Vrije Universiteit Amsterdam

Bachelor Thesis

Exploring modern chess engine architectures

Author: Pieter Bijl (2647306)
Author: Anh Phi Tiet (2638026)

1st supervisor: Prof. Dr. W.J. Fokkink
2nd reader: Prof. Dr. ir. H.E. Bal

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 5, 2021

CONTENTS

***Abstract*—The game of chess is over 1000 years old and is still not optimally played in the 21st century. This thesis aims to explore the technical history of chess computing over the last 40 years. Various techniques that have been employed throughout that time are implemented and tested to determine their performance differences. Similarly, classical evaluation functions that attempt to score a position based on handcrafted, human-like chess features are broken down and their parameters tuned. In conclusion, implementations of both array centric board representations and bitboards can result in equally performant move generators. However, for the purpose of a fully searching and evaluating engine, bitboards may be preferred. Furthermore, short-term sequential tuning methods will lead to a significant improvement in playing strength when dealing with low-ply depth engines.**

I. Introduction

Throughout the history of chess, players and avid enjoyers have always tried to improve their play. Heaps of books filled with human crafted chess-theory undoubtedly captivated the minds of the players. In 1997 many marked the start of a new era when reigning champion Garry Kasparov lost a six-game match against Deep Blue, a chess-playing computer developed by IBM [1]. Nowadays, the capabilities of a modern engine far exceed even the brightest of minds in chess. Chess theory, therefore, has adapted accordingly. Chess engines help shape the game of chess. The task of building such a program is monumental in the least. Luckily, the techniques which constitute a good engine are well documented by those who paved the way.

The first goal of this thesis is to describe the techniques and algorithms that top chess engines, such as Stockfish, Fat Fritz and Leela Chess Zero currently employ. In terms of chess board representation and move generation, various approaches exist and are described in detail in the following section. In addition, many of the search techniques and pruning methods are listed too, along with the intuitions behind why they are employed and how they work. Finally, various position evaluating algorithms exist. To this end, the focus is on a set of classical evaluation functions. This way of evaluating a chess position was the dominant evaluation style employed by chess engines up until 2016.

The techniques explained in *II. Basics of chess engine architecture* are all implemented into a working engine, which leads to the secondary goal of this thesis. Namely, to take some of these techniques and to compare and contrast their performance, both in terms of speed and playing quality. This testing will be done in an alpha-beta, quiescence search framework. First, the different types of board representations can all be implemented separately and performance tested with the same search and evaluation functions. This would lead to some interesting insights into whether it is worth investing time into optimizing these parts of the chess engine, and to see which approaches work well, and which do not. Moreover, if the move generation is the critical speed limiting factor in the engine, what type of board representation is best suited? To this end, this thesis will investigate four of the board types discussed and test them both in perft, where the move generator is often the limiting factor, and in a regular alpha beta style search. The first research question here is as follows: Which approach to board representation is the fastest for move generation, which suits a full general purpose chess engine, and in which subroutines of the engine is the choice of board representation most critical? We hypothesize that magic and PEXT bitboards will beat the others in both scenarios, where PEXT bitboards perform slightly better. In terms of where the choice between any type of board is the most critical, it is probably equal in most areas of the engine, except in the move generation itself. The second research question regards parameter tuning: Will 10 iterations of sequential parameter tuning make a significant difference in playing strength when it comes to engines playing at a relatively low depth? Though the evaluation parameters are a small part in the engine as a whole, slight fluctuations mean a lot in terms of how the engine will play. Especially at a low depth, where the evaluation heuristic is crucial for the engine to see further into the future, the parameters must play a significant role. A set of 6 differing parameters were chosen, to reflect the complexities in evaluating different aspects of various chess positions. Additionally the tuning happened over games played from 10 different starting positions, where each version played 1 game on either side.

Motivation and work distribution

After having examined some of the various techniques used in modern-day engines, we decided a C++ framework would fit the project the best. With wings of excitement, the planning phase commenced, where we came to a general plan of action and distribution of the tasks at hand. The entire framework would be accessed and updated through the use of Git. Almost every work session would be done whilst communicating through Discord, to enable quick exchanging of ideas and help wherever needed. While Pieter worked on the board representation and move generation, Anh Phi worked on the Evaluation. When the two parts were finalized, work was then done by both to bring them together in the search, which utilized both parts. After finishing the engine up to a certain degree, we looked at the notes we had made and the insights we had gained whilst creating it. With our minds still brimming with knowledge, we quickly started work on this thesis. Still, Pieter mainly worked on the board representation, while Anh Phi mainly stuck to Evaluation. Relying on each-other's input every now and then. After having finished both, we wrote the section explaining the different search techniques. All in all, the workload was distributed equally. Although we worked on different pieces, both of us know equally as much about the full content of this thesis.

II. Basics of chess engine architecture

A. Board representation

At the core of any chess engine is the internal state describing a given chess position. Every algorithm, whether related to searching or evaluating, is built on top of this. As such, the design considerations made in the process of

implementing the chess rules and the internal state dictate the design of every other algorithm [2].

Two-dimensional arrays

Perhaps the most intuitive way to start is to use a two dimensional 8 by 8 array of pieces. At first sight, this allows for easy off the board detection. Since the array is indexed by a rank and file combination, it can simply be verified that these fall in the range $0 \leq x < 8$. Unfortunately, when considering move generation, some performance issues arise immediately [2]. Every time the engine has to index into a 2 dimensional array, the compiler, under the hood, would have have to calculate $8 \cdot rank + file$ to get the appropriate memory location. Though a good compiler can change the multiplication by 8 into a left shift by 3, this would still take multiple instructions. Additionally, since both the upper and lower bounds of the rank and the file have to be checked, four compare instructions are needed in order to prevent memory issues. These issues are red flags to an efficient programmer. It is clear that some improvements are needed.

Single dimensional arrays

The easiest way to circumvent the issue of having to calculate the memory location of each square by rank-file combination, is to have a single, one dimensional array that is indexed by square. This transformation serves multiple purposes which may not be apparent at first. For starters, it makes indexing faster, but it also allows the engine to simply check whether the indexed square falls in the range of $0 \leq x < 64$, rather than having to do two range checks. When it comes to move generation, this approach gains efficiency too. Consider a bishop moving north-east from a1 on a two dimensional board. Here, simply increment both the bishop's file and its rank by 1. This results in `board[0][0]`, `board[1][1]`, `board[2][2]`, and so on. While elegant, it does require two addition operations. Compare this to the single dimensional solution. Here, instead 9 is added to the bishop's index into the board array. This results in `board[0]`, `board[9]`, `board[18]`, and so on. This again saves the multiplication and addition lookup, but also saves an addition when sliding over the board. Note that there could be some ambiguity as to which index maps to which square on an actual chess board. Throughout this thesis, a row major rank-file mapping will be assumed where relevant. This means that square a1 = 0, b1 = 1, ..., h1 = 7, a8 = 56, h8 = 63, as shown in figure 29 in the appendix.

The attentive reader will notice that through the elimination of the file bounds check, there is now a potential wrap around the board for pieces. Consider a bishop at index 38 (g5) moving north east. This would result in 47 (h6), and then 56 (a8). In terms of implementing a chess engine that uses this board model, it would be a nightmare to detect these wrap around issues. The following consideration is an attempt to solve this issue.

Padded single dimensional arrays

Consider an original chessboard enclosed in a larger array with sentinel values around the edges. Now, before moving a piece onto the board at index i, first check board[i] to determine whether it stores a sentinel value. If so, then the piece must have moved off the board. The traditional way of doing this is by using a board of 12 rows and 10 columns, with the "actual" board centered in the middle (a1 = 21, h1 = 28, a8 = 91, h8 = 98) [3]. The complete array indices and the location of the board are shown in figure 1. Two free rows are padded at both the top and the bottom because knights could move in vertical steps of two. Only one padded column on each side is needed because a knight that wraps around one side of the board would land on the padded column on the other side. At this point, this board design has eliminated expensive indexing lookup and reduced bounds checking to a single sentinel value comparison. The question is, is there an even quicker way?

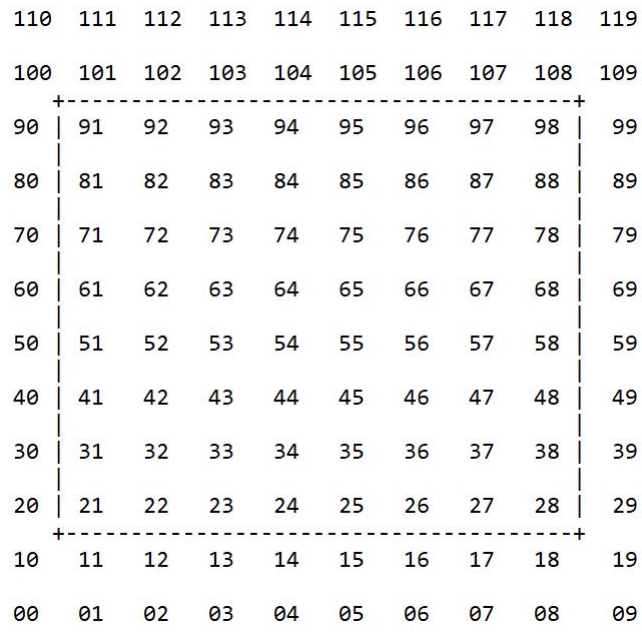

Fig. 1. 10x12 board layout

0x88 board representation

Unsurprisingly, the answer is yes. On older machines, often without caches, having to access the memory in order to check whether a square is invalid is expensive. The 0x88 board representation relies instead on checking whether the square index is out of bounds [4]. Imagine again a single dimensional array whose indices are of byte width. Now let the upper (most significant) nibble (4 bits) address the rank of a square in the array, and the lower nibble address the file of a square. If laid out this way, this leads to a single dimensional array representing a 16 by 16 board. However, a chessboard only needs 8x8 squares, which can be addressed by just 3 bits. Now, envision the board laid out in such a way that the 3 least significant bits of each nibble address

the squares of the actual board. Every invalid square on the larger board now has a 1 in the most significant bit of one or both of the nibbles. Should it be necessary to access a square, first determine whether it is invalid by checking if one or both of the nibbles has its most significant bit set. This can be done by verifying if the bitwise *and* of the index and the binary number 10001000 results in a non-zero value. In hexadecimal, this binary number is 0x88. As another optimization, technically only the lower half of the 16 by 16 board has to be allocated, since the upper half has its most significant bit index always set, which means that our algorithm should catch those indices out as invalid before attempting an invalid memory access. This reduces the board size to just 128 pieces, only 8 worse than 10x12.

70	71	72	73	74	75	76	77	78	79	7a	7b	7c	7d	7e	7f
60	61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e	6f
50	51	52	53	54	55	56	57	58	59	5a	5b	5c	5d	5e	5f
40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f
30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f
20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f

Fig. 2. 0x88 board layout

Square centric vs. piece centric board representations

All of the board representations above are square centric. This refers to the fact that every square can instantly be accessed and thus verified whether it is an empty square, an invalid square, or if there is some piece. Many simplistic implementations just use a byte to represent what is on each square, where every black and white piece has a unique numeric value, along with constants for empty and invalid squares [5]. In terms of move generation, this makes it incredibly easy to check whether a move to a certain square is valid. However, there is one detrimental shortcoming. Say, for instance, the engine wants to generate moves for every white bishop. Using the currently proposed padded arrays, it would be necessary to loop through every square on the board to find every white bishop, and only then is it possible to generate its moves. Consequently, this would take 128 loop iterations, which would be fatal for the efficiency of the engine. Here is where piece centric board representations come in. Rather than just keeping a list of squares with pieces on them, the engine should also keep a list of pieces that store some information about which square they are on. This way, the effort of having to find the pieces before their moves can be generated is saved. Keep in mind that using only a list of pieces without a list of squares results in having to loop through every piece to check whether a square is occupied. It is for this reason that many modern chess engines store both a piece centric and a square centric representation of the board.

It is worth noting that by using both representations the engine is effectively storing information twice. While a piece's location can be determined by simply checking the list of squares, the location is still stored along with the piece for efficiency reasons. The space-time tradeoff is a recurring theme throughout problems encountered by chess engines. More often than not, it seems that the results are in favor of needing less time, albeit increasing the memory footprint.

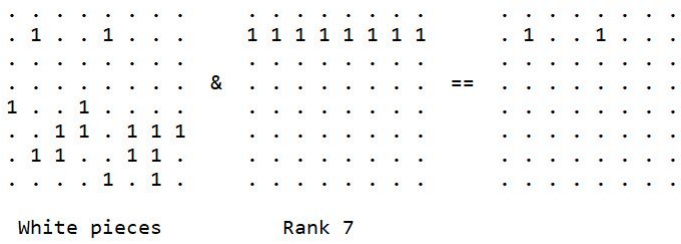

Fig. 3. Bitwise *and* operation on bitboards

Bitboards

One particular piece centric concept in board representation is the concept of bitboards, which dates back outside of chess to 1952 [6]. It was first applied to chess in 1970 [7]. The underlying idea is very simple: a chessboard has 64 squares. If every square was to be represented with one bit denoting whether there was a piece on that square or not, exactly 64 bits are needed. Given that nearly all modern processors use 64 bits for their instructions, this bitmap technique lends itself extremely well to very efficient chess engines. In essence, bitboards are nearly identical in use to arrays, except that bitwise operations can be used to compute information in just one CPU instruction. Consider a scenario where the engine is trying to determine if white has any pieces on the 7th rank of the board. In the array approach, it would either loop through the 7th rank of the board and see if there are any white pieces, or it would loop through every piece and see if there is one that is both white and on the 7th rank. For bitboards, however, it simply takes a bitboard representing white's pieces and a bitboard representing the 7th rank, and performs a bitwise *and* on them. A non-zero value would mean that white has pieces on the 7th rank, as shown in figure 3.

Note that for this to work effectively, a variety of bitboards have to be available from memory, or preferably cache. An engine relying on bitboards typically keeps a bitboard for each piece-color combination. A bitboard representing every piece on the board is in this case the logical sum (bitwise or) of all of the separate bitboards. Bitboards like every piece on the board or every piece of a certain color are used so often, that those too are stored in memory for quick access.

Throughout this thesis, where relevant, it will be assumed that one bitboard is stored in one 64 bit word. Bits in bitboards map to squares on a chessboard in little endian rank file manner. This means that a1 is the least significant bit, h8 the most significant bit, the first rank of the board is the least significant byte and the 8th rank is the most significant byte. This is useful, because if the board has to be mirrored

vertically, a quick byte swap (endianness swap) is done to accomplish this.

B. Move generation

A legal move generator is an essential piece to the workings of an engine, everything relies upon it being fast and correct. Out of all the possible moves, only a fraction are pseudo legal. Pseudolegal moves are moves that follow the rules of how each piece moves, but do not take into account whether the king is left in check after the move is made. The subset of pseudolegal moves that does not leave the king in check is called legal.

Pseudolegal move generation

Move generation for bitboards is fairly similar to move generation on arrays. Rather than moving a piece across the squares in an array, the engine simply shifts a bitboard representing the piece in the direction it's going. Bit shifting a bitboard vertically is simple, a board can be shifted up by doing a bitwise shift left of 8. Similarly on an array, you would add 8 to the current square index. Note that the 8th rank is shifted off the board and a fresh, empty rank is added at the bottom. When working with bitwise shifts, this behaviour is automatically added; bits that are shifted outside the range of the register are removed, and fresh zeroes are added on the other side. Horizontally shifting is slightly more complicated. Shifting a bitboard one step to the right can be done naively by left shifting the board by 1. Note, however, that bits on the H file will wrap around onto the A file one rank up. Since shifting a bitboard one to the right is supposed to leave fresh zeroes on the leftmost file, this issue is solved by subtracting the A file from the resulting shift (bitwise *and not* file A). Diagonal shifts and knightmove shifts can be implemented alike.

```
. . . . . .  .     1 1 1 1 1 1 1 1     . . . . . .  .
. . . . . .  .     1 1 1 1 1 1 1 1     . . . . . .  .
. . . . . .  .     1 1 1 1 1 1 1 1     . . . . . .  .
. . . . . .  . &   1 1 1 1 1 1 1 1  ==  . . . . . .  .
. . . . . .  .     1 1 . . . 1 1 1     . . . . . .  .
. . . . . .  .     . 1 . 1 . . 1 .     . . . . . .  .
. . . . . 1 1 1    1 . . . 1 . . 1     . . . . . . . 1
. . . . . 1 . 1    . 1 1 1 . 1 . 1     . . . . . 1 . 1

Pregenerated       bitwise negation    Pseudolegal
king moves on g1   of white's pieces   king moves in this
                                       position
```

Fig. 4. Removing one's own pieces from pseudolegal movesets

Knight and king moves

Pseudolegal knight and king moves are the easiest to compute because their pseudolegal move sets are independent of the state of the current board. A king on h8 will always be able to move to g8, g7 and h7. One objection you could make is that this is not fully true. A white king will not be able to move onto a square where another white piece is, and the same goes for a black king onto a black piece. However, if the possible destination squares of the king are stored in a bitboard, then, when taking into account all of the king's color's pieces, there could potentially be 2^8 different move sets just depending on which of the destination squares are occupied by friendly pieces. Instead, it makes much more sense to assume that any piece of any color can be captured, and then to remove every destination square with a friendly piece during legal move generation. This can very easily be done with bitboards and bitwise operations, as denoted in figure 4. There are multiple advantages to this approach. The first of which is that it is actually quite useful both in move generation and evaluation to assume as though one's own pieces can be captured. It allows to quickly determine whether pieces are defended and thus whether it makes sense to capture them as the other color. Secondly, since the knight and king move bitboards are now truly independent of the color or state of the board, there are only 64 different bitboards describing a king's (or knight's) moves. Namely, one for each different square. This is incredibly useful, as it allows the engine to pre-generate these bitboards and store them in an array accessed by square. Rather than needing to compute 8 shifts in each of the different move directions during the move generation procedure, every single move set is pre-generated while initializing the engine. As a result, the engine now only has to do a quick memory (cache) lookup during the move generation routine.

Sliding piece moves

Continuing this train of thought, it would be nice if the same were possible for pieces like rooks, bishops and queens. As it turns out, it is somewhat possible. First, consider the prior observation that a king's or knight's pseudo-legal moveset is independent of the state of the board, they are only dependent on the square the piece is on. This does not hold true for sliding pieces. Queens, for instance, can only move up until the next piece on the ray she is sliding along. If queens were only able to move upwards, a queen on a1 would have 7 distinct possible move sets, depending on where pieces are located on the A file. Each additional ray a piece can slide along is linearly proportional to the number of distinct movesets. An actual queen on d4 has $3^5 \cdot 4^3 = 15552$ different possible movesets. Precomputing and storing the different movesets for each square would require a lot of memory. Not only that, a fast algorithm that takes in the location of a piece and the locations of the other pieces on the board is needed to produce the correct move set.

So how do modern chess engines tackle this problem? The first thing to note is that it is not necessary to store or compute queen moves. The set of pseudolegal queen moves is just the union of the sets of pseudolegal rook and bishop moves generated from the same square with the same board state. This leaves just the rook and bishop moves to be generated. The simplest approach would be to generate these at runtime, by just sliding the piece along each of the attacking rays until another piece is reached. This is not all that bad, but can it be better?

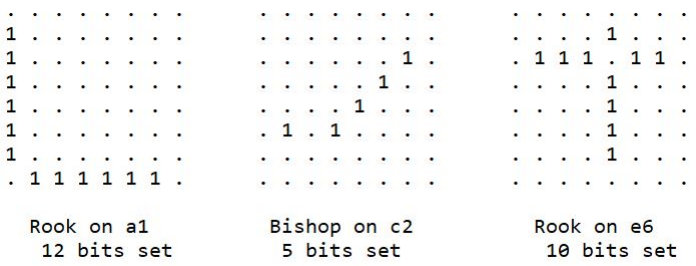

Fig. 5. Example occupancy masks

```
Bitboard mask = 0xffff;
Bitboard subset = 0;

do {
    // ...
    subset = (subset - mask) & mask;
} while (subset);
```

Fig. 6. Carry rippler subset enumeration of a bitboard pseudocode by M. Van Kervinck [8]

Magic bitboards

When inspecting the location of the pieces on the board in regard to sliding piece attacks, it does not matter what type of piece is on a square, nor does it matter which color the piece is. Recall from above, pieces from the piece's own color are assumed to be possible targets. As such, a bitboard representing every piece on the board can be used while generating moves. Furthermore, only the bits representing pieces that are on the attacking rays of the sliding piece are relevant. When generating moves for a rook on a1, whichever (if any) piece is on g6 is not relevant, because it never influences the results. Also note that the contents of a square at the end of a ray do not influence the resulting moveset. If in the case of the rook on a1 there was a piece on a8, then it could be captured. If there was no piece on a8, then the rook could move to a8. The occupancy of a square only affects the squares that come after it on the attacking ray. Consequently, squares that lie at the end of the ray are irrelevant. For a rook on a1, this leaves the squares between and including a2-a7 and b1-g1 as the only squares that are relevant. If these squares are represented as a bitboard mask, the bitwise *and* of this mask and the bitboard representing every piece on the board can be used to determine the relevant occupancy bits. A few example such masks are shown in figure 5. The relevant occupancy bits represent in essence the pieces on the board that influence a sliding piece's move set.

On the one hand, it is trivial to create such a mask for both rooks and bishops just by generating their moves on an empty board, and to remove the last square of the ray. On the other hand, it is quite difficult to see how this mask might be used to generate movesets. Upon further inspection, the number of bit subsets in the mask shows how many different combinations of relevant occupancies a piece square combination can have. In case of the rook on a1, there are 2^{12} unique subsets of relevant occupancy bits. This is because 12 bits are set in the occupancy mask. Theoretically, every one of those subsets can be generated by using the carry-rippler trick, shown in figure 6. Then, the correct rook moves for each of those subsets can be generated and stored somewhere in memory. The only thing that is missing then is to be able to go from a certain relevant occupancy subset to the correct moveset.

There are several ways to do this and this very topic has been highly discussed within the chess programming community for a decade or two now. Nowadays most competitive chess engines rely on the magic bitboard technique, which was first introduced in 2007 [9]. While there are various approaches, most magic implementations assume that an array of length 2 to the power of the number of bits in a square's occupancy mask bitboards is needed as hash table to properly implement a fast hashing algorithm that maps a set of occupancy bits to the correct moveset which is stored in this array. Now, while generating every possible subset of the occupancy mask, a magic number is found that, for each subset, when multiplied with this magic number, results in a unique index into the database of movesets. At runtime, when generating the moves for a sliding piece on square x, the occupancy mask for that piece on that square is bitwise *and*-ed with all of the pieces to obtain the relevant occupancy bits. This bitboard is then interpreted as a regular unsigned 64 bit integer and multiplied with the magic number for that square. The result of this multiplication is another unsigned 64 bit integer. In order to use this as an index into the hash table of movesets, only the upper n bits are used, where n is the number of set bits in the occupancy mask. This can easily be done by bitwise right shifting the resulting value by $64 - n$. This process is layed out in figure 7.

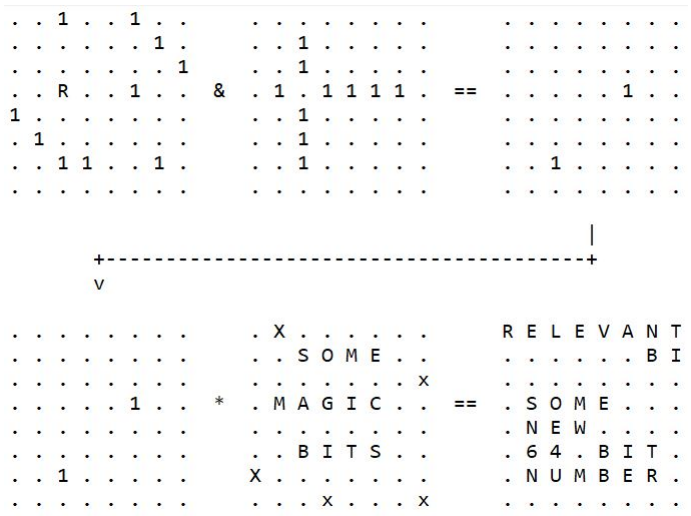

Fig. 7. Hashing algorithm used in magic bitboards

If this works, it is essentially a tricky hashing algorithm that

hashes a key, the relevant occupancy bits, into an index for the hash table. There is no guarantee that such a magic number exists for every single sliding piece square combination. As it turns out, however, these magics can be generated by using a trial and error approach within a reasonable time. Many implementations generate a random magic number with a sparse number of bits set in them. Then, every subset of occupancy bits is enumerated through and its correct moveset placed in the hash table by using this magic in the hashing algorithm. If there are collisions, the process starts over.

At this point, magic bitboards provide a constant time algorithm to generate sliding piece attacks by making use of a pregenerated hash table. While it seems difficult to do better than this, this approach is still far from a theoretical optimum. The time complexity is very low, but the space needed for such a hash table amounts to 102400 bitboards for rooks and 5248 bitboards for bishops. Since the size of any bitboard is 8 bytes, this amounts to roughly 860 KiB of memory. This is definitely manageable on modern computers, but it would be nice to shrink this table without losing any speed. This would considerably improve CPU caching.

Consider again a rook on a1, for instance in the chess starting position. Since square b1 is blocked by a knight, there is no way the rook could move to c1 or beyond. However, since there are pieces along the first rank, the bitboard representing the entire board has all of these bits set. The problem here is that, for a rook on a1, if there is a piece on b1, all of the occupancy bits along the c1-h1 squares are irrelevant; whether they are set or not, the final moveset for the rook does not change. Despite this problem the possible occupancy subsets of the squares c1-h1 are still each considered separately, resulting in the hash table containing many duplicate copies. Each ray of length n squares has 2^n occupancy subsets, but only n unique attack sets. In a perfect world, there would be some sort of very fast hashing algorithm that takes these different occupancy subsets but maps those to the same attack set. In this way, the speed of the original algorithm is retained but the space footprint is significantly reduced. One approach to achieve this is to reduce the size of the hash table and to try to find magic numbers that still satisfy the conditions. While some magic numbers have been found that reduce the size of the corresponding table, it is not clear whether these "perfect magics" exist.

PEXT Bitboards

If all that is cared about is time complexity and speed, the hashing algorithm used in magic bitboards was optimal until around 2013. In 2013, Intel released its haswell CPU architecture along with an x86_64 expansion of bit manipulation instructions. In particular, the BMI2 set comes with the PDEP and PEXT instructions. Parallel bit deposit and extraction are both instructions that suit bitboards perfectly. Both 64 bit variants of the instructions take a mask and an input. In the case of PDEP, the output is then the least significant bits of the input moved to the bit positions as dictated by the set bits in the mask. The PEXT instruction does essentially the opposite, by taking the bits in the positions of the set bits in the mask, and "extracting" them to the least significant bits in the output, like is shown with the bitboards from figure 7 in figure 8. In terms of bitboards, this means that given the bitboard representing the entire board and the relevant occupancy mask for a certain square-piece combination, the PEXT instruction transforms it directly into an index into the hash table. Essentially, PEXT bitboards are magic bitboards that replace the hashing algorithm by one instruction that completes in just one CPU cycle. Furthermore, it eliminates the need to pre-generate magic numbers, as PEXT does not rely on magic.

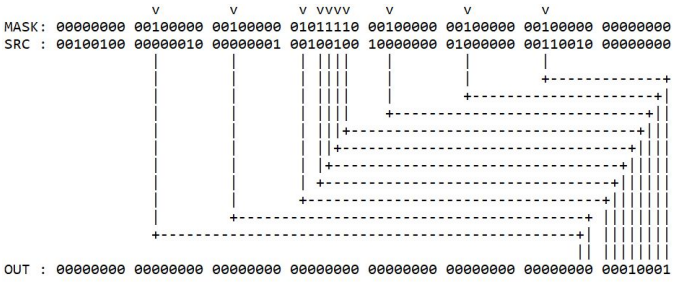

Fig. 8. PEXT instruction workings layed out graphically

Legality and the position class

While pseudolegal moves are largely independent of the actual piece types of the occupied squares, legal moves can not leave their own king in check. As such, more sophisticated information than just a bitboard representing every piece on the board is needed to generate legal moves. At this point, the move generator can generate bitboards representing attack sets for each piece in a location, but there is no structure or object that holds all of the information needed to execute the move generator. This is where most chess engines have one large object representing a position. This position holds all of the necessary information to be able to generate moves, make a move, search the moves, and to statically evaluate it. In order to accomplish the first, some sort of data structure is needed to hold all of the relevant bitboards. Many engines decide to store bitboards for each piece, but also for each color. Additionally, bitboards can also be used to determine castling masks, both to check if there are pieces blocking the castling path and to check if the castling path is attacked by an enemy piece. Naturally, the position also stores whose turn it is, a possible en passant square, the number of half moves (plies) played, and various other useful statistics. If the moves are generated for a position, a bitboard representing all of the pieces of a certain color-type combination can be looped through by clearing the least significant bit of the bitboard one-by-one. The PEXT pseudolegal move generator then looks up the relevant bitboard attack set, and it is serialized in a similar way to enumerate all of the moves. Any move can be represented by a short 16 bit unsigned integer. 12 total bits are needed for the source and

destination squares, 2 bits to indicate whether the move is a special move (normal, castling, en passant, promotion), and 2 bits to indicate the piece that is promoted to (knight, bishop, rook, queen).

Then, in order to actually play a move on the board, several steps have to be taken to modify the position object and leave it in the correct state. This includes, but is not limited to, updating the bitboards and other internal variables to move / remove any pieces, updating the Zobrist hash key of the position [10], determining possible en passant squares, updating castling rights, and so on. Zobrist hashes are used as keys in many hash tables inside a chess engine, and are essentially the xor sum of all the pieces on the board. Moreover, since multiple moves are often checked for one position, it would be preferable to be able to both play and undo moves rather than creating an entirely new position when a move is made. This approach fits bitboards nicely, as pieces can simply be "xor-ed" in and out very quickly. Finally, due to the xor nature of Zobrist hashes [10], a new position's hash can easily be determined by xoring the moved pieces with the previous position's hash.

To facilitate legal move generation, a few bits of extra information are essential to also be generated. First of all, it is necessary to determine whether the side to move is in check, because this significantly limits the number of available moves. The most powerful approach here is to consider this side's king only, and then to iteratively check if there is an enemy pawn checking it by considering the squares that would be attacked by a pawn of the king's color at the king's position, then to see if there is an enemy knight on any of the squares that would be attacked if there was a knight at the king's position, then to check if there is a bishop or queen on any of the squares that would be attacked if there was a bishop at the king's position, then similarly for rooks and queens. The results are logically added together to form one bitboard of checking pieces. This is useful, because if the resulting bitboard has more than one bit set, it must be a double check, which means the only legal moves are king moves. This saves quite some move generation cycles by not considering any regular pieces. If there is only one checker, consider the squares between the checker and the king. The only legal moves are to move the king to a non enemy controlled square, to move a piece on any of the squares pointed out above, or to capture the checker. Here it is useful to have these sets of squares between any two squares pre-generated. For squares that have no obvious horizontal, vertical or diagonal line between them, such as two squares a knight's move apart, these bitboards can simply be empty. Also, all of the squares controlled by the enemy can be generated as the bitwise *or* of all of the pseudolegal movesets of the enemy's pieces. When this is done, it is important to leave out the player's own king from the occupancy set. Any move can not leave the moving piece's color's king in check. One way to prevent moves that do this is to consider the new position after the move is made and to check if the king is in check. This approach is often taken to verify whether an en passant capture is legal, as it can be a real struggle to determine whether en passant is legal without making the move. In terms of more standard moves, it is possible to be a bit more clever. A piece that could leave the king in check after moving is referred to as being "pinned" to the king. Pinned pieces can only move along the ray they are pinned on. If there are no checking enemy pieces, when not considering any special moves, every non pinned piece's pseudolegal moveset is equal to their legal moveset. Similarly, the legal moveset of a pinned piece is obtained by using a logical and on the ray along which it is pinned and its pseudolegal move set. Special moves, such as castling, en passant and promotions, are often added separately with their own legality verifier. This is because they follow non-standard move rules.

Perft

In order to verify the correctness and performance of a chess legal move generator, a routine referred to as "perft" is used. Perft is an abbreviation for performance test and it recursively counts every leaf node in the legal move tree upto a certain depth in any position. It is used for various turn based games, among which are chess and checkers [11]. The results a move generator returns for a certain depth in a certain position can be compared to the results that are generally accepted in the computer chess community, as listed on the following page [12]. While to our knowledge these results have never been formally proven, all of the well-established chess engines return these same results. Given that they are not known for ever suggesting illegal moves, we can be reasonably sure they are correct. It would be interesting to see some method to formally prove the correctness of a move generator given the rules of chess. A C++ style pseudocode for perft is shown in figure 9. The perft routine is perhaps the simplest search function, since it recurses over the entire game tree. While it is 100% accurate when the move generator is 100% accurate, it is unfortunately far too slow to work as a strong chess engine. From the starting chess position, just 9 plies deep, there are already over 2.4 trillion leaf nodes in the game tree. Generating and evaluating all of these positions would take too much time for use in a generic min max algorithm. This is why chess engines put a large emphasis on making a search function that prunes a lot of the search tree while still retaining a good evaluation. Pruning a move with a large search depth still remaining significantly reduces the search load.

C. Search

The most standard optimization is that of alpha beta pruning. Alpha beta, for short, relies on two prerequisites. First, the maximizing (or minimizing) player must have a previous move already searched. Second, during the search of the next move, a line is found where the now minimizing (or maximizing) player has a better score in their favor when compared to the previously searched move. If both of these conditions are true, then that means the first player would never play the move that is currently being searched, as it allows the other player

```
Nodes Position::perft(Depth depth) {
    if (depth == 0)
        return 1;

    Nodes nodes = 0;
    Moves moves = generate_legal();
    for (Move m : moves) {
        play_move(m);
        nodes += perft(depth - 1);
        unplay_move(m);
    }

    return nodes;
}
```

Fig. 9. Perft pseudocode

to force a line that is worse than the best line in the previous move. As such, there is no need to search the rest of the moves that follow after this move. By this logic, when the strongest move is searched first, when the algorithm proceeds to search the next, worse move, it is likely to find a line that beats the initial best move, and it allows the rest of the moves to be pruned.

This is where the bridge between move generation and search is formed: move ordering [13]. In this case, instead of going through every generated move in random order, moves are first ordered in such a way that the likely strongest moves are searched first. Moreover, if the move generation is staged, and a tree cutoff occurs after a move while the rest of the moves have not been generated yet, it saves the effort of having to generate those moves. Without any prior knowledge, moves can be ordered as follows. First come the good captures, that is to say, any capture that leads to a net positive material gain, or that leads to a sequence of captures that is advantageous to the respective side. For instance, pawn captures queen is very strong, whereas queen captures pawn, and then pawn captures queen is very weak. The higher the net gain, the better the static evaluation of the move. Pawn captures bishop is good, but pawn captures rook is better. A routine often referred to as static exchange evaluation is used to rank every capture from strongest to weakest. Moves can then be ordered with the strongest captures first, up until the captures that are equal in material exchange. After this come all of the quiet moves and finally captures that lose material.

Thankfully, there is often some prior knowledge about positions before they are searched. Ideally, the game tree is searched in a breadth first search manner. This would lead to the depth gradually increasing for a stronger and stronger evaluation as the search continues. However, alpha beta search inherently works in a depth first manner. This is one of the reasons why chess engines and other two player turn based game playing AIs choose to use an iterative deepening framework with alpha beta search. This entails first searching a position from the start to depth 1, then from the start to depth 2, then from the start to depth 3, and so on, all in a depth first manner. At first glance, this looks very inefficient, as if the work done by the previous iterations is redone every time by the current iteration. While this is true to some extent, a clever engine uses information discovered in previous iterations in the current iteration. Firstly, by searching the best move discovered by the previous iteration first, alpha beta will likely have many cutoffs right off the bat. This can be done by storing some information for every searched position in a large hash table, with the position's Zobrist hash as its key [10]. When a new search is started, this table, which is often referred to as the transposition table [14], is consulted first to see if there is any information already available. If the information in the transposition table has a higher search depth than the current search, the stored results can be used directly. If the retrieved information has a lower search depth than the current search, the stored results can be used to achieve more cutoffs. Additionally, when the same position appears in different search lines, the global transposition table allows the position to only be searched once. This is essential when the search is multithreaded.

Moreover, not just capturing moves can be ordered from best to worst, quiet moves can be too. Consider a position where one side is threatening mate, or some other strong move. The move that threatens this is called the killer move. If it goes unanswered, this side will get a large advantage. If a move is detected after it causes a beta cutoff in a sibling node, it might do so in the current node too. It therefore makes sense to give moves like this a bonus that allows them to be sorted to the front in the move ordering. Similarly, there are countermove and history heuristics that work to order strong quiet moves higher in the quiet move ordering list.

Since minimax style algorithms have both a maximizing and a minimizing player, simple implementations often use two functions, or a function with an argument denoting whether to minimize or maximize. Though an implementation like this is often easiest to understand, the two functions can be combined into a single function in what is known as a negamax framework. Negamax relies on the fact from equation 1, and its pseudo code is shown in figure 10. Here, beta can be seen as the upper bound of the score, i.e., the score that the previous player is guaranteed to get, because it has a different line that gets this score. If any search returns a score higher than beta, a cutoff immediately happens. Alpha, on the other hand, is the score the current player is already guaranteed to get. It is the lower bound of the score.

$$\max(a, b) = -\min(-a, -b) \tag{1}$$

Quiescence search

One of a chess engine's biggest weaknesses is the horizon effect. This effect occurs when the search of a position stops at some point in the position where there could be a game changing set of moves right after the stop. Had the position been searched slightly deeper, this would have been picked up

```
Value search(Value alpha, Value beta, Depth depth) {
    if (depth == 0)
        return evaluate();

    for (Move m : moves) {
        play_move(m);
        Value score = -search(-beta, -alpha, depth - 1);
        unplay_move(m);

        if (score >= beta)
            return beta;
        if (score > alpha)
            alpha = score;
    }

    return alpha;
}
```

Fig. 10. Negamax alphabeta pseudocode

on by the engine. A lot of the time, it is easy to detect when a position must be searched further, but this is not always easy to determine. Most importantly, if the search stops right after a queen captures a knight, but does not continue to check whether the queen can be recaptured, the resulting position might be considered +3 for the queen's side. This will likely result in the engine preferring lines to this "free" knight, and it may even blunder material to achieve its goal, only for the queen to be recaptured in the very next move. In order to solve trivial cases such as the one described, a search can only be stopped and evaluated at a "quiet" position. Here, the term quiet refers to the fact that there aren't any key captures left that might drastically change the evaluation. This type of search is referred to as quiescence search and largely counters the horizon effect [15], and it is a much simpler type of search that only looks at good captures. Bad captures are often disregarded as they are very likely to be cut off either way. In case there are no captures to be played, the evaluation of the position is returned as the score of that position. Furthermore, the position is often evaluated before continuing the search as well. If this evaluation is already higher than beta, thus causing a cutoff, it is often safe enough to return this evaluation. This is based on the assumption that the position will not get worse for the player to move and is referred to as standing pat, from the poker term. Thus, if the position is already good enough to be cut off, there is almost certainly a move that will still allow it to be cut off, so it is unnecessary to go through the trouble of generating all of these moves. This null move observation is at the base of many pruning techniques. The only time it fails is when the position is in zugzwang. Luckily, zugzwang almost exclusively occurs in (late) endgames [16], so only at this point are pruning methods that rely on the null move observation disabled.

Search windows

In a regular alpha beta search, the initial values for alpha and beta are nearly always negative and positive infinity. This is because there is no good enough guess as to what the true evaluation at that depth might be. Interestingly, in an iterative deepening framework, there is already some idea of the score of a position at a certain depth, namely that of the previous depth. Consequently, this score can be used to create a window around this score as the starting point for the new search. By using a small window, more cutoffs are to be expected. This is referred to as using alpha beta with an aspiration window, and often leads to a considerable speedup [17]. If the final score falls inside the window, then no problems have occurred and there have only been some additional cutoffs. If the final score does not fall inside the window, that means the true score was cut off, and a search with a bigger window has to be done to resolve the issue, basically discarding the previous work. This is costly, but in the vast majority of positions this turns out to be well worth the risk.

Similarly, when in the middle of a search, it is possible to make some assumptions based on previous iterations of the iterative deepening searches. With proper move ordering, most importantly the best move as found by the previous iteration first, there is a very high chance that the next moves will be cut off. Therefore, a chess engine might consider the best line of moves as the principal variation. That is to say, a set of consecutive moves that is optimal for both players, thus not causing any cut offs. For all other moves it only needs to be shown that they have as lower bound the principal variation move. The exact score is not necessarily needed as long as it is shown that the range of possible scores falls outside the window. Principal variation search is currently the most widely used alpha beta search enhancement used throughout many chess engines after being determined to be the strongest out of various searching algorithms [18]. It uses the aforementioned observations by searching all non PV-nodes with a null window. This means that the distance between alpha and beta is zero, and a search like this will get more cutoffs quicker. If a move does not end up being cut off, it means that it is potentially a new PV-move and a costly research has to be done in order to find its exact score.

In scenarios where moves are not ordered from best to worst, principal variation search is unlikely to be faster than regular alpha beta pruning. This shows that a good move ordering algorithm in conjunction with prior knowledge gained from previous iterations of the iterative deepening framework are essential for such algorithms that rely on smaller windows to improve efficiency.

Pruning

Naturally there are various techniques being applied during the search process that stop nodes from being explored more if they can not, or are very unlikely to influence the search result. Perhaps the simplest form of pruning is mate distance pruning, which is a backward pruning technique. In a position with a forced mate in x moves already discovered, it is an algorithm that aims to cut off all of the branches of the search tree that are longer than x moves. Logically, this does not affect the search result since the game is already a forced win and neither does it improve the engine's playing strength by

a lot for the same reason. However, it helps a lot in puzzles where the shortest mate is often searched for.

Another pruning technique, which is similar to standing pat in quiescence, is the null move heuristic. In simple terms, if in a non PV-node, the opponent is allowed to move twice and our position is still strong enough, it is likely strong enough to be cut off without having to fully search it. Chess engines typically check if null move pruning is allowed first. For instance, the player can not currently be in check, as that would leave the position in an illegal state. Additionally, the previous move on the board can not also be a null move, to prevent two null moves in a row. If a null move is possible, a new search is done at a reduced depth after the move is made. If the search returns a score higher than beta, the branch can be pruned without a full search necessary [19].

Reductions and extensions

Often, fully pruning a non promising subtree might cause a possible principal variation to be lost. However, it may be possible to search this tree with a lower depth to confirm any suspicions. Similarly, some very promising subtrees may need to be searched deeper for a more in depth evaluation. This is where the search function employs depth extensions and reductions. One of the strongest such depth reducing alpha beta enhancements is late move reduction. It, in combination with null move pruning and futility pruning, reduces the effective branching factor of the search tree to around 3 [20]. Late move reduction relies on a similar observation as principal variation search does. If there is a good reason to assume that the first few moves are likely the strongest, then the later moves do not necessarily have to be searched at full depth, as they are probably going to be cut off. If one of the moves searched at a lower depth ends up returning a value above alpha, i.e. that is not as bad as the engine expected, a full depth search is carried out. Depth reductions can not be applied without care, as even tactical moves that are expected to be bad may turn out to be good. Most engines do not apply them to any move that is a good capture as decided by static exchange evaluation, gives a check, or is already at low depth. Other conditions may apply too. Of course, there are positions where a depth extension is warranted. For instance, in positions with tactics or very forcing lines, it may be worth searching a certain move slightly more deeply [20]. This is exactly what singular extensions are used for. If a move tends to be the only strong one out of many possibilities, this move's search depth should be extended. The drawback here is that it is very difficult to detect whether there is only a single strong move without having searched all of the moves.

D. Evaluation

After the engine has gone through the different move trees, it will eventually reach some leaf positions. Assigning a relative value to these positions using a heuristic function encompasses the meaning of evaluation. The value assigned to the position reflects the chance of winning for the respective side and is what the engine uses to determine the most optimal next move. The evaluation function in a modern-day chess engine is either a traditional heuristics function or a multi-layered neural network and sometimes a combintion of the two. Traditional heuristics range from light evaluation, focusing on the most basic of features, to heavy evaluation, where a lot of features and combinatory patterns are detected and evaluated. Given that engines usually operate in a finite amount of time, a heavier evaluation function will consume more processing power which could otherwise have been used in the search. Engines such as Fruit [21] have illustrated that light evaluation functions are capable of competing with heavier ones, even at the highest level. This tradeoff between search and evaluation is nothing new. A. Junghanns and J. Schaeffer weigh the advantages of search versus evaluation, concluding both search and evaluation eventually yield diminishing returns, and therefore suggesting a balanced tradeoff between the two [22].

Implementation

The first thing a classical chess player is taught in terms of evaluation regards piece values. Piece values are supposed to simulate the worth of a piece in terms of assisting their side in achieving victory. Intuitively, it makes sense for some pieces to be worth more than others. Most attempts to quantify the worth of an individual piece try to do so by setting the value of a pawn as a baseline, usually 100. This value is typically referred to as centipawns and is then used to assign other pieces a value. From early attempts to quantify the worth of pieces by H.M. Taylor in Chess by using the average mobility of a piece over the 64 squares. More recently, S. Droste and J. Furnkranz [23] attempt to assign value to pieces with the use of reinforced learning. In their paper, a conclusion is reached for the four chess types which they looked at. Relevant to this thesis, which purely looks at standard chess, they assigned the following values to the pieces:

Piece	Value
Pawn	1.0
Knight	2.7
Bishop	2.9
Rook	4.3
Queen	8.9
King	∞

Table 1: Piece values

It is important to note that this study [23] focused purely on the piece values, without looking at anything other than their values. In modern-day engines, however, the evaluation of material is generally not this simplistic, and piece values will differ if other factors are at play.

First of all, the value of pieces is relative to the game phase. Though simply assigning a handful of game phases is generally not considered good practice, as it leads to the concept of evaluation discontinuity, where a significant jump is made on the border of said game phases. A simple and intuitive improvement to this naive approach is that of tapered evaluation, where the game phase is projected as a range, and

positions are evaluated using interpolation between their early and endgame values. Each of these two values, reflects the relative score the parameter brings in the respective game phase. By interpolating between the two, a sort of suitable middle-ground is reached. With the use of this method, a position can now be evaluated without the idea of evaluation discontinuity. Second of all, a linear combination of all piece values may be a decent heuristic for some positions but offer no deeper going analysis; positions can be completely lost or winning regardless of the material advantage or disadvantage. Though it is important to note that some tactical sequences which lead to a significant material advantage are easier to spot with a good material evaluation. The next aspect to look at is the positioning and mobility of pieces. This heuristic will complement the static piece value evaluation. Even without looking at the threats, the positioning of pieces can really help boost the accuracy of the evaluation heuristic.

For instance, an uncontested outpost for a minor piece, the possession of the complementing bishop pair, and rooks on the relative seventh rank are but a few of the many indicators of good features in a position. The time spent evaluating features such as outposts is kept to a minimum through bitboard operations. Through a linear combination of the evaluation of all piece types, a score for the material alone is assessed. This linear combination is comprised by the sum of all piece evaluations functions for white, which the black piece evaluations are then subtracted from to form a single evaluation. This score, along with the threat assessment and initiative assessment makes up the total evaluation. Finally, the tallied score will go through the tapered interpolation, and return as the completed evaluation. With all these ideas in mind, the illustration in figure 18 of the appendix shows a simplified depiction of the outermost function, which is called on the final depth of the search algorithm.

Initialization

Before these evaluation functions are called, Evaluation Initialization is called, which stores relevant information in a temporary object accessible for all evaluation functions to improve performance. First, the game phase is determined, which will be used in the tapered evaluation to interpolate the mid-game and endgame scores to a single score. This game phase is based purely on summation of the amount of specific pieces left on the board. The determining of the game phase is implemented in a similar fashion to how Fruit [21] assesses the game phase.

After the game phase has been determined, it will eventually be used to interpolate the final evaluation score in the manner depicted in equation 2. Here, early denotes the tallied early game evaluation and end denotes the tallied endgame evaluation.

$$eval = \frac{1}{256}((early \cdot (256 - phase)) + (end \cdot phase)) \quad (2)$$

Next, for both white and black, all pseudolegal moves for all pieces are stored separately. Then, The bitboard for the squares which either side control is saved by simply using bitwise *or* operations on every pseudolegal move of every piece. Finally, the bitboards for squares that are controlled twice are stored. This bitboard is acquired by using bitwise *or* operations in a specific order on the controlled squares and pseudolegal moves.

To save a few LSB/MSB operations later down the road, both king's squares gets determined from their respective bitboards and stored as well.

Next is the king area, which is a precomputed bitboard of the region around the king and used to detect threats around the king. If a king is positioned on the edge of the board, i.e. the A/H files and first and eighth ranks, the king area will be as if the king were more centralized, moving the files and ranks 1 towards the centre respectively. This is illustrated in figure 11, which depicts the exact bitboards of three King positions and their respective King areas. Note that the leftmost King area is adjusted as mentioned above.

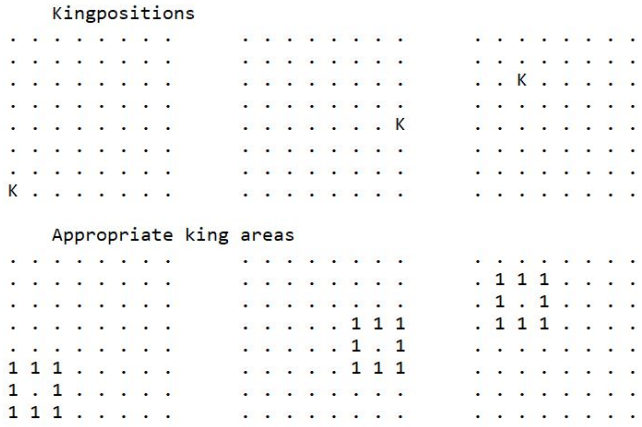

Fig. 11. Three King positions together with the respective King Area

Additionally, squares around the king which are protected twice by friendly pawns are subtracted from the king area as they are considered relatively safe from enemy threats. Finally, the pinned pieces, i.e. pieces which are pinned to the king by an enemy sliding piece attacker, and blocking pieces, i.e pieces that block pawns from advancing are computed as well before the mobility area is determined using those bitboards.

Piece evaluation - material score

The static material score is a score that is assigned to every piece excluding pawns. This score represents the static value of a piece regardless of the board state as mentioned before in implementation. The static material score for a given piece is the number of occurrences of said piece multiplied by the predetermined score. Because the static exchange evaluation needs every piece to have a score, the king gets assigned a score of ∞. The static material score is used in the evaluation of every piece, though the ∞ score of the king, is not added to the total for obvious reasons. A simplified depiction of this

function can be seen in figure 19 in the appendix

Piece evaluation - mobility score

The piece mobility score is computed with the mobility area, which is computed in the initialization phase. The mobility area itself is a bitboard consisting of all squares which are not occupied by blocked friendly pawns nor friendly pinned pieces as those pieces are generally not expected to move freely. The friendly king and queen position is also excluded together with squares that are controlled by enemy pawns. What remains is a bitboard of all the squares which are relatively feasible for a piece to move to in a handful of turns. After using the bitwise *and* operation on pseudo-legal moves of a certain piece, the popcount operation is then used to count the number of overlaps. This number is finally multiplied with the respective mobility score.

Figure 12 depicts a chess position, and the corresponding mobility area for both sides. As mentioned above, some squares which are considered infeasible to move to in the near future are left out, denoted by a '.' .

```
Black
1 1 1 . . 1 1 1
1 1 . 1 1 . 1 1
1 1 1 1 1 1 1 1
1 1 1 . . . 1 1
1 . . . . 1 1 1
. . . 1 . . . .
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1

White
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
. . . . . . . .
1 1 1 1 1 1 1 1
1 1 1 . . . 1 1
1 1 . 1 1 1 1 1
1 1 1 1 1 . 1 1
1 1 1 . . 1 1 1
```

Fig. 12. The classical variation of the Italian opening, the Giuoco Pianissimo, after *1. e4 e5 2. Nf3 Nc6 3. Bc4 Bc5 4. d3 Nf6 5. c3*. And the respective mobility bitboards

Piece evaluation - minor pieces & knights

A handful of evaluation heuristics is specific only to the minor pieces, meaning the knights and bishops. Starting with outposts, which, according to many [24], is defined as a square which is on the fourth fifth sixth or seventh rank, which is protected by a pawn and unable to be attacked by enemy pawns. Outpost detection is fast, and the function only looks at the relevant squares to determine whether a square is an outpost or not.

As minor pieces play a crucial role in the safety of the king, a slight penalty is given for every square which separates it from the friendly king. Using the Chebyshev distance [25], which is precomputed in a table, this distance is then multiplied by the King Distance Penalty. Equation 3 depicts the Chebychev distance for Cartesian coordinates, which is relevant to chess.

$$D_{Chebychev} = max(|x_2 - x_1|, |y_2 - y_1|) \tag{3}$$

```
5 4 3 2 2 2 2 2
5 4 3 2 1 1 1 2
5 4 3 2 1 K 1 2
5 4 3 2 1 1 1 2
5 4 3 2 2 2 2 2
5 4 3 3 3 3 3 3
5 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5
```

Fig. 13. Chebychev distance illustrated, K denotes the square which the distance is computed from

The outpost score, king distance score and a score for a minor piece which is shielded behind a pawn are applied per piece as the individual positioning of each piece matters. On the other hand, a minor piece also receives a bonus if it is defended. This is done using the global bitboard of every knight or bishop of a color and using the bitwise *or* operation with the precomputed bitboard of all controlled squares by that same color. The popcount of the resulting bitboard is then multiplied by the minor piece defended score. These shared evaluation heuristics coincidentally also make up the entirety of the Knight Evaluation, as shown in figure 20 in the appendix.

Piece evaluation - bishops

Aside from the shared evaluation heuristics for minor pieces mentioned above, the bishop is scored on many other heuristics. First, the bishops are checked for complementing colour-complex control, or in layman's terms, if there is both a dark and light-squared bishop. The idea of the bishop pair, which controls both colours is a well-known idea within the chess community. Simply checking if there are two bishops is insufficient, as a daring side could promote a pawn to a bishop of the same colour. Therefore, the bishops bitboard is compared to a precomputed bitboard of light and dark squares to determine if there is in fact a bishop pair. The importance of the bishop pair has been shown before by a paper by Mark Sturman [26].

Additionally, the bishop receives a very slight penalty for every enemy pawn it attacks, even through friendly pieces. This penalty is applied using the bitboard of enemy pawns and the precomputed sliding piece attack table using 0 as the occupancy bitboard and is meant to prevent maneuvering the bishop behind an enemy pawn chain. If a bishop attacks the enemy king area, it receives the bishop attacking king score as a bonus to facilitate long term attacking ideas and short term tactical sequences. Finally, to promote developing the bishop on the long diagonal early, a minor bonus is given for a fianchettoed [24] bishop in the early to mid-stages of the game. Notable is the fact that a score for the fianchetto

is only given for fianchettoes on the long diagonal, whether the shielding pawn is pushed once or twice does not matter. A simplified depiction of the bishop evaluation function is shown in figure 21 of the appendix.

Piece evaluation - rooks

Aside from the general evaluation heuristics for any piece, the rooks get an additional bonus if they defend each other. This is done by combining all the legal rook moves into a single bitboard using bitwise *or* operations and counting the occurrences using the popcount operation. The function is able to tell when at least one pair of rooks are defending each other by comparing $popcount > 2$. It is assumed that there are, at all times, two or fewer rooks are on the board. Underpromoting to a rook instead of a queen is rarely if ever a better option [24], so the performance gain by not looking deeper into the number of rooks is justified in that sense. An individual rook gets a minor bonus for being on the same file or rank as the enemy king or queen, this is to promote short tactical sequences and long-term attacking ideas as well.

Additionally, rooks on the respective seventh rank get a bonus if and only if there are enemy pawns on the seventh rank or the move cuts off the enemy king. A rook on a semi-open file, that is a file that is not blocked by a friendly pawn gets a bonus whereas a rook on a blocked file, a file that is blocked by a friendly pawn that is unable to move gets a penalty. Finally, by checking the mobility of the individual rooks and the positioning of the king and the rook, the rook gets a major penalty if it is blocked in by an uncastled king with no castling rights. This is a relatively common pattern in the opening stages of the game, which totally hinders the rook's mobility and effectiveness for at least a handful of moves. A simplified depiction of the rook evaluation function is shown in figure 22 of the appendix.

Fig. 14. Example of a blocked rook position resulting from an unusual Berlin defense after *1. e4 e5 2. Nf3 Nc6 3. Bb5 Nf6 4. d3 Bb4+ 5. Kf1*

Piece evaluation - queen

The queen, though considered the strongest piece, has a relatively simple evaluation function. Aside from the mobility and material scores, the queen only gets a penalty in evaluation if a sliding piece attack is lurking, this means that an enemy rook or bishop has a potential attack on the queen in the future. Because a queen contributes most in an attack, all the relevant threats and patterns are evaluated in the threat evaluation and king safety evaluation. Figure 23 in the appendix shows a simplified depiction of the queen evaluation function.

Piece evaluation - pawns & space

Though chess players are taught that pawns are worth a single point or 100 centipawns in value, pawns are used to dictate the flow of a game. Depending on the positioning, a pawn may be the most valuable piece on the board. The structures, levers and passed pawns must all be dealt with and evaluated. Therefore, the evaluation function for pawns must be adequate to evaluate these complexities.

Using Zobrist [10] hashing and indexing, if the pawn position is not already in the pawn hash table, the static evaluation function assigns it a value, which, along with the key, is stored in the pawn hash table. The static pawn evaluation scores pawns on a multitude of qualities. Pawns that are doubled early, backwards, isolated, those which are levered and unsupported, and blocked pawns all receive appropriate penalties. Pawns that are potentially passed pawns, and pawns that are connected and supported receive appropriate scores. Additionally, the static pawn evaluation also stores the pawn attacks and pawn attack span (all the squares they may attack in the future), which is used in fully scoring the passed pawns later down the road. The static pawn evaluation is stored without any use or knowledge of the other pieces, as these values only retain information regarding pawns. As such, those values may be used in future evaluations of the same pawn structure regardless of the other piece positioning.

A simplified depicted of the static pawn evaluation is shown in figure 24 of the appendix.

After the static pawn evaluation returns, the passed pawn evaluation then uses the stored information to calculate a full evaluation for the pawns. Using the passed pawn span, potentially passed pawns and relevant bitboards, the passed pawns are then scored based on king proximity, whether the passed pawn span is contested by enemy attacks and pieces or not, whether the path to queening is uncontested or not and whether the next square is uncontested or defended. All of these evaluations take the x-raying of rooks (a common pattern when dealing with passed pawns) into account and also only scores pawns which have advanced past the third rank respectively.

Finally, the spatial advantage is computed using a couple of values from the static pawn evaluation. Space is a concept used in chess to describe the mobility of pieces in the early game. In the engine it is used to promote centre domination and the improvement of piece mobility. The engine evaluates

up to 3 squares behind friendly pawns as safe and assigns a score to those squares if they are within the centre files(C D E F) and early ranks (2,3 and 4 for white, 7,6 and 5 for black). Additionally, blocked pawns and piece count is also taken into account in the space evaluation.

```
   White                Black
. . 1 1 1 1 . .      . . 1 1 1 1 . .
. . 1 1 1 1 . .      1 1 1 1 1 1 1 1
. . 1 1 1 1 . .      1 1 1 1 1 1 1 1
. . 1 1 1 1 . .      1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1      . . 1 1 1 1 . .
1 1 1 1 1 1 1 1      . . 1 1 1 1 . .
1 1 1 1 1 1 1 1      . . 1 1 1 1 . .
. . 1 1 1 1 . .      . . 1 1 1 1 . .
```

Fig. 15. The centre files and early ranks bitboard, used by the spatial evaluation

Piece evaluation - king & king safety

Evaluating the position of the king is crucial in order to grasp the position. Since the king position is what the chess game eventually revolves around, the evaluation must be adequate. First, the position is evaluated based on the relevant pawn-storm. A pawn storm is a concept in chess where a player advances their pawns in front of the enemy king in an attempt to dislodge their defenses and create an attack. The engine evaluates advances on the file of the king and the two adjacent files. A pawn-storm that is blocked by friendly pawns adds a lighter penalty than one which has not been.

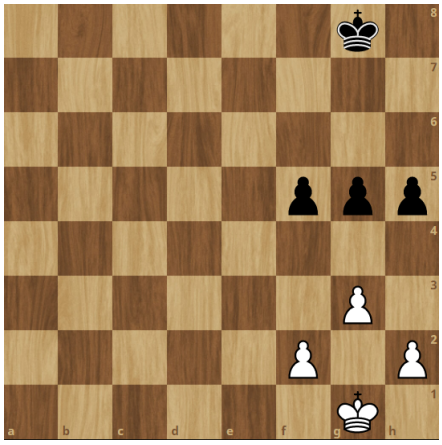

Fig. 16. Example of an unblocked pawnstorm without any other piece. *6k1/8/8/5ppp/8/6P1/5P1P/6K1*

Next, a penalty is given if the king is on an open file in the early stages of the game, as it considered to be vulnerable. The distance between the king and the nearest pawn is also scored based on the stage of the game. While it may not matter early, king positioning is integral in endgames and king maneuvering leading up to the endgame is decisive in most endgames.

Next is threat safety, which is the precursor to threat evaluation. After tallying all the weak squares around the king, potential enemy threats are assessed. Because a square

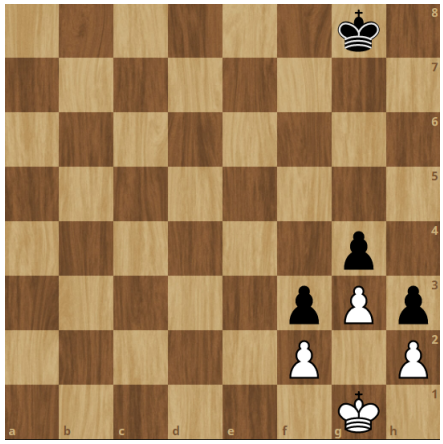

Fig. 17. Example of a blocked pawnstorm without any other piece. *6k1/8/8/8/6p1/5pPp/5P1P/6K1*

may only be utilized for a check by a single piece, it is important to take into account that some checks are more valuable than others. This way, the engine does not evaluate the same checking square twice, and will always play the strongest check in the position if at all. For each weak square around the king, a minor penalty is deducted. Finally, every unsafe check gets a slight penalty, and receives a slight bonus if the opponent does not have a queen, which is considered by many to be the strongest attacking piece. A simplified depiction of the king evaluation function is shown in figure 25 of the appendix.

Threat evaluation

Analyzing threats is essential for an accurate assessment of a given position. Aside from piece placement, the relevant threat that that piece makes on the enemy position is calculated in the engine as well. This way, the engine can see threats of captures and checks a handful of moves deeper. Whenever the search evaluates a leaf node, the threat analysis, along with a quiescence search will combat the horizon effect. Threat analysis in the engine is relatively simple. First, the enemy position is captured in bitboards. A bitboard for squares that are considered safe for the opponent, A bitboard of defended enemy pieces, and one for weak enemy pieces are used to calculate the threat scores.

First, threats are ranked based on the attacking and attacked piece, i.e. threats by a minor piece are ranked differently from those of a rook or the king. Additionally, the attacked piece is also taken into consideration. Because threats by a queen generally don't add or remove advantage from the position, the queen is not considered when calculating threats. Note that threats on the queen are considered. With this in place, excluding the queen, every piece attacking any other piece is scored accordingly. Aside from direct threats, threat detection also checks for potential threats created by pawn pushes on the next move, hanging pieces and assigns a small bonus for the number of squares controlled by either side.

Initiative evaluation

When it comes to advantages, it is common in engines to give a slight bonus to the side which has the advantage. Intuitively this makes sense. A pressed advantage may lead to an even bigger lead or in some cases a draw. Initiative is calculated for the side which has a better evaluation score. First, it is important to mention that adding the initiative bonus will never flip the side which is leading. At most, the advantage will be reduced to a drawn position.

The initiative bonus is assigned based on a number of criteria. Namely, based on the number of pawns and passed pawns, whether the king is outflanking the enemy king, whether there are flank pawns, and whether the king is infiltrating or not. If there are no flank pawns, and the king is not outflanking the enemy king, a large draw penalty will be added to the initiative bonus, sometimes cancelling out all other bonuses. After the initiative score has been calculated, the middle and endgame scores are calculated and adjusted to not flip the advantage to the other side.

Parameter tuning

With the evaluation in place, the next advance comes in terms of parameter tuning. With parameters tuning, what is meant is mostly the score and penalty tuning. After tweaking a certain score by a marginal amount, the engine then plays a match against a version with the previous score. After the match has concluded, the engine with the higher win percentage is then tweaked again until there is no more change in the desired parameters. This is the general idea for most parameter tuning algorithms. And the goal is to produce a set of parameters which is semi-optimal for playing common chess positions to the most of the position. Most parameter tuning algorithms handle the engine as if it were a black box. The starting values, i.e. the values assigned to the engine before tuning also play a crucial role in determining what the variables will eventually end up on. The selection of initial values relies mostly on intuition and common sense as there is no established optimal setup for any given engine.

Sequential tuning

Sequential tuning involves tuning a single variable until it no longer affects the win percentage. Then, the next variable would be tuned with the previous variable already set to the new value. Noteworthy is that because the variables are tuned one by one, the previously tuned variables will influence the tuning of future variables. While it can be said that this method is not mathematically sound to reach a global optimum of parameters, without a tremendous amount of resources it would be difficult to find the global optimum. This method of sequential tuning is a practical approach to improve the engine's playing strength. Also noteworthy is that the nature and inner workings of sequential tuning resembles the optimization method of gradient descent [27] closely, meaning a local optimum would probably be reached when parameters are tuned using this method.

Genetic algorithm tuning

In Genetic Algorithm Tuning, or GA tuning, the tuning of parameters relies on the fitness function described in most genetic algorithms [28]. While this would be a relatively sound method to reach a local optimum, it was not considered due the time frame of this undertaking. Through selective sampling and minor mutations in the parameters, the most-fit individual would move on to reproduce. This process resembles natural selection closely, as is befitting of an algorithm dubbed after that very process.

Probabilistic tuning

Finally, probabilistic tuning methods like simulated annealing [29] would be feasible in finding a set of optimal parameters. Though the engine relies on a lot of parameters, they would still be able to be tuned to a high degree by methods like gradient descent with simulated annealing. However, due to time constraints and the overwhelming practical benefits of sequential tuning, probabilistic tuning methods were not used in the tuning of the engine.

III. Methodology

The four different board representations and move generation techniques listed below and described in the previous section will be implemented and then compared and contrasted.

1) Two dimensional 8x8 square board representation
2) 0x88 based board representation
3) Magic bitboards
4) PEXT bitboards

Each of these will be subjected to testing in the following scenarios.

1) Perft
2) Alpha beta style search

Each of these will be run from 6 different starting positions, which are often used for verifying move generation correctness [12]. For the sake of reference, they are listed here too.

1) rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq
2) r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq
3) 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w -
4) r3k2r/Pppp1ppp/1b3nbN/nP6//q4N2/Pp1P2PP/R2Q1RK1 w kq
5) rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ
6) r4rk1/1pp1qppp/p1np1n2/2b1p1B1/2B1P1b1/P1NP1N2/1PP1QPPP/R4RK1 w -

In both two scenarios, the speed of the engine will be determined in nodes per second. A node here is defined as a node of the game tree that is processed by the engine. Note that this implies that a node is added on any call to the perft or search function. In perft, most often only leaf nodes are counted for the sake of verifying correctness of the move generator, but since the focus is on performance here, every node will be counted. Time will be started when the initial call to perft

or search with the highest depth is made. Time is stopped when this function returns. Time will be measured using C++'s `std::chrono::high_resolution_clock` .

For consistency purposes, all of the tests are executed on a single thread, and no hash table for transpositions is used. Both of these could significantly speed up the algorithms used above, but they would introduce discrepancies that would likely lead to a weaker comparison between the four approaches. Additionally, for a full search, time is measured to inspect how much of the total time is actually spent generating moves and evaluating a position. This can be compared to the total time taken for a search to see which areas use the most amount of time. At the end, the times calculated for each of the different positions will be averaged out and reported in the results.

The second part of the experiment involves parameter tuning, where the success of the engine will be determined by means of win percentage against the former version of itself. The win percentage will be calculated on different depths and will be calculated over a match consisting of 20 games. To avoid repeating the same game, the starting positions will vary. Each starting position will come forth from an opening book [24], where common chess openings are listed. The match will be decided either by checkmate, stalemate, or draw by threefold repetition or lack of sufficient material. For the sake of sanity, an evaluation score of +10, where both sides agree on the victor will also result in a win. In an official match between regular engines, like those that happen in the TCEC [30], a game will be adjudicated as won if both engines evaluate the position as +10 for 5 consecutive plies. After repeated testing, a mutual score of +10 for 1 turn by both sides was found to be enough for the sake of practicality and time efficiency. In the tests, virtually every match which came to a score of +10 resulted in victory for that particular side.

For the starting positions, the following were chosen. Each side played one game in the position as black, and one as white:

1) Ruy Lopez (Berlin Defense) - *1. e4 e5 2. Nf3 Nc6 3. Bb5 Nf6*
2) Scandinavian Defense (Main Line) - *1. e4 d5 2. exd5 Qxd5 3. Nc3 Qa5*
3) Sicilian Defense (Open Sicilian) - *1. e4 c5 2. Nf3 d6 3. d4 cxd4 4. Nxd4 Nf6 5. Nc3*
4) Queens Gambit Declined - *1. d4 d5 2. c4 e6 3. Nf3 Nf6*
5) Smith Morra Gambit (Accepted) - *1. e4 c5 2. d4 cxd4 3. c3 dxc3 4. Nxc3 Nc6*
6) French Defense (Exchange variation) - *1. e4 e6 2. d4 d5 3. exd5 exd5*
7) Petrov Defense - *1. e4 e5 2. Nf3 Nf6*
8) King's Indian Defense - *1. d4 Nf6 2. c4 g6 3. Nc3 Bg7*
9) Alekhine Defense - *1. e4 Nf6 2. e5 Nd5 3. d4 d6*
10) Englund Gambit - *1. d4 e5 2. dxe5 Nc6 3. Nf3 Qe7*

These particular openings were picked due to the diverse playing styles that usually come with them when they are played by humans. Additionally, the positions do not easily transpose into a similar position, making the games played unique. The positions are timeless, and all except the Englund gambit do not lose by force [24]. From the solid Berlin and French defense to the explosive Englund gambit, the hope is that by navigating these positions, a set of adaptive parameters is chosen to benefit the playing strength.

Each iteration of tuning, the tested parameters will get tuned by some value x, which amounts to 10% of the initial value. After 10 iterations of improvement, the initial set of parameters will be tested against the tuned ones, and the data from this match will be used to examine the improvement. The depths were chosen with the intent of observing different adaptations.

To better understand the complexities of the evaluation and search, the depth was kept low, to simulate a beginning - intermediate player. As a reference point, the engine at a depth of 6 plies was able to beat Emir, a 1000 rated engine on chess.com by a margin of 19/20, where 18 games were victorious and 2 game led to a draw. While the engine was not completely finished at this point in time, it goes to show the raw calculating capabilities of an engine, even at such a low depth.

In contrast, the engine at depth 6 was unable to defeat Nelson, a bot rated 1300 on chess.com. In all of its games, the glaring flaw was the inability to see past 6 plies. Long term plans and long tactical sequences eluded the engine, even though it did not blatantly blunder. The fact that it did not blunder is partially due to the quiescence search, which prevented it from blundering any hanging pieces past turn ply 6. Another drawback that comes with the inability to see past depth 6 is the inability to play good openings. The engine at a depth of 6 plies will always open with 1. h4, the Kadas opening, which is not considered to be optimal. Contrast this with the engine at depth 8, where it will always play 1. e4 and the engine at depth 9 where its opening move of choice is 1. e3, the Van 't Kruijs opening. While this will not matter much in the tuning process, it is a good indicator of playing strength.

After testing the engine enough to the point where it virtually never crashed nor played any illegal moves, the tuning commenced. The engine at depths: 4 up to 8 were tuned and pitted against the original versions in matches against itself at an equal depth. Any depth below depth 4 produced games which were too incoherent to judge improvement for, therefore those versions were not tested.

For each tuning iteration, a different parameter was chosen to tune, this way, after the final iteration, the tuned version of the engine would in theory be slightly better tuned than the original version in multiple fronts. The following parameters were tuned. The initial values are depicted with the early game score left, and the endgame score right. The scores are in centipawns.

IV. Results

A. Results

Table IV-A shows the results of the first test, which looks at the pure speed of the different types of board representations. Speeds are shown in mega nodes per second. In terms of move

Parameter	Initial value
Knight Material Score	350,350
Bishop Pair Score	50,50
Knight Outpost Score	25,25
Rook Stacked Score	25,25
Queen Pinned Penalty	30,30
King On Open File Penalty	50,0

Table 2: Initial Scores in centipawns

generation speed, both bitboards and 0x88 based positions seem to be nearly equal in speed. In an actual search however, 0x88 falls off as compared to bitboard approaches. Simple 2D array boards perform the worst in both cases.

Type	Perft speed (MN/s)	Search speed (MN/s)
2D array based	39.189	6.327
0x88 based	46.496	7.216
Magic bitboards	48.772	10.992
PEXT bitboards	48.740	11.038

Table 3: Perft and search node speeds

Table IV-A shows how much time is allocated to some of the subroutines of note for this thesis. First, move generation takes up, on average, about 10% of the resources. As shown, bitboard approaches seem to be quicker at evaluating positions than array based approaches. The final column of the table displays the rest of the time spent. It refers to searching, move ordering, playing and undoing moves and all of the other small discrepancies.

Type	Move generation	Evaluation	Other
2D array based	0.13	0.43	0.44
0x88 based	0.10	0.41	0.49
Magic bitboards	0.10	0.34	0.56
PEXT bitboards	0.10	0.34	0.56

Table 4: Fraction of time allocation in move generation

Table 5 illustrates the match results between the engine at different depths. Note that because the engine plays both sides, the match result will be a draw if both configurations are the same. Therefore, no control groups need to be created in order to test the results. Any score above equal will be considered a significant improvement over the previous version. Furthermore, Tables IV-A, IV-A, IV-A, IV-A and IV-A illustrate the tuned values on different depth. A graphical illustration of the changed variables plotted against the change in depth can be seen in figures 26, 27, and 28 respectively in the appendix.

Depth	Win percentage
4	75%
5	65%
6	70%
7	65%
8	60%

Table 5: Engine win percentages per depth

Parameter	Initial value	Tuned Value	Change
Knight Material Score	350,350	490,420	+140,70
Bishop Pair Score	50,50	25,20	-25,-30
Knight Outpost Score	25,25	32.5,15	+7.5,-10
Rook Stacked Score	25,25	50,50	+25,+25
Queen Pinned Penalty	30,30	27,18	-3,-12
King On Open File Penalty	50,0	30,0	-20, 0

Table 6: Parameter tuning at Depth 4

Parameter	Initial value	Tuned Value	Change
Knight Material Score	350,350	420, 385	+70,+35
Bishop Pair Score	50,50	30,30	-20,-20
Knight Outpost Score	25,25	37.5,15	+12.5,-10
Rook Stacked Score	25,25	50,50	+25,+25
Queen Pinned Penalty	30,30	30,30	0,0
King On Open File Penalty	50,0	15,-5	-35,-5

Table 7: Parameter tuning at Depth 5

Parameter	Initial value	Tuned Value	Change
Knight Material Score	350,350	385,350	+35,0
Bishop Pair Score	50,50	65,50	+15,0
Knight Outpost Score	25,25	42.5,30	+17.5,+5
Rook Stacked Score	25,25	50,50	+25,+25
Queen Pinned Penalty	30,30	30,30	0,0
King On Open File Penalty	50,0	15,-5	-35, -5

Table 8: Parameter tuning at Depth 6

Parameter	Initial value	Tuned Value	Change
Knight Material Score	350,350	420,315	+35,-35
Bishop Pair Score	50,50	40,70	-10,+20
Knight Outpost Score	25,25	42.5,30	+17.5,+5
Rook Stacked Score	25,25	50,50	+25,+25
Queen Pinned Penalty	30,30	21,21	-9,-9
King On Open File Penalty	50,0	35,5	-15, +5

Table 9: Parameter tuning at Depth 7

Parameter	Initial value	Tuned Value	Change
Knight Material Score	350,350	385,350	+35,0
Bishop Pair Score	50,50	80, 70	+30,+20
Knight Outpost Score	25,25	42.5,25	17.5,0
Rook Stacked Score	25,25	50,50	+25,+25
Queen Pinned Penalty	30,30	21,21	-9,-9
King On Open File Penalty	50,0	35, 5	-15,+5

Table 10: Parameter tuning at Depth 8

B. Result analysis

Board representation

It is interesting to see how well older approaches, such as 0x88, keep up with more modern approaches like PEXT bitboards in perft speed. The advantage of bitboards does not shine as much in just move generation as bitboards will inevitably have to be serialized into a list of moves for the engine to play. This means that much of the speed from instantly being able to calculate all of the moves is used similar to how an array based approach would simply loop through the array. In terms of evaluation speeds, however, bitboards seem to have an advantage. For the purposes of evaluating, a bitboard does not need to be serialized.

Also worth noting is that the two different bitboard approaches do not have any meaningful differences between

them. This is not too surprising, as the difference is only in the way sliding piece attacks are determined. Their time complexities are both constant, PEXT is just slightly more efficient. In terms of evaluation, they are identical. The small differences between the two in table 3 are likely due to noise.

A regular 2D board seems to be the clear loser out of the bunch. While it might be the most intuitive, it just has too many inefficiencies that can very easily be ironed out with some simple changes to the architecture.

Parameter tuning

After tuning, the new versions were able to get a decisive advantage over the old versions like hypothesized before. With an average increase of 15% win rate at all depths, it goes to show the importance and impact of parameter tuning, even if it is done over 10 iterations. Though some parameters stopped changing even before the 10th iteration, it is important to note that, here, they would probably have changed if different parameters were tuned before or after as well. Despite the changes, the engine was unable to overcome some positions. Especially, defending the Alekhine's Defense was tough for the engine, where it only managed to squeeze out a draw after the depth 8 version had been tuned, the rest of the games in this position, it lost. Though tuning was evidently successful, what is also interesting is the change in the tuned values. In all cases, the rook stacked score went up every iteration, indicating that on a relatively low depth, there is indeed a lot of value when it comes to the basic idea of doubling up rooks, at least, more than was initially speculated. This was also evident in the games, where sometimes games featured the doubling of rooks, where some reached a completely different position. What is also notable is the up trend in both early and endgame bishop evaluation. At a lower depth, the engine seems to disregard the idea of two bishops, whereas it will value a bishop pair more at higher depth.

Because most games at low depth are all about tactical patterns and 2-4 turn exchange sequences, it is interesting to note how the value of the knight changes from 490 at depth 4 to 385 at depth 8. This indicates that at a lower depth, the worth of having a material advantage may outweigh the positional advantages in the position. Whereas at a higher depth, the engine seems to come to an understanding about the value of the knight. The irregularity in the knight material score in the endgame also fluctuated a lot. This is credited partially due to the low depth and specific scenarios where sometimes a knight would not be enough for the engine to mate at its respective depth. Finally, it is noteworthy that most values did not stray too far from the initial value. As mentioned before this is partially because not all parameters were tuned, but still interesting to mention, as it goes to show the importance of the initial setup and intuitive understanding of the entire engine environment.

V. Conclusion

All in all, move generation speed can be very fast with both array based and bitboard based approaches. This is shown by the results regarding perft speeds. In terms of the complete chess engine, however, move generation is only a small fraction of where processing power is used. Consequently, micro optimizations in the move generator may not be as effective for the overall playing strength. It is still recommended to use bitboards, because they suit many areas of the code, especially evaluation, incredibly well. Here, the initial hypotheses were not correct. Bitboard approaches do not necessarily beat out array based approaches in move generation speed. From the results shown, it can not necessarily be derived that PEXT bitboards are clearly superior to magic bitboards. While the theory indicates that PEXT is faster, it is likely such an insignificant increase on modern processors that it is hard to see a difference in the results. Secondly, the observation that performance of board architecture is mostly only relevant in move generation is false. While move generators seem to do well with both architectures, evaluation speeds are benefited by bitboard style engines.

As for parameter tuning, a 15% increase in win percent constitutes a large enough margin where it can be called a significant improvement. As shown by the results, tuning only a fraction of the parameters by means of sequential tuning was more than enough to gain a decisive advantage against an engine of similar playing strength.

VI. Discussion

While great care went into implementing most of the algorithms described, it is difficult to compare specific parts of a chess engine when they are using different board architectures. While broader functions, such as evaluation, can be measured, the workings of each approach results in often completely different internals. It follows that at certain parts of the engine speed is to be gained, while at others there is some speed lost. This makes for a difficult time drawing parallels between approaches. Additionally, our move generator implementations generated only legal moves. Many top of the line engines currently generate sets of pseudolegal moves and do the legality check in the move ordering functions. This allows them to skip certain legality checks if a tree branch was pruned. In that sense, the results shown here might not be in line with similar results from current top engines. It would also be interesting to see some of the results when compiling and running on higher end processors. The results shown here are mostly to be taken in comparison to each other, so the magnitude of one number is not very interesting. However, if a test was done with a fully multithreaded search function with a high end multicore processor, the results might differ in a sense that even more time is allotted to searching and less to evaluating and move generating. As of 2021 the strongest Stockfish versions use NNUE evaluation [31], which is a relatively small neural network that runs very efficiently on processors using SIMD vector instructions. It is faster than Alpha Zero style deep networks, but slower than classical evaluations. This too would change the results shown. Furthermore, a chess engine often uses an endgame tablebase to solve known positions. Ronald de Man's popular syzygy tablebases are probed using

descriptions of the chessboard using bitboards. Addons such as this are another reason for use of bitboards in the chess engine architecture. These have not been implemented and thus are not reflected in the tests in this thesis.

Though a lot of work had been done to tune the engine to play at its highest capabilities, some features were not present during the testing. Take for example the Mate distance pruning, which would have helped in specific scenarios. Whenever the engine saw multiple lines which lead to mate, it would pick an arbitrary one, not the quickest path to mate. To illustrate this flaw, a game out of the match against Emir will be shown below.

1. h4 e5 2. e3 Nh6 3. Nc3 d5 4. Qh5 Qd6 5. Nb5 Qc6 6. Nf3 b6 7. Nxe5 Qxc2 8. Bc4 Bb7 9. Nxc7+ Kd8 10. Nb5 f5 11. Nd4 dxc4 12. Nxc2 Bd5 13. Rh3 a6 14. b3 cxb3 15. Nd4 g6 16. Qg5+ Be7 17. Qxh6 Bxg2 18. Rg3 Kc8 19. Rxg2 Bf6 20. Qf4 Bxe5 21. Qxe5 bxa2 22. Qxh8+ Kb7 23. Qg7+ Kc8 24. Qg8+ Kb7 25. Qd5+ Ka7 26. Qf7+ Nd7 27. Qxd7+ Kb8 28. Rxa2 Ra7 29. Nc6+ Ka8 30. Qxa7# 1-0

On move 28, instead of delivering checkmate with Nc6#, the engine evaluates Rxa2 as M2, and because it does not differentiate between M1 and M2, it plays Rxa2 and mates 2 moves later. Simple issues like this can be solved by using some form of mate distance pruning, or by giving shorter mates a slightly higher score. In this particular match, the engine was allowed to pick its own opening instead of a predetermined opening. To prove the point, in the game, the engine played 1. h4 the Kadas opening and still managed to win.

Furthermore, the state of the engine leaves it in an undesirable state to estimate its current ELO rating as compared to other chess engines. Whilst testing, we estimate its rapid and classical ELO at depth 8 at around 1400 ELO on chess.com. This is based on a couple of matches against the authors, one rated around 1400 and one rated around 1900 chess.com ELO. The following analyses describe two of the games played between the engine and the authors.

The engine (Depth 8) vs Pieter Bijl (1400)
1. e4 e5
2. Nf3 Nc6
3. Bc4 Bc5
Book stops here, this is where the engine and the player start making moves.
4. d4 Nxd4
The engine decides to strike in the center with d4, likely because the space evaluation here is good.
5. Nxd4 Bxd4
6. O-O Nf6
7. c3 Bb6
Knight is threatening a pawn, but the engine concluded that kicking the bishop away from the center was worth delaying the defense of the pawn.
8. Bg5 O-O
Now, since black's bishop moved back, Nxe4 is no longer possible due to the strong Qd5 reply. However, the engine decides that pinning the knight on f6 is still worth it.
9. Qb3 c6
It is unclear why the engine decide to play Qb3, here. We suppose the engine likes the attacking battery it forms together with bishop on c4 toward the black king.
10. Kh1 d5
11. exd5 cxd5
12. Bb5 Be6
Alternatively, 12. Bxf6 dxc4 13. Bxd8 cxb3 14. Bxb6 axb6 was in the position here, essentially trading down pieces leaving black with a somewhat awkward pawn structure. Engine decided this capture sequence was not advantageous enough in its eyes.
13. f4 Qd6
14. Bxf6 gxf6
15. f5 Bd7
16. a4 a6
a6 is a critical mistake from the human player here, as it leaves the bishop on b6 only protected by the Queen, which is also the only defender of the d7 square. The engine picks up on this and decides to capitalize.
17. Bxd7 Qxd7
At this point, Bc7 was the only saving move for black, as it starts generating mate threats on h2 with the bishop queen battery. However, the human player misses this too and the engine capitalizes once more.
18. Qxb6 Qc6
19. a5 Rfe8
20. Rd1 e4
21. c4 Qxc4
22. Nc3 Rad8
23. Qxf6 d4
At this point, the game is completely lost for the human player, however, the engine gets stuck in a repetition. Due to the way the moves were played on the engine's side, repetition counters weren't increased. We are confident that with this enabled, the engine would have detected the repetition, brought its rooks in toward the offense, and won the game.
24. Qg5+ Kh8
25. Qf6+ Kg8
27. Qg5+ Kh8
28. Qf6+ Kg8
29. Qg5+ Kh8
1/2-1/2

Anh Phi Tiet (1900) vs The engine (Depth 8)
1. e4 e5
2. d4 exd4
3. c3 dxc3
4. Bc4 cxb2.
After Bc4, the engine started playing and decided correctly that cxb2 is the way to go.
5. Bxb2 Nh6.
After Bxb2, Nh6 is unusual already, it does not fight for the central squares and leaves the knight on the rim. Perhaps the engine sees this manoeuvre as good in the coming 8 plies. However, chess theory dictates otherwise. Fun to note is that this did take Anh Phi out of chess theory, and any preperation he had on this particular line in the Danish Gambit Accepted.
6. Nf3 b5.
b5 is a decent plan to play c6 afterwards.
7. Bxb5 c6
8. Bc4 d5.
After Bc4, d5 is easily seen as a blunder, as it allows for a quick simplification of the position when black is already down material. Perhaps a better line would have been 8...Bb4+9. Nc3 O-O. This way, black may take control of the semi-open e-file and play a game of chess.
9. exd5 cxd5
10. Qxd5 Qxd5
11. Bxd5 Bb4+
12. Nc3 Ba6
13. O-O-O O-O.
After Ba6, taking the rook is not a necessity, it won't leave a8. Therefore Anh Phi castles long, as castling short is no longer an option.
14. Bxa8 Rc8.
After Bxa8, the engine played Rc8, seemingly with a double attack on the knight on c3. However, since the rook is tied down to protecting the back-rank checkmate, this threat is non-existent. Despite this fact, Anh Phi defended the knight anyway. Though Stockfish evaluated Kb1 200 centipawns higher than Kc2, during the game, Anh Phi missed that the rook was tied down to the defense of checkmate, and defended the knight on c3.
15. Kc2 Kh8.
Kh8, a prophylactic move in a losing position. The engine seems to love to play Kh1 or Kh1 when castling short. Usually, it is to step out of a pin, however, here it steps in to a pin. This move again seems inexplicable and needs to be looked into further.
16. Rd2 Bb5.
After Rd2, the engine plays Bb5, correctly predicting that the plan is to play Rd1 and double op on the D file. In this scenario, the engine would have played ba4, and won the rook on d1. This is why Anh Phi plays Bd5, to play Bb3 on Ba4.
17. Bd5 Nd7
18. Rhd1 Nf6
19. Bc4 Bxc4.
After Bxc4, the position seemed to have died for white, however. The following tactical sequence, which the engine should not have missed at a depth of 8 plies, brought the game a lot closer to easily winning.
20. Rd8+ Rxd8
21. Rxd8+ Nfg8
22. Rd4 Be6
23. Rxb4 Ng4
24. Rb7 Bc8
25. Ng5 h5.
Ng5 is a small improvement over Rxa7, as Rxa7 allows f6 or h6, which prevents the winning of the black f pawn.
26. Nxf7+ Kh7
27. Rxa7 Bf5+
28. Kb3 Be6+
29. Kb4 Bxf7
30. Rxf7 Kh8
31. Nd5 N8f6.
After N8f6, the game is over. After a quick simplification, anyone could convert this position into a win.
32. Nxf6 Nxf6
33. Bxf6 gxf6
34. Rxf6 Kg7
35. Rb6 h4
36. a4 Kf7
37. a5 Ke7
38. a6 Kd7
39. a7 h3
40. a8=Q hxg2
41. Qa7+ Kc8
42. Rb8# 1-0

Notable is that the engine, though set to play at depth 8, missed some tactical patterns which should have been spotted. This could be due to some bugs or unfinished workings in the engine. Additionally, trading when down material should also be avoided, especially when playing against a human opponent. Unsurprisingly, white took advantage of black's horrid play through easy-to-find moves, and though White did not play optimally in the end, it was more than enough to delete black from existence, concluding the two of the matches against humans.

For future work, like mentioned before, one could look at formally proving the correctness of a move generator. From experience, debugging the correctness of a move generator can be a real hurdle. Moreover, searches with more intricate pruning or evaluation all might change the results. This can be determined on a case by case basis, this thesis has tried to stay as close to the essential amount. Finally, a formal method for the parameter optimization in the form of sequential tuning could also be proven, though the results did indicate a significant improvement.

References

[1] M. Campbell, A. Hoane, and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[2] R. Hyatt, "Chess program board representations," 2004. University of Alabama at Birmingham.

[3] D. Spracklen and K. Spracklen, "First steps in computer chess programming," *BYTE*, vol. 3, no. 10, 1978.

[4] V. Manohararajah, "Rajah: The design of a chess program.," *ICCA Journal*, vol. 10, no. 2, 1997.

[5] V. Vučković, "The compact chessboard representation," *ICGA Journal*, vol. 31, no. 3, 2008.

[6] L. A. Lyusternik, V. I. S. Aleksandr A. Abramov, and M. R. Shura-Bura, "Programming for high-speed electronic computers," 1952. Izdatelstvo Akademii Nauk.

[7] G. Adelson-Velsky, V. Arlazarov, A. Bitman, A. Zhivotovsky, and A. Uskov, "Programming a computer to play chess," *Russian Mathematical Surveys*, vol. 25, pp. 221–262, 1970.

[8] M. van Kervinck, "Tricky bit tricks," 1994. On the rec games chess forum, Available: https://groups.google.com/g/rec.games.chess/c/KnJvBnhgDKU/m/yCi5yBx18PQJ.

[9] P. Kannan, "Magic move-bitboard generation in computer chess," 2007. Available: http://pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf.

[10] A. Zobrist, "A new hashing method with application for game playing," *ICCA Journal*, vol. 13, no. 2, 1990.

[11] A. Bik, "Computing deep perft and divide numbers for checkers," *ICGA Journal*, vol. 35, no. 4, 2012.

[12] G. Isenberg and A. Iavicoli, "Perft results." https://www.chessprogramming.org/Perft_Results, 2021.

[13] E. Thé, "An analysis of move ordering on the efficiency of alpha-beta search," Master's thesis, McGill University, 1992.

[14] M. Newborn, "Mac hack and transposition tables," *Kasparov versus Deep Blue: Computer Chess Comes of Age*, pp. 57–72, 1997.

[15] D. Beal, "A generalized quiescence search algorithm," *Artificial Intelligence*, vol. 43, no. 1, pp. 85–98, 1990.

[16] S. Plenker, "A null-move technique impervious to zugzwang," *ICCA Journal*, vol. 18, no. 2, 1995.

[17] R. Shams, H. Kaindl, and H. Horacek, "Using aspiration windows for minimax algorithms," in *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, (San Francisco, CA, USA), p. 192–197, Morgan Kaufmann Publishers Inc., 1991.

[18] A. Muszycka-Jones and R. Shinghal, "An empirical comparison of pruning strategies in game trees," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, 1985.

[19] O. David-Tabibi and N. S. Netanyahu, "Extended null-move reductions," in *Computers and Games*, (Berlin, Heidelberg), pp. 205–216, Springer Berlin Heidelberg, 2008.

[20] T. Marsland and Y. Björnsson, "Variable depth search," *Advances in Computer Games 9*, pp. 9–24, 2001.

[21] Wikipedia contributors, "Fruit (software) — Wikipedia, the free encyclopedia," 2021. [Online; accessed 3-July-2021].

[22] A. Junghanns and J. Schaeffer, "Search versus knowledge in game-playing programs revisited," pp. 692–697, 09 1998. Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI97.

[23] S. Droste and J. Furnkranz, "Learning the piece values for three chess variants," *ICGA Journal*, vol. 31, no. 4, 2008.

[24] D. Hooper and K. Whyld, *The Oxford Companion to Chess*. Oxford University Press, 2 ed., 1992.

[25] Wikipedia contributors, "Chebyshev distance — Wikipedia, the free encyclopedia," 2021. [Online; accessed 1-July-2021].

[26] M. Sturman, "Beware the bishop pair," *ICGA Journal*, vol. 19, no. 2, pp. 83-93, 1996.

[27] S. Ruder, "An overview of gradient descent optimization algorithms," 09 2016. https://arxiv.org/abs/1609.04747.

[28] S. Li and D. Li, *Genetic Algorithms*. 03 2021.

[29] D. Henderson, S. Jacobson, and A. Johnson, *The Theory and Practice of Simulated Annealing*. 04 2006. pp. 287-319.

[30] Wikipedia contributors, "Top chess engine championship — Wikipedia, the free encyclopedia," 2021. [Online; accessed 1-July-2021].

[31] Y. Nasu, "Efficiently updatable neural-network based evaluation functions for computer shogi," 2018. Accessible: https://github.com/asdfjkl/nnue/blob/main/nnue_en.pdf.

Appendix

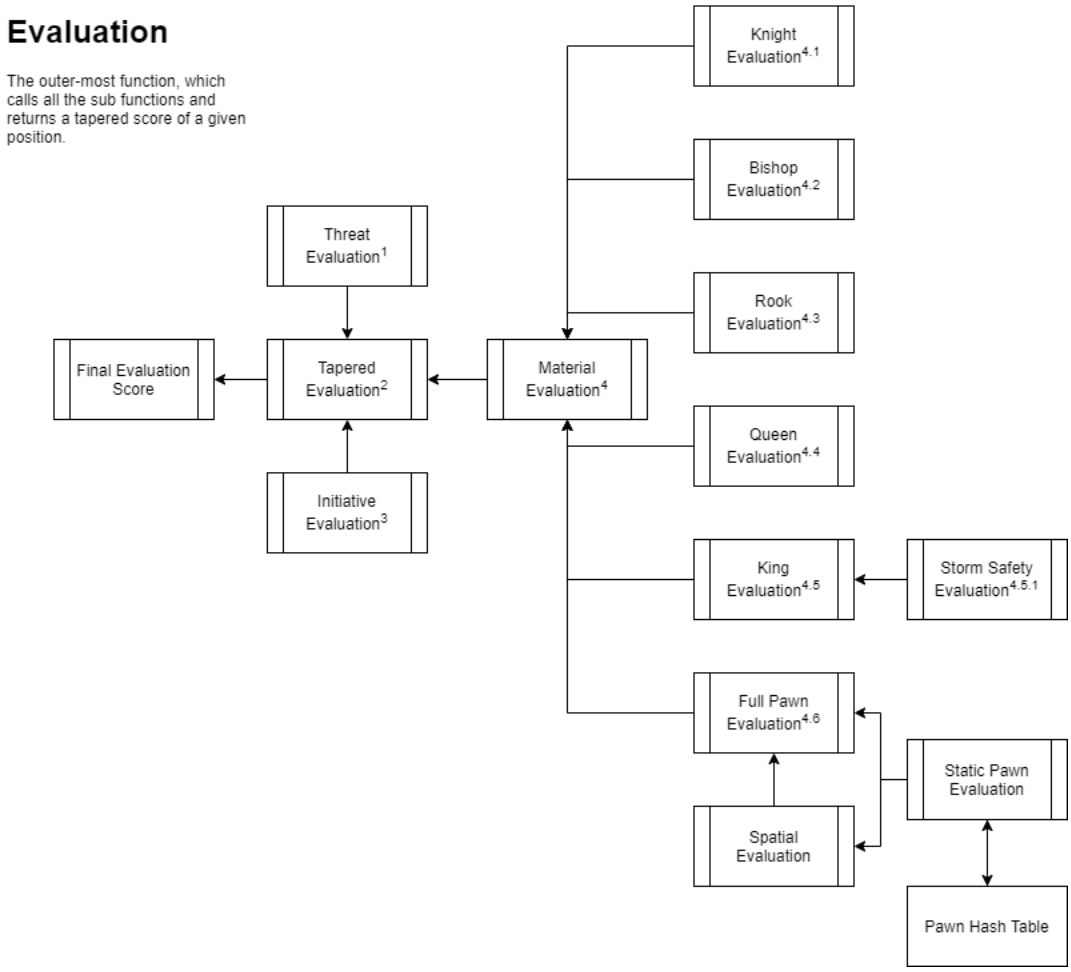

Fig. 18. Complete evaluation

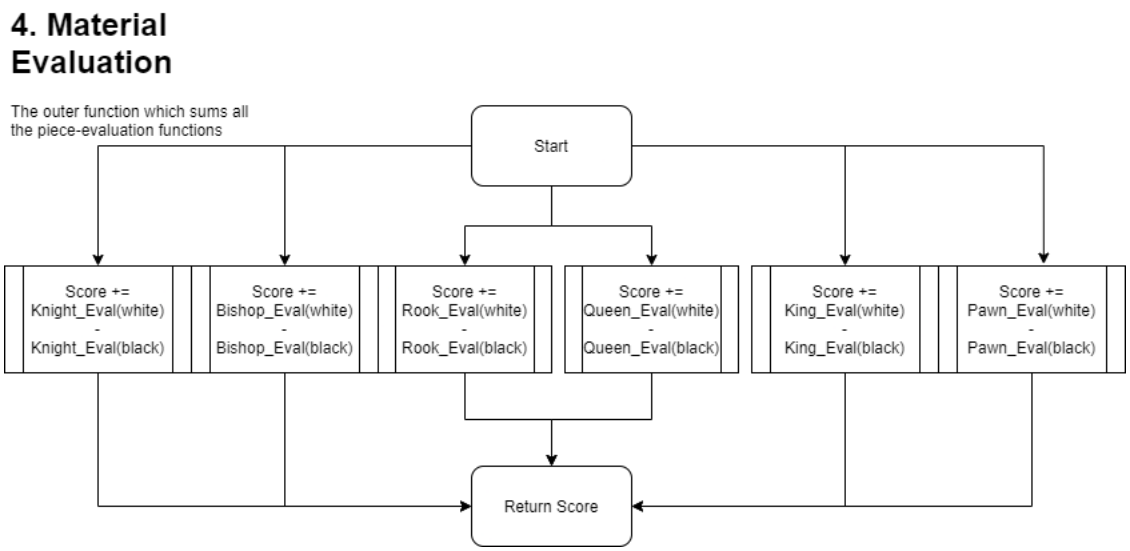

Fig. 19. Material evaluation

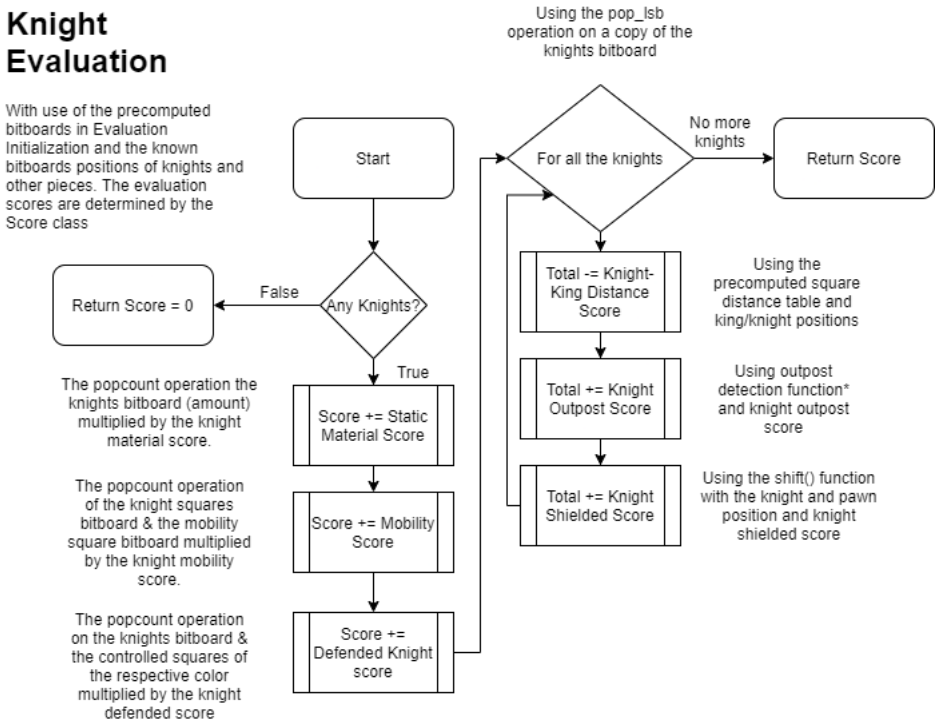

Fig. 20. Knight evaluation

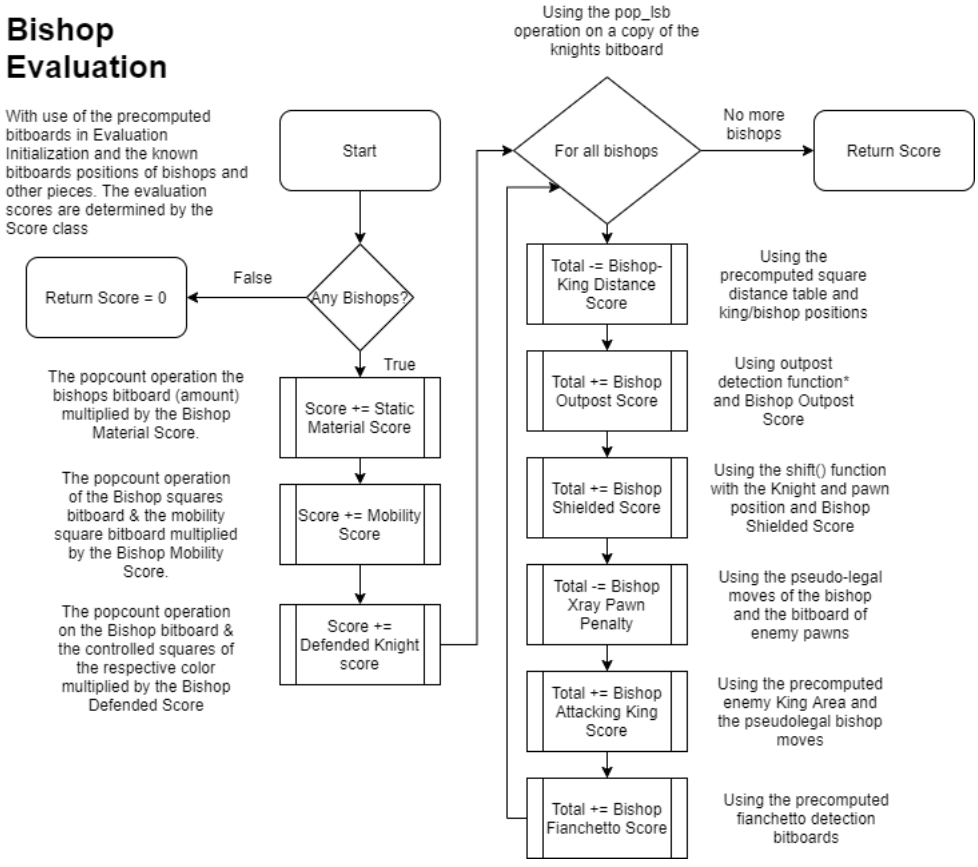

Fig. 21. Bishop evaluation

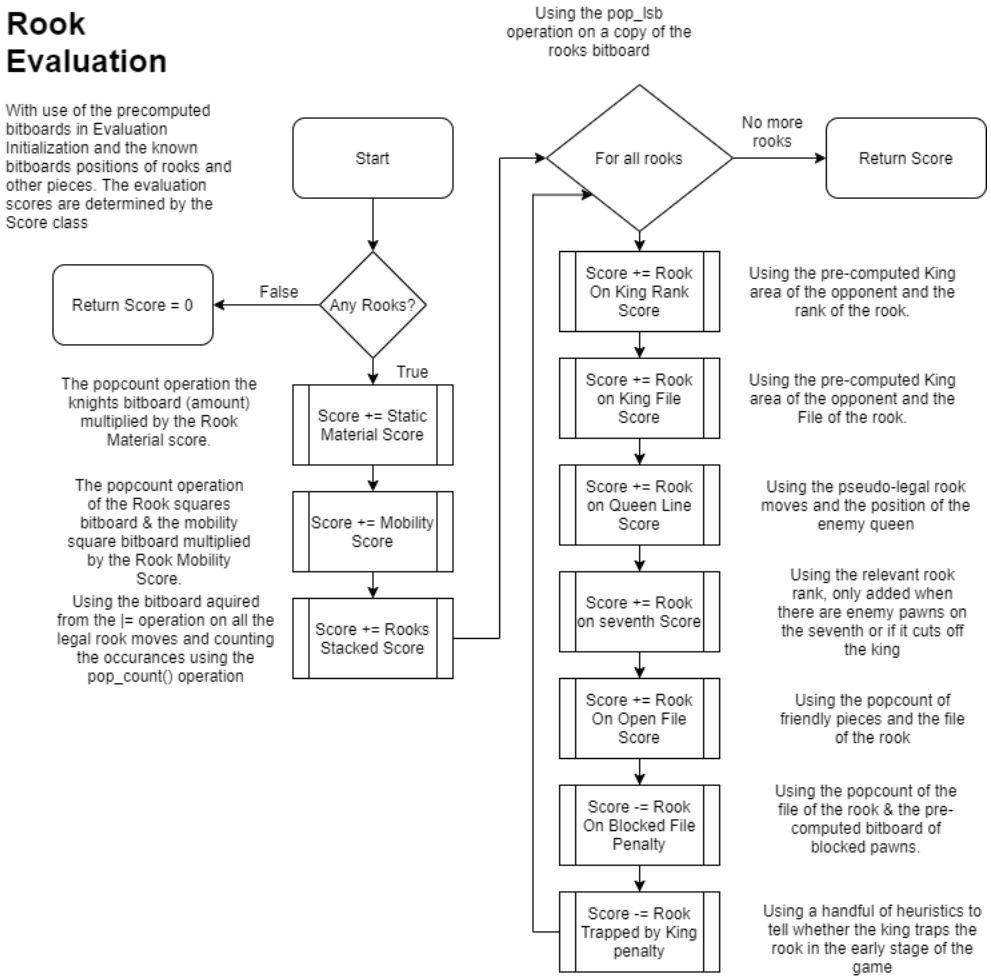

Fig. 22. Rook evaluation

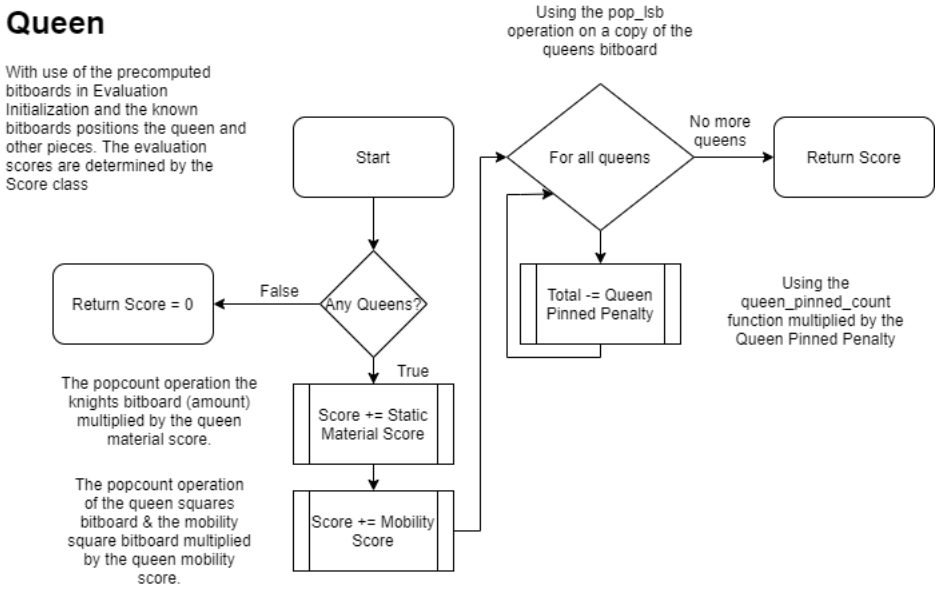

Fig. 23. Queen evaluation

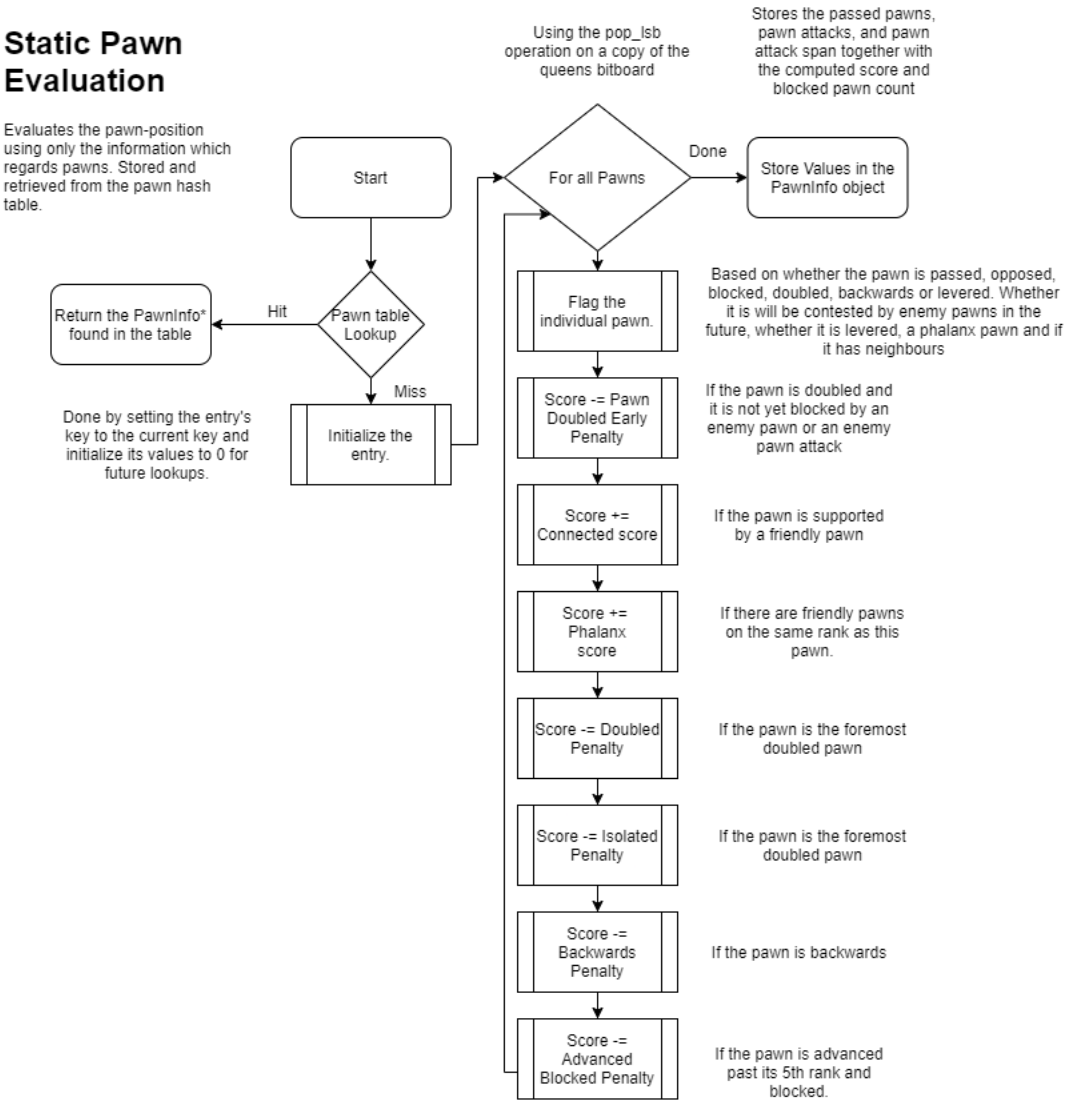

Fig. 24. Pawn evaluation

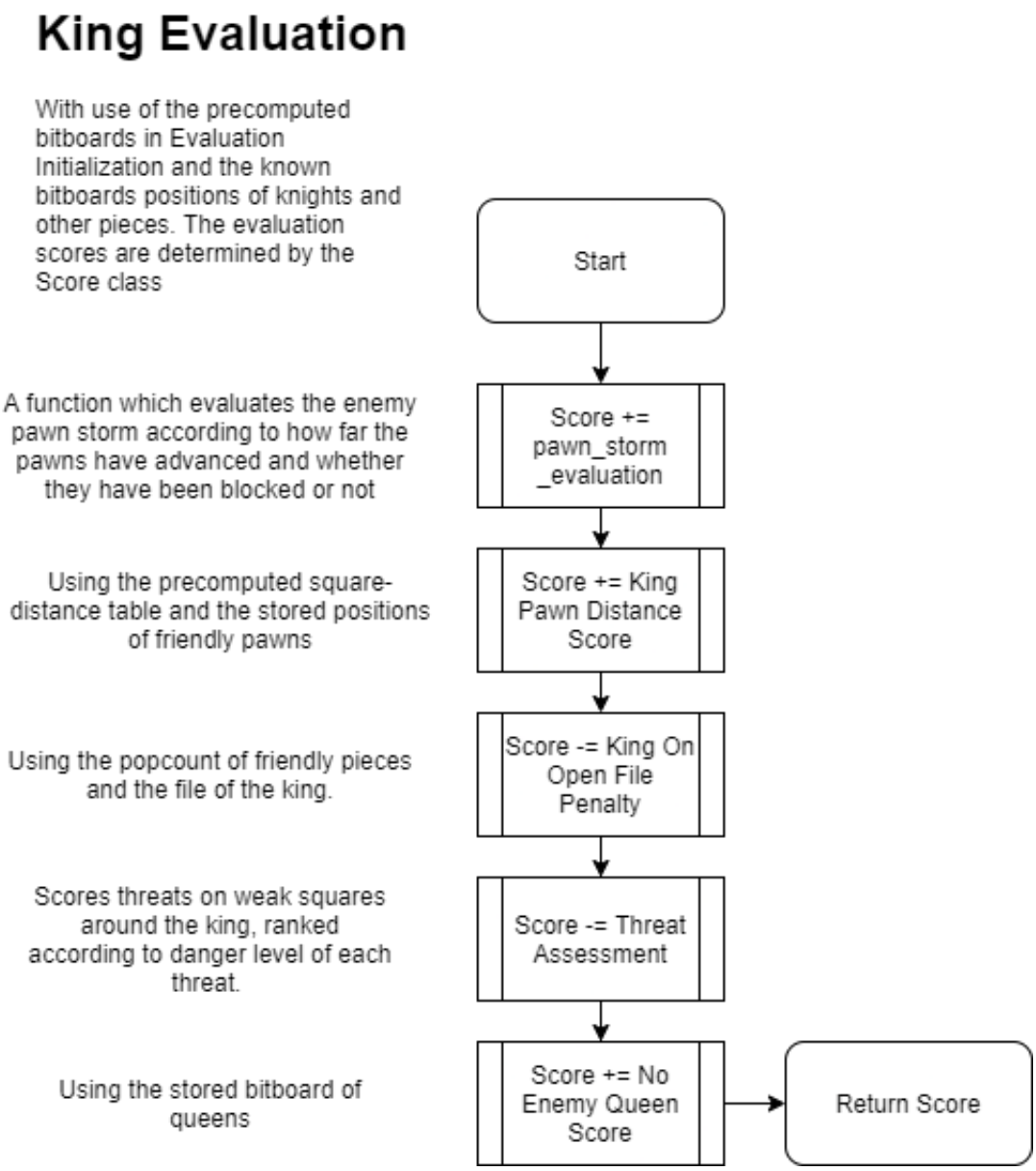

Fig. 25. King evaluation

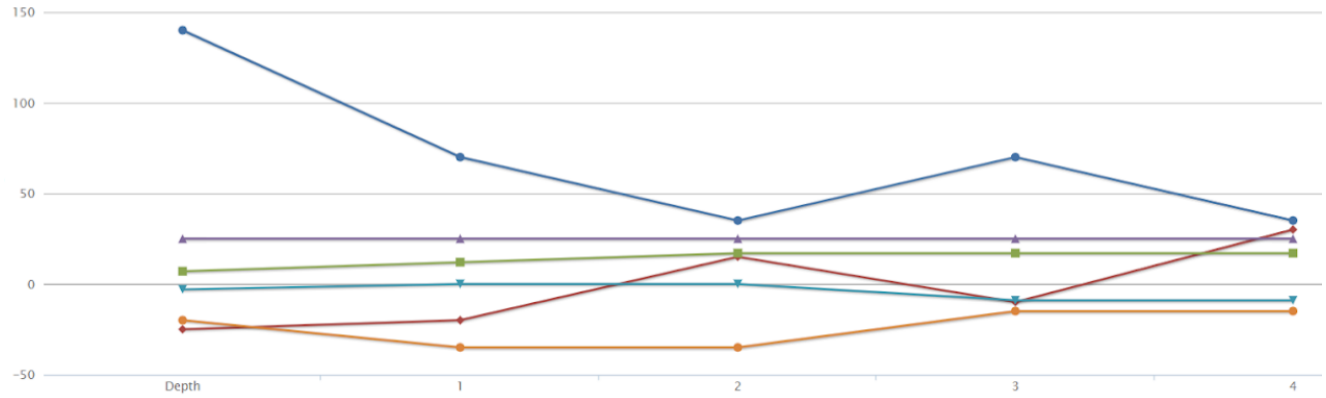

Fig. 26. Early - Change in parameters vs depth increase

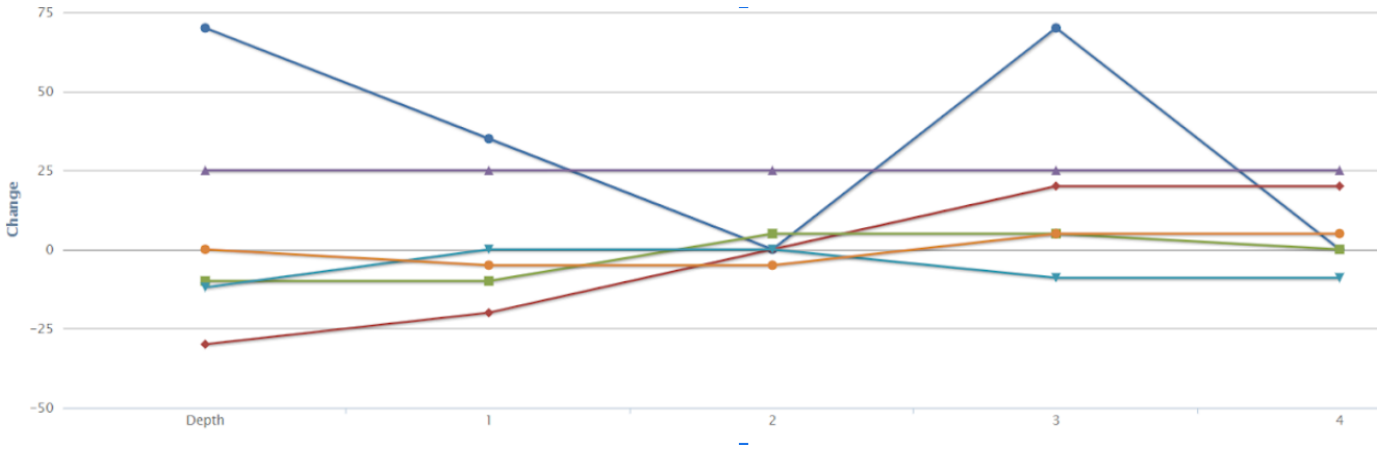

Fig. 27. Late - Change in parameters vs depth increase

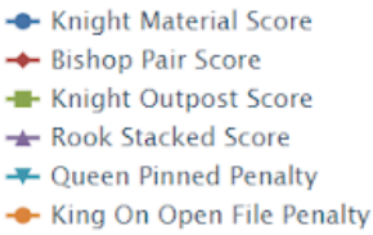

Fig. 28. Legend

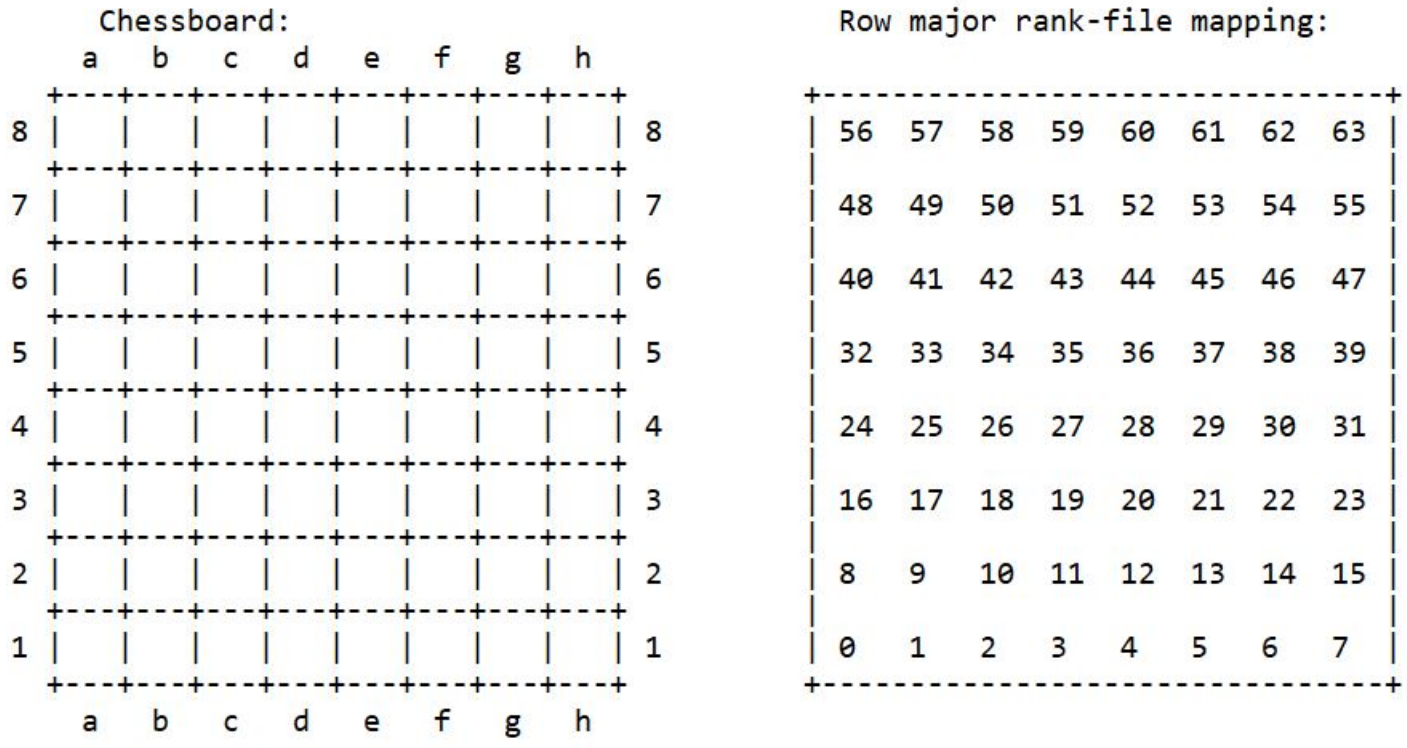

Fig. 29. Row major rank-file mapping of chessboard squares to array indices