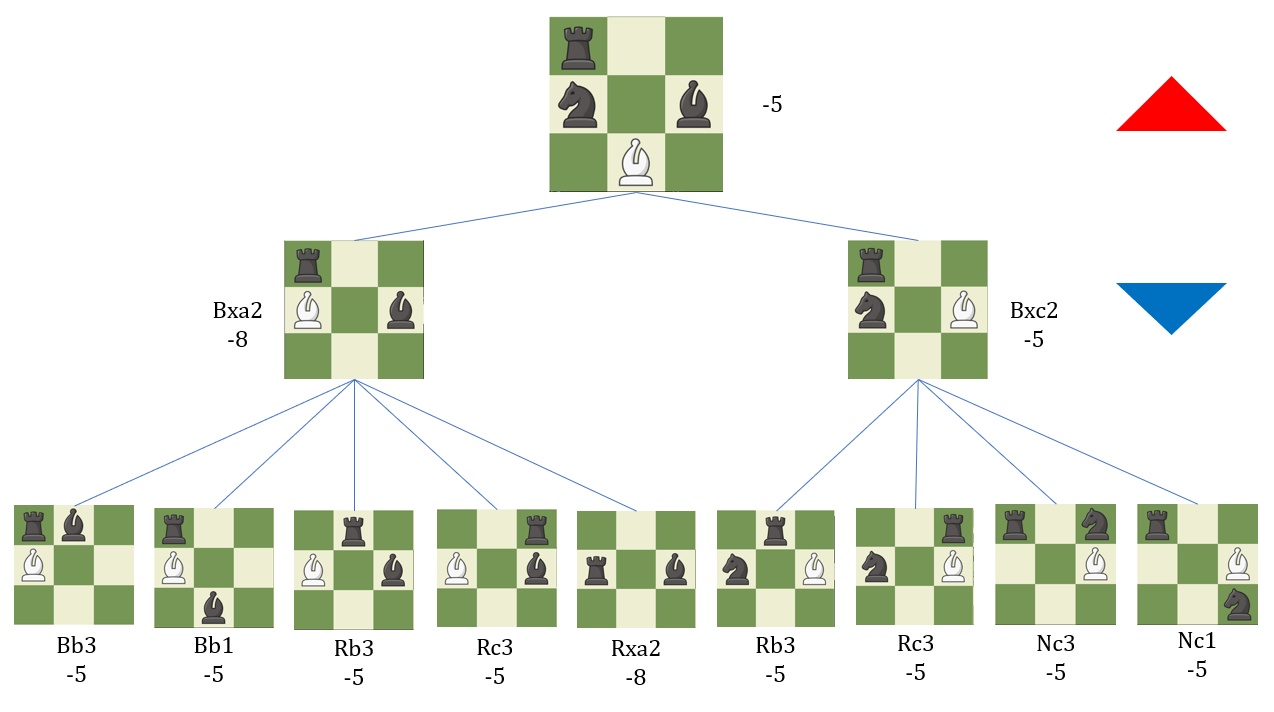
Before proceeding, I will first discuss how the Minimax algorithm works and how it can be applied to chess. With Minimax, one player is the “minimizer”, while the other is the “maximizer”. Every position in the game has a “value”, which is a measure of which side is more probable to win. The value is positive if the maximizer has the advantage, and negative if the minimizer is winning. When making moves, the minimizer tries to minimize the value, while the maximizer tries to maximize it. In chess, the maximizer is conventionally white, while the minimizer is typically black.

Depth-limited Minimax finds the “best” move by creating a recursion tree of all possible moves up to a certain depth. It then assigns a value to each position at the endpoints of the tree, known as leaf nodes, using an evaluationfunction. Starting from the leaves and going up, each parent node will be assigned a value based on either the minimum or maximum value out of its child nodes. If it is the minimizer’s turn, the smallest value out of the child nodes will be assigned to the parent position, and if it is the maximizer’s turn, the maximum child node value will be used (Hartikka). The “best move” in any position is the child node whose value the position inherits.

This concept will be illustrated in an example position (Hartikka). For simplicity, let the example use a 3x3 chessboard and go to a depth of two plies. The evaluation function is simple – the value of a leaf node is the sum of the piece values according to table 2.1.

|  |  |
| --- | --- |
| White knight: 3 | Black knight: -3 |
| White bishop: 3 | Black bishop: -3 |
| White rook: 5 | Black rook: -5 |

*Table 2.1: Piece values in example evaluation heuristic*



*Diagram 2.2: Sample Minimax tree with all nodes evaluated*

In diagram 2.2, the top position is the current position and it is white’s turn to move. All possible variations down to two plies deep are depicted as branches on the tree. The leaves of the tree, all nodes at 2 ply, are given a value based on table 2.1.

At 1 ply, it is black’s turn to move. Because black is the minimizer, the optimal move is the child node which has the lowest possible score. Hence, each of these two nodes at a depth of 1 will take on the lowest value out of all its direct child nodes. This means Bxa2 will have a value of -8, while Bxc2 is -5.

Returning to the current position, it is white’s turn to move. Because white is the maximizer, the child node with the highest possible score is the best move. Since Bxc2 yields a value of -5 while Bxa2 yields -8, Bxc2 is the optimal move. Hence, the current board position will also have an evaluation of -5.

However, while this example demonstrates how Minimax works, it is a huge simplification of the algorithm. A 3x3 board is used here, while a normal chessboard is 8x8. Additionally, this example stopped at a depth of 2 ply; engines such as Stockfish typically go to depths of 20 ply or more. In this example, there were only 9 leaf nodes to consider. However, in a normal chess game, there are 400 possible ways for both players to play their first turn. After just the second pair of turns, there are 197,742 lines. Understandably, this basic version of Minimax inevitably faces huge computational challenges when applied to real games. This is where optimizations come in to enhance the efficiency of the algorithm.

# 3: Optimizations on Minimax

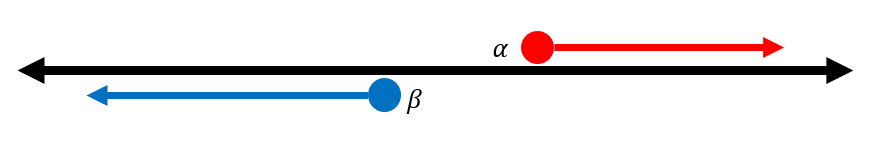
Optimizations are enhancements that make an algorithm more efficient. Hence, before continuing to discuss optimizations, I would first like to define the word “efficient”. With chess engines, there are mainly two types of efficiency – Node per Second (NPS) and depth/time. NPS is a measure of how much time is needed to evaluate a position. Despite discrepancies on what is considered a “node”, it is heavily based on the evaluation function. This paper is about the efficiency of the Minimax algorithm, which is the search algorithm; hence NPS efficiency will not be considered. On the contrary, I will be calculating depth/time efficiency, which is how deep an engine could go in a given time (Chessify). Depth/time efficiency is more suited for our needs because it demonstrates the ability of the search algorithm to find the best move efficiently. Running on the assumption that all nodes take equal time to evaluate, which is approximately true most of the time, how many nodes need to be evaluated in a position is a good indicator of depth/time efficiency. Therefore, optimization efficiency in this paper will be gauged by comparing the number of nodes that need to be evaluated in a given position before and after optimization.

Next, I will detail how some of the most well-known optimizations on the Minimax search work. There are hundreds of possible optimizations, so I will only focus on the most common ones – namely, alpha-beta pruning and the accompanying enhancements of transposition tables and iterative deepening. In section 4, I will investigate their effect on Minimax efficiency.

## 3A: Alpha-Beta Pruning

The most common optimization on Minimax is alpha-beta pruning. Alpha-beta pruning allows Minimax to avoid searching subtrees of moves which will not be considered. It does this by passing along two bounds – alpha () and beta () – which restrict the set of possible solutions based on what has been seen. is the lower bound of maximizing values, and is the upper bound of minimizing values (CS 161 Notes).

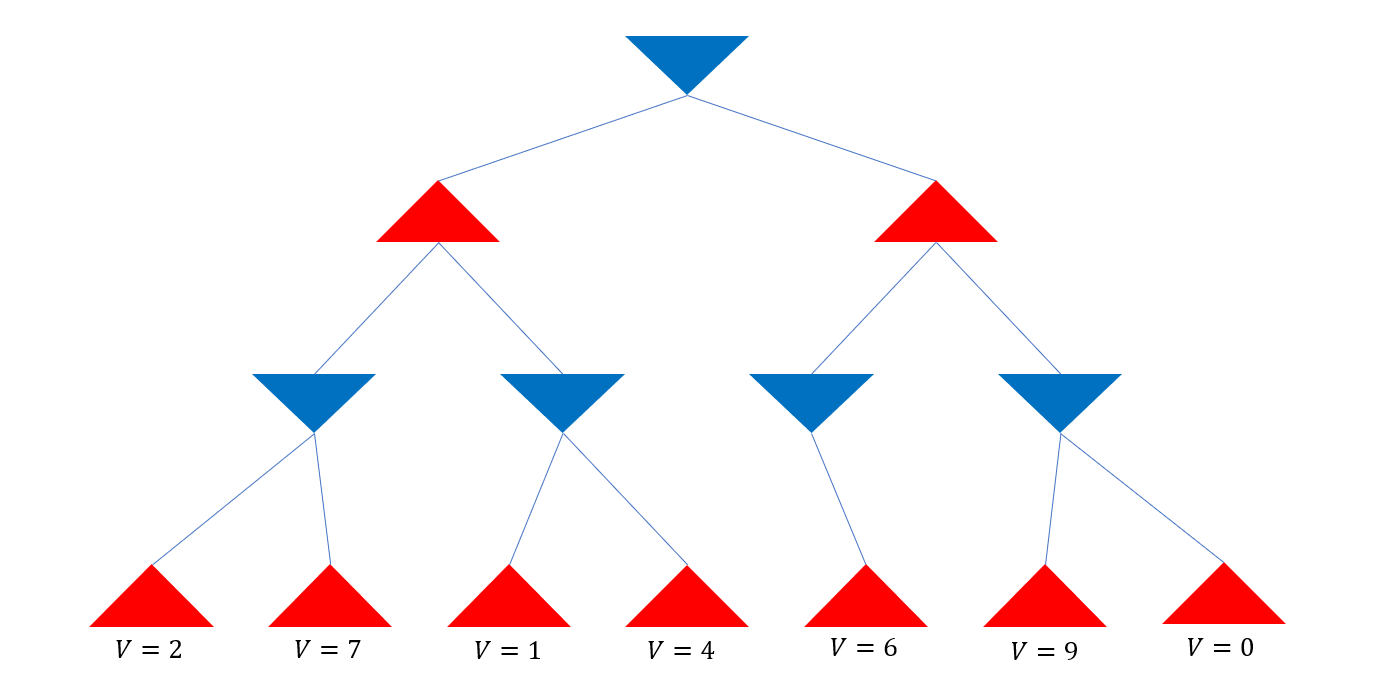
Any position which has a chance of being considered an optimal move for either side works only if its value satisfies for the bounds and passed to it. As the Minimax search progresses to greater depths, and get closer together. At some point during evaluation, the bounds of and may shift such that there is no more overlap in their ranges, as depicted in diagram 3A.1.



*Diagram 3A.1: At some point, there is no more overlap between and*

At this point, the engine prunes – or stops considering – the node and its children, because there is no way it or any positions that branch from it could be an optimal move for either side.

To demonstrate how alpha-beta pruning works on Minimax, the search tree in diagram 3A.2 with a depth of three plies will be used as an example (CS 161 Notes). Here, the evaluations of all leaf nodes are shown to depict the full tree. In practice, however, pruned nodes would not be evaluated.



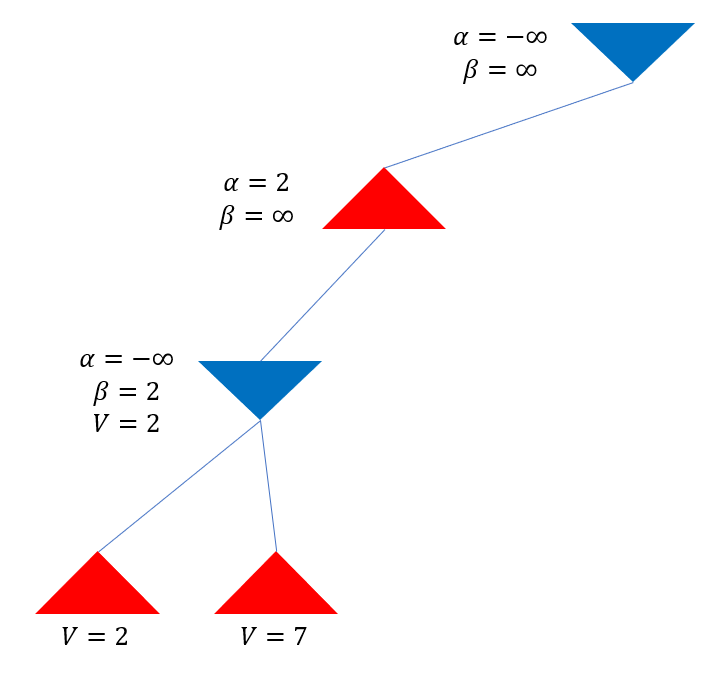
*Diagram 3A.2: Sample Minimax search tree*

Let Minimax traverse the tree from left to right. It starts from the root node with bounds of and , because the node can have any value. Next, it generates children until it reaches a leaf node at 3 ply, passing the bounds along the way.

When the value of the first child, 2, is passed to its minimizing parent node at 2 ply, it can be known that the parent value must satisfy because it takes the minimal child value. Hence, . This means the value of the parent node .

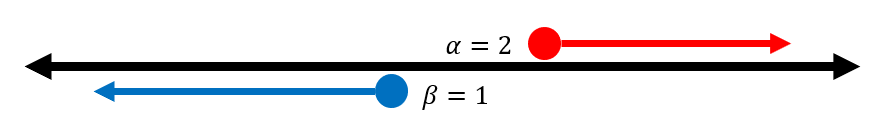
The next node at 3 ply has a value of 7. Because the parent is a minimizing node and , this child is ignored. Because all children of the parent node have been evaluated, the node at 2 ply takes on the value of .

The next parent at depth 1 ply is a maximizing node, so its value must satisfy because its first child has a value of 2 and it takes the maximum child value. Hence, , as depicted in diagram 3A.3.



*Diagram 3A.3: and bounds are passed up*

Two more children are generated to reach a node at desired depth, which is evaluated to have a value of 1. Since its parent at depth 2 ply is a minimizing node, the value of this node satisfies , so it is assigned . However, note that the node has also inherited the value , meaning . This is portrayed graphically in diagram 3A.4.

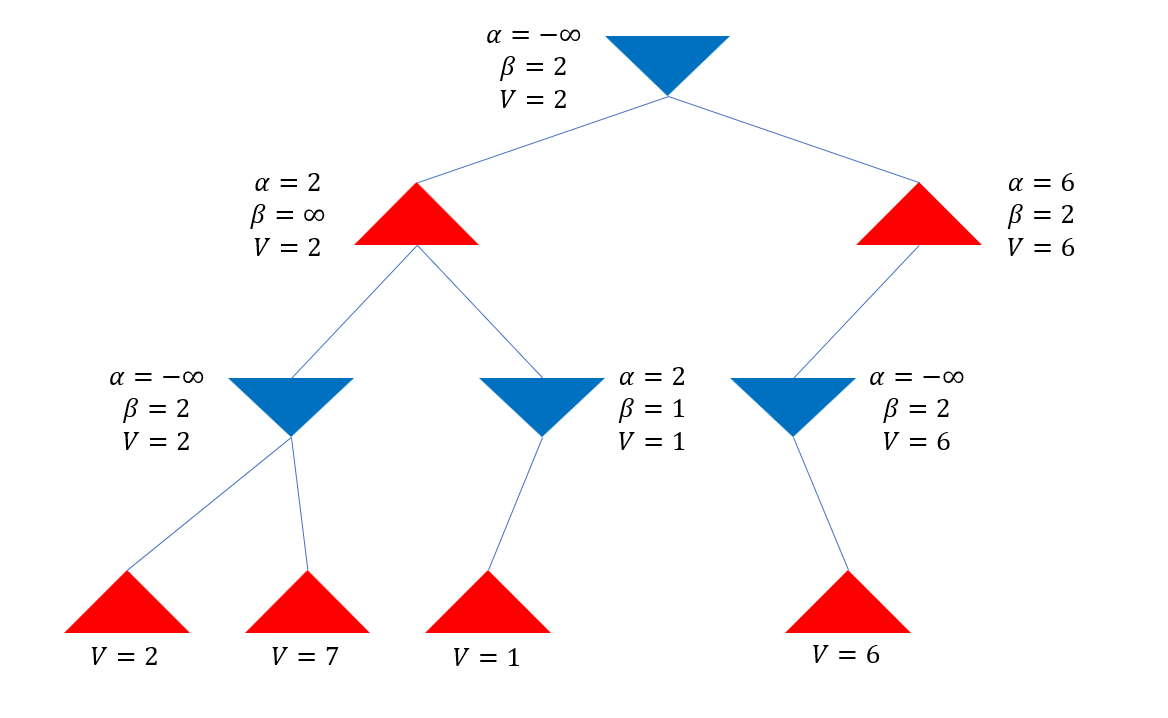


*Diagram 3A.4: When , the node can be pruned*

There is no more overlap between the ranges bound by and . Hence, this node and any of its children cannot possibly be in a sequence of optimal moves. Because this node will never need to be considered, the rest of its children are ignored and a value of 1 is returned to its parent (the bound changed). This is called a cutoff.

Next, the 1 ply maximizing node will pass the maximum child value of 2 to the root position. Because the root position is a minimizing node, it is given , which it passes to the second child at 2 ply. At the next leaf node, an evaluation of 6 is obtained; this is the only child node of its parent, so the parent at 2 ply takes on a value of 6 too.

Because the parent at 1 ply is a maximizing node and a value of 6 is passed up from 2 ply, . However, because the bound is passed down from the root position, . Because of this, the parent at 1 ply and all its branches may be pruned from further consideration. Completing the search, the final tree in diagram 3A.5 is obtained, with a root position evaluation of .



*Diagram 3A.5: Final alpha-beta search tree*

From this example, note that alpha-beta pruning may cut away huge chunks of the search tree – but only if the engine searches the better moves first. Nonetheless, in huge trees such as those used to evaluate typical chess positions, it is likely that alpha-beta pruning can cut down a lot of unneeded calculations.

Most other Minimax enhancements work on improving alpha-beta by letting it prune as much as possible. There are 3 main principles of further optimization:

1. Improving the order in which nodes are searched so that the best come first
2. Reusing information from prior parts of the search
3. Shrinking the size of the window between and more readily

Principle 3 is based on enhancements such as the aspiration window or minimal window, which aren’t easy to implement and may have controversial effects on optimization (Shams 192). Therefore, for simplicity, this paper and its experiment will only utilize optimizations based on principles 1 and 2.

## 3B: Transposition Table

Another optimization on the Minimax algorithm is the use of transposition tables. Transposition tables help avoid redundant searching of repeat positions and also optimize branch search order by reusing information.

In a chess game, not all positions are necessarily distinct. For example, the position shown in diagram 3B.1, known as the Italian Game, can be achieved in several ways from the starting position: *1. e4 e5 2. Nf3 Nc6 3. Bc4*, *1. e4 Nc6 2. Nf3 e5 3. Bc4*, etc.



*Diagram 3B.1: This transposition can be reached through various sequences of moves*

Previous calculations can be capitalized on through storing evaluated positions in memory, typically taking the form of a hash table. In this table, seen positions are recorded, along with their evaluation and the depth at which they attained that evaluation (Pettersson).

Whenever a new position is encountered, the table is polled to see if the position has been previously searched. If it has, there are 2 possible cases – either the node has been searched to a depth at or above the desired depth, or the node has been evaluated at a depth lower than the desired depth. If the position has already been evaluated to a depth at or above the desired depth, the evaluation from the hash table can simply be taken and used as the new position’s evaluation, which avoids a re-search of its entire subtree.

Otherwise, if the position’s evaluation is below desired depth, the evaluation can give insight into approximately how good the move is. This information can be used to start the search from positions with the highest heuristic evaluations, allowing to prune away much more branches (Chessprogramming Transposition).

## 3C: Iterative Deepening

A basic time management strategy used with alpha-beta pruning is iterative deepening. Iterative deepening makes it so that regardless of the time constraint, the algorithm can still come up with some sort of evaluation or “best move”. Additionally, iterative deepening also facilitates the search by enhancing move ordering.

Iterative deepening works by first searching to a depth , which is above the desired depth. These nodes at depth are evaluated as if they were leaf nodes, and the nodes at depth are then searched in descending order of their values at depth , so that the “best” moves may be searched first. In a time-constrained scenario, if time runs out, the engine can resort the best move found so far.

Otherwise, based on the best moves at shallower plies, the engine can reorder the moves at deeper plies to first search the most promising-seeming branches. This reordering of moves typically tends to pay off, as many inferior subbranches are pruned (Schaeffer 4).

# 4: Effect of Depth on Increase in Efficiency

## 4A: Experimental Setup

To evaluate the effect of depth on the extent to which optimizations make Minimax more efficient, I will compare the number of nodes evaluated per move under the optimizations mentioned in section 3 over varying depths with the projected number of nodes which would have been evaluated without those optimizations.

Tests will be run with a custom chess engine written in Java, the source code of which may be found in the appendix. Experimental setup will be as follows:

1. The engine will repeatedly play itself at various depths with the optimizations described in section 3
2. The main efficiency indicator – nodes evaluated per position – will be noted after each move, along with the other indicators of cutoffs and transposition cutoffs
3. The indicators for each move in a game will be averaged and counted as a trial
4. Those trials will be averaged to obtain average values per each depth
5. The average per depth will then be compared to the projected number of nodes which would have been evaluated had there been no optimizations

In this experiment, everything except for the search depth will stay constant. The same search algorithm will be used between different depths. In this way, the relationship of different optimization indicators with search depth can be tracked.

## 4B: Projected Unoptimized Performance

First, it is vital to estimate the performance of the engine without optimizations. The average branching factor from a given chess position is 35 – that is, given a random chess position, it is likely that there are around 35 possible moves to the made for the next turn. Depending on the position or the stage of the game, this number may fluctuate, but it centers around 35 (Chessprogramming Branching).

Therefore, the number of nodes which have to be considered in a search at a depth of ply with no optimizations is approximately:

Note that is not simply , because that is only the projected number of leaf nodes which would be evaluated; non-leaf nodes have to be accounted for as well.

## 4C: Experimental Data

The engine played a total of 120 games with itself under optimized conditions, with 20 at each search depth from 1 to 6 ply. For each move, 3 indicators were recorded – nodes evaluated, cutoffs, and transposition cutoffs. Nodes evaluated is the main indicator of engine efficiency – the effect of optimizations will be measured by comparing the nodes evaluated at a certain depth with optimizations with the projected number of nodes that would have been evaluated without optimization.

The other two indicators can be used to explain the trend demonstrated by the nodes evaluated indicator. cutoffs is the number of sub-branches which are pruned due to a value exceeding or bounds, and transposition cutoffs is the count of non-leaf nodes in the search tree which have been evaluated at equivalent or greater depth and can simply have their value pulled from a table. These are the two ways in which branches from the search tree may be removed from consideration, which decreases the number of nodes that need to be evaluated.

Table 4C.1 shows the average data from the indicators for each of 6 varying depths.

|  |  |  |  |
| --- | --- | --- | --- |
| Depth (ply) | Nodes Evaluated | Cutoffs | Transposition Cutoffs |
| 1 | 22.1 | 0.0 | 0.0 |
| 2 | 854.4 | 0.0 | 3.7 |
| 3 | 2687.9 | 743.0 | 3.4 |
| 4 | 25607.8 | 2126.9 | 99.4 |
| 5 | 77066.2 | 19338.2 | 481.9 |
| 6 | 1136111.4 | 129470.3 | 6941.8 |

*Table 4C.1: Average data from all trials*

## 4D: Analysis

To evaluate how depth affects the effects of optimizations, the average number of nodes evaluated per depth with optimization will be taken as a fraction of the projected number of nodes that would have been evaluated.

At a depth , the percent of nodes evaluated expressed in terms of the average number of nodes evaluated with optimizations at that depth and the projected number of evaluated nodes without optimization will hence be:

Running the average nodes evaluated for each depth through this function, table 4D.1 and graph 4D.2 are obtained.

|  |  |
| --- | --- |
| Depth (ply) | (%) |
| 1 | 63.216 |
| 2 | 67.812 |
| 3 | 6.090 |
| 4 | 1.658 |
| 5 | 0.143 |
| 6 | 0.060 |

*Table 4D.1: Percent of projected nodes evaluated in relation to depth*

First, note that the percent of projected nodes evaluated without optimization at 1 ply makes little sense. Looking at the data, there are no cutoffs or pruned branches at all at 1 ply, meaning that the number of nodes searched at 1 ply should be roughly equivalent to the projected number of nodes that would have been searched had there been no optimizations. A possible explanation for this discrepancy is that at 1 ply, the engine is moving with practically no tactics in mind, so it tends to spend more time in the endgame, where the branching factor is often lesser than 35 because there are less pieces on the board.

Nonetheless, it is apparent that the various optimizations do not prune away a constant percentage – the number of nodes evaluated as a percentage of decreases in a roughly exponentially decreasing manner as depth increases. In fact, if the first data point is removed, the resulting curve almost perfectly fits an exponential decay function, as demonstrated in graph 4D.3.

Such a conjecture is supported by the number of and transposition cutoffs as depth varies, depicted in graph 4D.4.

Note that both the number of transposition cutoffs and cutoffs increase as search depth increases. Since cutoffs eliminate a chunk of the tree, a cutoff at a random position in a tree of arbitrary depth is likely to eliminate a similar proportion of the tree as a cutoff somewhere in another tree of arbitrary depth . Hence, because the number of cutoffs increases with depth, it comes as no surprise that cutoffs prune away greater proportions of the tree at greater depths.

# 5: Conclusion

## 5A: Experimental Evaluation

This experiment had its merits, but also had drawbacks. One strength is that because the engine played full chess games against itself, the average number of nodes evaluated accounted for all stages of the game, from the opening to the endgame. Additionally, I had sufficient raw data – 6 data points and 20 trials each – to demonstrate a trend that can support a claim. The additional indicators – averages of cutoffs and transposition cutoffs – also logically supported my conjecture.

However, a weakness of this paper is that it failed to account for individual optimizations and how their value is affected by depth. When running the optimized engine, all optimizations were toggled on – pruning, transposition tables, and iterative deepening for move prioritization. Hence, the conclusion reflects how their combined effort at optimization is affected by depth, which fails to address trends provoked by each of these optimizations individually. Another weakness is that the data collected had a point that made little sense, thus decreasing the credibility of the data and the conclusion drawn from it. This could be attributed to the behavior of engines with low search plies, or it may be indicative of botched search algorithm code.

If the experiment could be redone, I would run the engine at greater search depths to collect more data points. I would also deduce the projected number of nodes without optimization experimentally rather than from a book value, so that my experiment is more self-consistent.

## 5B: Conclusion

In this paper, I first introduced the Minimax algorithm and explained algorithmic optimizations such as pruning. Next, I collected and analyzed data from an experiment in which a custom engine played itself at various search depths. I found that as search depth increases, the ratio of evaluated nodes to projected evaluated nodes decreases in a roughly exponential manner. *Therefore, search depth increases the extent to which optimizations on the Minimax search algorithm make the search more efficient, as applied to chess.*

With this knowledge, we obtain better understanding of what it takes to build optimized adversarial engines. Because the concept of Minimax – finding the best move given optimal play from the opponent – is at the core of adversarial networks, data concerning Minimax and its pruning techniques can be applied to a variety of other adversarial settings. For example, the Minimax GAN Loss is an important concept in generative adversarial networks (Brownlee). Minimax can also be applied to economics, where people have to anticipate optimal decision making by others when choosing their own course of action (Economicsdiscussion). Perhaps one day, if Minimax optimizations become sufficiently developed, it may be feasible to write engines which traverse entire search trees and never lose. When applied to other fields, such a breakthrough will have tremendous implications on the way humans interact and make decisions from then on.

# 6: Works Cited

Best\_Schools. (2020, March 23). A Brief History of Computer Chess. Retrieved August 30, 2020, from <https://thebestschools.org/magazine/brief-history-of-computer-chess/>

Branching Factor. (n.d.). Retrieved October 14, 2020, from <https://www.chessprogramming.org/Branching_Factor>

Brownlee, J. (2019, July 12). A Gentle Introduction to Generative Adversarial Network Loss Functions. Retrieved December 14, 2020, from <https://machinelearningmastery.com/generative-adversarial-network-loss-functions/>

Chessify. (2019, November 23). NPS vs Time-to-depth: What you should look at when analyzing with Stockfish. Retrieved August 15, 2020, from <https://chessify.me/blog/nps-vs-time-to-depth-what-you-should-look-at-when-analyzing-with-stockfish/>

CS 161 Recitation Notes - Minimax with Alpha Beta Pruning. (n.d.). Retrieved September 2, 2020, from <http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html>

Hartikka, L. (2017, March 15). A step-by-step guide to building a simple chess AI. Retrieved August 28, 2020, from <https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/>

N. (n.d.). 4 Strategies of the Game Theory – Explained! Retrieved December 14, 2020, from <https://www.economicsdiscussion.net/game-theory/4-strategies-of-the-game-theory-explained/3825>

Pettersson, J. (1970, January 01). [Guide] Transposition tables. Retrieved October 1, 2020, from <http://mediocrechess.blogspot.com/2007/01/guide-transposition-tables.html>

Rin, A. (2015, May 14). How Stockfish Works: An Evaluation of the Databases Behind the Top Open-Source Chess Engine. Retrieved August 30, 2020, from <http://rin.io/chess-engine/>

Schaeffer, J. (n.d.). The History Heuristic and Alpha-Beta Search Enhancements in Practice (Tech.). Retrieved from <https://pdfs.semanticscholar.org/b4d2/cf76e4c42b9325b52aac45d61e80a01de77b.pdf>

Shams, R., Kaindl, H., &amp; Horacek, H. (n.d.). Using Aspiration Windows for Minimax Algorithms [Scholarly project]. Retrieved from <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.1590&rep=rep1&type=pdf>

Steen, L. (1975). Computer Chess: Mind vs. Machine. Science News, 108(22), 345-350. Retrieved September 18, 2020, from <http://www.jstor.org/stable/3960294>

Transposition Table. (n.d.). Retrieved September 1, 2020, from <https://www.chessprogramming.org/Transposition_Table>

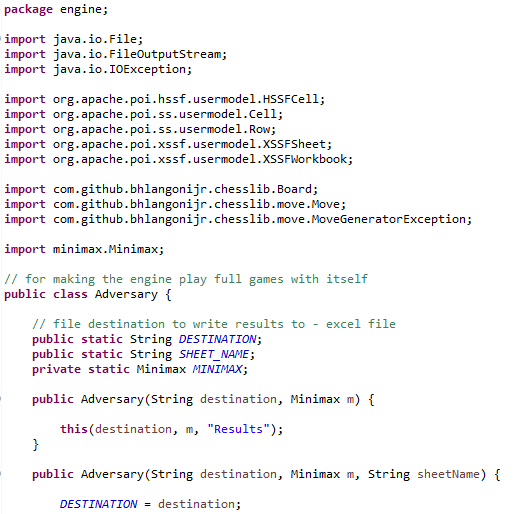
# 7: Appendix

## 7A: Project Code

/engine/Index.java



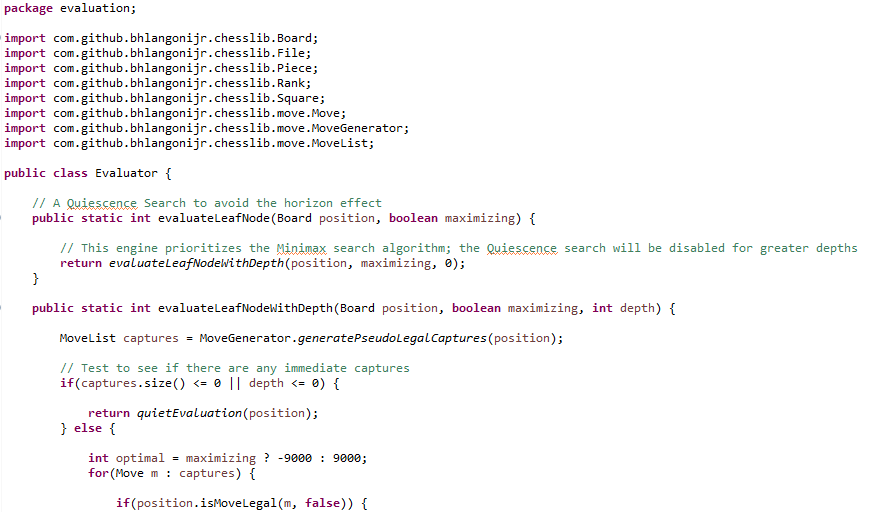
/engine/Adversary.java

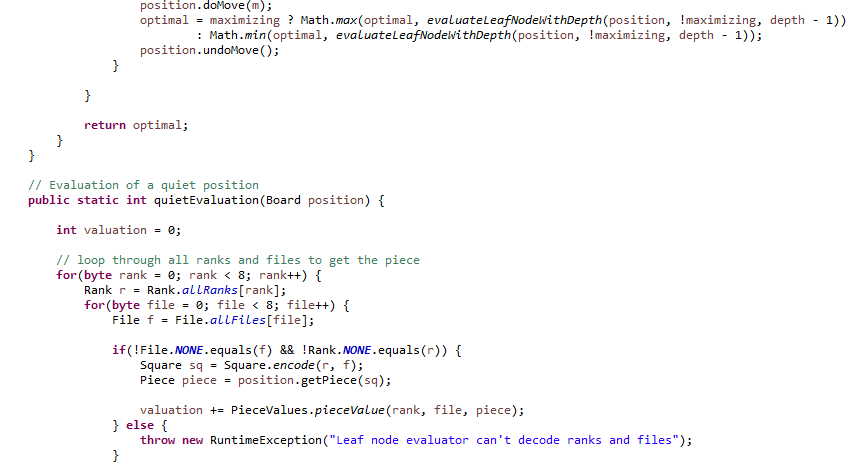


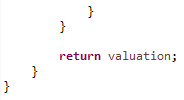




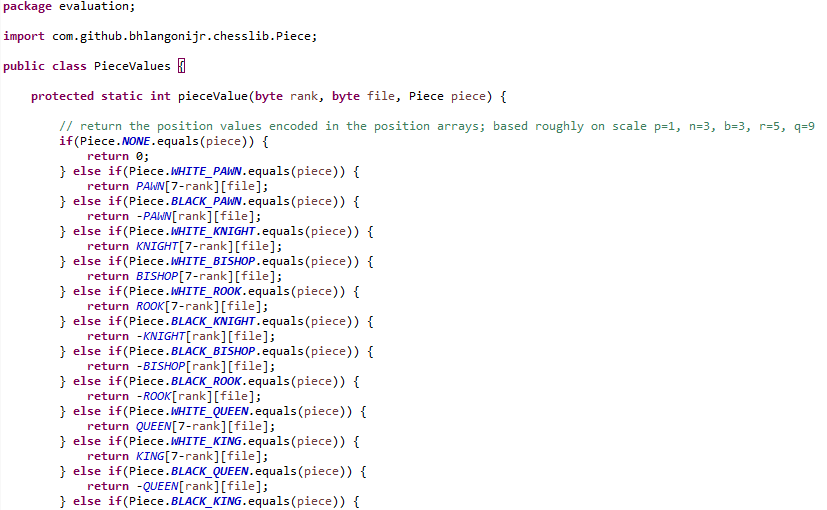
/evaluation/Evaluator.java

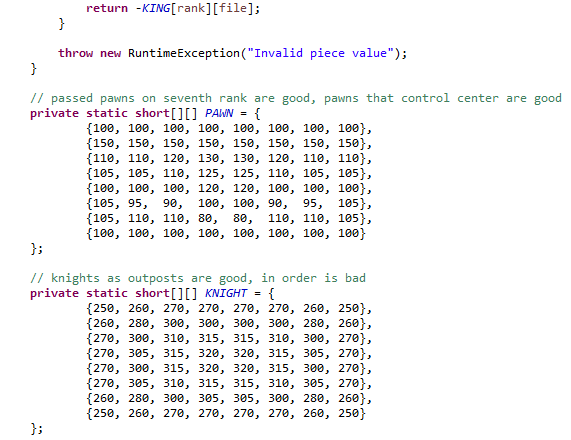


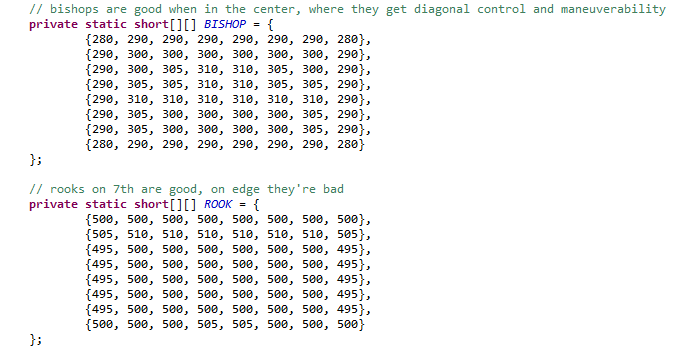


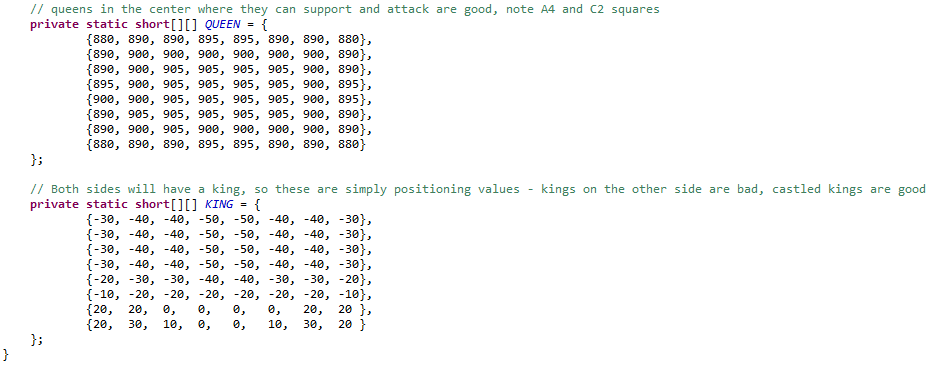


/evaluation/PieceValues.java

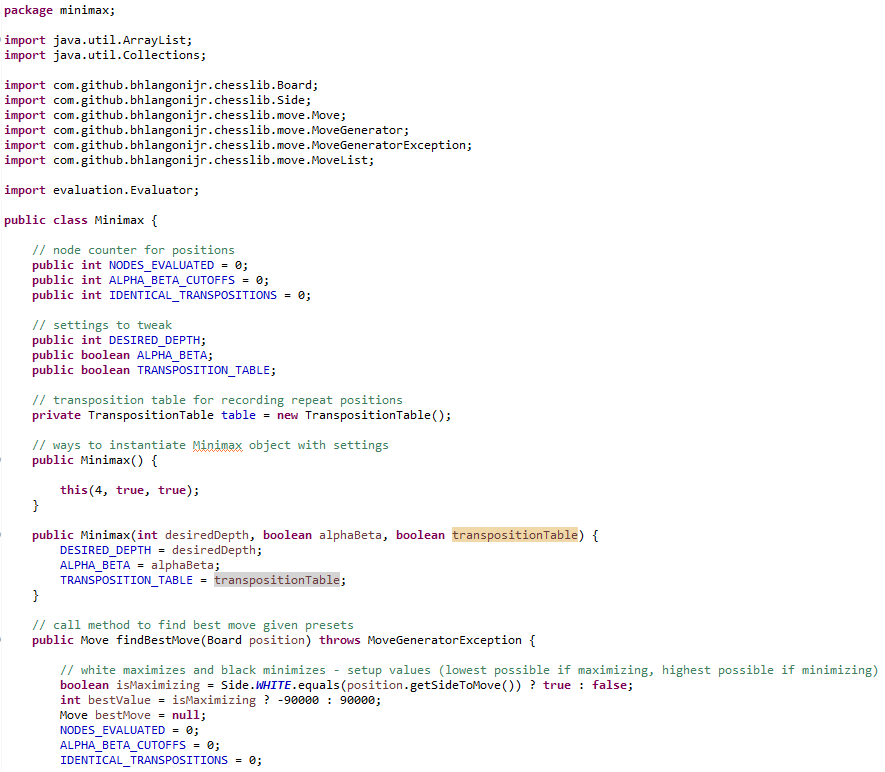


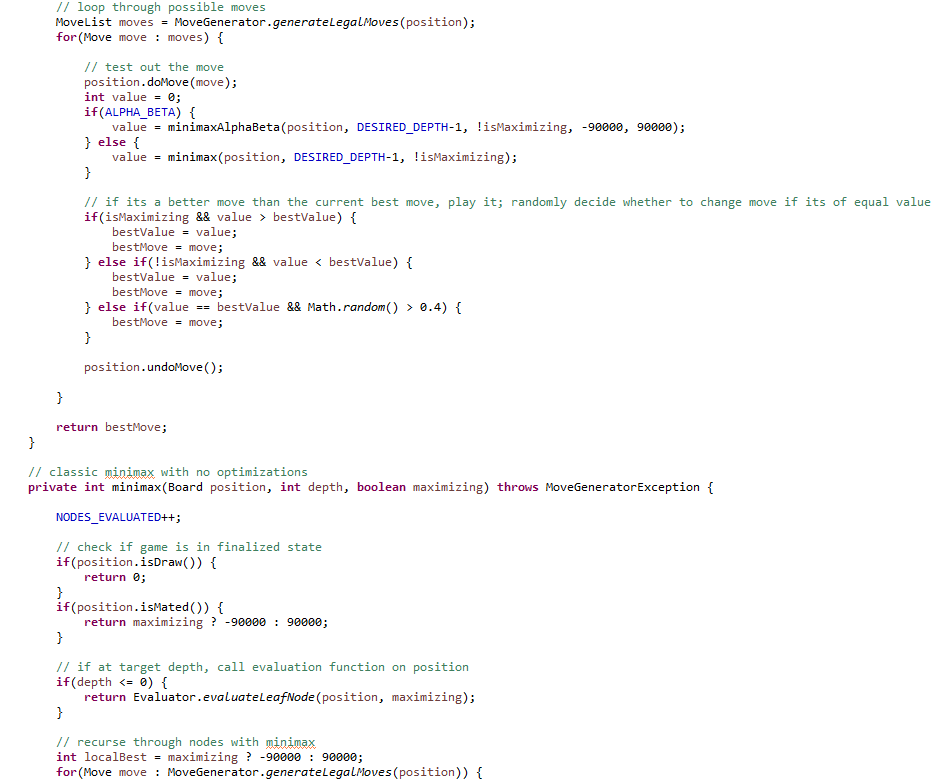


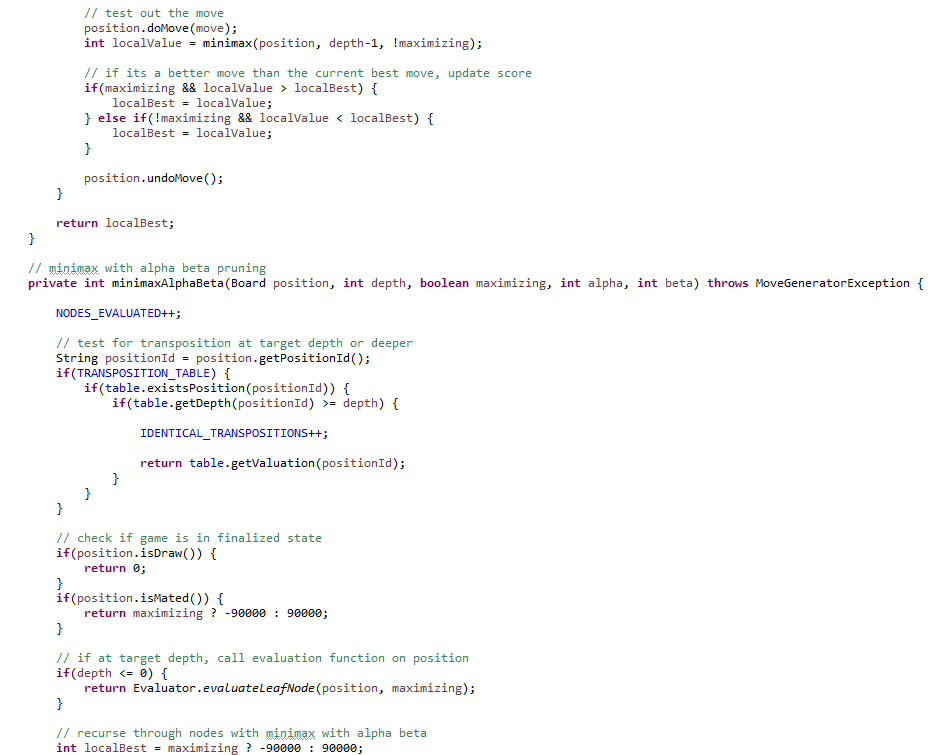




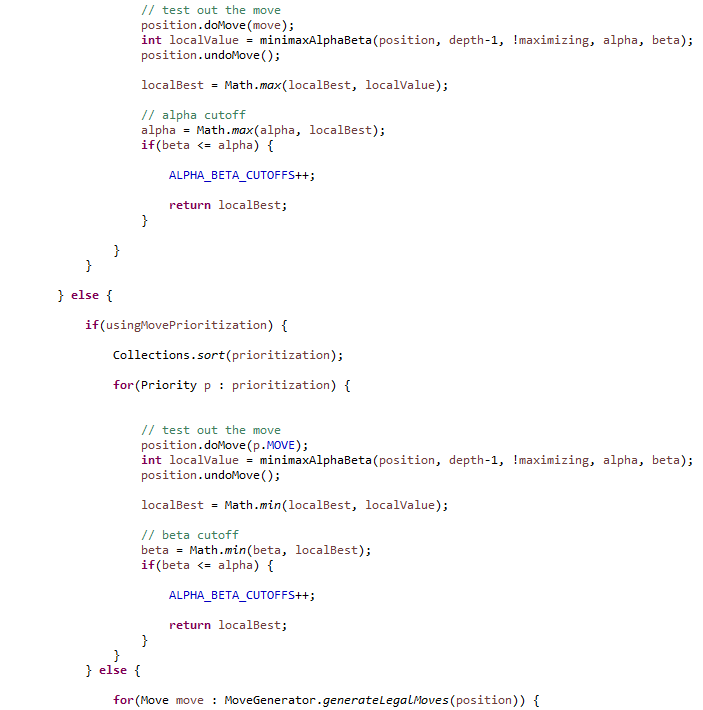
/minimax/Minimax.java





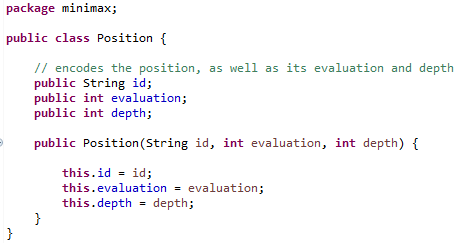




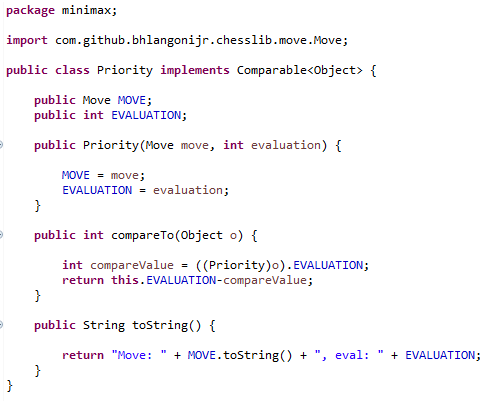




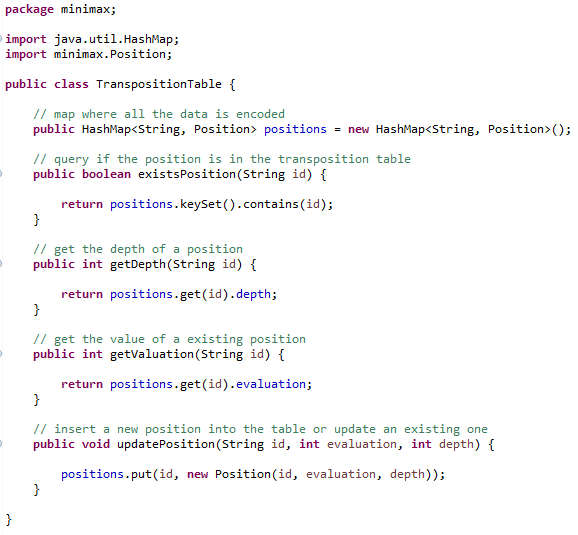
/minimax/Position.java



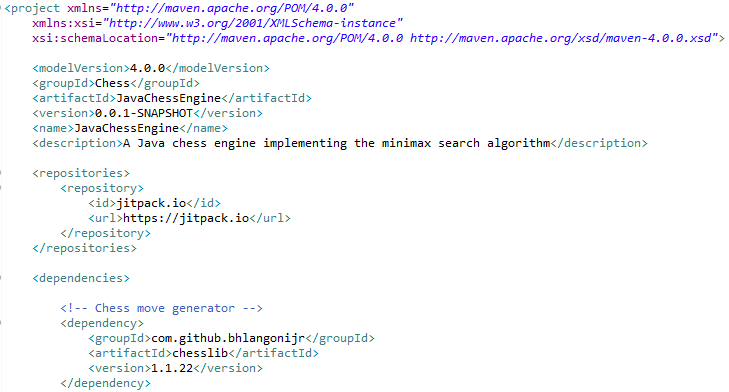
/minimax/Priority.java



/minimax/TranspositionTable.java



pom.xml





## 7B: Raw Data

A total of 120 trials were run – 20 each for depths of 1, 2, 3, 4, 5, and 6 ply. Each trial consisted of a full game that the engine played against itself.

The results were written to Microsoft Excel and then automatically averaged for each trial – the number of nodes evaluated, cutoffs, and transposition cutoffs for each move were averaged over the entire game.

Those average values for separate trials were then collected and placed into the table below for data processing.

