



UNIVERSITY OF
LIVERPOOL

COMP390

2022/23

Chess Engine AI

Student Name: Leon Szabo

Student ID: 201515134

Supervisor Name: Flávia Alves

Department of
Computer Science

University of Liverpool
Liverpool L69 3BX

Acknowledgements

First of all I would like to express my gratitude to Flávia Alves for her continued support and knowledge during the production of this Engine. I could not have done it without her, without the weekly meetings this project would never have gotten done. Next Igor Potapov, I had some great insight from some of the major gaps in my arguments from you, without it there would be a great lack of mentions of outside knowlege, and I thank you for this.

Korzh Maksim, the author of the Bitboard Chess engine, the youtube series was an excellent guide, without it I would have had no idea where to start or where to go. Thank you.

Abstract

Since 1950, Chess programming has remained a vast but incomplete field of research. However, many breakthroughs have occurred over the years, resulting in Chess Engines surpassing humans. This paper aims to provide a resource covering how to construct a Chess Engine by producing an overview of the implementation process of a 2300-rated Engine from start to finish while comparing the implemented techniques with other notable standards.

The paper covers the implementation of Bitboards, Negamax, alpha-beta pruning, Quiescence search, Iterative Deepening, Move ordering, Principal Variation Search, Late Move Reductions, Null Move Pruning, Futility Pruning, Evaluation Pruning, Mate Distance Pruning, Transposition Tables, Tapered Evaluation, Mobility, and King Safety.

Contents

1	Introduction	4
1.1	Background	4
1.2	Aims & Objectives	4
1.2.1	Aims	5
1.2.2	Objectives	5
2	Design	5
2.1	Program Overview	5
2.1.1	Overall structure and arrangement	5
2.1.2	Utility functions and Data structures	6
2.2	Connecting to a GUI	6
2.2.1	Commands	6
2.2.2	Use of Go's Concurrency	8
3	Implementation	8
3.1	Board Representation	8
3.1.1	Representing a Chessboard	8
3.1.2	The Bitboards strengths and weaknesses	9
3.1.3	Essential functions and variables	10
3.1.4	Commonly used bitwise operations	12
3.1.5	Attack sets, patterns, and initialization	14
3.1.6	Move Generation	16
3.1.7	Copy and Make board state	18
3.1.8	Making the move	18
3.2	Search	19
3.2.1	Negamax with alpha-beta pruning	20
3.2.2	Quiescence search	21
3.2.3	Iterative deepening	21
3.2.4	Move ordering	22
3.2.5	Extensions	23
3.2.6	Principal Variation Search	24
3.2.7	Pruning and Reduction techniques	24
3.2.8	Transposition table	26
3.3	Evaluation	26
3.3.1	Piece Values and Piece-Square Tables	27
3.3.2	Tapered Evaluation	27
3.3.3	Mobility	28
3.3.4	King safety	28
4	Testing & Evaluation	29
4.1	Testing	29
4.1.1	Perft	29
4.1.2	Trial and Error	29
4.1.3	Tricky positions	30
4.2	Performance Evaluation	31
4.2.1	Tournament	31
5	Conclusion	32
5.1	Aims & Objectives	32
5.1.1	Aims	32
5.1.2	Objectives	32
6	BSC Project Criteria	33
7	Appendices	39

1 Introduction

For centuries, Chess has remained a timeless classic, played and enjoyed by all while relying solely on the creativity and innovation of human intellect to push the game to new heights. However, 70 years ago, Claude Shannon released a paper named “Programming a Computer for Playing Chess” [1], which forever transformed how people approached the game. Today, the immense power and potential of Chess Engines powered by artificial intelligence have led to the emergence of the strongest players ever known[2][3]. As such, Chess Engines are now more critical than ever, enabling players to analyze misplays and improve their game, whether online or offline. In this paper, we aim to develop an Engine that can serve as a valuable resource and stepping stone for other engines and players looking to hone their skills.

While the strongest Chess players may not be able to compete with the strongest Engines, being able to provide a position to get a so-called ‘best move’ can be an incredibly potent tool for preparation and learning. However, this paper plans to create an alternative way to enhance a player’s skill by making an Engine that players can compete against, giving players an option to learn in real-time.

Chess Engines are usually tuned by slowly making changes and then playing against previous versions to see if the changes are beneficial. However, there is also the case of playing against other Engines to provide a better overview of its strength; if two engines of vastly different skill levels play against each other, this may create complete losers or winners, leading to an inaccurate performance measure; providing a diverse range of Engines to test and compete against removes this problem.

Nevertheless, the main reason for creating this Engine is to provide yet another resource for Chess Programming. Having a complete overview of the creation process of an Engine from start to finish while providing alternative implementations could help someone who is looking to similarly dive into the world of Chess programming to either create an Engine or enhance their current one.

The Design section (2) covers the general structure of the program, along with how it relates to the original plan and current implementation. Next, the Implementation section (3) will provide a comprehensive overview of how the Chess Engine was built, covering the Board representation (3.1), Evaluation (3.3), and Search (3.2). While the Board representation and Evaluation sections draw heavily from existing Engines as they are more practical and have a clear objective, the Search section involves more theoretical ideas, requiring extensive experimentation and tuning based on research papers, conferences, articles, and existing Engines.

Some understanding of computer science terminology is required, but understanding of Chess Programming as a whole is not, whenever uncommon terminology is used, it will be explained first. Obviously knowing about these topics will make some of the more complicated sections easier to read but the paper is written in a way to be understood by the general computer science audience.

1.1 Background

After the publication of [1], not long after, Alan Turing and David Champernowne released the first Chess program, aptly named Turochamp[4]. Even though it was not that strong, both Turochamp and “Programming a Computer for Playing Chess” laid the foundations of Chess Programming.

As decades pass and computing technology advances, the release of Deep Blue sets Chess Programming back into motion, “On May 11, 1997, an IBM computer called IBM Deep Blue beat the world chess champion” [5], this milestone was a turning point. After this, Engines just kept getting stronger, surpassing human limits.

The most recent development has been using Neural networks in Chess programming. Alpha-zero pioneered this advancement by “using neural networks together with advanced reinforcement learning algorithms” [6] to allow the Engine to practically reinvent Chess by how it played the game, unlike any other Engine at the time. Many Engines use neural networks now, such as Stockfish NNUE[7].

1.2 Aims & Objectives

The aims and objectives here were created before the implementation of the Chess Engine began, but they still briefly cover what is desired from this Engine. For example, it should be a competent opponent for both players and other Engines, and the Engine requires a way to interact; the desired solution is to create a GUI, but there may be other solutions. The Conclusion (5) covers if and how they were achieved.

1.2.1 Aims

- Program a chess engine
- Create a stylistic and simple GUI
- Add player vs AI compatibility

1.2.2 Objectives

- Develop a chessboard representation
- Develop a chess search and evaluation algorithm
- Implement automated testing through GitHub
- Evaluate performance of chess engine
- Add intractability to GUI

2 Design

This chess engine draws inspiration from various sources and engines. Initially, the Chess Programming Wiki was intended to be the sole resource used throughout the project. However, it quickly became apparent that it was only helpful for discovering new techniques rather than building a Chess Engine from scratch. Hence, a guide with reference materials was required, and Maksim Korzh, author of BBC[8], was chosen as the guide, using Counter[9], Crafty[10] and other Engines as reference materials. As my understanding of chess programming grew, I began to infuse more originality into the work, which occurred around the time of Section 3.2.

The original plan did not go too in-depth about many of the topics; the only concrete ideas discussed were bitboards (3.1.1), hand-crafted evaluation (3.3), and alpha-beta pruning (3.2.1), which were all implemented. As for the Testing and Evaluation, that has changed completely; unit tests were to be used alongside Githubs auto testing capability, but now that has been replaced with perft tests (4.1.1) in board representation (3.1), and trial and error (4.1.2) in search (3.2) and evaluation (4.2), with tournaments to find the ELO rating (4.2).

2.1 Program Overview

This section will cover the main structure of the Chess Engine. The specifics will be under Implementation (3).

The idea of using Go has stayed the same; the Engine is written entirely in Go. The main reason for this was that since many Chess Engines are written in C++ or C, purely for the speed benefit it offers, an Engine written in Go will be easier to write, read and understand; the language difference might also make the Engine play differently with possibly more interesting moves.

Almost every Chess Engine has the same skeletal structure: a board representation, a search, and an evaluation. However, as seen in Stockfish's source code [11], the stronger it is made, the more obfuscated it gets; this is the primary reason why only the core ideas of chess programming were implemented, with still a couple of interesting techniques, the core of the Chess Engine should still be seen and understood so that it could be used as a resource, strong Chess Engines exist out there, so there is not much point in making a clone of one.

2.1.1 Overall structure and arrangement

The program has four packages, 'board' covering Board Representation, 'engine' covering Search, 'eval' covering Evaluation, and 'uci' covered in Section 2.2, when going over the source code, it should be organized and commented well enough so that relating them to this paper will not be difficult.

Most functions are commented; only the pure utility functions used for testing are not commented as they do not relate to the Chess Engine and are only used for particular purposes. The core of the comments were generated using Github Copilot[12] and rewritten to make more sense or make them more relevant to the function they are describing. Since most of the commenting was done weeks after the functions were made, Copilot sped up the process of re-understanding the functions and made commenting much quicker.

2.1.2 Utility functions and Data structures

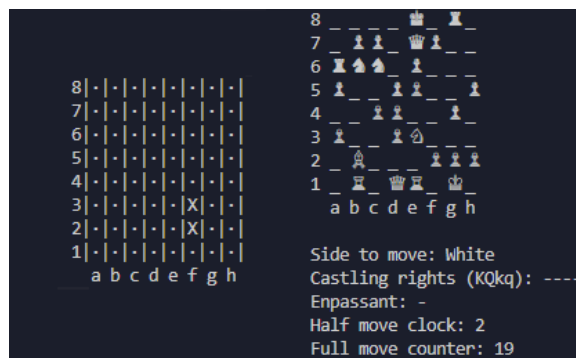


Figure 1: PrintBitboard() on the left, PrinChessBoard on the right

This section will go over the functions that are not used by the Engine but instead were used for testing (functions used in Engine in section 3.1.3). The most used utility functions were PrintBitboard() and PrintChessBoard() from the board package; since a bitboard 3.1.1 is just an int, to visualize what that int holds, a function is needed (left of figure 1). The other function is similar but holds information on the collection of bitboards, the ChessBoard; it prints an ASCII representation of a Chessboard showing all the pieces they hold (right of figure 1).

The Engine utilizes three primary data structures: the ChessBoard struct (see 3.1.1) referencing the Chessboard, the Move struct (see 3.1.6) usually extended to an array holding generated moves, and the Transposition Table struct (see 3.2.8) containing transposition data. Structs were chosen for their efficiency and ability to store information densely.

During development, there were instances where specific data structures significantly slowed down the Engine's performance. For example, in the initial implementation of the MakeMove() function (see 3.1.8), a map was instantiated every time the function was called, creating a significant bottleneck. However, this issue was resolved by replacing the map with an array, resulting in a surprising ten-fold speed improvement for move generation; later, all recursive calls to map instantiations were replaced with arrays, improving the Engine's speed.

2.2 Connecting to a GUI

Originally the plan was to create a GUI; it was an optimistic plan since I do not have much experience with creating any GUIs, not to mention the lack of knowledge about frontend languages; trying to balance creating a good Engine with a good GUI did not work out. However, that does not mean it cannot interact with an existing GUI; this is where the Universal Chess Interface (UCI) comes in; it allows an engine and GUI that both have the protocol implemented to interact with each other, another older protocol would be 'Winboard', although this has practically been phased out of Chess Programming as the UCI protocol is simpler to implement.

When the program first starts up, it starts to initialize all the tables required for the Engine, then it starts up the goroutine (essentially a cheap thread) with a channel that can communicate with the main Engine, and then it waits for a command to be received.

2.2.1 Commands

This Engine does not implement many of the commands available to the protocol, only the main ones needed to start a game. There are also 'option' commands in the protocol, which this Engine does not implement; the default parameters should not really be changed; if there is ever a need to, then editing the source code will do. UCI is easy to implement because it works through print statements, allowing any language with a print capability to implement this protocol.

- 'uci'

The first command sent to the program, when the engine receives it, it will print 'uciok' to let the GUI know that UCI is implemented, if 'uciok' is not returned, then it can be assumed that the engine does not implement UCI.

- ‘isready’

The GUI sends this to check if the Engine is ready to parse the position and search it. The initial implementation had no issues with it, but as Concurrency was implemented (more in 2.2.2), sometimes it would tell the GUI to start sending commands even though it was not ready; it took a while to fix this bug, it was found that the GUI does not stop sending commands, so without some buffer, the commands would get lost.

- ‘ucinewgame’

This command starts a new game by creating a Chessboard struct called ‘cb’; all other functions and operations within the program work from this cb variable. Additionally, this command resets the transposition table, an important component of the Engine’s search algorithm (more in 3.2.8).

- ‘position’

This command serves to parse positions from FEN strings. To accomplish this, the ParseFen() function is utilized. There are several ways to use this command to parse a position. For instance, a user can use the starting position by executing ‘position startpos’. Alternatively, users can execute “position fen *<fenstring>*” if they want to use a custom FEN string. As the game progresses and moves are made on the board, the GUI must communicate this information to the program. If a FEN string has been used and two moves have been made, the command “position fen *<fenstring>* moves *<move1>* *<move2>*” can be used to parse the position and moves. The Engine uses its MakeMove() function to parse the moves, as discussed in Section 3.1.8.

- ‘go’

After a new game has started, with a position parsed, the program runs ‘go’ to start the Search and Evaluation. Only a brief overview of the output (seen in 43) will be given here (more information in 3). The first output, ‘depth’, is how deep the search goes; a move increments whenever white and black make a move on board, but a Ply increments every half move, so when either white or black makes a move, this depth parameter goes up in Plies. The second output, ‘nodes’, is incremented whenever the game tree searches for a node. The ‘cp’ output is the abbreviation of centipawns, measured as 1/100th of a pawn; this evaluates the position for the current side to move. ‘time’ is measured and returned as milliseconds. Lastly, Every ply after ‘pv’ is the line of best moves for that current side the Engine finds.

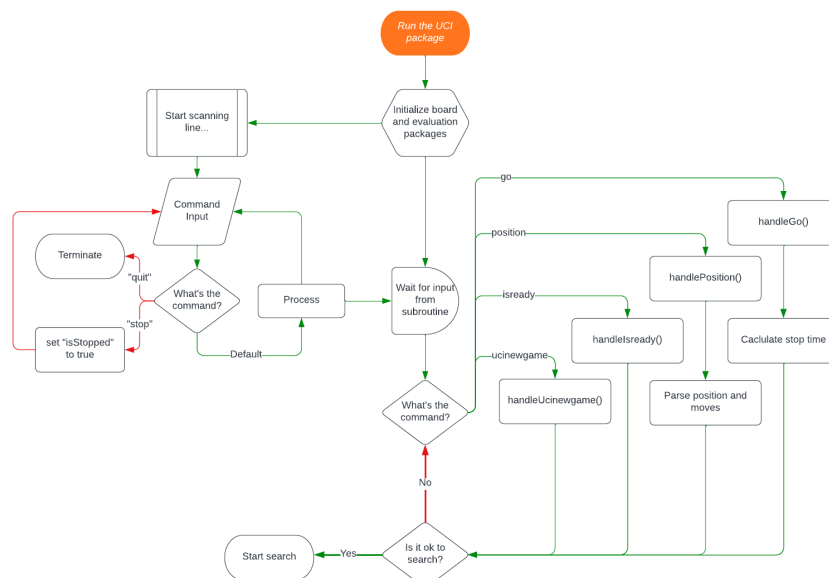


Figure 2: Flow chart from the start of UCI to searching the position

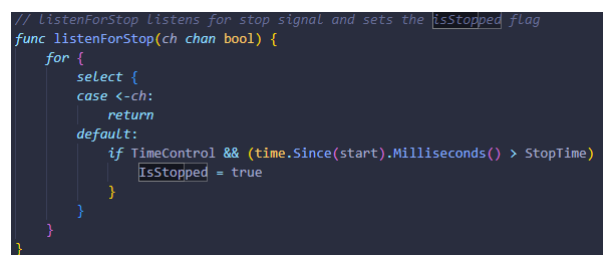
The commands were designed to be robust and tolerant of errors. To achieve this, a buffer is used to allow users to input multiple commands at once without the risk of losing any due to a full variable. Additionally, error handling was implemented to make the inputs resistant to incorrect formatting.

Since the Go language provides an easy-to-use error-handling mechanism, it was used to ensure that the program handles all errors gracefully.

Figure 2 shows a Flow chart of the UCI implementation. A separate sub-routine handles the commands so the previously talked about buffer can exist, along with robust line scanning. As errors propagate up the functions, they are all dealt with within their respective handlers. For example, sometimes a search should not be started if the correct commands have not been appropriately parsed or a game has not been set up.

2.2.2 Use of Go's Concurrency

Chess games are held with time controls, and they tell the Engine how much time they have left to finish the game or finish a certain number of moves within that time. The GUI communicates this information with the Engine, and the Engine then has to determine how long it should search for; some more potent Engines like Stockfish [11] have more complicated time control implementations, so they might vary the time for a move to be shorter or longer depending on if the position is tricky. This Engine has a simple time control implementation.



```
// listenForStop listens for stop signal and sets the IsStopped flag
func listenForStop(ch chan bool) {
    for {
        select {
        case <-ch:
            return
        default:
            if TimeControl && (time.Since(start).Milliseconds() > StopTime) {
                IsStopped = true
            }
        }
    }
}
```

Figure 3: Function running on a goroutine

Other Chess Engines, such as BBC [8], may have calls within their search function that communicate with the GUI to determine if the search should be stopped. However, this engine utilizes Go's built-in concurrency to achieve this functionality (Figure 2). Specifically, the check for whether the engine should be stopped runs on a separate goroutine while the search runs on the main thread. When the search passes the time for when it should be stopped, a variable (IsStopped) is set to true. The search function then checks this variable and stops the search if it is set to true (seen in figure 3).

3 Implementation

The implementation will be divided into three parts; Board representation - which will go through the process of generating legal moves from a chessboard; Search - as legal moves are generated and made on the board, there will need to be a way to reduce the number of moves to be able to reach a deeper depth; Evaluation - The positions are evaluated to determine whether a move should be ignored or looked at further. The search and evaluation depend on each other to work, but the sections will be separated to clarify what was implemented in each. Sections reference each other when appropriate.

3.1 Board Representation

Board representation is the foundation of any Chess Engine; this will be responsible for keeping track of the position, detecting repeated positions, and generating and returning legal moves given a position. As talked about on the Chess programming wiki[13], it is crucial to write bug-free code, as I have experienced, even minor bugs will cause huge problems, and pinpointing the source of these bugs becomes a tremendous pain as many large functions work in conjunction with each other, making it difficult to determine what is causing issues

3.1.1 Representing a Chessboard

There are two main approaches to representing a Chessboard, a piece-centric approach, where one would store what squares a particular piece occupies on the board, and the opposite square-centric approach, where one would store information on whether a square is empty or contains a particular piece[14].



Figure 4: A bitboard

Within these approaches are different ways to represent a board. There is still discussion on which approach is fastest; many strong Engines use a bitboard approach, examples would be Crafty[10], BBC[8], Counter[9], and some combine bitboards with a Mailbox approach such as Stockfish[11], there is no concrete standard yet, there are still new developments being made. The reason for choosing the bitboard approach in this Engine came down to how interesting some techniques are; for example, the magic bitboard enhancement (inside section 3.1.5) is a perfect hashing algorithm used to index a database.

```
type ChessBoard struct {
    WhitePawns  uint64
    WhiteKnights uint64
    WhiteBishops uint64
    WhiteRooks  uint64
    WhiteQueen  uint64
    WhiteKing   uint64

    BlackPawns  uint64
    BlackKnights uint64
    BlackBishops uint64
    BlackRooks  uint64
    BlackQueen  uint64
    BlackKing   uint64

    WhitePieces uint64
    BlackPieces uint64
}
```

Figure 5: A collection of bitboards representing a Chessboard

A single bitboard is not very helpful, as having information on every piece is necessary; one will need to represent a collection of bitboards and use them together as a single Chessboard; the Engine uses a struct of fourteen bitboards, with a bitboard for each piece (six for white and six for black) and two extra bitboards for the collection of all white pieces and all black pieces, these extra bitboards come in handy as it saves having to do a bitwise-OR operation of all pieces of a particular colour every time we need this information, the collections of bitboards idea came from Crafty[10]. There are different ways to represent these bitboards, such as storing particular pieces of both colours in a single bitboard; in this representation, white pawns and black pawns would be stored in a single bitboard called pawns, this is how Counter stores moves[9], and if one wants to only access information on white pawns, a cheap bitwise-AND on white pawns and white pieces will provide this, this saves space as fewer bitboards are stored, and as seen developing the Engine, it may have also saved time as many bitwise-ORs were used when trying to get both colours of a piece, changing the Chessboard struct would have taken too long and was not done. However, in hindsight, this other representation may have proved to be better.

3.1.2 The Bitboards strengths and weaknesses

These points are mentioned on the Chess programming wiki[15]; the points below have more information, sometimes referencing how it relates to the Engine.

There are benefits to using such an approach, such as

Information density

A single bitboard is only a 64-bit integer, and it can store information on every square of a chessboard, and this density scales as more squares on the board are 'on' (bit set to 1).

Bitwise instructions

Arguably the most substantial reason to use bitboards, bitwise operations are cheap and inexpensive, giving a solid performance benefit when certain information is required, information like all white pieces except pawns with only a single operation (WhitePieces AND NOT WhitePawns); other approaches do not allow this flexibility making bitboards a great option.

Built-in instructions

Some programming languages like C++ have built-in functions that allow for rapid computation for specific information like counting the number of 'on' squares on the board; these built-in functions are

operating system dependent, so they can make a Chess Engine less accessible. However, there are ways to work around this, and the considerable performance benefit outweighs the negatives of using other methods. The Chess Engine does not use these built-in instructions as the language of choice, Go, does not have these. functions; a reasonably fast alternative is used.

And also negatives, such as

Particular piece information

As bitboards store information so densely, information like finding what piece is on a certain square is lost and cannot be accessed directly, one would need to loop through all of the piece bitboards and then use a function to find square information.

Specific information

Although many operations are quick, a lot can not be directly accessed from bitboards, and a bit shift is needed. As seen in Stockfish[11], and mentioned on the Chess Programming Wiki[14], sometimes engines will keep a redundant Mailbox approach to make this information more accessible.

Working with bitboards

Although not inherently a weakness, working with bitboards can get complicated, and functions may need to be clearer to read or understand. Other approaches offer a much simpler way to represent Chessboards. Many chess engines use these more straightforward methods first to understand more about chess programming before later switching to bitboards. As we just wanted a fast approach right away, we chose to implement bitboards first to avoid rewriting the whole Engine by switching to bitboards later in development.

3.1.3 Essential functions and variables

```
var SquareToIndex = map[string]int{
    "a1": 0, "b1": 1, "c1": 2, "d1": 3, "e1": 4, "f1": 5, "g1": 6, "h1": 7,
    "a2": 8, "b2": 9, "c2": 10, "d2": 11, "e2": 12, "f2": 13, "g2": 14, "h2": 15,
    "a3": 16, "b3": 17, "c3": 18, "d3": 19, "e3": 20, "f3": 21, "g3": 22, "h3": 23,
    "a4": 24, "b4": 25, "c4": 26, "d4": 27, "e4": 28, "f4": 29, "g4": 30, "h4": 31,
    "a5": 32, "b5": 33, "c5": 34, "d5": 35, "e5": 36, "f5": 37, "g5": 38, "h5": 39,
    "a6": 40, "b6": 41, "c6": 42, "d6": 43, "e6": 44, "f6": 45, "g6": 46, "h6": 47,
    "a7": 48, "b7": 49, "c7": 50, "d7": 51, "e7": 52, "f7": 53, "g7": 54, "h7": 55,
    "a8": 56, "b8": 57, "c8": 58, "d8": 59, "e8": 60, "f8": 61, "g8": 62, "h8": 63,
    "-": 64,
}
```

Figure 6: Map, converts ‘squares’ to indexes

A utility map (seen in figure 6) defines what a square is; it maps the Chessboard squares to indexes on bitboards; this was made for testing purposes but could be used by a user.

At first, bitboards were not very intuitive to use, and it took a while to understand how to write some functions, but in the long run, it did end up being worth it as the performance benefit is evident, as will be seen in testing (see 4.1). Here are some functions with explanations used throughout the Engine that are general enough to not belong to any other section.

```

// setBit sets a bit on a bitboard
func SetBit(bitboard *uint64, square int) {
    *bitboard |= (1 << uint64(square))
}

// getBit returns a bit on a bitboard
// mostly replaced by indexMasks
func getBit(bitboard uint64, square int) uint64 {
    return bitboard & (1 << uint64(square))
}

// popBit pops a bit on a bitboard
func PopBit(bitboard *uint64, square int) {
    *bitboard ^= getBit(*bitboard, square)
}

// isBitOn returns true if a bit is on
func isBitOn(bitboard uint64, square int) bool {
    return bitboard == (bitboard | (1 << uint64(square)))
}

```

Figure 7: Utility functions

- **SetBit()**

SetBit (Figure 7) uses a bit shift to return the same bitboard with the specified square turned on (square’s bit set to 1); this function is used when creating the attack tables (Section 3.1.5), making a move (Section 3.1.8), and parsing the Chessboard.

- **PopBit()**

PopBit (Figure 7) is the opposite of SetBit; if a bit is set to 1 on the bitboard, this function will set it to 0; however, if the bit was already set to 0, then it will not change it. This function is only used to generate magic numbers (seen in 3.1.5) but could probably be phased out completely using another method, other engines may use this function a lot more, but in the programming language Go, whenever we loop over a bitboard “bitboard &= bitboard - 1”, is used to remove the next bit instead of using PopBit.

- **isBitOn()**

isBitOn (Figure 7) returns true if the indexed bit is set to 1; otherwise, it returns false. This function is used in move generation to check whether a piece is attacking another piece, treating these moves differently if this is the case.

```

// The "Brian Kernighan's way" of counting bits on a bitboard,
// implementation idea from chess programming wiki
func BitCount(bitboard uint64) int {
    var count int
    for bitboard != EmptyBoard {
        count++
        bitboard &= bitboard - 1
    }
    return count
}

```

Figure 8: "Brian Kernighan's Way"[16] to count bits on a bitboard

There are many reasons to count the number of 1-bits on a bitboard; as an example, let us say someone wants to know how many pawns are left on the board; they would need to use a function to count the number of 1-bits to get their answer, there are a plethora of ways to do this, the Engine uses Brian Kernighan’s bit count to get this answer, as seen in 8, it is a pretty simple method. “bitboard &= bitboard - 1” as used previously, lets you loop through the bitboard by turning every next bit to a 0 and then counting how many loops it takes to reach an all 0 bitboard. This function is used throughout the Engine, and as said on the Chess programming wiki[17], it is still considered fast for counting bitboards with few 1-bits but is not great for counting bitboards with many 1-bits; even with this bottleneck, it is still used as the only bit count method, C has a built-in pop count method that can be reasonably

assumed to be faster, and could be implemented as an alternative, but as stated before the choice was made to not use built-in methods.

```
// Kim Walisch's proposed ones' decrement to compute
// the least significant 1 bit used in BitScanForward()
func BitScanForward(bitboard uint64) int {
    const debruijn64 uint64 = 0x03f79d71b4cb0a89
    if bitboard != 0 {
        return index64[((bitboard^(bitboard-1))*debruijn64)>>58]
    }
    return -1
}
```

Figure 9: Kim Walisch’s suggestion for getting the least significant bit

Counting the number of 1-bits is helpful, but some functions require information on the location of the 1-bit, the other techniques above do not record this information. There are many ways to get this information, but we use de Bruijn sequences to index a 1[18]; this technique was originally proposed by Charles E. Leiserson, Harald Prokop and Keith H. Randall, but the small optimization “bitboard^(bitboard-1)” was proposed by Kim Walisch. This technique works very well and is used throughout the Engine.

As seen in these functions, they are not very complicated yet produce a fast, effective result; we do not sacrifice much performance choosing these approaches (seen in 4.1), readability and understanding are more important to us than having the fastest algorithms, even built-in operations may be CPU dependent, and limit uses to specific computers. Other smaller functions deal with smaller operations, but the shown functions are used quite often; most functions rely on these methods, so going through them in detail is vital for understanding the rest of this paper.

3.1.4 Commonly used bitwise operations

This section is mainly to build the reader’s understanding of how some bitwise operations are used within bitboards to get desired information and how this relates to the Engine; there will be figures that use the program’s PrintBitboard() function, this just pretty prints a bitboard, to help remember the many different operations used in the Engine, Chess programming wiki’s “General setwise operations” page will be used[19].

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

Figure 10: WhitePieces OR BlackPieces = AllPieces

As seen in figure 10, a bitwise-OR is used to find every piece on all the bitboards, and the same operation was used to find the WhitePieces and BlackPieces bitboards (all piece bitboards of colour were combined with Bitwise-OR operations), this operation is only used to find those three bitboards and other operations are used to work with them, let us say we wanted all non-white pawns, we could use an OR operation on all the white piece bitboards except white pawns, taking 4 OR operations, or we could do WhitePieces &^ WhitePawns, to save us some operations. So working with these collections of bitboards tends to be more accessible and takes fewer operations than working with all the bitboards individually. This paper might use OR or UNION interchangeably, depending on which fits best.

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

Figure 11: WhiteKnightAttacks AND WhitePieces = DefendedWhitePieces

As seen in figure 11, a bitwise-AND can be used to find which white pieces are defended by that particular white knight; there would be a couple of different ways to find all the defended white pieces; an example would be to find the union of all attacks, and then do a bitwise-AND with WhitePieces. One could similarly do WhitePieces & BlackPieces to find which black pieces the white knight is attacking and also expand this to all the black pieces that are being attacked with the same method as before; this information is more useful in the 3.3 than other parts, but still helpful none the less.

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

Figure 12: NOT BlackPieces = Every square excluding BlackPieces

As seen in figure 12, unlike the other bitwise operations, this one can get you square information outside of the defined bitboards and attack tables; this information could prove to be useful when combined with other bitwise operations (seen in figure 13), this ‘compliment’ operation is used throughout the Engine, for example, let us say we have a bitboard called ‘unsafeSquares’, this bitboard being the union of all attack tables of every piece, finding the compliment of this could get you a ‘safeSquares’ bitboard that could be used inside evaluation 3.3, there are other specific use cases where this trick comes in handy.

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

Figure 13: WhiteKnightAttacks AND NOT AllPieces = MoveableSquares

As seen in figure 13, this operation is helpful in some instances of the like move generation, but more often than not, we would store the NOT AllPieces in a separate bitboard and then later use a bitwise-AND with another bitboard, in essence, we are doing the same as in the figure but making the code slightly more readable. This operation is just an example of a combination of these bitwise operations; often, these operations are chained together to get desired information.

These operations are used a lot more than someone might think; for instance, the ‘&’ operation has been used ~350 times in the Engine, the ‘|’ operation has been used ~120 times, and the ‘^’ operation ~100 times, it is pretty clear how vital these operations are with regards to bitboards, even if these operations are quick, it is still important to minimize them wherever possible as in the case with storing the WhitePieces and BlackPieces bitboards, and with development, other such bitboards have come up that may improve the Engine with them in the main ChessBoard struct, such as WhitePieceAttacks and BlackPieceAttacks, being the union of all piece attacks of that colour, incrementally updating them whenever a move is made, these could be used in the evaluation function and other parts of the code.

3.1.5 Attack sets, patterns, and initialization

At this point, we have only discussed the definition of a bitboard and the set of bitboards as we define it (The ChessBoard Struct). However, it is still crucial to understand the concepts and operations to help ease into the bulkier functions. Firstly before going onto move generation, how specific pieces move needs to be defined; the pieces are categorised into two sets, leapers and sliders. Leapers ‘leap’ across the chessboard; this set includes the pawns, knights, and the king, while the sliders ‘slide’ across the chessboard; this set includes the bishops, rooks, and the queen. Defining how the two sets of pieces move is slightly different, but storing the attacks can be very different. The distinction is made later when defining the attacks.

It is important to note that we do not yet distinguish between legal and pseudo-legal moves (pseudo-legal moves do not take into account whether the king is in check), as these are lookup tables used when moves are generated 3.1.6, the distinguishment is made when the move is made on the board (seen in 3.1.8).

Defining Masks

Before defining the attack tables, we first need to initialize some ‘masks’ that will be useful; if we consider a bitboard, a mask is when certain bits on that board are set to 1, similar to, say, the WhiteKnights bitboard. However, masks are usually constants that we apply to the variable bitboards, and certain bitwise operations can be performed together to get some desired output. So, for example, if someone wants to get all the WhiteBishops on white squares, they should use a bitwise-AND with a mask that has all white squares set to 1.

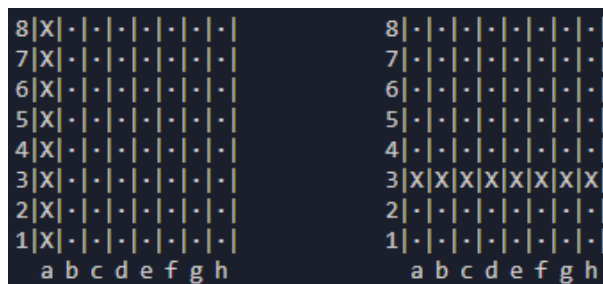


Figure 14: Left image is a file A mask, right image is a rank 3 mask

The most used masks inside the Engine are the File and Rank masks, eight masks for each file and rank, as seen in figure 14; they prove to be useful both when we generate the attack tables and inside the evaluation function 3.3. These masks are used when we want to know information like if the A file is empty or if it contains particular pieces.

Defining Attacks for leapers

Leaper attacks are more straightforward than slider attacks as they do not require special handling when a piece blocks their movement (referred to as a ‘blocker’), so bit shifts and bitwise operations suffice. If we take into account these blockers, a cheap AND NOT 13 operation could be used to remove the squares the piece can not move to, but again these are lookup tables, and this is done when we generate the moves, not when we add them to the table, this is slightly different for sliders.

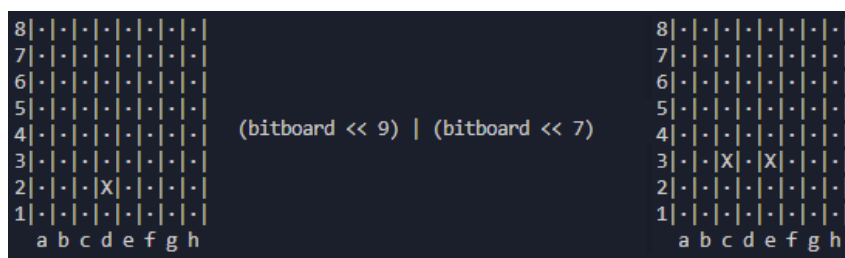


Figure 15: Using bit shifts to get an attack table

As seen in figure 15, two simple bitshifts can be used to get the attack tables for the white pawns; as white and black pawns attack in different directions, they use different bit shifts. However, these bitshifts

are not the whole story; without special edge cases, the bit shifts would push one of the bits over to the h file, creating an illegal move; this is where bit masks come in; in this case, if a NOT H mask is applied, it will get rid of that stray bit, fixing the problem. Figure 16 shows the correct implementation for white pawn attacks.

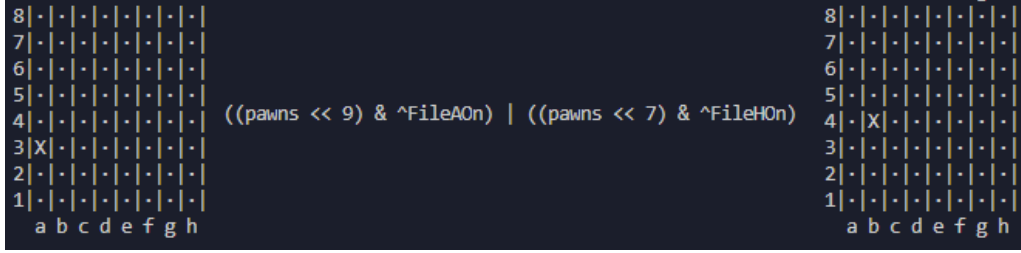


Figure 16: Using bit masks to fix the stray bit

In the implemented pawn attack table creation function, the input is not set on a single pawn, passing a bitboard with mutliple pawns set on it does work, this trick is used in the evaluation 3.3 function to look at all the pawn attacks on a given board without needing to loop through all the pawn attacks and finding the union of them, in every other case the value from the attacks tables is used, as it is cheaper than recalculating the attacks each time.

Since all the leaper attacks use a similar implementation, this paper will not go through the others (Knights, and kings).

Defining Attacks for sliders

The desired output for sliders is bitboard 4 (in figure 17). Many techniques could be used; a typical approach would be ‘Blockers and Beyond’; the approach does not perform well in the opening or middle game, as many pieces still exist on the board, creating excessive interactions with sliding pieces. However, it performs much better in the end game (as fewer pieces mean fewer interactions). The approach we went for stays reasonably consistent throughout the different game phases, and Chess Programming wiki[20] states it to be the de facto standard for bitboard engines. The ‘magic bitboard’ approach, initially motivated by Gerd Isenberg[21] and improved by others, uses a perfect hashing algorithm to index an attack bitboard table that retrieves desired information[20].

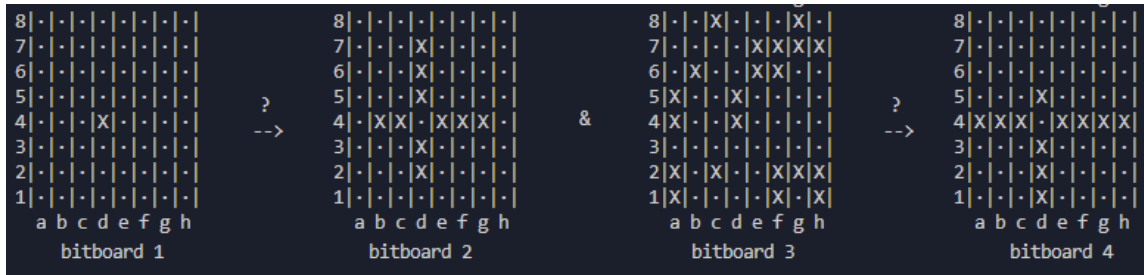


Figure 17: 1. Rook piece square, 2. Rook mask bitboard, 3. AllPieces bitboard, 4. Desired attack output

The process of ‘magic bitboards’ is in itself complicated, that is why the aid of a tutorial[22] to help write the implementation was used, and also an article[23] to help understand it, the article[23] explains magic bitboards very well, so the idea of magic bitboards will not be covered, but the implementation linking them to original authors’ ideas will be. Firstly, we use Lasse Hansen’s[24] ”Plain” magic bitboard implementation, there are many others that improve upon the idea and reduce the memory space required, but we use the ‘Plain’ implementation to save time for enhancing the Search 3.2 and Evaluation 3.3.

As discussed in the article, the ability to produce magic numbers makes magic bitboards work. Torm Romstad came up with the idea of using trial and error on different random numbers to brute force the generation of magic numbers[25]. The Engine has the same implementation when generating magic numbers, which are stored in an array for both bishops and rooks (seen in 40). The magic numbers should be generated before the Engine runs to avoid unnecessary time penalties. In addition, storing the

magic numbers makes it possible to find the best set of magic numbers, the best set is one that produces the fastest average search times on different positions.

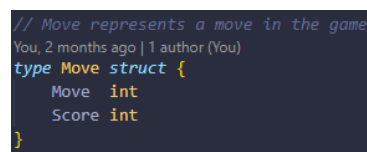
A unique feature of the Engine is the function `PerftTestFindMagic()`. This function produces magic numbers from a range of (random number) seeds and performs a timed exhaustive search to a given depth on a given chessboard. The search times are then compared to find which set of magic numbers produces the fastest search. Although this function is relatively inaccurate as it does not consider uncertainties from external factors such as CPU temperature, it produces magic numbers that seemingly perform better than others. This idea can be extended to find the top X% best magic numbers by repeating the search and finding the average time for these ‘best’ magic numbers to obtain a more accurate result.

3.1.6 Move Generation

Move generation is the first of four core functions, the others being ‘Making the move’, ‘Search’, and ‘Evaluation’. There will be references to previous sections when needed, as all the previously mentioned implementations and techniques will be used. A tutorial[26] was also used to guide the implementation process; there have also been optimizations from Counter[9] (a bitboard-oriented Engine written in Go). However, given the choice of how bitboards are represented inside the ChessBoard struct, there is originality in most of the work.

The Moves

The first step is representing a move; one can define a move by what piece is moving, what piece is being captured (if any), the square the piece is on, the square the piece is moving to, and the flags containing special moves (if any). An int variable stores all of this information; however, later on in move ordering 3.2.4, a score is given to these moves, so it is necessary to turn the move int into a Move struct containing two ints (seen in 18).



```
// Move represents a move in the game
You, 2 months ago | 1 author (You)
type Move struct {
    Move int
    Score int
}
```

Figure 18: Struct representing a move

Since both the start and end squares can be stored as indexes, 6 bits are used to represent the 64 squares available for both the start and end squares; next comes the piece types; unlike how the piece types are stored in the ChessBoard struct, these pieces are only represented by piece type, and not piece colour, so both white or black pawns would be stored as pawns, this is done since storing piece colour is unnecessary as it can be retrieved from the colour of the side to move, the last 4 bits are the flags used to represent what type of move it is, so a double pawn push would have a different flag than a kingside castle, this distinction is made as these moves require special handling when the moves are generated, this information could also be used by both Search and Evaluation for different reasons (seen in 3.2.4).

During the generation of moves, considerations on whether the move is legal are not made; this distinction is made later when the move gets made on the board in 3.1.8, the name for these types of moves is ‘pseudo-legal’ as they still obey the rules of Chess, but ignore whether it puts the king in check—simplifying the implementation in the generation of moves. More discussion on why this is done is in 3.1.8.

Encoding and Decoding

Encoding the moves can save space for moves stored in memory and make the Move struct easier to work with, as it deals with a single int. While it may seem expensive to constantly do bit shifts to turn these conditions into a single int, this technique can be very efficient. However, storing each condition as an int in the Move struct may have been a better decision for some engines.

```
// Values that represent the encoding for the move type

// 0000 0000 0000 0000 0011 1111 - Index of where the move starts
// 0000 0000 0000 1111 1100 0000 - Index of where the move ends
// 0000 0000 1111 0000 0000 0000 - Piece type
// 0000 1111 0000 0000 0000 0000 - Captured piece type (0 if no capture)
// 1111 0000 0000 0000 0000 0000 - Flag for the move type
```

Figure 19: Bits required to store information on the move

A call to function `EncodeMoves()` takes the five arguments given in figure 19, and then bit shifts are used to set the bits in their respective places. Decoding works differently; only the requested information is retrieved, so `GetMoveFlags()` would return the 4 bits the flags are stored inside.

Handling pawn and special moves

Since pawn moves require more conditions than every other piece, they get their own section. Before defining the moves, an array of Moves (also called the move list) gets instantiated; whenever finding a possible move, it gets appended onto the move list; since the move list parameter is a pass-by-pointer, there is no need for a return. The Engine then loops through every bitboard and makes all the available moves for that piece. Afterwards, the bit is popped off the board, and the Engine repeats the process until all the moves for every piece on that bitboard are exhausted, leaving the bitboard empty. This procedure applies to all the pieces. The only difference is in the conditions used to define the moves.

- **Promotions**

A promotion is when a pawn reaches the final rank; it then promotes to a more substantial piece; the conditions for the move apply whenever a pawn is on the seventh rank and not blocked from moving forward, then four moves get added, one promotion for each piece, the knight, bishop, rook, and queen, usually only the queen promotion gets used, but all moves get added for completeness. Since promotion calls take up room in the move generation, an intuitive idea from Counter[9] is to move these calls to a separate function.

- **Pawn pushes**

The second condition is probably where most pawn moves fall into, the single pawn push; if the pawn can move forward, then this move gets made, similarly there is a double pawn push; if the pawn is still on its initial starting square and can move two squares forward then this move gets made. However, double pawn moves create possible en passant opportunities for the opponent, so a double pawn push flag gets added to the move to be later used to create an en passant square; this is created in `MakeMove()` 3.1.8.

- **Captures**

Now come the capture moves. There is now a second inner loop responsible for looking at all the possible captures; as seen before, the Attack Tables 3.1.5 are responsible for showing every capture a piece can make, a bitwise-AND used together with the opponent's pieces gets all the possible attacks on the opponent pieces. There can be both promotion captures and regular captures. Promotion captures are similar to a regular promotion, but instead of moving to the final rank, the pawn captures a piece on the final rank, and regular captures are where the pawn captures a piece.

- **En passant**

En passant is one of the few special moves in Chess, seen in Figure 44, designing the move this way prevents pawns from escaping the threat of an attacking pawn. Although it may seem challenging to implement, the square where an en passant capture can occur (c3 in the case of Figure 44) is provided by the FEN string (parsed in 2.2.1), for consecutive moves, the en passant square is updated in the `MakeMove()` function. The condition for an en passant capture is if there is an en passant square and an opponent pawn that attacks this square, then a capture is made.

- **Castling**

Now come castling moves, also a special move, seen in figure 45; the move puts the king in a safer position. Unlike other moves in move generation, this one checks whether the king is in check; this distinction can not be made when making the move and will bring about errors if not handled correctly. Since the king moves two squares during castling, the move is not made if an opponent's

piece attacks either square. If pieces are blocking the move, it is also not made. However, when the king or rook move, this condition is not checked as, similarly to en passant moves, the available castling moves are retrieved from the FEN string.

Handling every other piece

Conveniently when factoring out all pawn and special moves, only non-captures and captures are left, so the move generation for knights, bishops, rooks, the queen, and the king all have a similar implementation. Similar to pawn moves, there are two loops, one looping through the pieces, and another inner loop, looping through the available squares and attacks to the opponent's pieces. If an opponent's piece is on an attack square, add a capture move; if not, add a non-capture.

3.1.7 Copy and Make board state

```
// CopyBoard copies the current board and iterates the ply
func (oldBoard ChessBoard) CopyBoard() {
    chessBoardCopies[Ply] = oldBoard
    hashKeyCopies[Ply] = HashKey
    aspectsCopies[Ply] = [5]int{SideToMove, CastleRights, Enpassant, HalfMoveClock, FullMoveCounter}

    repetitionTable[(RepetitionTableIndexOffset+Ply)%1000] = HashKey
    Ply++
}
```

Figure 20: Copy the board state, store it, and increment Ply

This section is relatively small but still crucial for handling the moves. Before the moves are made on the board with MakeMove() 3.1.8, the board state needs to be stored. MakeMove() must be able to find an illegal move and traverse back up the search tree. The copy and make functions solve these problems. Before making a move, the entire position consisting of the Chessboard, side to move, castling rights, en passant, and clocks are stored; this information is then indexed by Ply (discussed before, incremented whenever a side moves); this is called the copy function, or CopyBoard() 20 in the implementation.

```
// MakeBoard pastes the board from the previous ply
func (newBoard *ChessBoard) MakeBoard() {
    Ply--

    *newBoard = chessBoardCopies[Ply]
    HashKey = hashKeyCopies[Ply]

    SideToMove = aspectsCopies[Ply][0]
    CastleRights = aspectsCopies[Ply][1]
    Enpassant = aspectsCopies[Ply][2]
    HalfMoveClock = aspectsCopies[Ply][3]
    FullMoveCounter = aspectsCopies[Ply][4]
}
```

Figure 21: Decrement Ply, and paste (Make) the board state, store it

Now say the MakeMove() function makes a move on board and discovers that it is not legal; it is required to unmake the board and retrieve the previous state before any modifications; this is called unmaking the board, or MakeBoard() 21 in the implementation, using the stored information from CopyBoard(), decrementing Ply and indexing the arrays retrieves the old information. However, this is not the only condition to consider; if only illegal moves were ‘unmade’, trying to traverse back up the search tree would not be possible, this is why whenever entirely examining a move, it is also unmade, allowing traversal back up the search try.

Regrettably, the functions are misnamed; this only affects readability. For example, the Chess Programming Wiki refers to copying and pasting the board state as Make and Unmake, respectively; this implementation refers to them as CopyBoard() and MakeBoard(), as at the time it made sense, but in hindsight, it does not.

3.1.8 Making the move

MakeMove() is the second of the four core functions. After generating the moves in 3.1.6, a check of the legality of the moves in the move list is necessary. Without this, illegal moves may be made on

board, which should not happen. This function only accepts single moves, not the entire move list; since the main objective is to search through the move tree, moves are searched through one by one using `CopyBoard()` and `MakeBoard()` 3.1.7.

Making a move is similar to move generation 3.1.6, but it doesn't require exhaustive checks for each piece since all the pseudo-legal moves have already been generated. The only necessary checks are for the flags set in move generation, which indicate the special moves that require additional handling. Finally, the move is checked for legality.

An alternative method for checking the legality of a move is to ensure that several conditions are met before making the move, such that the king is not in check. However, this Engine uses a more straightforward method, which generates all potential moves and only removes those that put the king in check after making a move. This approach may be slower, as it generates many illegal moves that need to be 'cut' later. However, it simplifies the move generation process since only one condition needs to be met for a move to be considered legal, this idea is from BBC [8].

Handling pawn and special moves

There are many Bitwise-XOR operations. For example, the initial starting square gets popped off when moving, but the end square gets set; an XOR can do this with a single operation. The operation to pop the initial square's bit and set the end square's bit is the first operation done; there is no condition to be satisfied for this, it is done on every move, and all other operations work on this.

- **Promotion flag**

Since a pawn gets promoted to another piece, the pawn's bitboard has its end square popped off, and then the piece it promotes to has the end square set.

- **Double pawn push flag**

Since this was set in the initial operation (the XOR of the start and end square), only the en passant square is changed; a double pawn push creates an en passant square as seen in 44

- **Castling flag**

Since the king has already been moved, only the rook's bitboard is changed here, and this is also just an XOR of the initial square and the end square

Every other flag does not take any special conditions; the only extra addition to note is that since the side to move is used to index the piece's bitboard, there is no need to make an if-else statement for the side to move; the index takes care of this.

Keeping consistency

Whenever the function makes a move, it also needs to ensure that the parameters stay consistent. For instance, if the current side to move is white, then after a move is made, the side to move must be switched to black. Similarly, other parameters, such as castling and en passant, must be updated to maintain consistency with the changes in the position caused by the move.

An array is used to update the castling rights. The idea is from BBC [8]. During the move, if the position of the rooks or king changes, then the castling rights must be updated accordingly. This is done by bitwise-ANDing the current castling rights with an array to check whether it should be updated, and if it should, then the castling rights are updated.

Is the king in check

The final condition before finishing all of the move generation and move making, this same condition was used in move generation during generation of castling moves 3.1.6, `IsSquareAttackedBySide()`, it takes the square and side as arguments, the square is usually where the king resides, and usually, the side is the opposite to the side to move (the opponent); this is one of the bottlenecks of move generation and also a weakness of bitboards 3.1.2, this Engine checks whether a square is attacked by looping through every attack table for every piece for a given side to move, doing a significant number conditions, giving a severe time penalty. A possible fix would be storing all these attack tables as a single bitboard while generating these moves and then checking it against this single bitboard rather than the current implementation.

3.2 Search

After board representation comes searching the position, after employing even basic search enhancements and Evaluation, the overall strength significantly increased, many algorithms were tested and used during development, some better than others, and some even seemingly decreased the strength of the Engine;

since there are a significant number of nodes that get pruned, unlike Board representation 3.1 there are sometimes negatives to some of these techniques that need to be taken into account. Hence, a more theoretical approach needs to be taken.

Most of the techniques and algorithms implemented are well understood and used by many Engines. However, there still needs to be tuning and testing 4.1.2. Hence, to not unnecessarily limit how well they perform, even considering the number of algorithms out there, many were removed as they did not increase the Engine's performance; considering how the Evaluation or move ordering work, specific techniques may prune or reduce moves when they should not leading to unexpectedly horrible moves.

Before moving on, it is essential to understand the meanings of pruning and reducing. When a move is 'pruned', it is cut off from the search tree; this does not necessarily mean the move is disregarded since making taking note of some cut-offs can allow the Engine to repeat the same process when the same move shows up again. However, when a move is 'reduced', its original search depth is decreased; reduced moves may show some interesting results, so they are still considered, but not as much as moves that did not get reduced.

When creating a competent Engine, it is only necessary to implement alpha-beta 3.2.1, iterative deepening 3.2.3, and move ordering 3.2.4; these three bring about the core of search, every other technique prunes or reduces more, but without these main techniques the Engine would not function properly.

3.2.1 Negamax with alpha-beta pruning

Alpha-beta is the third of the four core functions. The `alphabeta()` function provides the basis of where everything from now is built. Negamax is a simplification of mini-max. There are two functions, one for a min player and the other for a max player; if, say, the side to move is white, for the first selection of moves, white tries to 'maximise' their score by picking the move with the highest Evaluation (score from the evaluation function 3.3) searching deeper to where the opponents moves (black) are now introduced, black tries to maximise their chances of winning by minimising whites score, this is repeated up the search tree. Negamax is a way to combine the min and max functions using the principle that $\max(a, b) \equiv -\min(-a, -b)$ [27].

Claude E, Shannon estimated there to be an average of 30 legal moves in a position[1]; since negamax will not reduce the node count, there will need to be techniques to try to reduce the branching factor (number of moves per position), the first technique implemented is alpha-beta pruning, as seen in the following pseudocode 3.2.1. Unlike most other methods discussed here, alpha-beta pruning has no downside, as there will be no chance of missing a better move when pruning a tree's branch[28].

Algorithm 1 Negamax with alpha-beta pruning using a Fail hard framework

```

1: function ALPHABETA(alpha int, beta int, depth int, cb *ChessBoard)
2:   if depth <= 0 then
3:     return quiescence(alpha, beta, cb)
4:   end if
5:   Set moveList to empty Move array                                ▷ Create the moveList
6:   GenerateMoves(moveList)                                          ▷ Fill moveList up with moves
7:   for Each move in moveList do                                     ▷ Exhaust moves until finished or cutoff
8:     Set score to -alphabeta(-beta, -alpha, depth-1, cb)
9:     if score is greater than alpha then
10:       $\alpha \leftarrow \text{score}$                                        ▷ A better move was found
11:      if score is geater than or equal to beta then                ▷ Fail hard beta-cutoff
12:        return beta                                              ▷ Fail high
13:      end if
14:    end if
15:  end for
16:  return alpha                                                    ▷ Fail low
17: end function

```

Alpha-beta pruning works as follows, when looking at a depth of 2 or lower, it is assumed that the second player will always try to make the first player's moves as bad as possible by choosing the best counters to those moves, so how does the first player ensure that without going through every move they only choose the best move? As an example, say the first player looks through the first move and all the counters to that move; they find that in each counter move, something is lost; they can lose a pawn, a

rook, or small positional advantages, assuming that the opponent plays the best possible counter, the only position considered is the one in which a rook is lost; this is because every other choice is worse for the second player. However, say the first player now examines the second move and sees that the best counter for black is a slight positional advantage; since this is the weaker counter than what was seen in the first move, they can safely ignore the first move and only consider this one; doing this for every move gives a lower bound (alpha in this case). However, now a depth of 3 or deeper is looked at, similar to how a lower bound is kept, the other player can go through the same method but do the opposite (trying to find a maximum for their minimum), which results in an upper bound (beta in this case)[28].

The Engine's implementation uses alpha and beta as hard bounds, resulting in a fail-hard framework[28]. However, an alternative approach is the fail-soft framework, where alpha and beta act as soft bounds. In this scenario, if no better move than alpha is found, the best move found is returned[29]. Soft bounds may be the better approach since the score relies heavily on the evaluation function. A move that appears better based on evaluation may not necessarily be so. To determine whether the alternate strategy is feasible, further testing is necessary.

3.2.2 Quiescence search

In the algorithm 1, when the depth is 0 or below, the function `quiescence()` is called. However, before discussing this function, it is essential to understand the 'horizon effect'. This effect refers to the fact that an Engine will never truly know what is beyond the search's 'horizon' (the searched game tree)[30]; this can lead to losing pieces or positional advantages. While this effect is unavoidable, it is also minimisable.

One way to minimise this effect is through a quiescence search. After searching the specified depth given to alpha-beta, the Engine drops into a final quiescence search. The implementation used in this Engine only searches captures until all possible captures are exhausted. Other implementations might search positions where the king is checked[31].

```
var standPat int = eval.Eval(*cb)

// found a better move
if standPat > alpha {
    alpha = standPat
    // fails high
    if standPat >= beta {
        return beta
    }
}
```

Figure 22: Standing pat implementation in Quiescence search

The Quiescence search implementation in the Engine is similar to the alpha-beta skeleton, many other more substantial Engines like Stockfish[11] and Crafty[10] have sizeable Quiescence functions, but since the main focus was around alpha-beta, this did not get built up as much. One enhancement was the 'Standpat' score (figure 22); before generating the move list, a score of the position is produced using the Eval function from Evaluation 3.3, then if it exceeds both alpha and beta, the node fails high, and since the Engine uses a Fail-hard framework, it returns beta; but if it does not exceed beta, then alpha is set to the new evaluation. This enhancement sometimes avoids searching all captures and boosts performance by a small amount. This idea is from BBC[8] and the Chess Programming Wiki[32].

Since the implementation only evaluates captures, and the move generator and move maker are for all moves, two new functions are made to only deal with captures. This optimization does not reduce node count but speeds up the `Quiescence()` search by a small amount. Implementing the Quiescence search reduced the depth significantly. However, the Engine played much more reasonably, giving away fewer pieces, leading to safer positions.

3.2.3 Iterative deepening

```
position startpos
go
info depth 6 nodes 159596 score cp 0 time 177 pv d2d4 g8f6 b1c3 d7d5 g1f3 b8c6
bestmove d2d4
```

Figure 23: Search at a fixed depth with no iterative deepening

Iterative deepening involves incrementally increasing the depth when the Engine searches a position[33]; the main reason for doing this is for time management[34]. Until iterative deepening, regardless of how complicated or straightforward the position is, it was always searched to the same depth (seen in 23). Iterative deepening allows the Engine to stop searching when the stop time (calculated in UCI 2.2) has been reached. More efficiently using the time significantly increases how well the Engine can play; taking the same time on every move lets the Engine think for longer, possibly creating a better move. Again stronger Engines like Stockfish[11] vary this time if the search is unsure of the move. The idea came from BBC[8], thought to be first implemented in Chess 4.5[35].

```
// Aspiration window
if depth > movesAvoidAspiring && score > -MateValue && score < MateValue {
    // If the score is outside the window, we research without the windows
    if (score <= alpha) || (score >= beta) {
        alpha = minScore // -INFINITY
        beta = maxScore // INFINITY
        currDepth--
        continue
    }

    // set window up for next iteration
    alpha = score - aspirationWindow
    beta = score + aspirationWindow
}
```

Figure 24: Aspiration window starting using values from the previous search

Another improvement made before the search begins is the use of aspiration windows. Typically, the value of alpha is initialized to a very low negative number (referred to as negative infinity or minScore within the Engine), such that any evaluation of the position would not exceed it. Similarly, beta is set to infinity; using this range ensures the search captures every value. Aspiration windows work by taking the score from the previous search and putting a ‘window’ around it (seen in 24). In the Engine, the window is set to around a third of a pawn, as this value produced the best results during trial and error 4.1.2. Although this technique has some downsides, such as missing a possibly good move if it is outside the window, the benefits outweigh the negatives, so it is currently used. A complete re-search at the same depth must be performed if the desired value is not within the window. The aspiration window implementation is from Bruce Moreland[36], and originally tested by multiple sources[37][38].

Initially, implementing aspiration windows proved beneficial; however, as more moves got pruned or reduced, problems arose; sometimes, it would search more nodes with the aspiration windows than without, possibly due to how many re-searches were needed. However, simply choosing to start aspiration windows at a later depth fixed this, so now they start from a depth of 4 instead of a depth of 1. This idea came from Counter[9].

3.2.4 Move ordering

After implementing alpha-beta pruning in the engine, the following technique added was move ordering[39]. Although alpha-beta does perform cut-offs based on the quality of moves 3.2.1, better ordering of moves leads to more accurate updates of alpha and beta values, resulting in more pruning, reduced branching per move, and allows for deeper tree searching. As depicted in figure 46, the difference in search time between a search with and without move ordering is significant; this clearly shows that the time taken increases exponentially without move ordering, making deeper depths infeasible within regular gameplay.

Move ordering is performed after move generation, scoring moves based on certain conditions. Move ordering occurs ‘on-the-fly,’ meaning that the highest-scoring move is placed in the first position of the move list before performing any operation. This move is then looked at first. Then, after an iteration through the loop, the second-highest-scoring move is placed into the second position and looked at next. This process continues until a cut-off occurs or until the move list is exhausted. By doing it on-the-fly, there is no need to sort every move into the correct place, which saves time if a cut-off occurs. The idea to perform move ordering ‘on-the-fly’ comes from the Rustic Engine [40].

Since move ordering can bring about significant speed enhancements, this was also a heavily tested area, and many techniques were added and removed from various authors and Engines. As seen in 48, most moves in the move list are scored based on some condition. There are many conditions because all unscored, unordered moves get treated the same in alpha-beta, so the more these moves are scored, the better alpha-beta works.

- **PV node**

A PV node is the best move found at some some depth, iterative deepening 3.2.3 produces PV nodes, in fact this is why iterative deepening works in the first place, if say a depth of 1 is searched, the best move found can be retrieved and then ordered highly when searching to a depth of 2, this is repeated until the search eventually stops.

- **Most Valuable Victim - Least Valuable Attacker**

Also referred to as MVV_LVA[41], since the order of piece value increases as follows, pawns, knights, bishops, rooks, the queen, and the king, MVV_LVA provides a way to order how valuable to captures are, for example, a pawn capturing a queen would be much more valuable than a queen capturing a pawn. The table used is from the Rustic Engine[42] with a few alterations; this is the core move ordering. Ordering the captures provides the most significant speed benefit compared to the other ways to order moves within the implementation. However, there is an alternative better way to order moves called the Static Exchange Evaluation (SEE); whereas MVV_LVA only sees one capture ahead, SEE looks down the path of captures to see whether it genuinely gains material[43], although more computationally expensive since MVV_LVA only indexes an array, it could prove to be better as it does order captures more accurately.

- **Killer move heuristic**

First introduced by [44], Killer moves are a way to sort ‘quiet moves’ that caused a beta-cutoff in a sibling node[45], as seen in 49. A killer move is a move that does not capture any material but requires the opponent to react to it, leading to many moves getting cut. For example, in the figure, ‘isSafe’ is true if there are no captures or promotions; this is because captures and good promotions should have already been sorted; without this condition, potential killer moves may be thrown out and replaced by these other already sorted moves. The original move ordering of killer moves came from BBC[8]; it now comes from REBEL[46]. Since REBEL also orders older Killer moves, there is a slight performance advantage within it.

- **Promotions and castling**

The idea is from Rebel[46]; since all unordered moves get treated similarly, it makes sense to order promotions and castling highly since these moves are usually desirable.

- **Relative History Heuristic**

Although it may appear that most moves are getting a score given the number of conditions, this is far from the case. [47] created a way to store “interior nodes of search trees”, as seen in 49, the hhScore table represents the history heuristic, although not necessarily limited to this location, this table is only updated within a beta-cutoff. However, similar to other move ordering techniques, as the search tree shrank, the power of the history heuristic significantly decreased and ended up slowing down the search at some point. [48] devised a better way to score these history moves by comparing them to butterfly boards (scored in 48), also seen in 49, the bfScore table holds the ‘butterfly boards’[49], the original idea incremented the bfScore table if it was not a beta-cutoff, but Robert Hyatt proposed the current implementation as it is now[50] (looping over all previously found safe moves when a beta-cutoff occurs), this better way to score history moves ended up improving the search by a reasonable margin and is still used.

3.2.5 Extensions

```
// Check extension
// https://www.chessprogramming.org/Check_Extensions
if isOpponentChecked {
    extension = 1
}
```

Figure 25: Increase search depth if king is in check

To explore certain moves further, the Engine employs extensions, which involve increasing the search depth if a move meets certain conditions. The Engine currently only implements the Check Extension (as shown in Figure 25) as it is simple and safe. By applying this extension, the Engine can search deeper into positions where checks occur, resulting in more accurate move evaluations. Discussed initially in

[51], the Engine adopts the implementation from BBC[8] but refines it by incorporating it at the start of move search, as inspired by Counter[9]. Counter also implements Singular Extensions[52], which extends the depth if there is a move much better than the rest. Implementing this would require changing how moves are generated and picked, so it was not implemented, but it could be in future versions.

3.2.6 Principal Variation Search

Principal Variation Search (PVS) was first introduced in [53] and is widely used in many game-playing engines, including those referenced in this paper. The PVS technique is used to reduce the number of nodes searched while slightly altering the search result. The implementation of this technique was originally from Bruce Moreland's Gerbil Engine[54] but later changed to Counter's implementation[9] as shown in Figure 26.

3.2.7 Pruning and Reduction techniques

There are several ways to prune and reduce moves; the ideas behind the techniques are usually intuitive, but their implementations may need to be revised. Therefore, a brief overview of all techniques used within the Engine will be reviewed without the granular details.

There are two categories of pruning techniques: Backwards pruning, where the search result remains unaffected, only reduces the total nodes seen; examples include alpha-beta pruning 1 and mate distance pruning; then there are forward pruning techniques, which can affect the search result but can significantly decrease the total nodes seen. Forward pruning is inherently riskier, but as there are many more techniques that it can offer, it can significantly boost the search depth of an engine[55].

Then there are reduction techniques, which are used to decrease the node count to reach a deeper depth. However, unlike in pruning techniques, where entire trees are cut, reduction techniques only reduce the depth the search originally used. As seen in 53, reductions may vary depending on how safe it is to reduce.

Late Move Reductions

```
score = alpha + 1
// LMR
// https://www.chessprogramming.org/Late_Move_Reductions
if reduction > 0 {
    score = -alphabeta(-alpha-1, -alpha, depth-reduction+extension, LMRSearch, extension, cb)
}
// if the move fails high, we search deeper
// PVS
// https://www.chessprogramming.org/Principal_Variation_Search
if score > alpha && beta != alpha+1 && i > 0 {
    score = -alphabeta(-alpha-1, -alpha, depth-1+extension, PVSSearch, extension, cb)
}
// Normal search
if score > alpha {
    score = -alphabeta(-beta, -alpha, depth-1+extension, StandardSearch, extension, cb)
}
```

Figure 26: Late Move Reductions (LMR), Principal Variation Search (PVS) along with the normal recursive negamax equation

Late Move Reductions (LMR) can be compelling[56]. LMR (Figure 26) may reduce the depth of moves if they occur later in the move order, as seen in Figure 53, if the move is below a particular score and certain conditions apply, it may get reduced. Many other Engines like BBC[8] and Counter[9] only apply LMR after a certain number of moves have been made. After some testing, this Engine saw a decrease in total nodes visited without much change in the search when applying the condition that only moves below a particular score (mostly unordered) get reduced.

Null Move Pruning

```

// Null Move Pruning (NMP)
// https://www.chessprogramming.org/Null_Move_Pruning
if !pvNode && !isChecked && depth >= nullMoveDepth && !eval.IsEndGame(*cb) && flag != NullMovePruningSearch && staticEval >= beta {
    cb.MakeMoveNull()
    // vary reduction based on depth, idea from the CounterGo chess engine
    reduction := (nullMoveDepth + (depth / 6))
    score = -alphabeta(-beta, -beta+1, depth-reduction, NullMovePruningSearch, 0, cb)
    cb.MakeBoard()

    // check if the search should be stopped, time is checked concurrently
    if IsStopped {
        return 0
    }

    if score >= beta && (score > -MateScore) {
        return beta
    }
}

```

Figure 27: Null Move Pruning (NMP)

Null Move Pruning (NMP) relies on the null move observation, that if a player gives the opponent a free extra move, and if this still causes a beta-cutoff, then prune the move; this can be done further up the tree before any moves are generated[57]. As seen in Figure 27, the ideas to vary the reduction factor based on depth and to only perform if the Evaluation of the position is above beta is from Counter[9].

Futility Pruning

```

// skipping moves given conditions
if depth <= futilityDepth && legalMoveAvailable && !isChecked &&
    moveList[i].Score < moveOrderOffset-10 && (score < MateScore) {
    // futility pruning
    // skip moves that likely won't improve alpha
    // https://www.chessprogramming.org/Futility_Pruning
    if staticEval+(eval.PieceValuesTapered[board.Pawn]*(depth+1)) <= alpha {
        continue
    }
}

```

Figure 28: Futility Pruning

Futility Pruning[58] is a pretty aggressive form of pruning[59]. Similar to LMR, only moves that are not scored highly are pruned. However, dissimilarly, this is done on moves below a certain depth instead of applying to moves above a certain depth (seen in Figure 28). Other engines typically use this pruning only at depths at or below 2. Before making a move, the Engine calculates the position's Evaluation. If this value plus some margin still needs to improve alpha, the move is skipped as it likely is not better than the already searched moves. The idea to change the margin based on depth is from Counter[9].

Evaluation Pruning

```

// Reverse Futility Pruning
// https://www.chessprogramming.org/Reverse_Futility_Pruning
if !pvNode && !isChecked && depth <= futilityDepth && (kingSafety <= 1 && kingAttack <= 2) {
    var score = staticEval - (eval.PieceValuesTapered[board.Pawn] * (depth))
    if score >= beta {
        return staticEval
    }
}

```

Figure 29: Evaluation Pruning or sometimes referred to as Reverse Futility Pruning

Evaluation pruning, also known as Reverse Futility Pruning (RFP)[60], is pretty much identical to Futility Pruning in how it works (seen in Figure 29), however instead of skipping moves below alpha, RFP returns the evaluation if a moves evaluation plus some margin is below beta, this uses the same principle in NMP, in that it relies on the Null move observation. Basing on whether to skip the move based on king safety and king attack is original, implementing these ideas made the Engine play safer, whereas returning the evaluation instead of beta is from Counter[9].

Mate Distance Pruning

```
// Mate Distance Pruning (MDP)
// https://www.chessprogramming.org/Mate_Distance_Pruning
if alpha < -MateValue+board.Ply {
    alpha = -MateValue + board.Ply
}
if beta > MateValue-board.Ply {
    beta = MateValue - board.Ply
}
if alpha >= beta {
    return alpha
}
```

Figure 30: Made Distance Pruning (MDP)

Mate Distance Pruning (MDP) is a technique that does not necessarily enhance an Engine’s overall performance. However, it can help find shorter mates more quickly, as shown in Figure 47. Although the effectiveness of MDP diminishes as the Engine becomes more efficient, the technique is still valuable in optimizing certain aspects of the search process. This implementation is from Crafty[10], but described more in [61].

3.2.8 Transposition table

During a search, the Engine encounters many nodes, and among these nodes, some positions may be identical. Such positions are called transpositions, allowing the Engine to store and retrieve information about them. Three flags are made based on the type of move they represent. If a move fails low, meaning no better move was found, and no cutoff occurred, it is given the Alpha flag. On the other hand, if a move fails high, meaning a better move was found, but a cutoff was produced, it is given the Beta flag. Finally, if a move is better than alpha and does not cause a cutoff, it is given the Exact flag. This idea was first used in Mac Hack VI[62]

```
type TranspositionTable struct {
    key      uint64
    depth    int
    score    int
    flag     int
    age      int
    bestMove board.Move
}
```

Figure 31: The struct responsible for holding transposition data, this is then initialized into an array holding $1 * 1028 * 1028$ entries (1MB * elements in struct)

Zobrist hashing[63] is used to identify if a position is a transposition. This method assigns a unique hash value to all the information whenever generating a new position. The idea is that two identical positions should have the same hash value, regardless of when they are generated. This hash value could then be used to index a colossal array holding this transposition data (Figure 3.2.8).

Since random numbers generate Zobrist hash keys, two different positions can have duplicate ones. Multiple factors are taken into account to minimise this chance. An example of a factor is the “randomness” of the random number generator, as true randomness cannot be achieved within a computer. This Engine assumes that Go’s random number generator is exemplary in terms of its randomness. Another factor that affects the probability of collision is the size of the key itself. This has been evaluated in [64], and the Engine uses a 64-bit unsigned integer to store this information.

3.3 Evaluation

Evaluation is the fourth and final core function. As seen in both `alphabeta()` (1) and `quiescence()` 3.2.2, positions need to be evaluated and given a score, this function serves that purpose. When comparing the scores inside these functions, there is no requirement for how they are scaled, but when printing them to the terminal, they should be scaled to centipawns (cp) as mentioned in Section 2.2.1. Since this Engine already uses the granularity of the centipawn scale within the Evaluation function, scaling is unnecessary. Stockfish is an example where the granularity is different; there is a scale of $1/256$ instead of the usual $1/100$ scale, and the difference in scale results in ‘finer’ values.

The problem comes with evaluating the position and giving it a score; in a perfect world, having perfect information about every position would allow one to know whether the game is lost, drawn, or won[65]. The Evaluation aims to find an approximation to a position’s ‘true’ value by providing a heuristic. The more detailed and accurate each measure within the heuristic, the more accurate the

Evaluation, but the more costly each call to the Evaluation function becomes. Trying to balance the Evaluations complexity with the cost per Evaluation and if it is genuinely worth it becomes a significant part of building this function. However, this does not necessarily matter in the early stages as every new addition gives the Engine more insight into the ‘true’ value.

This Engine uses Hand Crafted Evaluation (HCE) instead of neural networks. Stockfish does have an open-source neural network available, but given that this paper tries to cover the general principles, using it would have provided little insight into what Evaluation is.

3.3.1 Piece Values and Piece-Square Tables

```
var (
    PieceValuesMG      = [6]int{82, 337, 365, 477, 1025, 0}
    PieceValuesEG      = [6]int{94, 281, 297, 512, 936, 0}
    PieceValuesTapered = [7]int{0, 82, 337, 365, 477, 1025, 0}
)
```

Figure 32: Piece values of all piece in the middle and end game from pawn to king[66]

The first definition estimates a piece’s worth (Figure 32). Although most engines have slight differences in how they score each piece, the general values have remained similar to how they were defined in 1950 by Shannon[1]. Pawns are worth 1, Knights are worth 3, Bishops are worth 3, Rooks are worth 5, and Queens are worth 9. Assigning a value to the king is unnecessary as it cannot be captured.

```
var pawnMG = [64]int{
    0, 0, 0, 0, 0, 0, 0, 0,
    98, 134, 61, 95, 68, 126, 34, -11,
    -6, 7, 26, 31, 65, 56, 25, -20,
    -14, 13, 6, 21, 23, 12, 17, -23,
    -27, -2, -5, 12, 17, 6, 10, -25,
    -26, -4, -4, -10, 3, 3, 33, -12,
    -35, -1, -20, -23, -15, 24, 38, -22,
    0, 0, 0, 0, 0, 0, 0, 0,
}
```

Figure 33: Piece-Square table of pawns in the middle game[66]

Now comes creating values for each piece’s squares, a king in the middle game should usually stay around its start squares, so they are scored highly compared to the rest of the board. However, a knight usually does not want to stay close to the king but stays close to the middle, so they can ‘see’ the board more. Each piece has a different table, but it should also be noted that the middle game and end game are treated very differently, so each piece should have two tables, one for the middle game and the other for the end game (Figure 32).

Within the Engine RofChade by Ronald Friederich, only Piece-Square tables are used[66], they are used since they provide excellent values for the ‘general’ position, and implementation is from TSCP by Tom Kerrigan[67]. After implementing these tables, since they are so well-tuned, there was an immediate effect on the Engine’s performance; this effect slowly decreased as the heuristic strengthened.

3.3.2 Tapered Evaluation

First mentioned in [68], Tapered Evaluation tries to interpolate the values between the middle and end game to remove evaluation discontinuity[69]. Without tapered evaluation, capturing a piece would move the Engines game phase from the middle to the end game instantly. Since the Engine treats the middle and end games vastly differently, this could completely disrupt and break the Engines search.

The implementation this Engine uses is also from TSCP[67], as seen in Figure 54. The Engine performs tapered Evaluation by trying to find an average of the game phase using the pieces left on the board as a scale.

3.3.3 Mobility

Also, an idea laid out by Claud Shannon in 1950 [1], ‘mobility’. There are different ways to consider what this means, but generally, it is how many squares are available to the piece; it could also be how many legal moves are available to this piece. However, within the Engine, mobility is defined as how many squares the piece can move to AND NOT pawn attack squares, pawn attack squares are removed since this would more often than not result in the capture of a piece.

When the Engine has the number of squares it can ‘see’ available, it indexes an array using this information. The array is from Stockfish[11]. Initially, each piece had its calculation for mobility, but indexing an array meant having an individual value for each square that the piece could see, allowing for more accurate Evaluations.

3.3.4 King safety

King Safety is a fundamental concept in the game of Chess, as the objective is to checkmate the opponent’s king. There are various ways to incorporate this concept into Chess programming. Implementing different forms of King safety has resulted in the most significant improvement in the Engine’s strength. As observed in Section 4.1.3, the Engine’s lack of King Safety caused numerous unfavourable positions. The implementation of King Safety enhanced the Engine’s strength and made its gameplay considerably safer.

```
// attacking king zone
// Idea from https://www.chessprogramming.org/King_Safety#Attacking_King_Zone
mg[side] += SafetyTable[Min(attackingPiecesCount, 99)]
// Add how many pieces are attacking the opponent for use in search.
KingAttackingPieces[side] = attackingPiecesCount
attackingPiecesCount = 0

// pawn shield
mg[side] += board.BitCount(board.KingAttacks[square]&*pieceArr[side*6]) * pawnMultiplier

// king tropism
mg[side] -= kingTropismPenalty[board.BitCount(board.GetQueenAttacks(square, allPieces))]
```

Figure 34: How both the King tropism and Attack units are used to index the arrays for Evaluation.

The Engine utilizes the technique of king tropism as a form of king safety (Figure 3.3.4). To evaluate the king’s safety, a Queen replaces the king, and the number of attack squares the queen sees is counted. This count is then used as an index for a table, providing a simple yet effective method for evaluating the squares that are ‘seen’ by the king[70]. Although the implementation does not result in a significant performance boost, it is straightforward, ensuring minimal time is wasted.

```
queenAttacks := board.GetQueenAttacks(square, allPieces) &^ unsafeSquares[side]

mobileSquares := board.BitCount(queenAttacks) // - 2
mg[side] += queenMobility[mobileSquares][0] // * rookMobility[0]
eg[side] += queenMobility[mobileSquares][1] // * rookMobility[1]

// king attack update
attackingPiecesCount += 5 * board.BitCount(queenAttacks&kingSquares[otherSide])
```

Figure 35: Calculating the ‘value’ of the Queens attacks on the enemy king

Instead of calculating the attack squares seen by the king, calculating the attacks on the king by the opponent’s pieces makes much more sense. The current implementation uses calculations similar to Figure 35 to calculate the attacks from all enemy pieces (except pawns). Stockfish[11] uses the idea of ‘Attack units’, as seen in the Figure, applying the idea that “the whole is greater than the sum of parts”[70], the squares that the queen sees on the kingside are multiplied by five since the queen’s attacks are more potent than any other piece a higher value is used, the knights will multiply this by two, this is similar to all pieces. Then the sum of these piece attacks is used to index an array; this array is from Stockfish[11].

4 Testing & Evaluation

For Board Representation 3.1, there is a way to test the completeness of the move generator, checking if it correctly returns a list of all the legal moves given a position with no errors. This test is referred to as a ‘Perft’ test (seen in 4.1.1). However, testing the Search and Evaluation components of the Engine is more complex and requires different techniques. For example, the evaluation functions weights could be tuned using an automated tuning method, a common one being Texel’s Tuning Method[71], the method used on this Engine was trial and error (seen in 4.1.2). Another reason to test is to uncover problems in the search or eval; a move might get pruned or reduced if it really should not have. These methods are only used to improve the Engine; evaluating how ‘good’ an engine is and improving its performance is the main reason any of this is done (seen in 4.2).

4.1 Testing

Testing was vital in this Engine. Unlike other past projects, every test done had uncovered bugs, especially in the Trial and Error phase 4.1.2. Although bug testing within the Search and Evaluation functions cannot be exhaustive, it still led to many different optimizations and performance enhancements.

4.1.1 Perft

The Perft test is a crucial and exhaustive test used to verify the accuracy of the board representation section in the Chess Engine. While other Chess Engines may use additional unit tests, the Perft test alone is sufficient to ensure that the Chess Engine is complete regarding legal move generation. This test involves generating all legal moves for a given position to a certain depth and counting the number of nodes in the resulting move tree. If the Chess Engine being tested returns a node count that matches the result from a previously tested working engine such as Stockfish[11], it can be safely assumed that the Chess Engine is complete and accurate in its move generation (Perft only works in Board representation 3.1).

The output of the Perft test 41 for the Engine was designed to match Stockfish. Conventionally all the legal moves at a depth of 1 are listed with their node counts, so if there is a problem with one of the node counts, the tester can ‘step through’ (seen in 42) the search tree to find what move is not appearing.

4.1.2 Trial and Error

Although rudimentary, Trial and Error was valuable in uncovering elusive problems within the Search function. As seen in the Perft tests 4.1.1, the sheer number of moves to consider meant that trying to look at every move individually to see how it should be handled was practically impossible. When incorrectly handled, specific moves could propagate through and ultimately disrupt the Search, often by poorly set parameters within functions that pruned or reduced moves (as discussed in 3.2.7). To address this issue, the Search function was fine-tuned by playing games against different Engines. If there were no errors, but the Engine played poorly, the parameters within the Search were altered to cut more nodes. However, this trial and error process had risks; while testing against other Engines, changing these parameters could drastically alter how the Search functioned. Sometimes, this led to sudden drops in Evaluation resulting in losses, while other times, the accumulation of slightly wrong moves resulted in a loss.

There are still inherent issues with this approach. There are instances when the Engine just performed badly, not because of the parameters but because of the overall strength of the Engine being weaker. The concern with this is that, no matter how tuned the parameters are, the Engine will always play worse than its opponent, possibly skewing the tuned parameters. A possible solution would be to only play against an Engine that it can consistently win and lose against so that an accurate measurement can be taken; this also has an inherent weakness, the Engine may be over-tuned to its opponent, it may play very well against this one other Engine, but compared to other similar rated Engines, it performs poorly. Although these ideas have limitations, if tuned for long enough against various Engines, it can be assumed that the tuning is good enough.

Later in development, a similar but different approach was taken. Initially, 2 minutes per side per 40 moves time controls gave the Engine time to ‘think’; if it could not produce even a decent move compared to previous iterations, then a previous version was rolled back. This new approach used the same idea but had very short time intervals. Compared to the 2 minutes, it was now 5 seconds per side per 40 moves since, later in development, the Engine had to think less to produce the same results. The

short time controls allowed the Engine to play far more games in less time, leading to faster results and better tuning.

Unlike uncovering bugs in the Search, Trial and Error was primarily used to tune values inside the Evaluation manually. Although archaic and inaccurate, it worked for the most part. However, in hindsight, calculating these values using Texel's automated tuning would have cut down on the time required to get the same results and produced better results.

4.1.3 Tricky positions

During matches against other Engines, certain positions led to significant spikes in the evaluation. This was often caused by cutting moves that should not have been cut, but sometimes it was an inherent issue with the Engine. Improving the Search and Evaluation fixed these inherent issues, and now there should not be any significant problems.



Figure 36: Lichess[72] used for Chessboard, Stockfish 14[11] used for the best move, Position FEN: r3k2r/ppp1bpp1/6q1/3pP3/6p1/P1P1P2P/1B3P2/R2Q1RK1 w kq - 0 17

In 36, it can be seen that black is winning; although white is losing, the Engine should still try to avoid making errors as sometimes a losing position could be turned around. Originally the Engine made a catastrophic mistake; by making the move h3g4 (white pawn takes black pawn), a losing position turned into a mate in 5 for the opponent. Although a human could easily see the tremendous threat on the kingside posed by the black queen and black rook, the Engine did not recognize this because King Safety was not yet appropriately evaluated (before 3.3.4). As seen in 50, the Engine no longer makes the same mistake.



Figure 37: Lichess[72] used for Chessboard, Stockfish 14[11] used for the best move, Position FEN: 5rrk/p1p1R3/n2pRp2/1Q1P1q1p/1P1P4/P5Pp/5P2/5NK1 w - - 1 31

Unlike in the previous example, in Figure 37, white is winning, but in this situation, the Engine made the move b5a6, capturing the black knight with the white queen. Unfortunately, the move was horrible; the evaluation turned from +7.0 to -26.8 (according to Stockfish). Even though it is not a mate, an evaluation that low is practically a completely lost game. Again this was because King Safety was

overlooked, but changing the Late Move Reduction (LMR) parameters fixed this issue for the most part. Since the current implementation can properly evaluate King Safety (In Figure 51), the LMR changes were reversed.



Figure 38: Lichess[72] used for Chessboard, Stockfish 14[11] used for the best move, Position FEN: 1n1b2rk/5r2/p1p1p3/3pP3/1P1PqNp1/Q5P1/3B1P2/2RR2K1 w - - 3 48

In figure 38, white is winning, but their king is under some threat. The previous Engine iterations disregard this and make some moves. Again a recurring theme, if not correctly accounting for the king's safety, the opponent's threats pile up, potentially making the situation dire. However, the final version of the Engine makes a viable move (as seen in Figure 52).

4.2 Performance Evaluation

A performance measure is used to find the level of players that can comfortably play against the Engine, allowing players to judge if this Engine is right for them.

4.2.1 Tournament

My Tournament						
Rank	Name	Elo	+/-	Games	Score	Draw
1	ChessEngineAI	458	nan	30	93.3%	6.7%
2	Napoleon	95	131	30	63.3%	6.7%
3	bodo_02b	-70	128	30	40.0%	6.7%
4	tscpl81	-585	nan	30	3.3%	0.0%

60 of 60 games finished.

Figure 39: 2 minutes per side; 40 moves; tournament time control.

Cutechess and many GUIs have a 'tournament' feature, allowing users to make two or more Engines compete against each other. An 'Elo' difference is calculated from these games depending on the Engine's playing strength; this is just a relative measure of the Engine's performance. It should be noted that the rating system used is usually different on sites like Chess.com[73], FIDE, and CCRL[74], they are more measures to compare the different players on their respective sites, so it cannot be transferred over.

As seen in 39, many games can be played in series to determine whether specific Engines are better than one another. The time control seemed short in concept, but playing 60 games with 2 minutes per side took around five hours to finish. The site that these Engines are from (CCRL[74]) play games with 15 minutes per side every 40 moves to calculate a rating; this would lead to a more accurate measure of the Engine's playing strength, but playing enough games with this more extended time control would have taken days to calculate since the focus is not around making the strongest Engine, a refined analysis is not needed.

The three Engines and their ratings according to CCRL[74] used are 'TSCP' by Tom Kerrigan rated 1761 ELO[67]; 'Bodo' by Joel Veness rated 2219 ELO[75]; and 'Napoleon' by Marco Pampaloni rated

2284 Elo[76]. There is no real reason as to why these engines were chosen, except that they hovered around the ratings of ChessEngineAI (the Engine covered in the paper). Although an Engine stronger than ChessEngineAI would give a better measure of the strength since the win rate is 93.3%, it can be assumed to be better than Napoleon. Taking the absolute minimum of this would put the Engine around 2300 Elo regarding the CCRL rating system.

5 Conclusion

During the year working on the Chess Engine, many techniques have been implemented, improving it to a state we are happy with, it does not compete with the highest-ranking engines out there, but the reason for this Engine's existence does not require it to.

The Chess Engine still serves a vital purpose; strengthening the Engine by playing against other engines was vital during development; the hope is that this Engine may also be used to strengthen other Engines to find potential weak areas in play and for players to be able to use as a competent opponent. However helpful, this was different from the intended goal. Originally this Engine was meant to be another resource for Chess programmers, and through thorough explanations of the topics used throughout this paper, it should be able to serve as one.

Chess programming is in itself vast. Everything covered in the Engine's implementation is still only a fraction of the problem area. It is effortless to get lost in improving the Engine and getting it to a state that the developer is happy with, and though the stopping point in this Engine was determined by how much time new techniques would have taken, to me, it is in a finished state.

5.1 Aims & Objectives

The aims and objectives here were thought up before the implementation of the Chess Engine began, leading to some redundancy; however, most of the aims and objectives were achieved; the other unachieved ones were altered and achieved in some way.

5.1.1 Aims

- **Program a chess engine**

A pretty broad aim. The original interpretation was, can the Engine finish a game as black or white, and find a checkmate, without the Engine crashing. It can do that, as can be seen in Testing 4.1. The goal was achieved not too far into the search and evaluation, around after Iterative deepening 3.2.3 in the search and after PeSTO Tables 3.3.1 in the evaluation.

- **Create a stylistic and simple GUI**

This aim was not achieved; however, the Engine does implement the UCI protocol 2.2, allowing the Engine to connect to any GUI that allows the UCI protocol. Making a GUI would have put a severe time constraint on every other aspect of the Engine, and as there were many other GUIs out there, this just seemed like the best choice.

- **Add player vs AI compatibility**

As mentioned in the last aim, implementing UCI allows the Engine to connect to a GUI with this capability.

5.1.2 Objectives

- **Develop a chessboard representation**

Bitboards were the first addition to the Engine; these bitboards were combined into a single ChessBoard struct 3.1.1.

- **Develop a chess search and evaluation algorithm**

Also needed to make a well-functioning Engine, Negamax was implemented as the search algorithm 3.2.1, with Hand Crafted Evaluation as the evaluation algorithm 4.2

- **Implement automated testing through GitHub**

This objective was not implemented. Perf tests 4.1.1 could have been used alongside automated

testing in GitHub. However, passing a suite of Perft tests ensures Board Representation is complete and correct, making it unnecessary.

- **Evaluate performance of chess engine**

The Engine was evaluated using CuteChess's [77] tournament feature (more in 4.2.1). Overall, it is strong compared to other Engines and players, but this performance measure is unnecessary. An Engine's strength should not be the only indication of its usefulness. For example, a Low-rated Engine that plays interesting moves but sticks to known concrete principles could be an even more significant asset to more players than other Engines that only try to get a higher rating.

- **Add intractability to GUI**

Again since the Engine implements the UCI protocol, this was unnecessary.

6 BSC Project Criteria

- **An ability to apply practical and analytical skills gained during the degree programme**

The combination of all the efforts from everything I have learned over the years in university made up this quality solution; without learning anything, the Engine would be both weak and unusable. However, as seen, this is far from the case.

- **Innovation and/or creativity**

Many minor optimizations spread throughout the Engine can be seen as having completed this objective; these optimizations allow less code to be written to accomplish the same objective, sometimes with a speed enhancement. Also, some of the more Construde bugs found in 4.1.2 needed a creative solution; there were plenty of instances when this objective was completed.

- **Synthesis of information, ideas and practices to provide a quality solution together with an evaluation of that solution**

Starting from lacking any knowledge about Chess Programming to creating an Engine that can beat a 2300-rated Chess Engine (In 4.2) alone answers this.

- **That your project meets a real need in a wider context.**

As mentioned in the Introduction 1, there are plenty of players in Chess, and a new Engine to compete against is not something to be ignored; this, combined with a complete overview of creating a Chess Engine, could also be used by someone else who would similarly like to get introduced into the world of Chess Programming.

- **An ability to self-manage a significant piece of work.**

The initial draft of a working Engine was completed long before any deadlines; this allowed me to improve it over a significant period, a primary reason why the playing strength is as strong as it is. The time spent on many areas, such as the tuning of parameters in the Search and Evaluation, could have been done better, these major time wastes were because, initially, I was unsure of how long some implementations or techniques took to implement, so I decided to implement or do the simple versions first, only the core parts of the Engine did not have these basic versions since as the name states, they are fundamental to how the Engine functions. Even with my current ability, some implementations will take some time.

Unlike the Engine, writing the paper started much later, actually since I wanted to get the Engine into a state I was happy with, I only started working on the paper around a month before the deadline, but because of how familiar I was with all the elements in Chess Programming, the writing of the paper's material did not take too long, but as seen with the sheer size of the reference list, trying to find the source of the techniques while also remembering where the implementations came from took a long time. In hind-sight, I should have made notes about the sources of the implementations.

- **Critical self-evaluation of the process.**

From the start, I knew very little, not where to start, nor the commonly used techniques, so there were many gaps in my knowledge at the time but over the course of the year my understanding of this area has vastly improved. The Engine in its final state still, while it could be improved in

many ways a stopping point was needed, as considering how many ideas there are within Chess, the programming of the Engine could have covered years.

In the end, however, the Engine is in a pretty good state, as seen with the number of techniques in 3 and performance in 4.2. So along with the Engine, this paper is similarly in a state I am happy with; everything I wanted to cover has been covered within the time frame given. However, if given more time, I would improve the evaluation, create a framework for automated tuning, and possibly try to create and implement an original technique.

References

- [1] C. E. Shannon, ‘Programming a computer for playing chess,’ *Philosophical Magazine Series 7*, vol. 41, no. 314, 1950.
- [2] ‘Live chess ratings.’ (9th May 2023), [Online]. Available: <https://www.2700chess.com/> (visited on 28/04/2023).
- [3] ‘Comparison of top chess players throughout history.’ (9th May 2023), [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_top_chess_players_throughout_history (visited on 28/04/2023).
- [4] ‘Turochamp.’ (25th Feb. 2021), [Online]. Available: <https://www.chessprogramming.org/Turochamp> (visited on 28/04/2023).
- [5] IBM, *Deep blue*. [Online]. Available: <https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/> (visited on 10/05/2023).
- [6] N. Tomašev, U. Paquet, D. Hassabis and V. Kramnik, ‘Reimagining chess with alphazero,’ *Commun. ACM*, vol. 65, no. 2, pp. 60–66, 2022, ISSN: 0001-0782. DOI: 10.1145/3460349. [Online]. Available: <https://doi.org/10.1145/3460349>.
- [7] nodchip *et al.* ‘Stockfish nnue.’ (), [Online]. Available: <https://github.com/nodchip/Stockfish> (visited on 10/05/2023).
- [8] K. Maksim. ‘Bbc,’ Github. (2nd Oct. 2020), [Online]. Available: https://github.com/maksimKorzh/chess_programming/tree/master/src/bbc (visited on 29/04/2023).
- [9] ChizhovVadim. ‘Counter,’ GitHub. (23rd Nov. 2022), [Online]. Available: <https://github.com/ChizhovVadim/CounterGo> (visited on 28/04/2023).
- [10] T. Riegle. ‘Chess program board representations,’ Crafty Chess. (21st Sep. 2016), [Online]. Available: <https://craftychess.com/hyatt/boardrep.html> (visited on 13/10/2022).
- [11] R. Tord, C. Marco and K. Joona. ‘Stockfish.’ (4th Dec. 2022), [Online]. Available: <https://stockfishchess.org/> (visited on 26/04/2023).
- [12] ‘Github copilot,’ Github. (25th Apr. 2023), [Online]. Available: <https://github.com/features/copilot> (visited on 29/04/2023).
- [13] *Main page*, 2021. [Online]. Available: https://www.chessprogramming.org/Main_Page (visited on 12/10/2022).
- [14] ‘Board representation,’ Chess programming wiki. (28th Jan. 2020), [Online]. Available: https://www.chessprogramming.org/Board_Representation (visited on 13/10/2022).
- [15] ‘Bitboards,’ Chess programming wiki. (12th Mar. 2022), [Online]. Available: <https://www.chessprogramming.org/Bitboards> (visited on 05/05/2023).
- [16] B. W. Kernighan and D. M. Ritchie, ‘The c programming language,’ in 2nd ed. Pearson Education, 1988, ISBN: 9780131103627.
- [17] ‘Population count,’ Chess programming wiki. (16th Nov. 2020), [Online]. Available: https://www.chessprogramming.org/Population_Count (visited on 27/04/2023).
- [18] C. E. Leiserson, H. Prokop and K. H. Randall, ‘Using de bruijn sequences to index a 1 in a computer word,’ *Available on the Internet from http://supertech.csail.mit.edu/papers.html*, vol. 3, no. 5, 1998.
- [19] ‘General setwise operations,’ Chess programming wiki. (18th Jan. 2022), [Online]. Available: https://www.chessprogramming.org/General_Setwise_Operations (visited on 27/04/2023).
- [20] ‘Magic bitboards,’ Chess programming wiki. (8th Jul. 2021), [Online]. Available: https://www.chessprogramming.org/Magic_Bitboards (visited on 28/04/2023).
- [21] I. Gerd, ‘Efficient generation of moves and controls of sliding pieces,’ Workshop Chess and Mathematics, 21st Nov. 2008. [Online]. Available: https://www.chessprogramming.org/Efficient_Generation_of_Sliding_Piece_Attacks (visited on 28/04/2023).
- [22] K. Maksim. ‘Bitboard chess engine in c: Generating magic number candidates,’ YouTube. (25th Aug. 2020), [Online]. Available: <https://www.youtube.com/watch?v=KqWe0Vy0oyU%5C&list=PLmN0neTso3Jxh8ZIylk74JpwfiWNI76Cs%5C&index=14> (visited on 27/04/2023).

- [23] R. R. Elliott, ‘Fast chess move generation with magic bitboards,’ 19th Jan. 2019. [Online]. Available: <https://rhyssre.net/fast-chess-move-generation-with-magic-bitboards.html> (visited on 27/04/2023).
- [24] ‘Fast(er) bitboard move generator,’ Winboard forum. (14th Jun. 2006), [Online]. Available: <http://www.open-aurec.com/wbforum/viewtopic.php?t=5015> (visited on 28/04/2023).
- [25] R. Tord. ‘Magic move generation.’ (3rd Aug. 2006), [Online]. Available: https://www.talkchess.com/forum3/viewtopic.php?topic_view=threads&p=175834&t=19699 (visited on 28/04/2023).
- [26] K. Maksim. ‘Bitboard chess engine in c: Implementing make move function (moving pieces),’ YouTube. (4th Sep. 2020), [Online]. Available: <https://www.youtube.com/watch?v=coVPpTJN9iU&list=PLmN0neTso3Jxh8ZIylk74JpwiWNi76Cs&index=32> (visited on 29/04/2023).
- [27] ‘Negamax,’ Chess programming wiki. (27th Apr. 2018), [Online]. Available: <https://www.chessprogramming.org/Negamax> (visited on 08/05/2023).
- [28] ‘Alphabeta,’ Chess programming wiki. (1st Dec. 2021), [Online]. Available: <https://www.chessprogramming.org/Alpha-Beta> (visited on 08/05/2023).
- [29] ‘Fail-soft,’ Chess programming wiki. (12th Mar. 2022), [Online]. Available: <https://www.chessprogramming.org/Fail-Soft> (visited on 08/05/2023).
- [30] ‘Horizon effect,’ Chess programming wiki. (11th Aug. 2020), [Online]. Available: https://www.chessprogramming.org/Horizon_Effect (visited on 08/05/2023).
- [31] L. R. Harris, ‘The heuristic search and the game of chess - a study of quiescence, sacrifices, and plan oriented play,’ in *International Joint Conference on Artificial Intelligence*, 1975.
- [32] ‘Quiescence search,’ Chess programming wiki. (15th Jul. 2021), [Online]. Available: https://www.chessprogramming.org/Quiescence_Search (visited on 08/05/2023).
- [33] ‘Iterative deepening,’ Chess programming wiki. (4th Oct. 2019), [Online]. Available: https://www.chessprogramming.org/Iterative_Deepening (visited on 08/05/2023).
- [34] ‘Time managment,’ Chess programming wiki. (22nd Apr. 2022), [Online]. Available: https://www.chessprogramming.org/Time_Management (visited on 08/05/2023).
- [35] D. J. Slate and L. R. Atkin, ‘Chess 4.5—the northwestern university chess program,’ in *Chess Skill in Man and Machine*, P. W. Frey, Ed. New York, NY: Springer New York, 1983, ISBN: 978-1-4612-5515-4. DOI: 10.1007/978-1-4612-5515-4_4. [Online]. Available: https://doi.org/10.1007/978-1-4612-5515-4_4.
- [36] ‘Aspiration windows,’ WayBackMachine. (31st Oct. 2007), [Online]. Available: <https://web.archive.org/web/20071031095918/http://www.brucemo.com/compchess/programming/aspiration.htm> (visited on 08/05/2023).
- [37] R. Shams, H. Kaindl and H. Horacek, ‘Using aspiration windows for minimax algorithms,’ in *IJCAI*, 1991. [Online]. Available: <https://www.ijcai.org/Proceedings/91-1/Papers/019.pdf>.
- [38] H. Kaindl, R. Shams and H. Horacek, ‘Minimax search algorithms with and without aspiration windows,’ *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 12, 1991.
- [39] ‘Move ordering,’ Chess programming wiki. (12th Mar. 2022), [Online]. Available: https://www.chessprogramming.org/Move_Ordering (visited on 08/05/2023).
- [40] V. Marcel. ‘How move ordering works.’ (2019), [Online]. Available: <https://rustic-chess.org/search/ordering/how.html> (visited on 08/05/2023).
- [41] ‘Mvv_lva,’ Chess programming wiki. (1st Nov. 2019), [Online]. Available: <https://www.chessprogramming.org/MVV-LVA> (visited on 08/05/2023).
- [42] V. Marcel. ‘Mvv_lva.’ (2019), [Online]. Available: https://rustic-chess.org/search/ordering/mvv_lva.html (visited on 08/05/2023).
- [43] ‘Static exchange evaluation.’ (1st Aug. 2022), [Online]. Available: https://www.chessprogramming.org/Static_Exchange_Evaluation (visited on 08/05/2023).
- [44] S. G. Akl and M. M. Newborn, ‘The principal continuation and the killer heuristic,’ in *Proceedings of the 1977 Annual Conference*, ser. ACM ’77, Seattle, Washington: Association for Computing Machinery, 1977, pp. 466–473, ISBN: 9781450339216. DOI: 10.1145/800179.810240. [Online]. Available: <https://doi.org/10.1145/800179.810240>.

- [45] ‘Killer move,’ Chess programming wiki. (20th Jul. 2021), [Online]. Available: https://www.chessprogramming.org/Killer_Move (visited on 08/05/2023).
- [46] S. Ed. ‘How rebel plays chess.’ (2004), [Online]. Available: <https://www2.seas.gwu.edu/~simhaweb/champalg/chess/papers/SchroderInsideRebel.pdf> (visited on 08/05/2023).
- [47] J. Schaeffer, ‘The history heuristic,’ *ICGA Journal*, vol. 6, no. 3, pp. 16–19, 1983. DOI: 10.3233/ICG-1983-6305.
- [48] M. Winands, E. Werf, H. Herik and J. Uiterwijk, ‘The relative history heuristic,’ Jan. 2006, pp. 262–272, ISBN: 978-3-540-32488-1. DOI: 10.1007/11674399_18.
- [49] D. Hartmann, ‘Butterfly boards,’ *ICCA Journal*, vol. 11, no. 2/3, pp. 60–63, 1988.
- [50] ‘Relative history heuristic,’ Chess programming wiki. (19th Dec. 2018), [Online]. Available: https://www.chessprogramming.org/Relative_History_Heuristic#Alternatives (visited on 08/05/2023).
- [51] H. Kaindl, ‘Searching to variable depth in computer chess,’ Jan. 1983, pp. 760–762.
- [52] T. S. Anantharaman, M. Campbell and F.-h. Hsu, ‘Singular extensions: Adding selectivity to brute-force searching,’ *Artif. Intell.*, vol. 43, pp. 99–109, 1990.
- [53] T. A. Marsland and M. Campbell, ‘Parallel search of strongly ordered game trees,’ *ACM Comput. Surv.*, vol. 14, no. 4, pp. 533–551, 1982, ISSN: 0360-0300. DOI: 10.1145/356893.356895. [Online]. Available: <https://doi.org/10.1145/356893.356895>.
- [54] M. Bruce. ‘Principal variation search,’ WayBackMachine. (11th Apr. 2002), [Online]. Available: <https://web.archive.org/web/20040427015506/http://brucemo.com/compchess/programming/pvs.htm> (visited on 08/05/2023).
- [55] N. Sato and K. Ikeda, ‘Three types of forward pruning techniques to apply the alpha beta algorithm to turn-based strategy games,’ *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, 2016.
- [56] ‘Late move reductions,’ Chess programming wiki. (4th Apr. 2022), [Online]. Available: https://www.chessprogramming.org/Late_Move_Reductions (visited on 08/05/2023).
- [57] ‘Null move pruning,’ Chess programming wiki. (25th Dec. 2020), [Online]. Available: https://www.chessprogramming.org/Null_Move_Pruning (visited on 08/05/2023).
- [58] J. Schaeffer, ‘Experiments in search and knowledge,’ Ph.D. dissertation, Department of Computing Science, University of Waterloo, Canada, 1986.
- [59] ‘Futility pruning,’ Chess programming wiki. (8th Jul. 2021), [Online]. Available: https://www.chessprogramming.org/Futility_Pruning (visited on 08/05/2023).
- [60] ‘Reverse futility pruning,’ Chess programming wiki. (29th Apr. 2018), [Online]. Available: https://www.chessprogramming.org/Reverse_Futility_Pruning (visited on 08/05/2023).
- [61] E. A. Heinz, *Scalable search in computer chess: Algorithmic enhancements and experiments at high search depths*. Springer Science & Business Media, 2013.
- [62] R. Greenblatt, D. Eastlake and S. D. Crocker, ‘The greenblatt chess program,’ in *Proceedings of the AfiPs Fall Joint Computer Conference*, ACM, vol. 31, 1967, pp. 801–810.
- [63] A. L. Zobrist, ‘A new hashing method with application for game playing,’ *ICGA Journal*, vol. 13, pp. 69–73, 1990.
- [64] H. Robert and C. Anthony, ‘The effect of hash signature collisions in a chess program,’ *ICGA Journal*, vol. 28, no. 3, 2005.
- [65] ‘Evaluation,’ Chess programming wiki. (9th Sep. 2021), [Online]. Available: <https://www.chessprogramming.org/Evaluation> (visited on 08/05/2023).
- [66] F. Ronald. ‘New uci engine: Rofchade,’ TalkChess.com. (23rd Jan. 2018), [Online]. Available: <https://www.talkchess.com/forum3/viewtopic.php?f=2&t=68311&start=19> (visited on 08/05/2023).
- [67] K. Tom. ‘Tscp.’ (1997), [Online]. Available: <http://www.tckerrigan.com/Chess/TSCP/> (visited on 08/05/2023).
- [68] H. Berliner, ‘On the construction of evaluation functions for large domains. iii. smoothness,’ in *IJCAI*, vol. 79, Tokyo, 1979.

- [69] ‘Tapered eval,’ Chess programming wiki. (24th Jun. 2021), [Online]. Available: https://www.chessprogramming.org/Tapered_Eval (visited on 08/05/2023).
- [70] ‘King safety,’ Chess programming wiki. (29th Mar. 2021), [Online]. Available: https://www.chessprogramming.org/King_Safety (visited on 08/05/2023).
- [71] ‘Texel’s tuning method.’ (28th Mar. 2022), [Online]. Available: https://www.chessprogramming.org/Texel%27s_Tuning_Method (visited on 28/04/2023).
- [72] Lichess. (4th Jul. 2022), [Online]. Available: <https://lichess.org/> (visited on 12/10/2022).
- [73] chess.com. (2nd Nov. 2022), [Online]. Available: <https://www.chess.com/> (visited on 12/10/2022).
- [74] B. Graham, B. Ray, B. Sarah, K. Kirill and S. Charles, CCRL. (29th Oct. 2022), [Online]. Available: <https://www.computerchess.org.uk/ccrl/> (visited on 23/10/2022).
- [75] V. Joel. ‘Bodo 0.2b,’ Computer-Chess Wiki. (3rd May 2019), [Online]. Available: http://www.computer-chess.org/doku.php?id=computer_chess:wiki:download:engine_download_list (visited on 11/05/2023).
- [76] P. Marco. ‘Napoleon 1.8,’ Github. (28th Jun. 2017), [Online]. Available: <https://github.com/crybot/Napoleon> (visited on 11/05/2023).
- [77] P. Ilari and J. Arto, Cute Chess. (9th Aug. 2020), [Online]. Available: <https://cutechess.com/> (visited on 24/10/2022).
- [78] ‘Grammarly.’ (2023), [Online]. Available: <https://www.grammarly.com/> (visited on 12/05/2023).
- [79] J. Hammersley and J. Lees-Miller, *Overleaf*, 2023. [Online]. Available: <https://www.overleaf.com>.
- [80] L. Lamport, *Latex: A document preparation system*, Addison-Wesley, Reading, MA, 1986.

7 Appendices

```
// Best magic number seed found so far: 15

// Rook magic numbers, use the SquareToIndex map to index these numbers
var rookMagicNumber = [64]uint64{
    1188950372496965666, 1170936040556331072, 648531610290888704, 5044036049991417992, 1585284669612508168, 144119622663143696, 288230930773182480, 4755801481985165586,
    1189372515164895872, 38421471764152448, 2308235615243608064, 563027264932928, 2523423187686918144, 2306405976615550984, 10674376643465838596, 2378463558642205193,
    90213279793692672, 75041670696961, 576602039816036352, 4504149517406336, 1442278884473047840, 37718748672819456, 144679241936134912, 10905393642756,
    2684145517499269248, 36312496792961056, 36436720132071425, 2289221864784128, 6756503348170204, 761671304226021302, 9259968186185417216, 2305860910229679753,
    141012374659072, 141012374659072, 301741346834613249, 20301438546612225, 5226427384768497792, 288232577330840576, 3531387256851988484, 576742915582201604,
    117234466312716288, 5769112581883125760, 144132780798804096, 2452491609787400200, 2533309151248388, 8444387008971776, 576462951393853696, 145258972505571329,
    4616260177925046528, 2306485682366851584, 619316443076362496, 20266234898546816, 659289165065715840, 306385520739745920, 9223380841555559424, 72339077612978432,
    9259436301730480146, 288270242340208770, 158468220749954, 576470648445210625, 9252927201503881217, 577023708837982210, 2449994484127629860, 146649567232000386,
}

// Bishop magic numbers, use the SquareToIndex map to index these numbers
var bishopMagicNumber = [64]uint64{
    290517652400111648, 9802093389561212928, 1162511462273384448, 2287534547664898, 1157574778073124912, 9296012381182246928, 324823291398225922, 142940817881408,
    869212328892566024, 176207737830834704, 18150130125312, 19144713785626624, 238719369324633216, 9016143256748354, 4684317566393518786, 2522015937390075904,
    2251869106227202, 4900205103304573000, 9228441120125758210, 1139265882810560, 289356542381105266, 4612847107010215936, 5620773861407250352, 288266668659048960,
    325402674392531204, 585768290371242000, 3461029587908256020, 1170940301167100065, 18295944369946624, 3386771774119936, 563224865407536, 142386791715840,
    5884160155787264, 5765258451114657824, 585503342105986176, 1152925919833752800, 4611968610095726624, 11538260733599025176, 2594381545108799748, 4653556549732663809,
    36314679710064771, 144406077690621952, 9512167570724620801, 10377560454305189920, 79371265114368, 2458969865491316768, 2333014142791256192, 292808813438894212,
    5206732923911278592, 1153062827302332416, 18859924042809352, 2324561104862741120, 9228016618525950340, 22526931803079680, 5647153061101568, 9009003149427216,
    11547248694889963584, 76968030921232, 722854132909641728, 4719772478205888514, 578727946423915520, 19140315865678404, 18021065104098562, 232963647627227584,
}
```

Figure 40: The magic numbers generated using Torm Romstad’s method[25]

```
go perf ft 5
a2a3: 181046
a2a4: 217832
b2b3: 215255
b2b4: 216145
c2c3: 222861
c2c4: 240082
d2d3: 328511
d2d4: 361790
e2e3: 402988
e2e4: 405385
f2f3: 178889
f2f4: 198473
g2g3: 217210
g2g4: 214048
h2h3: 181044
h2h4: 218829
b1a3: 198572
b1c3: 234656
g1f3: 233491
g1h3: 198502

Total nodes: 4865609
```

Figure 41: Perf ft Output from UCI 2.2. Starting position used, searched to depth 5.


```

position startpos moves a2a3
go perft 4
a7a6: 7754
a7a5: 8551
b7b6: 8568
b7b5: 8573
c7c6: 8499
c7c5: 8934
d7d6: 10963
d7d5: 11402
e7e6: 12103
e7e5: 12129
f7f6: 7754
f7f5: 8189
g7g6: 8568
g7g5: 8551
h7h6: 7754
h7h5: 8553
b8a6: 8163
b8c6: 8960
g8f6: 8936
g8h6: 8142
Total nodes: 181046

```

Figure 42: Perft Output from UCI 2.2. Example as a way to expand incorrect Perft tests to pinpoint the problem

```

info depth 1 nodes 26 score cp 52 time 0 pv g1f3
info depth 2 nodes 174 score cp 0 time 1 pv g1f3 g8f6
info depth 3 nodes 350 score cp 49 time 1 pv g1f3 g8f6 b1c3
info depth 4 nodes 2533 score cp 0 time 8 pv g1f3 g8f6 b1c3 b8c6
info depth 5 nodes 4545 score cp 48 time 15 pv g1f3 g8f6 b1c3 b8c6 d2d4
info depth 6 nodes 21109 score cp 0 time 45 pv g1f3 g8f6 b1c3 b8c6 d2d4 d7d5
info depth 7 nodes 33250 score cp 36 time 63 pv g1f3 g8f6 b1c3 b8c6 d2d4 d7d5 c1f4
info depth 8 nodes 101037 score cp 2 time 145 pv c2c4 g8f6 g1f3 b8c6 b1c3 d7d5 c4d5 f6d5
info depth 9 nodes 178762 score cp 27 time 235 pv c2c4 e7e5 b1c3 c7c5 g1f3 b8c6 e2e4 g8f6 f1d3
info depth 10 nodes 276899 score cp 7 time 348 pv c2c4 d7d5 c4d5 c7c6 g1f3 g8f6 d5c6 b8c6 d2d4 c8f5
info depth 11 nodes 350808 score cp 21 time 438 pv c2c4 b8c6 b1c3 e7e5 g1f3 g8f6 e2e4 f8d6 f1d3 c6b4 d3e2
info depth 12 nodes 936857 score cp 10 time 1176 pv g1f3 g8f6 b1c3 d7d5 d2d4 b8c6 c1f4 c8f5 e2e3 e7e6 f1d3 f8e7
info depth 13 nodes 1055801 score cp 25 time 1345 pv g1f3 g8f6 b1c3 d7d5 d2d4 b8c6 c1f4 c8f5 e2e3 c6b4 a1c1 c7c6 f4e5
info depth 14 nodes 1206208 score cp 24 time 1627 pv g1f3 g8f6 b1c3 d7d5 d2d4 b8c6 c1f4 c8e6 e2e3 h7h6 f1d3 g7g5 f4e5 f8g7
info depth 15 nodes 1491491 score cp 25 time 2000 pv g1f3 g8f6 b1c3 d7d5 d2d4 b8c6 c1f4 e7e6 e2e3 f8b4 f1d3 f6e4 d3e4 d5e4 f3e5
info depth 16 nodes 5071199 score cp 14 time 6498 pv e2e4 e7e5 g1f3 b8c6 d2d4 e5d4 f3d4 g8f6 d4c6 b7c6 f1d3 f8d6 b1c3 e8g8 e1g1 c8b7
info depth 17 nodes 6012559 score cp 23 time 7581 pv e2e4 e7e5 g1f3 b8c6 d2d4 e5d4 f3d4 g8f6 d4c6 d7c6 f1d3 e8d8 b1c3 f8d6 c1g5 d8e7 g5f6 e7f6 e1c1
info depth 18 nodes 7668220 score cp 22 time 9392 pv e2e4 e7e5 g1f3 b8c6 b1c3 g8f6 f1b5 f8b4 e1g1 e8g8 d2d3 f8a8 c1g5 b4c3 b2c3 h7h6 g5e3 d7d5
info depth 19 nodes 10417349 score cp 19 time 12396 pv e2e4 e7e5 g1f3 b8c6 b1c3 g8f6 f1b5 f8b4 e1g1 e8g8 a2a3 b4c3 d2c3 f6e4 f1e1 d7d5 b5c6 b7c6 f3e5
info depth 20 nodes 14210084 score cp 20 time 16621 pv e2e4 e7e5 g1f3 b8c6 d2d4 e5d4 f3d4 g8f6 d4c6 b7c6 f1d3 f8d6 e1g1 e8g8 c2c4 c8b7 b1c3 d6e5 c1e3 d7d5

```

Figure 43: The starting position searched to a depth of 20 using command ‘go’

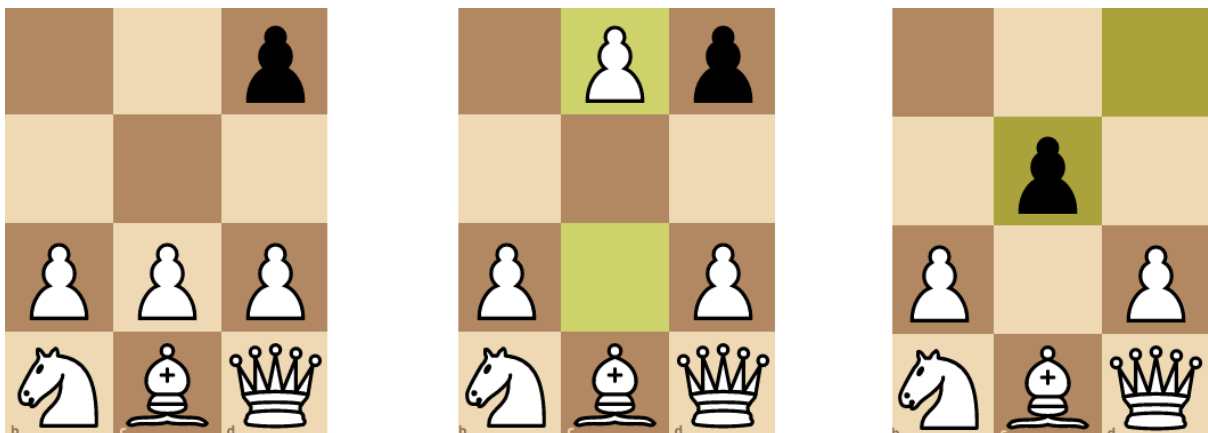


Figure 44: The prerequisites for an en passant capture, the third image is the en passant, images were made on Lichess[72]

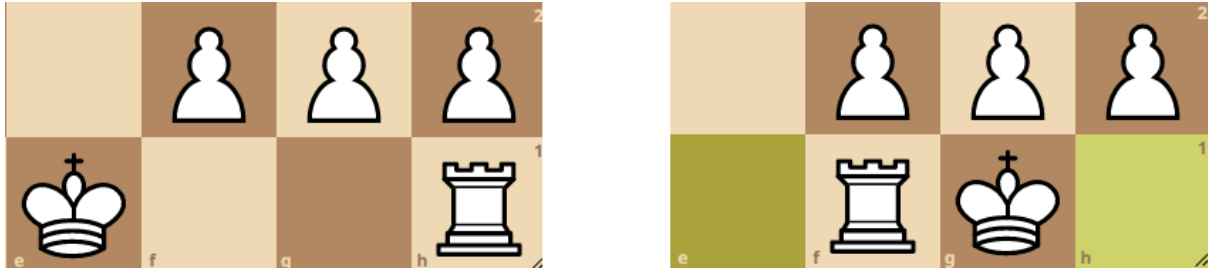


Figure 45: King side castle, similar move possible on the queen side, images were made on Lichess[72]

```

                                With move ordering
info depth 1 nodes 26 score cp 52 time 0 pv g1f3
info depth 2 nodes 214 score cp 0 time 0 pv g1f3 g8f6
info depth 3 nodes 423 score cp 49 time 0 pv g1f3 g8f6 b1c3
info depth 4 nodes 2564 score cp 0 time 4 pv g1f3 g8f6 b1c3 b8c6
info depth 5 nodes 4624 score cp 48 time 7 pv g1f3 g8f6 b1c3 b8c6 d2d4
info depth 6 nodes 18142 score cp 0 time 39 pv g1f3 g8f6 b1c3 b8c6 d2d4 d7d5
info depth 7 nodes 61001 score cp 19 time 123 pv e2e4 d7d5 e4d5 d8d5 d2d4 d5e4 c1e3 g8f6
info depth 8 nodes 186754 score cp 2 time 333 pv c2c4 g8f6 g1f3 b8c6 b1c3 d7d5 c4d5 f6d5
info depth 9 nodes 402931 score cp 27 time 596 pv c2c4 g8f6 g1f3 c7c5 b1c3 b8c6 e2e4 e7e5 f1d3
info depth 10 nodes 504336 score cp 7 time 706 pv c2c4 g8f6 d2d4 d7d5 c4d5 c7c6 d5c6 b8c6 g1f3 c8f5
info depth 11 nodes 896932 score cp 27 time 1194 pv c2c4 g8f6 d2d4 e7e6 b1c3 c7c5 d4d5 f8d6 c3b5 d6e5 g1f3
info depth 12 nodes 1331713 score cp 13 time 1682 pv c2c4 g8f6 d2d4 e7e6 b1c3 c7c5 d4d5 e6d5 c3d5 f6d5 c4d5 f8d6
info depth 13 nodes 2525173 score cp 15 time 2866 pv c2c4 g8f6 d2d4 e7e6 g1f3 c7c5 b1c3 c5d4 f3d4 f8b4 c1f4 e8g8 a1c1
info depth 14 nodes 3690549 score cp 15 time 4063 pv c2c4 g8f6 g1f3 c7c5 b1c3 b8c6 e2e4 e7e6 d2d3 f8d6 c3b5 d6e7 e4e5 f6g4
info depth 15 nodes 7143892 score cp 23 time 7645 pv c2c4 g8f6 g1f3 c7c5 b1c3 b8c6 e2e4 e7e6 d2d4 c5d4 f3d4 d7d5 c4d5 e6d5 c1g5

                                Without move ordering
info depth 1 nodes 26 score cp 52 time 0 pv g1f3
info depth 2 nodes 519 score cp 0 time 1 pv g1f3 g8f6
info depth 3 nodes 2957 score cp 49 time 3 pv b1c3 g8f6 g1f3
info depth 4 nodes 15836 score cp 0 time 18 pv b1c3 b8c6 g1f3 g8f6
info depth 5 nodes 34692 score cp 27 time 37 pv c2c3 d7d5 d2d4 g8f6 g1f3
info depth 6 nodes 198398 score cp 6 time 170 pv d2d4 g8f6 c1g5 f6e4 g1f3 b8c6
info depth 7 nodes 838111 score cp 19 time 614 pv e2e4 d7d5 e4d5 d8d5 d2d4 d5e4 c1e3 g8f6
info depth 8 nodes 4453908 score cp 13 time 3067 pv e2e4 c7c6 d2d4 d7d5 e4e5 c8f5 b1c3 b8d7
info depth 9 nodes 14154095 score cp 28 time 9539 pv e2e4 d7d5 e4d5 c7c6 d5c6 b8c6 b1c3 g8f6 g1f3
info depth 10 nodes 48271327 score cp 12 time 30187 pv c2c4 c7c5 b1c3 e7e5 e2e3 b8c6 g1f3 d7d6 d2d4 c8f5
info depth 11 nodes 102074280 score cp 17 time 67548 pv c2c4 e7e5 d2d4 e5d4 d1d4 d7d6 b1c3 b8c6 d4d1 c8e6 c3d5
info depth 12 nodes 421392711 score cp 10 time 290689 pv c2c4 e7e5 d2d4 e5d4 d1d4 d7d6 g1f3 b8c6 d4e3 f8e7 b1c3 g8f6 c3d5

```

Figure 46: Showing the difference in the performance for the move ordering technique

Without Mate Distance Pruning

```
info depth 21 nodes 1718528 score cp 685 time 1273 pv h8h3 d2e2 b7c6 e2f2
info depth 22 nodes 2036353 score cp 698 time 1500 pv h8h3 d2e2 b7c6 e2f2
info depth 23 nodes 3878380 score mate 14 time 2711 pv h8h3 d2e2 b7c6 e2f2
info depth 24 nodes 5952474 score mate 14 time 4142 pv h8h3 d2e2 b7c6 e2d2
info depth 25 nodes 6146975 score mate 14 time 4273 pv h8h3 d2e2 b7c6 e2d2
info depth 26 nodes 6491065 score mate 14 time 4505 pv h8h3 d2e2 b7c6 e2f1
info depth 27 nodes 6841894 score mate 13 time 4758 pv h8h3 d2e2 b7c6 e2f1
info depth 28 nodes 7356091 score mate 12 time 5106 pv h8h3 d2e2 b7c6 e2f1
info depth 29 nodes 7945881 score mate 12 time 5460 pv h8h3 d2e2 b7c6 e2f1
info depth 30 nodes 8685061 score mate 12 time 5929 pv h8h3 d2e2 b7c6 e2f1
bestmove h8h3
```

With Mate Distance Pruning

```
info depth 21 nodes 1718528 score cp 685 time 1291 pv h8h3 d2e2 b7c6 e2f2
info depth 22 nodes 2036374 score cp 698 time 1526 pv h8h3 d2e2 b7c6 e2f2
info depth 23 nodes 3914926 score mate 14 time 2919 pv h8h3 d2e2 b7c6 e2f2
info depth 24 nodes 4431804 score mate 14 time 3308 pv h8h3 d2e2 b7c6 e2f2
info depth 25 nodes 5099210 score mate 14 time 3750 pv h8h3 d2e2 b7c6 e2d2
info depth 26 nodes 5415133 score mate 14 time 3967 pv h8h3 d2e2 b7c6 e2d2
info depth 27 nodes 5834173 score mate 13 time 4256 pv h8h3 d2e2 b7c6 e2d2
info depth 28 nodes 6316383 score mate 13 time 4581 pv h8h3 d2e2 b7c6 e2d2
info depth 29 nodes 6891089 score mate 12 time 4969 pv h8h3 d2e2 b7c6 e2d2
info depth 30 nodes 7687187 score mate 12 time 5475 pv h8h3 d2e2 b7c6 e2d2
bestmove h8h3
```

Figure 47: Showing the difference in the performance for the Made Distance Pruning technique

```

// move ordering
// https://www.chessprogramming.org/Move_Ordering
// Give each move a score depending on the type of move
func scoreMoves(movelist *[]board.Move, bestMove board.Move) {
    moves := (*movelist)

    for i := range moves {
        flags := moves[i].GetMoveFlags()
        switch {
        case moves[i].Move == bestMove.Move: // Best Move / PV Move
            moves[i].Score = moveOrderOffset + 72
        case flags == board.MoveQueenPromotionCapture: // Queen Promotion Capture
            moves[i].Score = moveOrderOffset + 71
        case flags == board.MoveQueenPromotion: // Queen Promotion
            moves[i].Score = moveOrderOffset + 70
        case flags & board.MoveCaptures != 0: // Captures
            moves[i].Score = moveOrderOffset + MVV_LVA[moves[i].GetMoveCapturedPiece()][moves[i].GetMoveStartPiece()]
        case moves[i].Move == killerMoves[0][board.Ply].Move: // Killer Moves
            moves[i].Score = moveOrderOffset - 1
        case board.Ply > 1 && moves[i].Move == killerMoves[0][board.Ply-2].Move:
            moves[i].Score = moveOrderOffset - 2
        case moves[i].Move == killerMoves[1][board.Ply].Move:
            moves[i].Score = moveOrderOffset - 3
        case board.Ply > 1 && moves[i].Move == killerMoves[1][board.Ply-2].Move:
            moves[i].Score = moveOrderOffset - 4
        case flags == board.MoveKingCastle: // King-side Castle
            moves[i].Score = moveOrderOffset - 5
        case flags == board.MoveQueenCastle: // Queen-side Castle
            moves[i].Score = moveOrderOffset - 6
        case flags >= board.MoveKnightPromotion: // ALL other promotions
            moves[i].Score = moveOrderOffset - 7
        default: // Relative history heuristic
            moves[i].Score = hhScore[moves[i].GetMoveStart()][moves[i].GetMoveEnd()] / (bfScore[moves[i].GetMoveStart()][moves[i].GetMoveEnd()] + 1)
        }
    }
}

```

Figure 48: All the move ordering conditions, explanations for each in 3.2.4

```

// found a better move
if score > alpha {
    // fails high
    if score >= beta {
        hashFlag = hashFlagBeta

        if isSafe {
            if bestMove.Move != killerMoves[0][board.Ply].Move {
                killerMoves[1][board.Ply] = killerMoves[0][board.Ply]
                killerMoves[0][board.Ply] = bestMove
            }

            hhScore[bestMove.GetMoveStart()][bestMove.GetMoveEnd()] += 100

            for _, prevMove := range prevSafeMoves {
                bfScore[prevMove.GetMoveStart()][prevMove.GetMoveEnd()] += 1
            }
        }

        break
    }
}

```

Figure 49: the Killer Move Heuristic and the Relative History Heuristic

```

position fen r3k2r/ppp1bpp1/6q1/3pP3/6p1/P1P1P2P/1B3P2/R2Q1RK1 w kq - 0 17
go
info depth 1 nodes 53 score cp 0 time 0 pv h3g4
info depth 2 nodes 319 score cp -25 time 0 pv d1a4 b7b5 a4b5 c7c6
info depth 3 nodes 1665 score cp -45 time 2 pv h3g4 e8c8 d1f3
info depth 4 nodes 5014 score cp -69 time 8 pv d1g4 h8h3 g4g6 f7g6
info depth 5 nodes 13126 score cp -74 time 19 pv d1a4 c7c6 a4g4 h8h3 g4g6 f7g6
info depth 6 nodes 23115 score cp -62 time 33 pv d1g4 h8h3 g4g6 f7g6 e5e6 e8c8
info depth 7 nodes 37289 score cp -60 time 54 pv d1a4 c7c6 a4g4 g6g4 h3g4 e8c8 a3a4 h8h4 f2f3
info depth 8 nodes 61069 score cp -67 time 85 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 a1d1 h4h3 f1f2
info depth 9 nodes 99840 score cp -67 time 126 pv d1a4 c7c6 a4g4 g6g4 h3g4 h8h4 f2f3 e8c8 a1d1 h4h3 f1f2
info depth 10 nodes 161324 score cp -84 time 198 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 a1d1 e7c5 f1e1 h4h3 g1f2
info depth 11 nodes 188266 score cp -85 time 231 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 a1d1 e7c5 f1e1 d8h8 g4g5 h4h1 g1f2
info depth 12 nodes 254567 score cp -88 time 309 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 a1d1 e7c5 b2c1 c7c6 g1g2 d8e8 g2g3
info depth 13 nodes 346316 score cp -98 time 411 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 a1d1 e7c5 b2c1 c7c6 g1g2 d8h8 f1f2 h4h2 g2g3
info depth 14 nodes 557382 score cp -122 time 638 pv d1a4 c7c6 a4g4 g6g4 h3g4 h8h4 f2f3 e8c8 a3a4 e7c5 f1e1 d8e8 b2a3 c5a3 a1a3 e8e5
info depth 15 nodes 783746 score cp -147 time 897 pv d1a4 c7c6 a4g4 g6g4 h3g4 h8h4 f2f3 e8c8 g1g2 d8h8 f1h1 h4h1 a1h1 h8h1 g2h1 e7c5 b2c1
info depth 16 nodes 1285045 score cp -91 time 1434 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 g1g2 d8h8 f1h1 h4h1 a1h1 h8h1 g2h1 e7c5 b2c1 a7a5 h1g2
info depth 17 nodes 1447129 score cp -116 time 1638 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 g1g2 d8h8 f1h1 h4h1 a1h1 h8h1 g2h1 e7c5 b2c1 c8d7 h1g2 d7c6
info depth 18 nodes 1721260 score cp -124 time 1926 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 g1g2 e7c5 f1h1 h4h1 a1h1 c5e3 h1h7 e3h6 g2g3 a7a5 a3a4 c8b8 f3f4
info depth 19 nodes 2061983 score cp -116 time 2345 pv d1g4 g6g4 h3g4 h8h4 f2f3 e8c8 g1g2 e7c5 f1h1 h4h1 a1h1 c5e3 h1h7 d8g8 g2g3 c8b8 f3f4 b7b5 g4g5 b8b7
info depth 20 nodes 3952769 score cp -109 time 4407 pv d1g4 g6g4 h3g4 h8h4 f2f3 e7c5 g1g2 c5e3 g2g3 h4h8 b2c1 e3c5 c1g5 e8d7 a3a4 a8e8 f1e1 a7a5 a1d1 d7c6 e1e2
info depth 21 nodes 4950296 score cp -113 time 5558 pv d1g4 g6g4 h3g4 h8h4 f2f3 e7c5 g1g2 c5e3 g2g3 h4h8 b2c1 e3c5 c1g5 e8d7 a3a4 a8e8 f1e1 a7a5 a1d1 d7c6 e1e2 f7f6
info depth 22 nodes 8578056 score cp -124 time 9916 pv d1a4 c7c6 a4g4 g6g4 h3g4 h8h4 f2f3 e8c8 g1g2 e7c5 f1h1 h4h1 a1h1 c5e3 b2c1 e3c1 h1c1 c8c7 c1d1 d8e8 f3f4 a7a5 a3a4 e8h8

```

Figure 50: Search for Position FEN: r3k2r/ppp1bpp1/6q1/3pP3/6p1/P1P1P2P/1B3P2/R2Q1RK1 w kq - 0 17

```

position fen 5rrk/p1p1R3/n2pRp2/1Q1P1q1p/1P1P4/P5Pp/5P2/5NK1 w - - 1 31
go
info depth 1 nodes 51 score cp 380 time 0 pv b5a6
info depth 2 nodes 193 score cp 380 time 0 pv b5a6 f5d5
info depth 3 nodes 952 score cp 381 time 1 pv b5a6 f5d5 a6d3
info depth 4 nodes 3347 score cp 370 time 6 pv b5a6 f5d5 f1e3 d5d4
info depth 5 nodes 5013 score cp 370 time 12 pv b5a6 f5d5 f1e3 d5d4 e7c7
info depth 6 nodes 10410 score cp 372 time 23 pv b5a6 f5d5 f1e3 d5d4 e7c7 d4a1 g1h2
info depth 7 nodes 20372 score cp 374 time 42 pv b5a6 g8g7 a6a7 g7e7 e6e7 f5d5 e7c7
info depth 8 nodes 298205 score cp 267 time 441 pv b5e2 g8g7 e2a6 f5d5 f1e3 d5d4 e3f5 h3h2 g1h2
info depth 9 nodes 309617 score cp 209 time 476 pv b5e2 g8g7 e2a6 f5f3 f1e3 h3h2 g1h2 f3f2 h2h1 g7g3 a6a7
info depth 10 nodes 561093 score cp 289 time 827 pv b5e2 a6b8 f1e3 f5g5 e2f1 g8g7 f1h3 f6f5 e7e8 f8e8
info depth 11 nodes 650156 score cp 293 time 966 pv b5e2 a6b8 f1e3 f5g6 e2f1 h3h2 g1h2 g8g7 f1h3 g7e7 e6e7 b8a6
info depth 12 nodes 722863 score cp 297 time 1073 pv b5e2 a6b8 f1e3 f5g6 e2f1 f8f7 f1h3 b8d7 e3f5 f7e7 e6e7 g8d8
info depth 13 nodes 805023 score cp 293 time 1194 pv b5e2 a6b8 f1e3 f5g6 e2f1 h3h2 g1h2 g8g7 f1h3 b8d7 e3f5 g7e7 e6e7 f8d8
info depth 14 nodes 931978 score cp 290 time 1390 pv b5e2 a6b8 f1e3 f5g6 e2f1 f8f7 f1h3 b8d7 e3f5 f7e7 e6e7 g8d8 e7g7 g6e8
info depth 15 nodes 1114437 score cp 297 time 1640 pv b5e2 a6b8 f1e3 f5g6 e2f1 f8f7 f1h3 b8d7 e3f5 g8f8 e7f7 f8f7 e6e8 f7f8 e8e7 f8d8
info depth 16 nodes 1520049 score cp 294 time 2165 pv b5e2 a6b8 f1e3 f5g6 e7c7 h5h4 g3g4 f8f7 c7f7 g6f7 g1h2 b8c6 e3f5 c6d8 f5d6 f7g6
info depth 17 nodes 1707478 score cp 294 time 2441 pv b5e2 a6b8 f1e3 f5g6 e7c7 h5h4 g3g4 f8f7 c7f7 g6f7 g1h2 b8c6 e3f5 c6d8 f5h6 f7g6 h6g8
info depth 18 nodes 2329141 score cp 304 time 3270 pv b5e2 a6b8 f1e3 f5g6 e7c7 h5h4 g3g4 f8f7 c7f7 g6f7 g1h2 b8c6 e3f5 c6d8 e6d6 d8b7 d6e6 b7d8
info depth 19 nodes 4505258 score cp 325 time 6044 pv b5e2 f5d5 f1e3 d5g5 e2a6 h5h4 a6d3 g8g7 e7g7 h8g7 g3g4 g7h8 d3e4 a7a5 b4a5 g5a5 e3d5 a5a3 d5f6

```

Figure 51: Search for Position FEN: 5rrk/p1p1R3/n2pRp2/1Q1P1q1p/1P1P4/P5Pp/5P2/5NK1 w - - 1 31

```

position fen 1n1b2rk/5r2/p1p1p3/3pP3/1P1PqNp1/Q5P1/3B1P2/2RR2K1 w - - 3 48
go
info depth 1 nodes 61 score cp 314 time 0 pv f4e6
info depth 2 nodes 203 score cp 281 time 0 pv f4e6 d8b6
info depth 3 nodes 2143 score cp 293 time 4 pv f4e6 g8e8 d1e1
info depth 4 nodes 22474 score cp 189 time 34 pv a3e3 e4f5 d2c3 d8b6
info depth 5 nodes 27948 score cp 204 time 42 pv a3e3 e4f5 e3e2 d8b6 d2e3
info depth 6 nodes 59247 score cp 168 time 89 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8
info depth 7 nodes 66637 score cp 179 time 99 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1
info depth 8 nodes 91379 score cp 150 time 129 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1 d8b6
info depth 9 nodes 111882 score cp 163 time 152 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g7 d1h1 d8b6 h1h5
info depth 10 nodes 230036 score cp 144 time 310 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g7 c1a1 g7g8 d1h1 d8b6
info depth 11 nodes 262633 score cp 166 time 351 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g7 c1a1 g7g8 d1h1 d8b6 h1h6
info depth 12 nodes 310320 score cp 169 time 408 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g7 d1h1 f7b7 e3d2 b7f7 f4h5 g7g8 h5f4
info depth 13 nodes 375570 score cp 171 time 485 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g7 d1h1 f7b7 f4d3 d8e7 d3c5 e7c5 b4c5
info depth 14 nodes 467747 score cp 171 time 584 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g7 d1h1 f7b7 f4d3 d8e7 e3d2 e8d8 d3c5 e7c5
info depth 15 nodes 689134 score cp 164 time 837 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1 f7h7 h1h5 h7h5 f4h5 g8f7 h5f4 d8e7 f4d3
info depth 16 nodes 887648 score cp 163 time 1067 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1 f7h7 h1h7 g8h7 f2f3 g4f3 g2f3 h7g8 f2g2 a6a5 e3d2
info depth 17 nodes 2692556 score cp 160 time 2940 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1 f7h7 h1h7 g8h7 e3d2 h7g8 f4d3 d8b6 d3c5 b6c5 d4c5
info depth 18 nodes 5191235 score cp 169 time 5635 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1 f7h7 h1h7 g8h7 c1h1 h7g7 f4d3 b8d7 h1a1 d7b8 d3c5 d8b6 a1d1
info depth 19 nodes 5821302 score cp 171 time 6342 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1 f7h7 f4d3 h7h1 c1h1 d8e7 h1h6 g8f7 d3c5 e7c5 d4c5 f7g7 e3f4
info depth 20 nodes 6428340 score cp 168 time 7040 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1 f7h7 h1h7 g8h7 c1h1 h7g7 f4d3 b8d7 h1a1 d7b8 d3c5 d8b6 a1b1 g7g6 b1d1
info depth 21 nodes 7150577 score cp 180 time 7838 pv a3e3 e4e3 d2e3 g8e8 g1g2 h8g8 d1h1 f7h7 h1h7 g8h7 c1h1 h7g7 f4d3 b8d7 h1h6 a6a5 b4a5 d8a5 h6h4 a5c3 h4g4 g7f7 e3h6

```

Figure 52: Search for Position FEN: 1n1b2rk/5r2/p1p1p3/3pP3/1P1PqNp1/Q5P1/3B1P2/2RR2K1 w - - 3 48

```

if (depth > reductionLimit) && i > fullDepthMoves && (moveList[i].Score < moveOrderOffset) && isSafe {
    reduction = int(math.Sqrt(float64(depth-1)) + math.Sqrt(float64(i-1)))
    // Idea from CounterGo, incrementally reduces the reduction

    // If the node is a PV node, then reduce the reduction
    if pvNode {
        reduction -= 2
    }
    // // If the move is a killer move, then reduce the reduction
    if moveList[i].Score >= moveOrderOffset-10 {
        reduction -= 1
    }
    // If the move is threatening, then reduce the reduction
    if isChecked || isOpponentChecked {
        reduction -= 1
    }
    reduction = Max(0, Min(reduction+extension, depth-reductionLimit))
}

```

Figure 53: How reductions are defined and increased

```

// Tapered evaluation implementation from TSCP.
// Caculate scores and phases
var scoreMG int = mg[board.SideToMove] - mg[1-board.SideToMove]
var scoreEG int = eg[board.SideToMove] - eg[1-board.SideToMove]
var phaseMG int = gamephase
if phaseMG > 24 {
    phaseMG = 24
}
var phaseEG int = 24 - phaseMG

// Save Tapered Piece Values
PieceValuesTapered[0] = ((PieceValuesMG[0] * phaseMG) + (PieceValuesEG[0] * phaseEG)) / 24

// adjust king attacking pieces depending on game phase
KingAttackingPieces[0] = (KingAttackingPieces[0] * phaseMG) / 24
KingAttackingPieces[1] = (KingAttackingPieces[1] * phaseMG) / 24

// return tapered evaluation
return ((scoreMG * phaseMG) + (scoreEG * phaseEG)) / 24

```

Figure 54: Tapered Evaluation as implemented in TSCP[67]