BACHELOR'S THESIS 2021

Evaluating Heuristic and Algorithmic Improvements for Alpha-Beta Search in a Chess Engine

Henrik Brange

ISSN 1651-2197
LU-CS/HBG-EX: 2021-08

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

Evaluating Heuristic and Algorithmic Improvements for Alpha-Beta Search in a Chess Engine

LUND UNIVERSITY
Campus Helsingborg

LTH School of Engineering at Campus Helsingborg
Department of Computer Science

Hbg-2021-08

Bachelor thesis:
Henrik Brange

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2021

Abstract

In making a chess engine using alpha-beta search there are many ways to reduce its execution time and thereby improve overall performance. One way is by using move ordering heuristics, which are heuristics that reduce the search space by attempting to make a rough estimation about which moves are most promising before starting a search of a board position. Another example is using a transposition table, which can reduce redundant computations. A third example is multithreading, which is an attempt to utilize the several cores found in most modern computer to improve the amount of information found per unit of time.

The purpose of this thesis is to investigate four different commonly used algorithms and heuristics to see how much they can reduce the execution time of a chess engine. These algorithms are the move-ordering heuristic known as MVV-LVA, a transposition table, iterative deepening and Lazy SMP parallel search.

A Java chess engine named KLAS was implemented and used to measure the impact of these four algorithms and heuristics. The largest impact came from the MVV-LVA move ordering heuristic which decreased average execution time by 68.5%. Second most significant was the transposition table, which led to an average decrease of 39.8% execution time. Lazy SMP and iterative deepening resulted in average execution time reduction of 33.1% and 28.7% respectively.

Keywords: chess engine, alpha/beta-search, minimax, move ordering, hash table, iterative deepening, Lazy SMP parallel search.

Sammanfattning

För att förbättra exekveringstiden av en schackdator som använder alfa/beta-sökning finns det många olika metoder. Ett sätt är att använda dragsorteringsheuristiker, det vill säga heuristiker som minskar sökutrymmet genom att göra grova uppskattningar av vilka drag som är mest intressant innan en sökning av en brädesposition påbörjas. Ett annat exempel är att använda en transpositionstabell, som är en datastruktur vars syfte är att spara information som funnits tidigare så att den kan återanvändas. Ett tredje exempel är flertrådning, som försöker att utnyttja de flera kärnor man finner i de flesta moderna processorer så att man kan undersöka fler möjligheter per tidsenhet.
Detta examensarbetes syfte är att undersöka fyra olika algoritmer och heuristiker som ofta används inom schackprogrammering för att se hur mycket de kan minska en schackdators exekveringstid. De algoritmer som undersökts är: dragsorteringsheuristiken Most Valuable Victim – Least Valuable Aggressor, en transpositionstabell, iterativt fördjupande och flertrådning enligt Lazy SMP.

För att kunna undersöka dessa tekniker implementerades en schackdator i Java vid namn KLAS. Mätningarna visade att MVV-LVA hade störst påverkan på exekveringstiden och orsakade en genomsnittlig minskning på 68,5%. Transpositionstabellen orsakade den näst största genomsnittliga minskningen vilken blev 39,8%. Lazy SMP-flertrådning och iterativt fördjupande minskade exekveringstiden med 33,1% respektive 28,7% i genomsnitt.

Keywords: chess engine, alpha/beta-search, minimax, move ordering, hash table, iterative deepening, Lazy SMP parallel search.

Acknowledgements

I would like to thank my supervisor Jesper Öqvist for being patient and providing useful insight and advice which has greatly benefitted this work in several ways.

Contents

Definition of Terms

The following terms are used in this thesis.

All-node – A node where no moves cause a cut-off nor increase alpha or beta.

Bitboard – A binary number where each bit represents the location of a piece.

Board position – A chessboard including its pieces and their locations.

Branching factor – The factor at which a search tree branches, or in other words, the number of possible moves on a given board position.

Cut-node – A node where a move caused a beta or alpha cut-offs.

Depth – The distance, in nodes, from the root node to the current node.

KLAS – The engine implemented as part of this thesis.

Line – A continuous series of moves in a game of chess.

Node – Represents a state in a game where a search algorithm is employed. In chess specifically, the nodes represent board positions.

Principle Variation – The specific line of moves that have led to the result returned by an alpha-beta search.

PV-node – A node where at least one move increased alpha or beta and no move caused a cut-off.

Search – A chess engine considering different moves and the positions they lead to.

Transposition Table – A hash table containing data regarding different board positions.

1 Introduction

Chess engines are computer programs used for playing and analysing the game of chess. The goal of a chess engine is to find the best moves in a game as quickly as possible, however the immense number of possible moves makes it impossible to fully analyse every chess position to always find the best move. Instead of exhaustively analysing every possible chain of chess moves, different optimizations are used to limit the amount of moves to be analysed and thereby achieving faster results.

This thesis investigates the performance of algorithmic and heuristic optimizations that are commonly used to improve the speed and accuracy of chess engines. These optimizations are MVV-LVA, Iterative Deepening, Transposition Tables and Lazy SMP multithreading. Evaluation of these optimizations is done in a custom chess engine named KLAS. We pose the following questions related to these algorithmic/heuristic optimizations:

In this thesis we hope to provide an answer to the following questions:

- RQ1 How much execution time can MVV-LVA save on average?
- RQ1.2 How much execution time can Transposition Tables save on average?
- RQ1.3 How much execution time can iterative deepening save on average?
- RQ1.4 How much execution time can Lazy SMP save on average?
- RQ2 How do each of these optimizations achieve a decrease in execution time?

This thesis was carried out on behalf of the department of Computer Science at LTH.

2 Background

The history of chess engines is almost as old as computers themselves. Around the time of the Second World War many advancements were made in the new scientific field of computer science. Between 1942 and 1945, Konrad Zuse developed a new programming language called Plankalkül which he used to describe a chess playing algorithm which can be considered the first chess playing program ever written [1].

In 1948, Alan Turing wrote the chess engine Turochamp for a computer that had not yet been invented. Turing tried to implement Turochamp on one of the computers available to him at the time, but unfortunately it failed as the computer did not have the computing power necessary. Turing used algorithms in Turochamp that are still used in chess engines today, such as: a minimax search method, variable search depth and an evaluation method which considers the mobility of chess pieces, king safety and the ability to castle. Turochamp was capable of playing chess on par with an inexperienced amateur [2].

Claude Shannon was also among the first to investigate the possibility of using computers for playing chess. In 1949, Shannon estimated the complexity of chess and determined that the total number of possible games of chess is at least 10^{120}. This number has since came to be known as the Shannon number [3].

The time required for a computer to fully analyse a game of chess can be estimated by dividing Shannon's number by the number of floating-point operations the computer can perform per second and multiplying by the number of such operations needed to analyse a single board position. To get a very optimistic approximation we can pretend that analysing a board position takes one floating-point operation.

The fastest supercomputer in the world as of July 2021[1] has a computing power of about 400 petaFLOPS, that is, $4 \cdot 10^{17}$ floating-point operations per second. If we use this number with the estimation method above, we get $10^{120} / (4 \cdot 10^{17})$ seconds or $8 \cdot 10^{85}$ billion years to fully analyse a chess game from the initial position. This shows that it is clearly beyond the capabilities of conventional computers to optimally play chess.

To figure out what moves are good and which ones aren't, chess engines perform something called a "search". Searching is when a chess engine considers what moves are available on the board as it is currently and then plays them on its internal board representation to then repeat the process. At some point, the engine must stop considering possibilities and instead evaluate the boards it has found so far as the possibilities in chess rise exponentially with each move.

[1] The supercomputer Fugaku located at the RIKEN Center for Computational Sciencce in Japan.

The ever-present lack of computing power gives rise to what is called the horizon effect in the fields of AI and game theory. When a chess engine searches on a board position, or in other words, considers different possible outcomes of the available moves in that position, the horizon effect must be considered.
It arises, in a turn-based game, when the computer searches down to a limited depth of possibilities and makes a move based on the information it has acquired from this search. There is always the possibility that, if the computer had searched even deeper, it would have found that this move is a bad choice. However, lacking this search depth, it proceeds with the move believing it to be a good choice. The horizon effect can never be fully eliminated in chess because the search depth and information of the engine is always limited until nearing the end of the game and so one must find algorithms and heuristics that can evaluate different chess positions and make good move even when it is uncertain what will happen later during the game.

2.1 Chess Engine Components

Chess engines can be described as consisting of the following main components:

- A **board representation** that describes the location of all chess pieces on a chessboard.
- A **move generator** that enumerates the available moves given a certain chess position.
- An **evaluation method** that determines a favourability score of a chess position for the two players.
- A **search method** that evaluates moves based on the possibilities that may unfold after they have been played.

The board representation contains all the relevant information about a chess board. It includes the pieces that are still on the board, their location as well as whether either of the two players has castled or has lost the ability to castle on the kingside and/or the queenside.

The evaluation method is a heuristic method that takes a board representation as a parameter and returns a numeric evaluation of how favorable the position is for both players. Chess is a two-player game, and so only a relative value is needed, as a position that is good for the player with the white pieces is equally bad for the player with the black pieces. Usually, a positive value corresponds to a position that is considered favorable for the player with the white pieces, while a negative value is considered favorable for the player with the black pieces and the value zero is considered equally favorable for both players.

The evaluation method is static, it does not consider what moves can be played nor the possibilities that may occur after any moves have been played as that is the responsibility of the search method. Traditionally evaluation was hand-crafted, perhaps by skilled chess players, but today many of the more powerful chess engines apply neural networks to perform the evaluation.

For a chess engine to be able to accurately determine how good a move is, it needs to know what may occur after the move has been played. Therefore, a search method is used for exploring possible future moves. The search method requires an internal board representation and evaluation method to do just that.

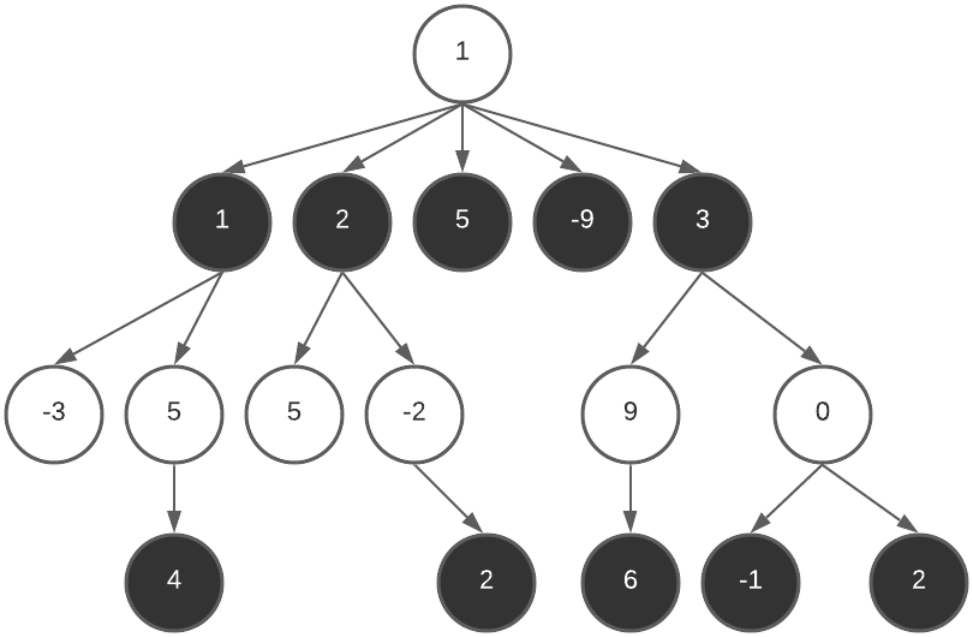


Figure 2.1.1. A search tree. In the analysis of engines playing two-player turn-based games, search trees are frequently used to represent the logical flow of the engine's calculations. In these trees the edges represent moves and the nodes are the resulting positions. The numbers inside of the nodes are the evaluations of the position by the engine.

In the implementation of the four main components of a chess engine, different heuristics and algorithms may be used. There are also other commonly used optimizations that do not fall under any of these four categories, that serve to make the engine more effective in its calculations and thus able to make better moves faster. One such optimization is move ordering, where moves are sorted in such a way that the most promising move is examined first.

Many optimizations for improving chess engine performance involve the concept of "pruning". Pruning is when a chess engine rejects moves that it deems unlikely to be played, to avoid unnecessarily dedicating time towards analysing them. Through smart heuristics, an engine can realize that certain moves are surely worse than others and thus not worth considering further. The more pruning that occurs, the faster an engine can reach a certain search depth; however, this also sometimes entails the risk of a good move being pruned despite warranting consideration.

One of the most common archetypical search methods for turn-based, two-player games is the minimax algorithm. A minimax algorithm is a recursive algorithm that seeks to alternately maximize and minimize the score of the game. This is based on the fair assumption that both players are trying to play their best moves, which for one player means maximizing the evaluation and for the other, minimizing it. The result it seeks is either the lowest evaluation the maximizing player is assured of or the highest evaluation the minimizing player is assured of, given a certain position. In this thesis, an enhanced form of a minimax algorithm is implemented and examined, specifically using an optimization called alpha-beta pruning.

Optimizations used to improve the speed and playing ability of chess engines can be divided into groups based on how they achieve this goal:

- Selectivity-based optimizations, also called pruning optimizations, that either decrease execution time by reducing the search space of the engine or that improve the playing ability of the engine by increasing the search space when necessary.

- Evaluation heuristics - Heuristics that seek to improve the engine's evaluation of positions based on chess concepts such as the value of pieces and their mobility, king safety etcetera.

- Space-time trade-off. Some optimizations involve using memory in order to save execution time, such as opening tables, allowing the engine to save time by playing predetermined moves at the start of a game.

- Algorithmic optimizations - Optimizations that improve performance through better algorithms that avoid unnecessary calculations. Examples include parallelization to make better use of multithreaded hardware.

3 The KLAS Chess Engine

The repository for the KLAS chess engine can be found on Github at:

https://github.com/henrikbr21/examensarbete21

This chapter explains the inner workings of the chess engine KLAS as well as any technical information that is necessary to understand this paper in its entirety. The KLAS chess engine contains the following critical components, each explained in its own section:

- A board representation in the form of so-called “bitboards”.
- A move generator, which generates the available moves given a certain position on the board.
- An evaluation method, which takes into whether either player is checked, the material for the two players as well as the location of different pieces.
- An alpha-beta search method.
- Check & checkmate methods, used in both the move generator and evaluation method.
- A makeMove function, that updates the internal board representation when different moves are considered.

In addition to the vital components listed above, KLAS employs four different optimizations for improving execution time:

- Move ordering according to MVV-LVA as well as ordering after the principal variation.
- Transposition tables that allow KLAS to remember the results from previous searches and reuse them.
- Iterative deepening, an optimization used for time management and to make better use of the move ordering optimizations.
- Lazy SMP Parallel search, an algorithm for parallelization of the alpha-beta search method.

3.1 Search Trees

In combinational game theory, game trees are graphs used to represent all the possible outcomes, and their intermediates, in games such as chess. As the complexity of chess is much too large for a full game tree to be made for it, search trees are commonly used in chess programming. These search trees are subsets of the game tree containing the possibilities that an engine considers during a search. These possibilities are called nodes, in chess specifically, they represent different positions on the board. Rather than being a well-defined data structure, they are graphs used to represent the logical flow of an engine.

3.1.1 The Minimax Algorithm

A minimax algorithm is a recursive depth-first algorithm that is used in two-player turn-based games. Given that the evaluation is such that a positive value is considered favorable for the player with the white pieces and a negative value is considered favorable for the player with the black pieces, the minimax algorithm can be defined as a co-recursive algorithm consisting of two methods where one seeks to maximize the evaluation and the other minimizes it. This algorithm is based on the idea of optimal play being achieved when a player presumes that their opponent is going to play the best possible moves. The minimax algorithm is shown below:

```
function minimax(board, depthLeft, playerColor)
    if depthLeft == 0 or checkmate(board) then
        return evaluate(node)
    end
    if player == white then
        value = NEGATIVE_INFINITY
        moves = generateMoves()
        for each move in moves do
            board.makeMove(move)
            value = max(value, minimax(board, depth - 1, BLACK))
        return value
    else (* black *)
        value = POSITIVE_INFINITY
        moves = generateMoves()
        for each move in moves do
        board.makeMove(move)
        value = min(value, minimax(board, depth - 1, WHITE))
    return value
end
```

At the start of the function is the end condition, where if the depth left to be searched is equal to ‘0’ the method returns the value given by the evaluation method for the node. If the end condition is not fulfilled, the method continues by finding the child nodes, i.e., the moves available, and then playing each of them on the internal board representation and then calling itself to repeat the process. When all the children of a node have been searched, the greatest of their evaluation is returned if the parent node is a board where it is the white player’s turn to move, otherwise if it’s the player with black pieces whose turn it is to move, the smallest value of the child nodes is returned.

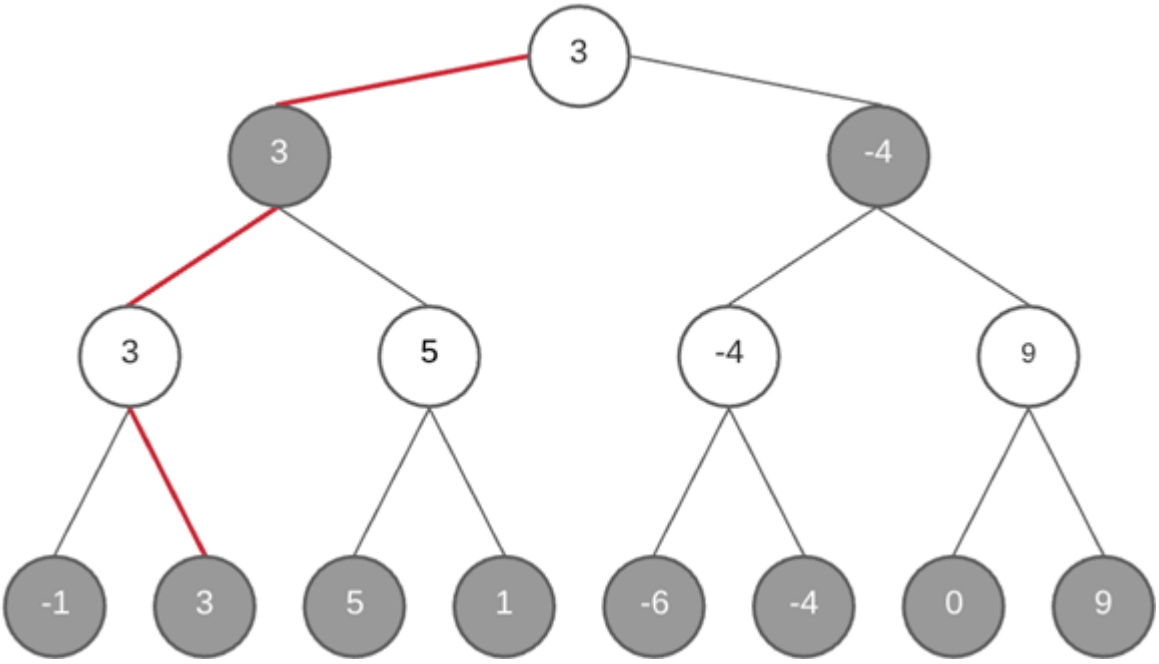


Figure 3.1.1. An example of a search tree for a minimax search algorithm. The bold, red line represents the principal variation, or in other words, the moves the algorithm expects will be played. The edges represent moves, and the nodes are the resulting board positions. The numbers inside of the nodes are the evaluations of the position by the engine.

Figure 3.1.1 shows an example of a search tree representing the flow of a search method in a game of chess. In this, tree the light nodes correspond to board positions where it is the maximizing player's turn to make a move and the dark nodes correspond to positions where it is the minimizing player's turn to make a move. The numbers in the leaf nodes are the values given by the evaluation method. After a leaf node has been evaluated the evaluation is returned to the parent node which in turn returns either the maximum or the minimum value of its child nodes. Whether or not the smallest or greatest value is returned depends on which player's turn it is to move. If it is the maximizing player's, i.e., the player with the white pieces, turn to move then the greatest value is returned. Conversely, if it is the minimizing player's turn to move, the smallest value is returned. The result of a minimax algorithm called at the root node is either the highest or lowest (depending on which of the two recursive methods are called), evaluation the player is guaranteed to achieve given expected play. This expected outcome in a given position is also called the principal variation, which is the sequence of moves that an engine expects to be played given a certain board position.

3.1.2 Alpha-Beta Pruning

Alpha-beta pruning is a method of improving the minimax algorithm by reducing the search space to save execution time. In the basic minimax algorithm, without any sort of pruning, some nodes are examined and evaluated despite them not possibly having the potential to affect the result. Pruning is done through so-called “cut-offs” which is when an engine avoids visiting a node to save execution time. In alpha-beta pruning this is achieved through the maintenance of two variables, alpha and beta. Alpha corresponds to the smallest evaluation value the maximizing player is assured of, and beta is the greatest value the minimizing player is assured given the information gathered from the nodes searched at a given point in time.

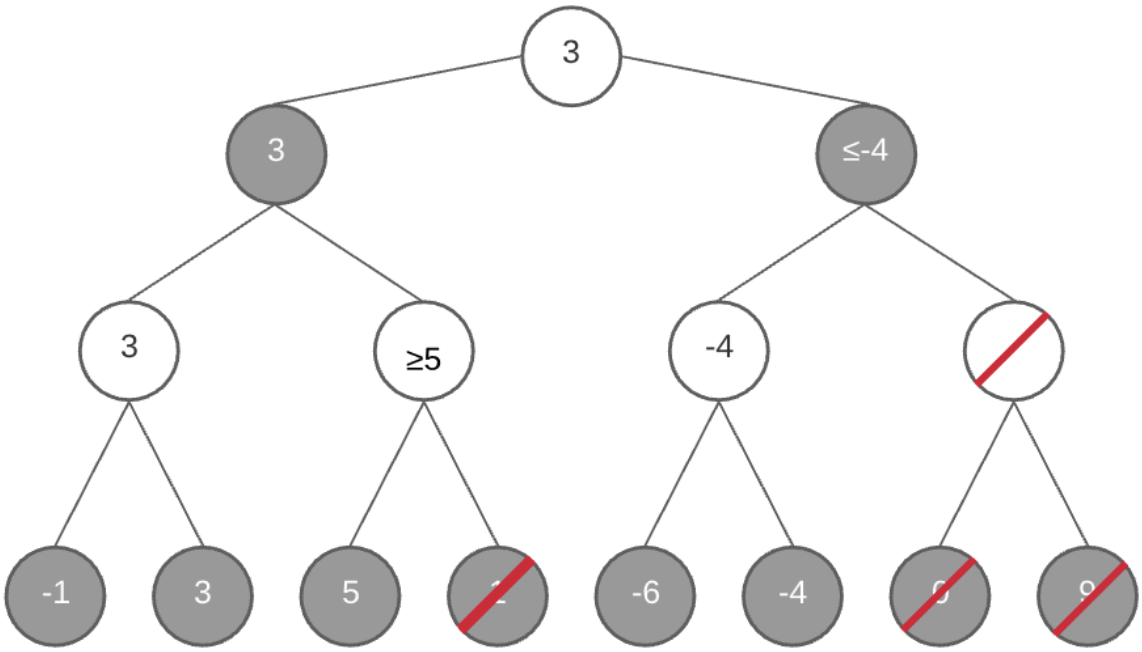


Figure 3.1.2. A search tree of an alpha-beta algorithm demonstrating nodes that have been pruned. Pruned nodes are crossed out with a red line.

In an alpha-beta algorithm, all nodes are evaluated, and the alpha and beta values are updated, up until the point when a node's evaluation falls outside the bounds of alpha and beta. Because such a node cannot change the outcome of the search, it may be pruned without affecting the result. An example of pseudocode demonstrating the maximizing method of the alpha-beta pruning algorithm is shown below:

```
function alphaBetaMax(board, alpha, beta, depthLeft)
     if(depthLeft == 0)
          return evalPosition(board)
     moves = generateMoves(board, "WHITE")
     for(move in moves)
          simBoard = new Board(board)
        simBoard.makeMove(move);
        score = alphaBetaMin(simBoard, alpha, beta, depthLeft -1)
    if(score >= beta)
        return beta //beta cut-off occurs
    if(score > alpha)
        alpha = score
    return alpha
```

The method above is the maximizing half of the alpha-beta search routine and it calls the minimizing method for each of its children, which in turn calls the maximizing method and so on. This makes these two methods corecursive. The minimizing method is shown below:

```
function alphaBetaMin(board, double alpha, double beta, int depthLeft) is
     if(depthLeft == 0)
          return evalPosition(board)
     moves = generateMoves(board, "BLACK")
     for(Move move in moves)
          Board simBoard = new Board(board)
        simBoard.makeMove(move);
        score = alphaBetaMax(simBoard, alpha, beta, depthLeft -1)
    if(score <= alpha)
        return alpha //alpha cut-off occurs
    if(score < beta)
        alpha = score
    return beta
```

The minimizing method, alphaBetaMin, is mostly identical to the maximizing alphaBetaMax except for a few differences. Firstly, it generates moves belonging to the player with the black pieces as this is the minimizing player in KLAS. Secondly, the usage of alpha and beta is reversed, as alpha is to the maximizing player what beta is to the minimizing player.

Alpha-beta pruning gives a large average reduction in execution time in chess, without the risk of overlooking any relevant possibilities. The worst-case scenario of alpha-beta search is when the worst move is always examined first, meaning no pruning can be made. In this scenario, the alpha-beta search performs the same as the minimax algorithm. Assuming a constant number of moves B available in a game of chess, and with a search depth of N, the total number of leaf nodes is B^N for both the minimax and alpha-beta algorithm. However, in a best-case scenario the number of leaf nodes that needs to be examined is $b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$ [4]. To maximize the benefit of alpha-beta pruning one needs accurate move ordering (See chapter 3.7).

Many chess engines, including KLAS, build a principal variable as they search for the purpose of debugging and to extract the best moves according to the engine. The principal variation contains the line of moves the engine expects to be played.

3.1.3 Node types

In the search trees used to represent chess and similar games, there are three commonly recognized node types as defined by Donald Knuth, an American computer scientist and mathematician. In his 1975 monography, the Art of Computer Programming, Knuth analysed a number of algorithms, among them the minimax algorithm with alpha-beta pruning. The node types he defined for alpha-beta search trees are as follows [5]:

- Type 1-nodes or “PV-nodes,” are nodes whose evaluations lies between the bounds defined by the alpha and beta values. The PV-nodes connected from the root node of a tree together make up the principal variation.

- Type 2-nodes or “Cut-nodes” are nodes whose evaluation result in an alpha or beta cut-off.

- Type 3-nodes or “All-nodes” are nodes where the evaluation does not cause an update in the alpha or beta values nor does it cause any alpha or beta cut-off.

When KLAS searches a node, it determines its type and stores that along with its score in the transposition table for later use (see chapter 3.8). The node type determines what the score means. For PV-nodes the score is exact and for cut-nodes and all-nodes the score is either an upper or lower bound of the actual score. This is because of the alpha-beta pruning which doesn’t need to determine the exact score of nodes that are not PV-nodes as all that is relevant is whether a score is lower or greater than the bounds alpha and beta or not.

3.2 Board Representation

A chess position in KLAS is represented by several bitboards. Bitboards are 64-bit variables where each bit is used to represent where pieces are located on a chessboard. The squares on the chessboard are numbered 0 through 63 and the bit on the corresponding index is set to ‘1’ if there is a piece on the square and if not, ‘0’. Because chess involves six different piece types of two colors, a total of twelve bitboards are needed to represent a chessboard and all its pieces. Bitboards are an effective board representation for chess on modern computers as an entire bitboard can be stored in a register of a 64-bit processor.

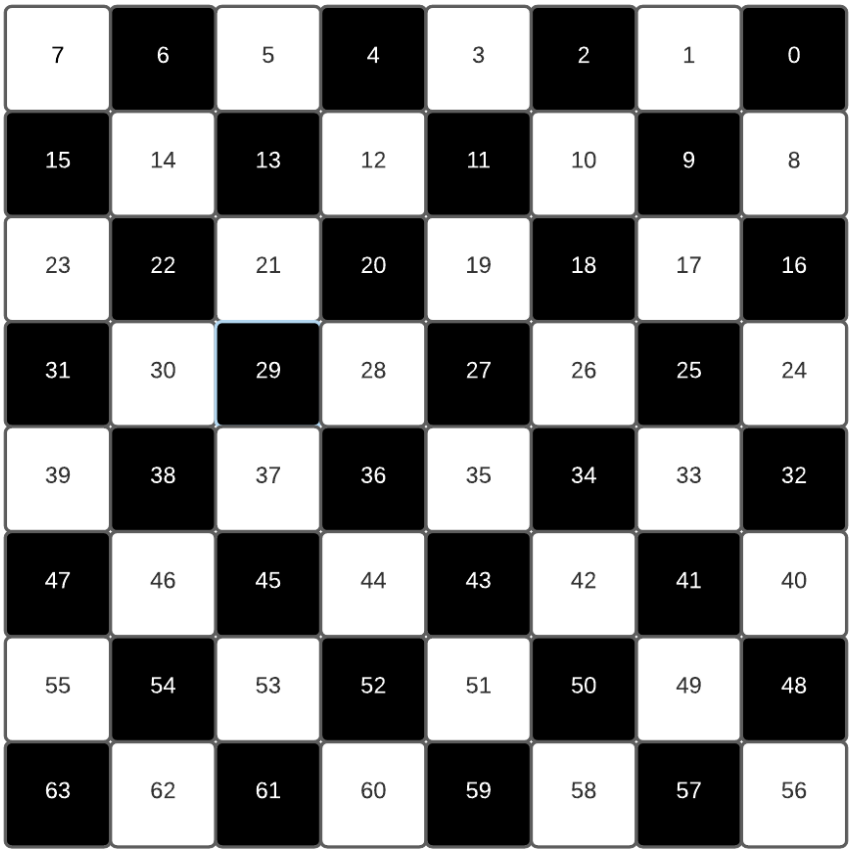


Figure 3.2.1. The mapping in KLAS of squares to indexes in the bitboards as seen from the perspective of the player with the white pieces.

In KLAS, the least significant bit i.e., the right-most bit of a binary number, LSB, is mapped to the top right square. The most significant bit, which is also the left-most bit, MSB, is mapped to the bottom left square. As an example, the following binary number corresponds to the position of the white pawns are the start of a chess game.

”0000000011111111000000000000000000000000000000000000000000000000”

Bitstring representing the white pawns at the start of a chess game.

The binary number above is shown more pedagogically in figure 3.2.2.

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 3.2.2. The bitboard for the white pawns are the start of chess game.

Lastly, the board representation contains a few other attributes related to castling and en passant moves. The use of these is explained in chapter 3.5.

3.3 Finding Moves

Chess engines use move generators to find all available moves on a given chessboard. In KLAS, the move generator first generates moves that may not be legal as they may leave the moving player's king in check. After the move generator has produced the list of moves, each move is individually checked for validity. In case a move in the list causes the player's own king to be in check, the move is illegal and thus removed from the list.

Vital to the move generator's function are four general use bitboards: the empty-bitboard, the occupied-bitboard, the friends-bitboard and the enemies-bitboards. The empty-bitboard contains 1s for each index at which the square is not occupied on the board and the occupied-bitboard is the inverse the empty-bitboard. The friends-bitboard is a subset of the occupied-bitboard containing 1s at the indexes where the square is occupied, and the occupying piece is friendly to the player for which the move generator is currently operating. In turn, the enemies-bitboard contains 1s at the indexes where the square is occupied, and the occupying piece belongs to the opponent.

3.3.1 Pawns

To generate moves for the pawns, KLAS uses bitshift to create a new bitboard containing the squares that the pawns may move to. Shifting a bit eight steps to the right is equivalent to moving a piece on step upwards on the board. For example, the bitboard for the white pawns at the start of a game is shown in figure 3.3.1.

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 3.3.1. Bitboard for the white pawns at the start of a game.

The bitboard is then bit shifted eight steps to the right to create a new bitboard that contains the possible one square pawn pushes.

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 3.3.2. The bitboard in figure 3.3.1 bit shifted eight steps to the left.

This bitboard is again bit shifted eight steps to the right to create the bitboard containing the available two step pawn pushes.

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 3.3.3. The resulting bitboard after the bitboard containing the white pawns has been bit shifted a total of 16 steps to the right.

To prevent the move generator from generating moves where a pawn is pushed to a square that is occupied, these bitboards are intersected with the empty bitboard using bitwise AND.

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 3.3.4. The empty-bitboard at the start of a game.

To generate the pawn attacks, nine and seven step bit-shifting is used. After a nine step rightwards bit shift of the bitboards for the white pawns, the resulting bitboard will contain their rightward attacks. A pawn cannot attack a square that does not contain an enemy, so to remove such moves the bitboard is also ANDed with the enemies bitboard.

3.3.2 Knights and Kings

In KLAS, knight and king moves are generated using pre-calculated movement bitboards which contain all the possible moves for a piece given its position, assuming it is not blocked by a friendly piece. At startup, KLAS initializes a movement bitboard for each possible square a knight can inhabit for a total of 64 bitboards.

8	0	0	0	0	0	0	0	0
7	0	0	0	1	0	1	0	0
6	0	0	1	0	0	0	1	0
5	0	0	0	0	0	0	0	0
4	0	0	1	0	0	0	1	0
3	0	0	0	1	0	1	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 3.3.5. The movement bitboard for a knight located at e5.

When the move generator finds a knight or king on a square, it retrieves the corresponding movement bitboard and XORs it with the friendly-bitboard to avoid the capture of one's own pieces.

3.3.3 Sliding Pieces

In chess, the bishop, rook and the queen are often referred to as "sliding pieces". This refers to their movements as they can slide along rays as far as the chessboard allows unless they are stopped by another piece.

The rook moves along four different rays: one in an upwards direction, one downwards and two left- and rightward directions. The moves for each of these rays are generated in the same way, although separately. First, pre-calculated movement bitboards are used to find the movement ray corresponding to the position of the rook for which moves are to be generated. The movement bitboard for the upward ray of a rook on square e3 is shown in figure 3.3.6.

8	0	0	0	0	1	0	0	0
7	0	0	0	0	1	0	0	0
6	0	0	0	0	1	0	0	0
5	0	0	0	0	1	0	0	0
4	0	0	0	0	1	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 3.3.6. The bitboard containing the upward ray of a rook located on e3.

Unlike with the knights and kings, a simple intersection of the movement bitboards and the occupied/friendly-bitboards is not enough to calculate the pseudo-legal moves of this rook. If the square e7 is blocked by an enemy pawn, the rook cannot attack the square behind it, e8. To remove moves that are blocked in this way, the movement bitboard for the ray is bitwise XORed with a ray in the same direction starting at the piece behind the first occupied piece along the original ray. For example, if e7 was occupied, a ray starting at e8 in the same direction is used and removed from the original movement ray.

3.3.4 Extraction of Moves

We have thus far only shown how the move generator calculates bitboards representing the available moves. However, in order to be used, the actual moves must be extracted from the bitboards. The move generator iterates over each bitboard containing the moves for each piece and then adds a move to the move list whenever it encounters a square with a value of 1. As one bitboard is generated for each individual piece, it is known which piece moves according to which bitboard. However, in the case of pawns, all the one step pawn pushes are stored in a single bitboard, and all the two step pawn pushes are stored in another. This is possible because pawn pushes are of course always vertical – for example if there is a possible pawn push to e4, it must be the pawn in the e column that can perform it.

Only the bitboards containing pawn attacks need to be separated when the moves are to be extracted as it is possible for two different pawns to attack the same square.

3.4 Check & Checkmate Detection

To detect whether either player is in check, or has been checkmated on a given board, a check and a checkmate function are needed. For each node visited during a search, the move generator is called, and the only moves that should be considered are of course legal ones. In KLAS, the move generator generates pseudo-legal moves, which are moves that are legal assuming they do not leave the player's king in check. To determine which of the pseudo-legal moves are legal, a function called findMoveList is used.

The findMoveList function first obtains all pseudo-legal moves from the move generator, then checks if each move is legal by playing them on its own internal board. After a move has been played, a check function is called to determine whether the move resulted in the player being checked. If the move leaves the player in check, the move is rejected and if the player is not in check, the move is added to the list of fully legal moves.

In chess, checkmate has occurred when a player is in check and will remain in check regardless of what move they attempt to make. In KLAS checkmate is found by considering all the pseudo-legal moves that are available to a player in check. Should none of them be legal, as they all result in the player still being checked, the player is checkmated.

It is very important that the check testing function is as fast as possible as it is called once for each pseudo-legal move and many times for each checkmate test. Checkmate is tested by the evaluation method, explained in chapter 3.6, for each node visited and thus resulting in many calls to the check function.

3.5 The makeMove Function

The makeMove function is responsible for updating the internal board representation when a move is made. For most moves, this involves simply removing a piece from its location on its bitboard using an XOR operation and then moving the piece to its new location also using XOR. For taking moves, the enemy piece that is taken also needs to be removed in the same way. However, for castling moves, the move generator in KLAS simply returns the move for the king and then the makeMove function identifies that it is a castling move and continues to perform both the necessary moves for the king and the rook in question.

As castling moves may only be played once per player, the move generator must know if a castling move has already occurred. Therefore, the Board class has these additional attributes:

```
boolean castleWQValid = true;
boolean castleWKValid = true;
boolean castleBQValid = true;
boolean castleBKValid = true;
```

These attributes keep track of whether castling is allowed on either side. Similarly, information about whether en passant moves are possible is also stored in the Board class.

The makeMove function is responsible for updating the castling and en passant attributes when moves are made on the board.

3.6 Evaluation Methods

Evaluation methods give a score to different board positions, i.e., a numerical estimation of how favorable a board position is for the two players. Negative values are considered more favorable for the player with the black pieces and positive values are considered more favorable for the player with the white pieces. The magnitude of the value indicates the relative favorability – a large positive value is more favorable for the white player than a small positive value.

If a chess engine had infinite computing power, the evaluation would only need to evaluate checkmate as the search method could explore the whole chess game tree, where leaf nodes are either checkmate or stalemate. However, it is usually not possible to search all the way to leaf nodes, except when very few pieces remain. Better playing ability can then be achieved for the chess engine by using a heuristic evaluation of various qualities of a board position – giving an estimation of how likely it is to win for both players in that position.

Common qualities of a board position often evaluated in chess engines include material (i.e., the pieces still on the board), piece mobility, king safety, pawn structure and the ability to castle. The evaluation score is a sum of the score given for each quality examined.

The evaluation of the material on a board position in KLAS is show in the following pseudocode:

```
function evalPosition(board) is
    for each piece on board
        if piece == WHITE_PAWN
            points += 100
        else if piece == WHITE_KNIGHT
            points += 320
        else if piece == WHITE_BISHOP
            points += 330
        else if piece == WHITE_ROOK
            points += 500
        else if piece == WHITE_QUEEN
            points += 900

        else piece == BLACK_PAWN
            points -= 100
        else if piece == BLACK_KNIGHT
            points -= 320
        else if piece == BLACK_BISHOP
            points -= 330
        else if piece == BLACK_ROOK
            points -= 500
        else if piece == BLACK_QUEEN
            points -= 900
```

Each piece on the board is given a value based on its type. If the piece belongs to the white player, the value is added to the total score, and conversely subtracted if it belongs to the black player. This means that positions, where the white player's material is identical to the black player's, are given the material score of 0. The piece values used in KLAS are widely accepted in the chess community as appropriate approximate values.

Piece-Square Tables

Another heuristic optimization implemented in KLAS is Piece-Square Tables, PSTs, which are two-dimensional arrays of numerical values with an entry for each square on a chessboard. These values apply a bonus or deduction on the value of a piece depending on its location on the board. In chess, there are statistical advantages of placing pieces on certain squares and so the PSTs increase the playing ability of the engine.

3.7 Move Ordering

Move ordering is used to improve the performance of the alpha-beta search routine by attempting to prioritize the most favorable moves first. Alpha-beta search routines perform pruning when a move has been shown to result in a worse evaluation than a previously examined move. In the best case, the best available move is examined first causing other moves to be quickly pruned as they are shown to be worse. The faster the engine finds a good move, the more cut-offs may be performed to reduce the search space.

Move ordering optimizations involve a rough estimation of which moves appear the most promising before the engine begins to examine them. The moves, which lead to different board positions, or "nodes", are then sorted based on this estimation, in the hope that a good move is found early. The best-case scenario occurs when the first child of each node is determined to be the PV-node out of all the child nodes.

It should be noted that move ordering optimizations have no effect on the actual result of the search. A chess engine will decide on the same move, whether it uses any move ordering because it is an optimization that only serves to reduce search space through enabling more alpha and beta cut-offs.

In figure 3.7.1 a search tree for an alpha-beta method is shown. Inside each node, the move that leads to the node is shown. The move “c2-c4” is the move from the root node which gives rise to the leftmost child node. Without move ordering, this move is explored first for arbitrary reasons that are completely unrelated to how promising the move may or may not appear.

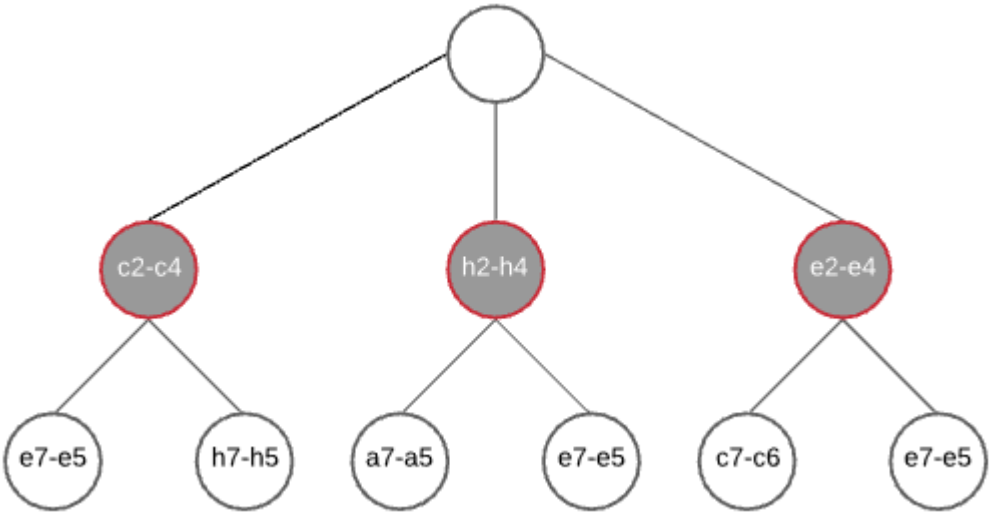


Figure 3.7.1. An alpha-beta search tree before any move ordering has occurred.

In figure 3.7.2 the same search tree is shown after move ordering has been performed at the root node. The sorting has resulted in the move “e2-e4” being the highest priority move of all the root nodes’ children, should this move turn out to be the most favorable move at the root, the search space and thus execution time will be reduced significantly.

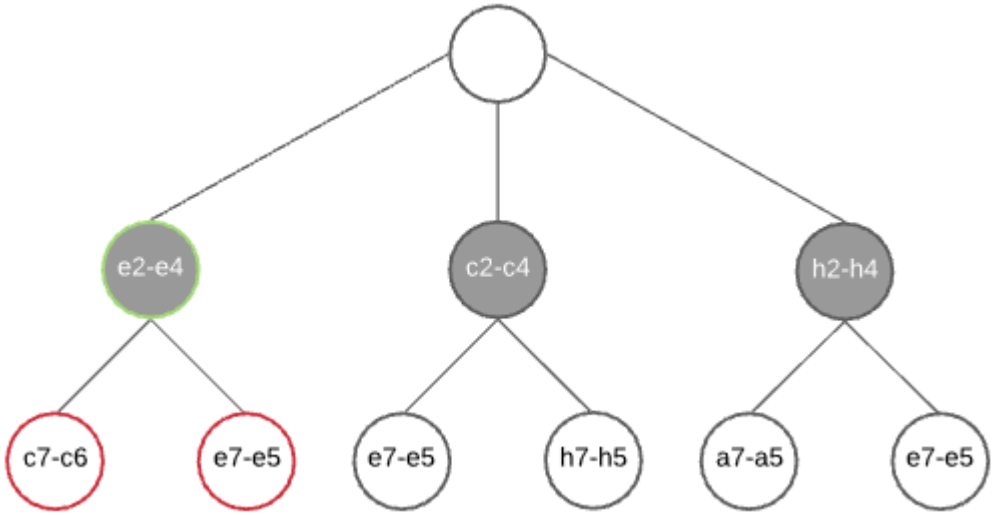


Figure 3.7.2. The search tree from figure 3.7.1 after the move ordering has been performed at the root node.

After the move ordering routine has been applied at the root node and the most promising child node is visited, it is applied once again on the new node “e2-e4”. This time the move ordering routine finds that the “e7-e5”-move appears more promising than the “c7-c6”-move, resulting in a new order as shown in figure 3.7.3.

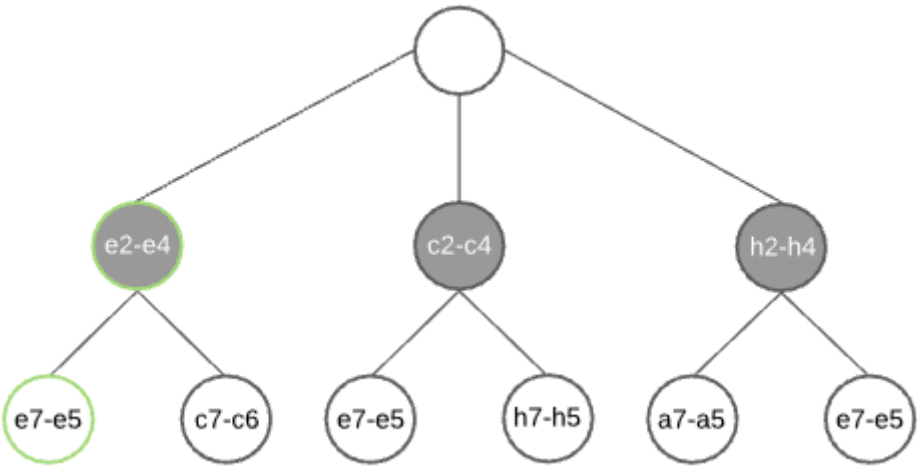


Figure 3.7.3. The search tree from figure 3.7.2 after the move ordering routine has been performed at the “e2-e4”-node.

The move ordering optimization is based on the concept that chess moves may be divided into categories, some of which are statistically more likely to result in favourable positions. However, if the heuristic behind the move ordering is inaccurate, it will result in a lower performance than if there is no move ordering whatsoever. In a position where the actual best moves are given low priority, move ordering will fail to decrease the search space and increase execution time.

3.7.1 MVV-LVA

An example of a commonly used heuristic for move ordering is MVV-LVA (Most Valuable Victim – Least Valuable Aggressor). In chess, moves, where an opponent's piece of high value is taken by a piece of low value of one's own, are often good moves. The MVV-LVA heuristic uses this reasoning by applying a score to each move based on the difference in value between a taking piece and the piece that is taken. The higher the value of the taken piece, and the lower value of the taking piece, the higher the priority score. For example, a very promising move, when available, is a move where one player may take the opponent's queen with their own pawn and this move is thus given the highest priority by the MVV-LVA heuristic. Pseudocode for the heuristic is shown below:

```
function prioritize(moves)
    for(move in moves)
        movingPiece = move.movingPiece
        takenPiece = move.takenPiece
        switch(movingPiece)
            case "PAWN"
                move.priority += -1
            case "KNIGHT"
                move.priority += -3
            case "BISHOP"
                move.priority += -3
            case "ROOK"
                move.priority += -5
            case "QUEEN"
                move.priority += -9
            case "KING"
                move.priority += -10

        switch(takenPiece)
            case "PAWN"
                move.priority += 10
            case "KNIGHT"
                move.priority += 30
            case "BISHOP"
                move.priority += 30
            case "ROOK"
                move.priority += 50
            case "QUEEN"
                move.priority += 90
```

Each move's priority is decreased depending on the value of the piece that moves from one square to another and then increased depending on which piece, if any, is taken. The bonus for taking a piece is greater than the penalty of moving a piece and so taking moves are given a greater priority than others.

3.7.2 PV-Ordering

Another common move ordering optimization is to prioritize the moves that either were a part of the principal variation of the previous search or those that, for a node, caused an alpha or beta cut-off. These moves have already been determined to be the either the best moves for the current node or at least good enough to cause a cut-off. KLAS prioritizes these moves the highest, even higher than the most promising move according to MVV-LVA. KLAS stores these moves in the transposition table, and they are retrieved for the current node at the start of both alpha-beta methods. See chapter 3.8.1.

3.8 Transposition Tables

During each search a chess engine performs, it encounters different nodes that are identical because in chess it is possible for different sequences of moves to result in the same board position. For example, the line “e4 → e5 → Nf3” results in the same board as the different line “Nf3 → e5 → e4”. These two identical board positions as reached by different sequences of moves are called transpositions. Whenever a chess engine encounters a transposition, it has already been evaluated and so re-evaluating the node would be a waste of time. To avoid this, the engine stores results in a transposition table so that they may be used later, either during the same search or new ones.

Transposition tables are hash tables that store hashes of board positions along with relevant information about them so that it may save time re-evaluating them. In KLAS the transposition table is implemented using a HashMap where the keys are the hash values of boards, and the values are objects of type TPTEntry. The TPTEntry data structure contains the following fields:

- The hash of the board that corresponds to the node in question.
- The score of the node.
- The depth at which the node was searched.
- The node type.
- Whose turn it is to move on the node.

As chess engines encounter a very large number of different board positions during a search, the data for all of them cannot be realistically stored in the transposition table for the entire duration of the game. It would simply grow too large and so the transposition table in KLAS is fixed size. When the table has been filled and a new entry is to be added, the oldest entry in the table is overwritten.

3.8.1 Usage of the Transposition Table

During a search, KLAS computes a hash value for each node it examines and tries to find a match in the transposition table. Should a match be found, the corresponding TPTEntry is retrieved, and its depth is compared to the remaining depth to be searched. As the score of a node becomes more accurate the deeper it is searched, KLAS rejects any matches where the depth field is lesser than the depth remaining to be searched.

The retrieval of entries in the transposition table occurs at the start of both two alpha-beta methods. The usage of the entries depends on its depth attribute as well as the node type attribute. An entry is only used if its depth attribute exceeds or is equal to the depth left to be searched at that point in time. See the following pseudocode:

```
function alphaBetaMax(board, depthLeft, alpha, beta)
    hash = tpt.hash(board)
    entry = tpt.get(hash)
    if(entry.depth >= depthLeft){
        if(entry.nodeType == PVNODE){
            return entry.score
        }
        else if(entry.nodeType == CUTNODE && entry.score >= beta)
            return beta;
        }
        else if(entry.nodeType == ALLNODE && entry.score <= alpha){
            return alpha;
        }
        ...
    }
```

The attribute nodeType allows KLAS to know whether the score attribute is an exact score or an upper or lower bound of the actual score. Depending on which type it is, KLAS uses the score differently. Should the node be a PV-node, the score is known to be exact and is thus returned immediately. However, if the node is either a cut-node or an all-node, the score is only usable if it exceeds the current alpha and beta bounds as we then know that this node couldn’t possibly affect these values.

3.8.2 Creating Entries

Because of the alpha-beta pruning, some results returned by the alpha-beta method are not actually exact. When a beta cut-off occurs in the maximizing method of the alpha-beta search routine, the value returned is beta which is an upper bound of the actual score of the node. If none of the node’s children causes a cut-off and none of them succeeds in increasing alpha, the node is known to be an all-node with the score being a lower bound of the actual score. In case no child nodes cause a beta cut-off and there is at least one child that increases alpha, the node is now known to be a PV-node with an exact score.

The exact same logic is valid for the minimizing half of the alpha-beta search routine except it is, in a sense, reversed. In the minimizing method, child nodes with a score smaller than alpha cause alpha cut-offs and the lower bound, alpha, is returned. A node where this happens is a cut-node. If none of the node's children causes an alpha cut-off and none of them succeeds in decreasing beta, the node is an all-node with a score that is an upper bound. If no child nodes cause an alpha cut-off and there is at least one child that decreases beta, the node is known to be a PV-node with an exact score.

During the alpha-beta search methods, KLAS creates entries and sets their nodeType attribute to their corresponding node type so that their score attribute is not misinterpreted when the entries are used.

3.8.3 Zobrist hashing

The Zobrist hash function was invented by Alexander Zobrist as an efficient way to calculate hash values for chess positions using bitwise operations and a number of random numbers. This type of hashing function is common in chess engines and is used in KLAS [6].

Hashing is commonly used in chess engines so that board positions can be efficiently compared to each other. The purpose of this comparison is simply to determine whether two positions are identical or not. This allows us to retrieve information about specific board positions from the transposition table without having to compare each field in the Board class.

In KLAS, a two-dimensional array of randomized 64-bit integers is initialized at startup. The size of one dimension is 64 and the other 13. These $64 \cdot 13$ random integers correspond to each square on the chessboard and each of its possible states as a square can either be empty or contain six different piece types, each of two different colors for a total of 13 possibilities.

To generate the hash of a board, the engine iterates over the squares and when examines what state the square is in. For each square, it takes the random integer corresponding to that square and performs an XOR operation on it and the hash in its current state. This results in a 64-bit hash value with a very low probability of collisions.

Pseudocode for the hash algorithm is shown below:

```
function hash(board) is
    for each piece on board
        if(pieceType == WHITE_PAWN)
                hash ^= randomNumbers[WHITE_PAWN][piece.position]
        else if(pieceType == WHITE_KNIGHT)
                hash ^= randomNumbers[WHITE_KNIGHT][piece.position]
        else if(pieceType == WHITE_BISHOP)
                hash ^= randomNumbers[WHITE_BISHOP][piece.position]
        else if(pieceType == WHITE_ROOK)
                hash ^= randomNumbers[WHITE_ROOK][piece.position]
        else if(pieceType == WHITE_QUEEN)
                hash ^= randomNumbers[WHITE_QUEEN][piece.position]
        else if(pieceType == WHITE_KING)
                hash ^= randomNumbers[WHITE_KING][piece.position]
        else if(pieceType == BLACK_PAWN)
                hash ^= randomNumbers[BLACK_PAWN][piece.position]
        else if(pieceType == BLACK_KNIGHT)
                hash ^= randomNumbers[BLACK_KNIGHT][piece.position]
        else if(pieceType == BLACK_BISHOP)
                hash ^= randomNumbers[BLACK_BISHOP][piece.position]
        else if(pieceType == BLACK_ROOK)
                hash ^= randomNumbers[BLACK_ROOK][piece.position]
        else if(pieceType == BLACK_QUEEN)
                hash ^= randomNumbers[BLACK_QUEEN][piece.position]
        else if(pieceType == BLACK_KING)
            hash ^= randomNumbers[BLACK_KING][piece.position]
```

The reason why Zobrist hashing is used in many chess engines is because of how it may be efficiently updated during a search. Knowing the previous hash value and having access to the previous board position the hash for the new board may efficiently be generated using only the changes incurred by the new move to be performed. [6]

3.9 Iterative Deepening

Iterative deepening is an optimization where searches are performed iteratively, with each iteration being deeper than the previous one. The purpose is to manage the amount of time spent per move as well as speed up execution. For example, one may execute a search with a depth of 2 on a board position only to then search the same position again with a depth of 4. As the transposition table will be filled with information from the previous iteration using a depth of 2, the subsequent search to a depth of 4 will be faster. In fact, with PV-ordering, it is possible for an iterative deepening search to a given depth to be faster than an immediate search to the same depth [7].

3.10 Lazy SMP Parallel Search

Lazy Symmetric MultiProcessing is a simple algorithm for parallelizing an alpha-beta search method used by many powerful modern chess engines such as Stockfish [8]. The method uses multi-threaded execution, running simultaneous searches in different execution threads.

On a computer with 4 hardware threads, ideal parallelization of the search could speed up the computation by a factor of 4 in real-time. In many cases parallelizing an algorithm can be very challenging, however the “Lazy” in “Lazy SMP” refers to how easy it is to implement as it requires no direct communication between threads other than a shared transposition table.

Lazy SMP can speed up searching by different threads sharing search results in a transposition table. A number of helper threads are started, along with the main thread and begin to search the board position. The transposition table allows each thread to store data regarding the different board positions it has searched thus far and then the other threads use this data instead of researching the position. Lazy SMP has proven to be an effective optimization and is capable of halving execution time in some cases [9].

If all the threads are searching the same node at the same time no speedup is achieved and so to decrease the probability that this occurs, different move ordering for each thread may be used. In KLAS the helper threads use randomized move ordering at the root node to decrease the probability of two of them searching the same node at the same time.

3.11 Other Optimizations

To minimize the execution time penalty incurred by the JVM's garbage collection, we made efforts to avoid initializing new objects when possible and instead reuse old ones. One class often instantiated is the Move class which is an abstraction of a chess move. To avoid instantiating new moves, a class called MoveArrayListManager (MALM) was implemented. The MALM is responsible for providing instances of type MoveArrayList (MAL), which are lists of moves.

The MALM class provides two methods: obtainMoveArrayList() and renounceMoveArrayList(). When a MAL has fulfilled its purpose, the method renounceMoveArrayList() is called which adds the MAL to the MALM's internal pool of recyclable MALs. When obtainMoveArrayList() is called, the MALM instantiates a new object and returns it only if there are no MALs available. Otherwise, it returns a MAL from its internal pool. This method helps reduce the number of lists instantiated by KLAS.

MAL, in turn, is a class that contains an ArrayList<Move> and a size attribute. Whenever a MAL has fulfilled its purpose and is added to the MALM's pool, its size attribute is set to 0. Then, whenever the MAL is reused and a new move is to be added, an old move in the ArrayList is simply updated to be identical to the new move. This way moves in the MoveArrayList are recycled too.

The transposition table also recycles its entries in a similar manner to the MoveArrayList. Whenever an old entry is to be overwritten, its attributes are updated to the values of the new entry.

4 Assessment

To answer the research questions, we needed to measure the performance of KLAS and analyse it from several perspectives. This chapter presents the methodology of the measurements, the results as well as the analysis of the results, each in its own section.

The optimizations that are discussed in this chapter are: iterative deepening, move ordering according to MVV-LVA and PV-nodes, Lazy SMP as well as the usage of the transposition table. To be able to examine each optimization independently of one or several of the other optimizations a number of different variants of KLAS are defined:

- MT – Complete, multithreaded KLAS which employs all the above optimizations.
- ST – Complete KLAS except Lazy SMP-multithreading is disabled.
- MO – Single-threaded KLAS without MVV-LVA move ordering.
- TT – Single-threaded KLAS without using transposition table for alpha/beta pruning.
- ID – Single-threaded KLAS with no iterative deepening.

The table below shows what optimizations are active in each variant.

Variant	**Lazy SMP**	**MVV-LVA**	**Transposition Table**	**Iterative Deepening**
MT	Yes	Yes	Yes	Yes
ST	No	Yes	Yes	Yes
MO	No	No	Yes	Yes
TT	No	Yes	No	Yes
ID	No	Yes	Yes	No

We measured the performance of each of these variants of KLAS by searching many different board positions. Each version of KLAS was made to search 600 different board positions, each being restricted to fifteen seconds of search time and a maximum depth of six. While each version was searching, information was stored regarding the execution time of each iteration of iterative deepening, the number of matches in the transposition table at each depth as well as the type of entry that was found.

4.1 Test System

All measurements were taken on a Windows 10 system with an i5 6600K CPU running at 3.5 GHz with 8192 MBs of DDR4 memory running at 1800 MHz. The JDK distribution used was Azul version 13.0.7. All the measurements were run using IntelliJ IDEA 2021.1.3.

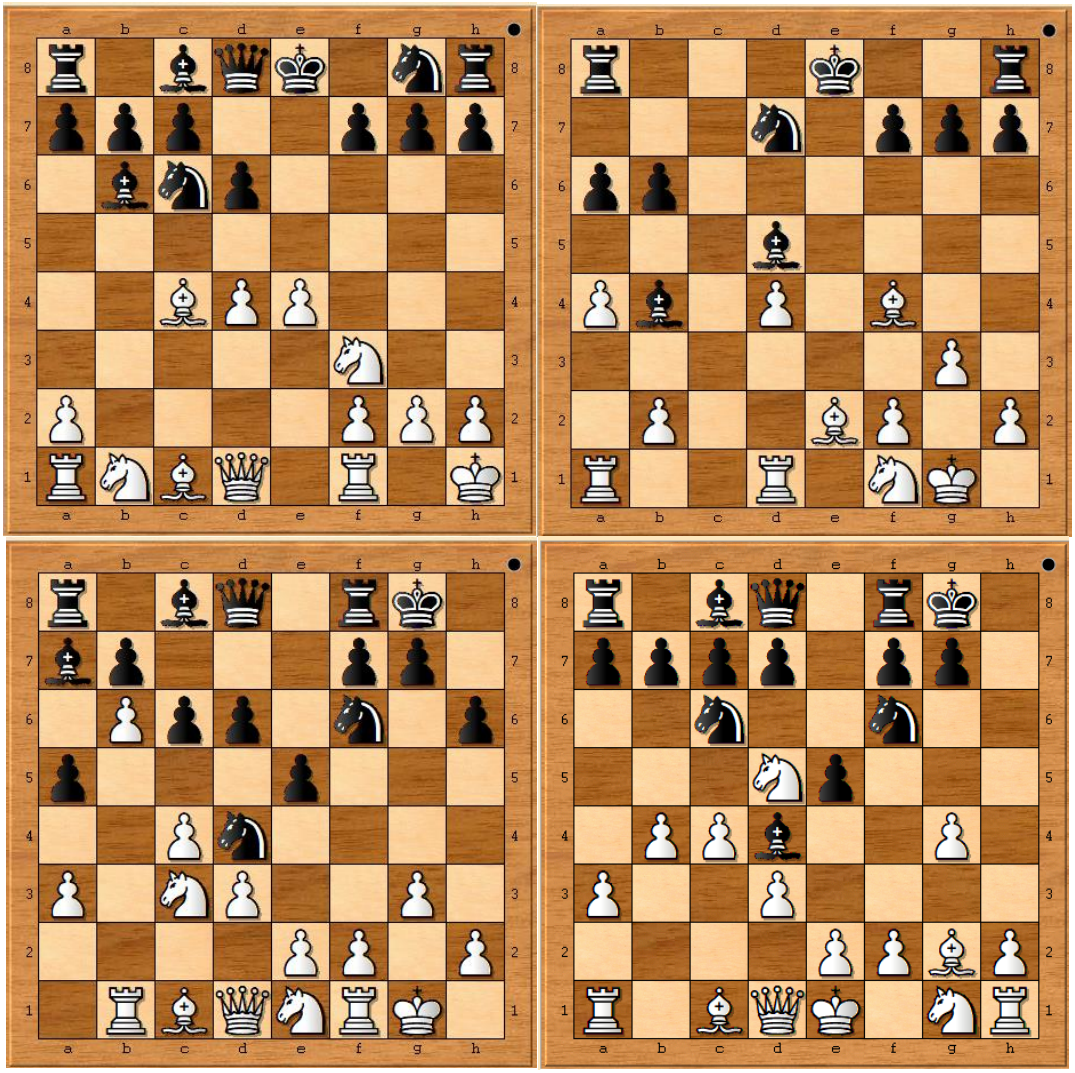
4.2 Board Positions

The performance of the different KLAS versions was measured using sets of ten connected lines, that is, ten board positions where each board is identical to the previous one except for one move having been made. The first board in each of these sets of ten will be referred to as an “original position”. Each original position was chosen from different online resources like chess lessons and professional matches. We used 60 sets for a total of 600 board positions.

The 600 board positions can be divided into two different groups, each based on how the boards from the original board were generated. These groups are:

1. Random-line boards – where each board following the original board is identical except for one randomized move having been made.
2. KLAS boards – where each board following the original board is identical except for one move having been made as chosen by KLAS’ search to a depth of 6.

Most board positions used for the measurements are in the early/midgame. Examples of the board positions are shown below:



4.3 Metrics

A number of different metrics were recorded during the 600 searches. Each metric recorded is listed and explained below.

- **Execution time** - the time to execute each search to a depth of 6 from the first to the last iteration of iterative deepening, if activated.
- **Memory usage** - the difference between the total memory and the free memory in the Java Virtual Machine. This was recorded at the end of each iteration of iterative deepening, for a total of three data points per search.
- **Good Move Index (GMI) 0-5** - the index defined to aid analysis of the move ordering heuristics. This was recorded as an average per node at each depth of each search during the last iteration of iterative deepening. The definition of this index is explained in chapter 4.3.1.
- **TP-Hits 0-6** – the total number of matching entries found in the transposition table at each depth of search during the last iteration of iterative deepening. Some of these transposition hits may not have been used during the search.
- **PV-hits 0-6** – the number of matching entries found in the transposition table where the score is exact. These were recorded at each depth of search during the last iteration of iterative deepening.
- **CUT-hits 0-6** – the number of matching entries found in the transposition table where the entry was at a node that was determined to be a cut-node. These were recorded at each depth of search during the last iteration of iterative deepening.
- **ALL-hits 0-6** - the number of matching entries found in the transposition table where the entry was at a node that was determined to be an all-node. These were recorded at each depth of search during the last iteration of iterative deepening.
- **GC collection time** – the total amount of time the JVM's garbage collector spent collecting garbage during a search.
- **GC collection count** – the number of times the JVM's garbage collector were called during a search.

4.3.1 Good Move Index (GMI)

To determine the effectiveness of MVV-LVA we defined the metric “Good Move Index”. The Good move index is the index of the move in a move list for a given node which either causes an alpha/beta cut-off or is the last node to increase either the alpha or the beta bounds. Such a move has the potential of being the best move in the move list and is in the worst-case scenario, at least good enough to cause a cut-off.

A lower average GMI results in a decrease in execution as figure 4.3.1 demonstrates using a scatterplot. The trendline shows the correlation between the average GMI at all depths and the execution time of a search of the MO variant.

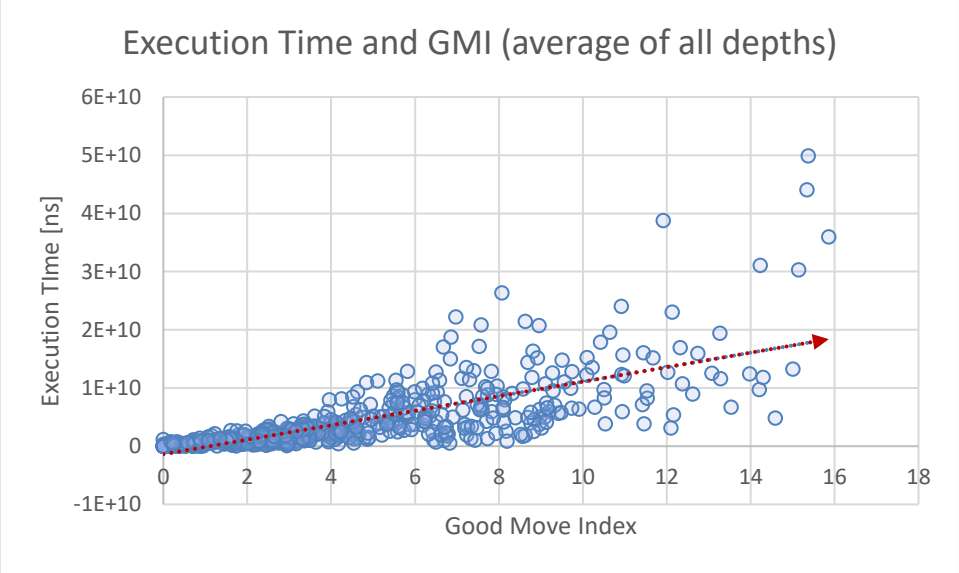


Figure 4.3.1. Scatterplot of each search’s execution time and its average Good Move Index (GMI) of all depths combined. The trendline demonstrates the correlation between the two.

Although there are also other factors at play, a lower Good Move Index is a predictor of lower execution time.

4.4 Results

This chapter presents the results from the measurements of the five different variants of KLAS, each in their own section.

4.4.1 MVV-LVA Move Ordering

To investigate the effectiveness of MVV-LVA, the MO variant of KLAS is compared to the ST variant, as the only difference between them is the MVV-LVA move ordering heuristic. The table below demonstrates the average execution time of these two variants:

Variant	**Execution time [ms]**
MO	4066
ST	1280

The MO variant, without MVV-LVA, executed the searches with an average execution time of 4066 milliseconds while the ST variant in only 1280 milliseconds. In terms of percent, MVV-LVA move ordering resulted in a decrease of 68.5% in execution time on average.

The average Good Move Index of both the ST and MO variants are presented in the figure 4.4.1.

Variant	**GMI (depth 0)**	**GMI (depth 1)**	**GMI (depth 2)**	**GMI (depth 3)**	**GMI (depth 4)**	**GMI (depth 5)**
MO	5.84	1.60	5.43	3.37	8.97	7.31
ST	4.89	0.98	2.00	1.35	1.48	0.32

Figure 4.4.1. The average Good Move Index at each depth of the MO and ST variants of KLAS.

The ST variant, using MVV-LVA move ordering, achieved a reduction in GMI at each depth of search. The effectiveness of MVV-LVA was bigger at greater depths with the GMI being reduced at depth 5 from an average of 7.31 to 0.31.

4.4.2 Transposition Table

The transposition table in all variants of KLAS, where activated, had a maximum size of 1.000.000 entries. In the TT variant of KLAS, the transposition table was not fully deactivated, but the entries were only used for PV-ordering. No entries found during a search in the TT variant were used to avoid researching a node.

The average execution time of the TT and ST variants of KLAS are presented in the table below.

Variant	**Execution time [ms]**
TT	2124
ST	1280

The ST variant, which used the transposition table for alpha/beta cut-offs performed the average search with a 39.8% reduction in execution time.

The average Good Move Index at each search depth of both the TT and ST variants is presented in figure 4.4.2.

Variant	**GMI (depth 0)**	**GMI (depth 1)**	**GMI (depth 2)**	**GMI (depth 3)**	**GMI (depth 4)**	**GMI (depth 5)**
TT	4.61	1.17	2.03	1.26	1.25	0.17
ST	4.89	0.98	1.99	1.35	1.48	0.32

Figure 4.4.2. The average Good Move Index at each depth of the ID and ST variants of KLAS.

In terms of move average GMI, the TT and ST variants are relatively similar at most depths. At depth 5 specifically the relative difference is significant, with the ID version achieving an almost twice as low GMI.

4.4.3 Iterative Deepening

Iterative deepening is an optimization which by itself gives no performance boost. However, when paired with move ordering by PV-nodes from previous iterations as well as a transposition table for achieving cut-offs, it may give a significant reduction in execution time. Therefore, we use the ID variant of KLAS, which has move-ordering as well as a transposition table enabled, but no iterative deepening, and compare this to the ST variant. This way, iterative deepening is the only differentiating factor, allowing us to investigate it.

In KLAS, the search depth is incremented by two every iteration. First, KLAS searches to a depth of two, then four and then six. As the ID variant of KLAS uses no iterative deepening memory usage was only measured at the end of the search at depth 6. The table below demonstrates the execution time of the ID variant and ST variant.

Variant	**Execution time [ms]**
ID	1796
ST	1280

As can be seen in the table above, iterative deepening with PV-ordering caused a decrease in execution time of 28.7% on average.

Iterative deepening resulted in significantly more transposition table hits as demonstrated in figure 4.4.3.

Variant	**TP-Hits, (depth 0)**	**TP-Hits, (depth 1)**	**TP-Hits, (depth 2)**	**TP-Hits, (depth 3)**	**TP-Hits, (depth 4)**	**TP-Hits, (depth 5)**	**TP-Hits, (depth 6)**
ID	0.0	0.5	0.1	358.0	1120.1	23735.2	2500.0
ST	0.8	9.7	24.4	718.5	594.1	16030.6	825.9

Figure 4.4.3. Average number of transposition table hits at each depth of search of the ID and ST variants of KLAS.

As can be seen in figure 4.4.3, the number of transposition table hits at lower depths increases drastically with iterative deepening. However, at greater depths the number of hits decreased with iterative deepening.

The average Good Move Index at each depth was recorded for the ID variant of KLAS as well. This data is presented in figure 4.4.4.

Variant	**GMI (depth 0)**	**GMI (depth 1)**	**GMI (depth 2)**	**GMI (depth 3)**	**GMI (depth 4)**	**GMI (depth 5)**
ID	10.32	1.77	2.86	1.94	1.95	0.40
ST	4.89	0.98	1.99	1.35	1.48	0.32

Figure 4.4.4. The average Good Move Index (GMI) at each depth for the ID and ST variants (lower is better).

The ST variant of KLAS, with its iterative deepening, achieved a much lower and thus better GMI at every depth of search. Most significant is the decrease at lesser depths while the difference is much lesser at greater depths.

4.4.4 Lazy SMP

The effect on the performance Lazy SMP had was measured using the MT variant of KLAS. MT uses a total of four threads during each search, three of which are helper threads with randomized move ordering. The fourth thread, the main thread, uses MVV-LVA and PV-ordering.

The average execution time and memory usage at each iteration of iterative deepening from the MT and ST variants of KLAS is presented below in the table below.

Variant	**Execution time [ms]**	**Memory usage (iteration 1) [MB]**	**Memory usage (iteration 2) [MB]**	**Memory usage (iteration 3) [MB]**
MT	850	914.7	944.8	765.8
ST	1280	759.1	776.3	756.3

The MT variant of KLAS performed the same 600 searches with an average execution time of 850 milliseconds while the single-threaded ST variant of KLAS performed the same searches with an average execution time of 1280 milliseconds. In other words, Lazy SMP decreased average execution time by approximately 33.4%.

The MT variant used significantly more memory at the first two iterations of iterative deepening: an increase of 155.6 or 20.5% and 168.5 MBs or 21.7% respectively. A much smaller difference in memory usage was recorded after the last iteration to depth 6, specifically 9.5 MBs or 1.2%.

The average number of transposition table hits at each depth per search is presented in figure 4.4.5.

Variant	**TP-Hits, (depth 0)**	**TP-Hits, (depth 1)**	**TP-Hits, (depth 2)**	**TP-Hits, (depth 3)**	**TP-Hits, (depth 4)**	**TP-Hits, (depth 5)**	**TP-Hits, (depth 6)**
MT	5.4	245.3	145.7	2030.6	3465.2	33299.4	3539.2
ST	0.8	9.7	24.4	718.5	594.1	16030.6	825.9

Figure 4.4.5. Average number of transposition table hits at each depth of search of the MT and ST variants of KLAS.

Overall, the multithreaded MT variant of KLAS achieved significantly more transposition table hits than the single-threaded ST variant at every search depth. The biggest relative difference is at depth 1, where the MT variant achieved 2528% more hits on average than the ST variant. The biggest absolute difference was at depth 5, where the MT variant got 17269 more hits per search.

In the MT variant of KLAS the average GMI was recorded at each depth of search for only the main search thread. The GMI of the helper threads was not recorded. The GMI of the MT and ST variants are presented in figure 4.4.6.

Variant	GMI (depth 0)	GMI (depth 1)	GMI (depth 2)	GMI (depth 3)	GMI (depth 4)	GMI (depth 5)
MT	4.59	1.16	2.04	1.44	1.42	0.29
ST	4.89	0.98	1.99	1.35	1.48	0.32

Figure 4.4.6. The average Good Move Index (GMI) at each depth for the MT and ST variants (lower is better).

For the MT variant the average GMI varied from its lowest value of 0.29 at depth 5 to its highest value of 4.59 at depth 0. Meanwhile, for the ST variant, the GMI varied between 0.32 to 4.89. The MT variant achieved a better GMI at depths 0, 5 and 5 while the ST variant performed better, in terms of GMI, at depths 1, 2, 3.

The correlation between the average GMI and the execution time shown in chapter 4.4.1 was observed in the MT variant as well. Figure 4.4.7 shows a scatterplot of the GMI as an average of all depths and the execution time.

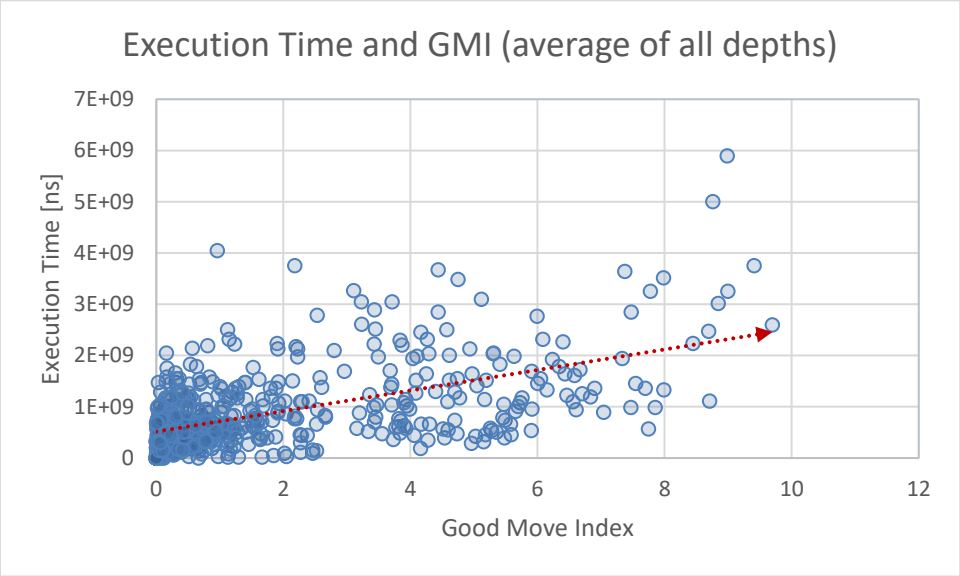


Figure 4.4.7. Scatterplot of each search's execution time and its average Good Move Index (GMI) of all depths combined. The trendline demonstrates the correlation between the two.

The correlation between the GMI and execution time is significantly weaker in the MT variant of KLAS than the MO variant (see figure 4.3.1, chapter 4.3.1).

4.4.5 Garbage Collection

The Java Virtual Machine's garbage collection was recorded both in terms of the number of calls to the garbage collector as well as the time used to collect the garbage at the end of each search. The average for all five variants of KLAS is presented in the table below.

Variant	GC Count	GC Execution Time [ms]	Total Execution Time [ms]	Garbage Collection/Total Execution Time [%]
MT	5.5	208.1	850	24.5
ST	2.6	100.0	1280	7.8
MO	8.5	312.5	4066	7.7
TT	4.4	143.7	2123	6.8
ID	3.7	141.5	1796	7.9

Figure 4.4.8. The number of calls to the garbage collector as well as its average execution time to collect the garbage relative to the total execution of the average search.

As can be seen in figure 4.4.8, the execution time penalty incurred by garbage collection was quite similar for all the variants of KLAS except for one, the MT variant. On average, garbage collection accounted for around 7 to 8% of the execution time of the four lowest variants while it accounted for more than 24% of the execution time of the MT variant.

4.4.6 Playing Strength of KLAS

No serious attempt at measuring KLAS' playing ability accurately was made. However, we tested KLAS by having it play against different chess bots on the website Chess.com. KLAS was able to beat a bot that the website has given an estimated rating of 2100 ELO. KLAS was given 30 seconds of time per move during this game.

4.5 Results Analysis

The optimizations move ordering, iterative deepening, Lazy SMP, and Transposition Tables all gave a significant boost to KLAS' performance, in our empirical measurements. Most significant was MVV-LVA move ordering which gave a 68.5% reduction in average execution time. In second place comes using the transposition table for achieving alpha/beta cut-offs, which gave a 39.8% reduction in average execution time. Lazy SMP gave the third greatest reduction in average execution time at 33.1% with up to four threads searching simultaneously. Least effective was iterative deepening with PV-ordering which reduced average execution time by 28.7%.

Below each optimization is discussed individually, in its own section.

4.5.1 MVV-LVA Move Ordering

Move ordering according to MVV-LVA was tested to answer research question 1: how much execution time can MVV-LVA save on average? In our tests, MVV-LVA move ordering resulted in a decrease in execution time of 68.5% on average and was thus the single most effective optimization applied at improving the performance of KLAS.

MVV-LVA achieves this performance boost by improving the ordering of moves at every depth of search tested, although the improvement was far greater on greater depths. No attempts at measuring the overhead incurred by MVV-LVA were made but we draw the conclusion that it is highly justified, at least past depth 2.

Although MVV-LVA is very effective at deeper search depth, it does not improve move ordering that much at lower depths (see Figure 4.4.1 chapter 4.4.1). By using iterative deepening and PV-ordering in conjunction with MVV-LVA, move ordering is improved at both greater and lower depths.

4.5.2 Transposition Table

The usage of a transposition table was measured to answer research question 1.2: how much execution time can transposition tables save on average? The answer found through our testing is 39.8% when the entries are used for alpha/beta cut-offs.

The transposition table in KLAS is also used for PV-ordering as well as achieving cut-offs from previously found transpositions. Together with PV-ordering, the performance boost that the transposition table results in is significantly higher than 39.8%.

The TT variant of KLAS managed to achieve a lower average GMI on search depth 0, 3, 4 and 5. This could be explained by more nodes being searched again due to fewer cut-offs having happened.

The transposition table is also the primary source of memory needed. The increase in memory usage of the certain variants was in all cases significantly lower than the memory the transposition table used.

4.5.3 Iterative Deepening

Iterative deepening with a transposition table was measured to answer research question 1.3: how much execution time can iterative deepening save on average? The answer we found through our testing was 28.7%.

Iterative deepening achieves this performance boost by greatly increasing the number of transposition table hits at lower depths and improving move ordering at lesser search depths. The reason why fewer transposition table hits were found at greater depths we believe is a result of more hits being found at lesser depths.

4.5.4 Lazy SMP

Lazy SMP was tested to answer research question 1.4: how much execution time can Lazy SMP multithreading save on average? Through our measurements we found Lazy SMP to reduce average execution time by 33.1%.

The reduction in execution time due to Lazy SMP comes from several helper threads helping the main thread in its search by filling the transposition table with relevant data for the main thread to use. In our testing, the helper threads succeeded in increasing the number of hits in the transposition table for the main thread at every depth of search, as can be seen in Figure 4.4.5.

Lazy SMP improved the move ordering by decreasing the GMI at every depth of search as well. Why this is we do not know but we suspect it is somehow a side effect of more alpha-beta pruning.

Memory usage was significantly increased at two of the three iterations of iterative deepening when using Lazy SMP. Why memory usage was not increased after the third iteration as well we do not know.

Lazy SMP brings with it a much higher impact of the Java virtual machine's garbage collection. We conclude that the usage of Lazy SMP warrants greater effort to be taken in reducing this impact.

4.5.5 Garbage Collection

The average execution time penalty incurred by the Java Virtual Machine's garbage collection was relatively consistent across 4 out of 5 variants of KLAS, varying between 6.8% to 7.9%. The exception is the multithreaded MT variant where the penalty incurred accounted for 24.5% of the total execution time. We conclude that taking effort to reduce garbage collection is worthwhile when implementing a chess engine in Java, regardless of the specific optimizations used. From our tests, this is particularly warranted when using Lazy SMP multithreading.

4.5.6 Choice of Optimizations

The four optimizations examined in this thesis were chosen for different reasons. Firstly, it was our hope that they would be effective for the goal of reducing execution time. Secondly, they were also chosen as they are all different from each other in how they function. It would have been less interesting to investigate four optimizations that all operate in a similar manner. The expectation was also that they would interfere less with each other. For example, if four different move ordering heuristics were implemented, the effectiveness of each individual heuristic optimization would probably be diminished due to the other three.

4.6 Threats to validity

To ensure that measurements of execution time are accurate, and generalizable, outside factors need to be eliminated. Any other processes running on the test system when measurements are made can interfere with the results and so execution time is measured in CPU cycles instead of wall-time using the Java class ThreadMXBean [9]. Other possible outside factors include the optimization performed by the Java Virtual Machine as well as interference from its garbage collector.

4.6.1 Board Positions

The board positions used for all measurements consist of original boards and boards generated from the original boards. The original boards were taken from various online resources and mostly constitute example positions from chess lessons as well as positions from various professional matches. It is possible that the board positions used are not representative of chess as a whole, as most of them are from the middle game. The reason why we chose middle game positions is because this is where the greatest complexity and greatest branching factor is found in chess.

4.6.2 Java Virtual Machine Optimization

The Java Virtual Machine dynamically optimizes Java programs during runtime, this is done for example when a particular method has been run many times – it is optimized to run faster on subsequent calls. However, these dynamic optimizations introduce variance in runtime performance. To reduce these variances in runtime, we let KLAS search method run 60 times without any usage of a transposition table at the start of each measurement to make the JVM optimize as much as possible before actually running a measurement. This should, along with the sample size used for our results, reduce the JVM's optimization as a significant source of error.

4.6.3 Interconnectivity of Algorithms and Heuristics

The different heuristics and algorithms examined in this thesis are interconnected, meaning they affect each other's function. For example, the different threads used in the MT variant of KLAS all use MVV-LVA move ordering, which means that the results of how effective Lazy SMP is might be dependent on the MVV-LVA heuristic. It's possible that Lazy SMP would be more effective or less effective if MVV-LVA move ordering had not been used.

5 Conclusion

Four different algorithmic and heuristic optimizations were examined in this thesis: MVV-LVA move ordering, transposition tables, iterative deepening and Lazy SMP multithreading. All four optimizations resulted in significant performance boosts to the KLAS chess engine with MVV-LVA move ordering having the single biggest impact. MVV-LVA resulted in an average reduction in execution time of 68.5% and the transposition table, when used to avoid researching nodes, resulted in a reduction of 39.8% of average execution time. Lazy SMP parallel search and iterative deepening resulted in the average execution time being reduced by 33.1% and 28.7% respectively.

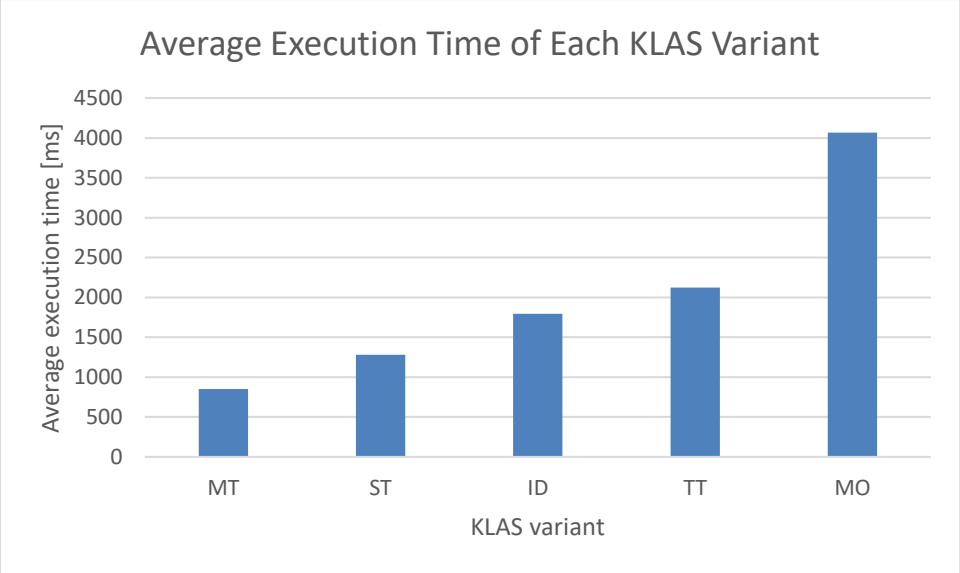


Figure 5.1. The average execution time of each variant of KLAS, demonstrating the impact of each of the four algorithms/heuristics.

We conclude that all four optimizations are worth implementing in a chess engine, particularly MVV-LVA move ordering as it has few downsides. The transposition table, while it is a very powerful optimization, has the downside of using a lot of memory. As long as there are no strict memory constraints then it is a very worthwhile optimization.

Lazy SMP is effective when implemented in Java and run on an Intel CPU with four cores. This effect is however dependent on the system having more than one core. It would be interesting to see how well Lazy SMP scales with an even higher number of available cores.

A significant increase in the amount of time the Java Virtual Machine spent collecting garbage was recorded in the MT variant of KLAS. We conclude that it is warranted to take effort to reduce the amount of garbage generated if implementing Lazy SMP like in KLAS.

Iterative deepening has a significant positive impact on the performance of a chess engine and if ease of implementation is a consideration, then it is very worthwhile. In KLAS the effect of iterative deepening was dependent on the existence of the transposition table, however it is possible to implement it without any sort of hash table.

6 References

1. J.A.N Lee. *Konrad Zuse.* IEEE Computer society. URL: https://history.computer.org/pioneers/zuse.html
2. Alan Turing, Jack Copeland (ed.) (2004). *The Essential Turing, 572.* Oxford: Oxford University Press.
3. Claude Shannon (1950)*. Programming a Computer for Playing Chess* (PDF)*. Philosophical Magazine. **41** (314).* Retrieved September 6, 2021*.*
4. Daniel Edwards, Timothy Hart (1961). *The Alpha-Beta Heuristic*. AIM-030. Massachusetts: Massachusetts Institute of Technology. URL: http://dspace.mit.edu/handle/1721.1/6098.
5. Knuth, Donald E., and Ronald W. Moore*. An Analysis of Alpha-Beta Pruning*. Artificial Intelligence, vol. 6, no. 4, 1975, pp. 293–326. DOI: 10.1016/0004-3702(75)90019-3
6. Albert Zobrist (1970*). A New Hashing Method with Application for Game Playing.* Madison: Computer Sciences Department, University of Wisconsin. *URL:* https://research.cs.wisc.edu/techreports/1970/TR88.pdf. Retrieved September 20, 2021.
7. T.A Marsland (1991). *Computer Chess and Search, pp* 12. Computer Science Department. Edmonton: University of Alberta. URL: https://webdocs.cs.ualberta.ca/~tony/RecentPapers/Marsland-CCandSearch-1991.pdf
8. Daylen Yang (2016). *Stockfish 7. URL: https://stockfishchess.org/blog/2016/stockfish-7/. Retrieved October 6, 2021.*
9. Østensen, Emil (2016). *A Complete Chess Engine Parallelized Using Lazy SMP.* Department of informatics, University of Oslo.
10. Oracle, Java interface ThreadMXBean documentation. URL: https://docs.oracle.com/en/java/javase/11/docs/api/java.management/java/lang/management/ThreadMXBean.html

Henrik Brange
nat15hbr@student.lu.se

A Faster Chess Engine

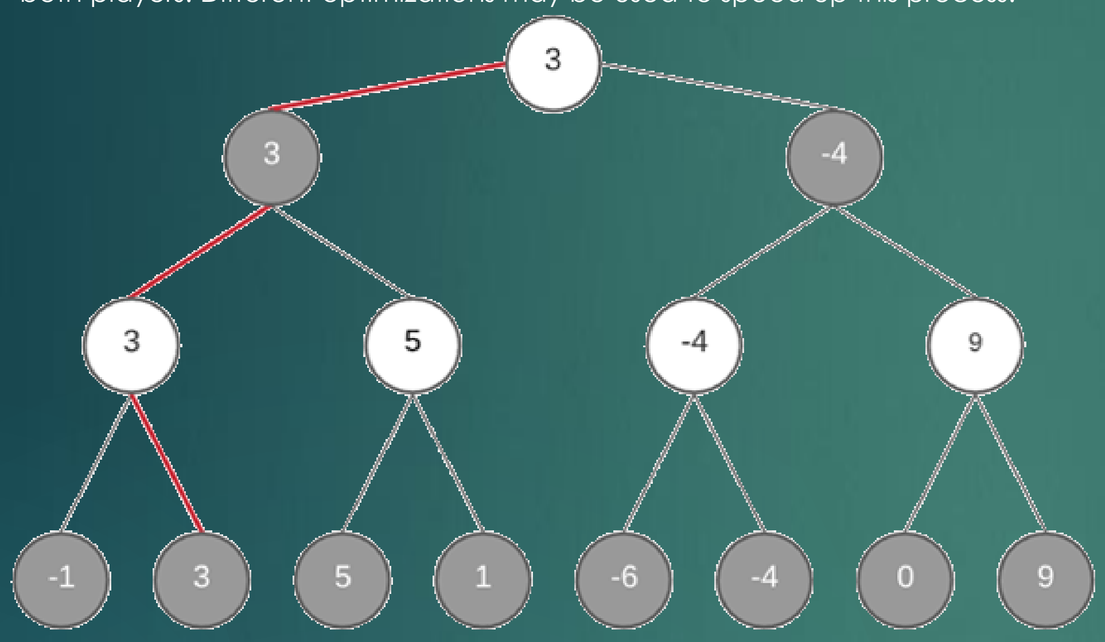
GIVING A CHESS ENGINE A BOOST

Background

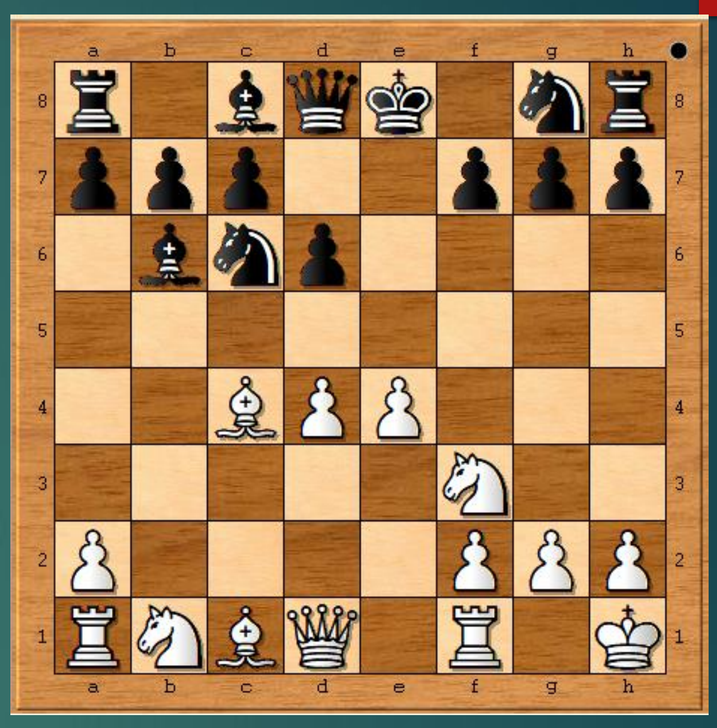
One of the pioneers of computer science, Claude Shannon, published in 1949 his paper *Programming a Computer for Playing Chess.* In which he described how a computer could play chess using a minimax algorithm. The minimax algorithm is based around the idea that both player are trying to maximize and minimize a value alternately. This is the case in chess, if we imagine that a board positions that favors the player with the white pieces are given positive values while positions that are favorable for the player with the black pieces are given negative values.

Search

Search refers to when a chess engine considers what moves are available on a board position, and then plays them on an internal board representation to see what the result is. A chess engine may repeat this process several times to a so called "depth", which is the number of moves from the original board position the engine considers. The result of a search is a list of moves the chess engine expects will be played as it considers them to be the best moves available for both players. Different optimizations may be used to speed up this process.



A search tree where the nodes represent board positions and the edges represent moves leading from the previous position to the next. The numbers in the nodes represent how favorable the position is for the two players. The tree as a whole represents the logical flow of the chess engine.



An example of a board position.

The Optimizations

The purpose of this thesis was to implement a chess engine called KLAS and optimize it to play faster using four different optimizations. These optimizations are:

Most Valuable Victim, Least Valuable Aggressor (MVV-LVA) – A move ordering heuristic which prioritizes moves where an enemy piece is taken. The more valuable the enemy piece and the less valuable the taking is piece is, the higher the priority.

Iterative deepening – An algorithm that involves deepening the search iteratively. Iterative deepening means that the chess engine first searches two steps ahead, then four and then six, and so on.

Transposition table – A data structure storing information about so called previously investigated board positions so that they may be reused at a later point in time.

Lazy SMP parallel search - An algorithm for multithreading the search of the chess engine. It uses the transposition table for sharing information between the different threads.

Methodology

In order to investigate the potential of these four optimizations a Java chess engine named KLAS was written and five different versions or "variants" of it were defined:

MT – Complete, multithreaded KLAS which employs all four techniques.
ST – Complete KLAS except Lazy SMP-multithreading is disabled.
ID – Single-threaded KLAS with no iterative deepening.
TT – Single-threaded KLAS without using a transposition table for alpha/beta pruning.
MO – Single-threaded KLAS without MVV-LVA move ordering

These five different versions were used to search 600 different board positions of chess and the execution time, as well as other relevant information, was measured. After the measurements were performed, the results were analysed to find out how these techniques function in practice and how well they perform.

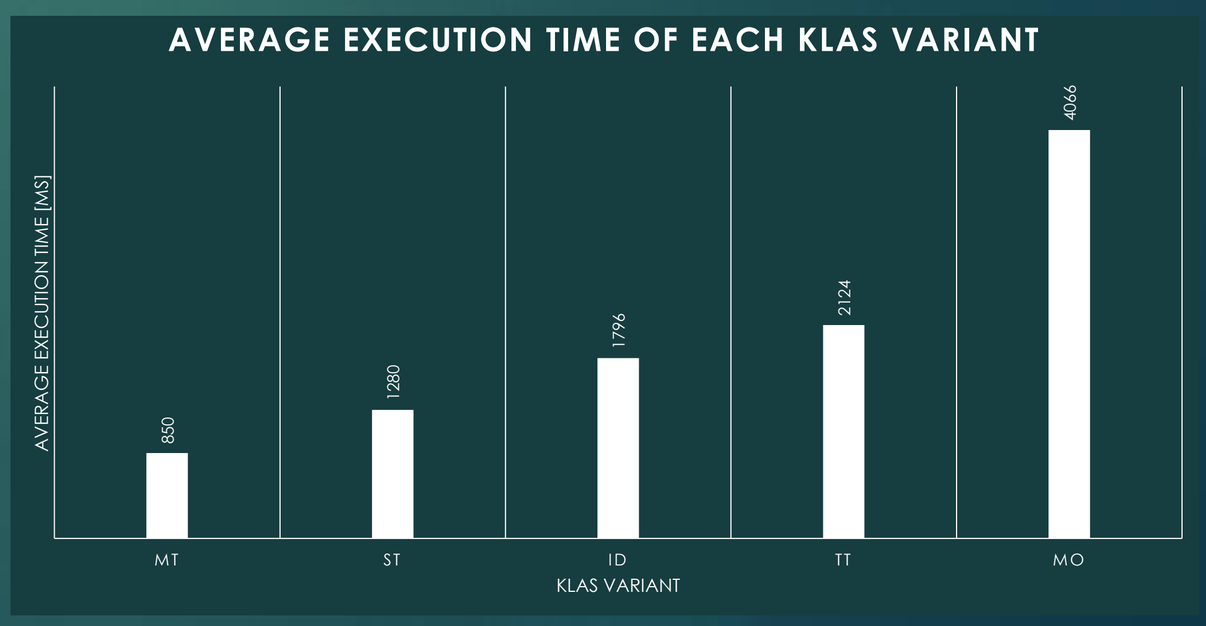
Results

All four techniques resulted in significant performance boosts to the KLAS chess engine with MVV-LVA move ordering having the single biggest impact. The **MVV-LVA** heuristic decreased average execution time by **68.5%.**

The **transposition table**, is a very powerful technique as well decreasing average execution time by **39.8%** on average when used to avoid researching board positions.

Lazy SMP decreased average execution time by **33.1%** on average on the four core test system. However it also significantly increased both the memory usage as well as the time the Java virtual machine spent on garbage collection.

Iterative deepening reduced average execution time by **28.7%.**



Conclusion

We concluded that each of the four techniques were effective in decreasing KLAS' execution time with different upsides and downsides. The move ordering heuristic MVV-LVA provides the biggest reduction in execution time and has few downsides. The transposition table is another very powerful optimization but it has the downside of requiring a lot of memory. Lazy SMP is effective when running on the Intel i5 6600K test system, which has four cores, but is of course dependent on the CPU having more than one core.