

APPLYING THEORETICAL CLASSICAL CHESS RESEARCH TO CREATE A HUMAN-LEVEL “MONSTER CHESS” ENGINE

Aditya Gupta EPQ – Artefact

Contents

Project Brief	5
Time Log: Pre-Hiatus	6
Week 1: 4/1/21 – 10/1/21.....	6
Week 2: 11/1/21 – 17/1/21.....	6
Week 3: 18/1/21 – 24/1/21.....	7
Week 4: 25/1/21 – 31/1/21.....	8
Week 5: 1/2/21 – 7/2/21.....	9
Week 6: 8/2/21 – 14/2/21.....	10
Week 7: 15/2/21 – 21/2/21.....	11
Week 8: 22/2/21 – 28/2/21.....	11
Major Break: 29/2/21 – 5/9/21.....	13
Time Log: Post-Hiatus	14
Week 1: 6/9/21 – 12/9/21.....	14
Week 2: 13/9/21 – 19/9/21.....	17
Week 3: 20/9/21 – 26/9/21.....	18
Week 4: 27/9/21 – 3/10/21.....	21
Small Break: 4/10/21 – 17/10/21	25
Week 5: 18/10/21 – 24/10/21.....	25
Week 6: 25/10/21 – 31/10/21.....	27
Week 7: 1/11/21 – 7/11/21.....	31
Week 8: 8/11/21 – 14/11/21.....	33
Week 9: 15/11/21 – 21/11/21.....	34
Small Break: 22/11/21 – 12/12/21	38
Week 10: 13/12/21 – 19/12/21.....	39
Week 11: 20/12/21 – 26/12/21.....	40
Week 12: 27/12/21 – 2/1/22.....	43
Week 13: 3/1/22 – 9/1/22.....	44
Small Break: 10/1/22 – 6/2/22	44

Weeks 14 and 15: 7/2/22 – 20/2/22	44
Week 16: 21/2/22 – 27/2/22.....	44
Research.....	45
Coding Practices	45
How Chess Engines Fundamentally Work	45
Mailbox vs Bitboards.....	46
Magic Bitboards vs Obstruction Difference	48
The Minimax Algorithm	48
Shannon’s Type A vs Shannon’s Type B	50
Self-Made Static Evaluation vs Neural Static Evaluation.....	50
Python vs C++	51
wxWidgets vs Qt.....	52
Major Restructuring.....	53
Design	57
Algorithm Design.....	57
User Interface Design	60
Implementation	63
Language, User Interface and Portability	63
Move Generation	63
Board Representation.....	64
Lookup Tables	64
King Movement	64
Pawn and Knight Movement.....	66
Bishop, Rook and Queen Movement.....	67
Converting Bitboards to Chessboards.....	68
Checks and Combining Functions	70
Move Selection	72
Minimax (Negamax) Algorithm	72
Improvements – Alpha-Beta Pruning	72

Static Evaluation	72
Further Improvements.....	73
Testing	74
Move Generation Testing	74
User Interface Testing.....	75
Evaluation	76
Academic Results.....	76
Project Planning and Research.....	76
Programming Skills	76
Development Process.....	78
Final Artefact	78
Ease of Use	79
Conclusions.....	79
Bibliography	80
Appendix 1 – Presentation Slides	83

Project Brief

My Project involves building a chess engine to play a ‘variant’ of chess called ‘Monster Chess’. There are many variants of chess – chess-like games with altered rules. These altered rules lead to dramatically different play. Monster Chess was introduced to me by my father when I was very young. My project thus combines my love for programming and chess, with the desire to beat my father at the game he taught to me.

Monster Chess differs from chess in two crucial ways. White gets two moves for every one move Black gets, and as recompense, Black starts with their original set of pieces while White starts with only a king and four pawns. The power of the double-move king is what give Monster Chess its ‘Monster’ name – the Monster White King.

My engine consists of a move generator, which modifies traditional engine generators by generating legal moves for Monster Chess, and a move selector, which chooses what the engine thinks is the best move.

The engine is built in C++ and can be easily ported onto any device running Windows Operating System. I have also built a User Interface so that non-programmers can easily play against my engine.

This project tested my skills of time management, handling and building large codebases, using Object Oriented Programming (OOP) in C++, in addition to my analytical skills of modifying and creating new algorithms to apply them to new scenarios.

‘Monster Chess’ initially started as a way for me to be productive with the extra time that came as a result of the COVID lockdowns in 2021 – when schools reopened, I realized I had a prime opportunity to explore the area in more depth by doing an EPQ around the topic. As a result, some of my evaluation and log has been done out of order and/or retrospectively.

Time Log: Pre-Hiatus

Week 1: 4/1/21 – 10/1/21

My project initially started as a way for me to be productive with the extra time that came as a result of the COVID lockdowns in 2021 – when schools reopened, I realized I had a prime opportunity to explore the area in more depth by doing an EPQ around the topic. However, many of the initial steps I took were very much in line for an EPQ regardless of how formalized the process was.

After some thought, I decided that I wanted to try and create a ‘chess engine’ that could play a variant of chess called ‘Monster Chess’. This project combined my love for programming and investigating algorithms with my passion for chess and its variants. I also wanted to create an algorithm to beat my father in the game he taught to me.

The first thing I had to investigate was how modern chess algorithms worked on traditional chess. I would then have to explore how to apply these ideas to chess variants in general and Monster Chess specifically.

Week 2: 11/1/21 – 17/1/21

This week I investigated various modern state-of-the-art chess engines such as Stockfish and Leela Chess Zero. Some of the most powerful modern chess engines are open source whose code can be freely viewed.

Through my research into these engines, I was able to get a broad outlook on how chess engines are built. Primarily, chess engines go through two distinct phases – move generation and move selection. I realized I would have to consider both these aspects carefully when translating from traditional chess engines to my own Monster Chess engine.

However, I also realized that current state-of-the-art engines are quite complicated and have very large codebases. It didn’t seem feasible or useful to try and analyze the inner workings of these extremely strong engines before first getting a general overview for how various algorithmic techniques have developed over time. I decided to do this research first.

Sources:

https://www.chessprogramming.org/Main_Page

<https://www.chessprogramming.org/Mailbox>

Week 3: 18/1/21 – 24/1/21

This week I investigated the two main phases I had discovered last week – move generation and move selection. I found a great resource meant for budding chess program developers - the website chessprogramming.org. This website explains various concepts and techniques related to programming chess engines at a level which is complex enough to be of practical use but is more than just reams of code (as the source code for Stockfish and other engines was). Though primarily intended for traditional chess, I feel this site will become an invaluable resource as a starting point for my research.

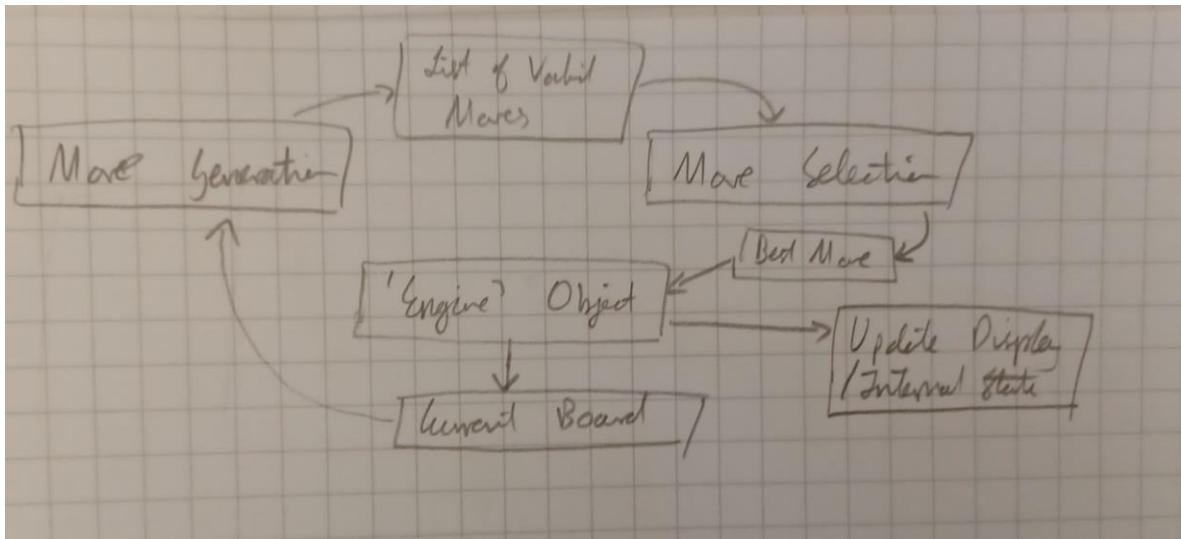


Figure 1- The base model for how my engine will work. An 'engine' will look at the current board, generate the possible legal moves, select the best move, and then update the display to show which move it has selected.

Sources:

https://www.chessprogramming.org/Main_Page

<https://www.stmintz.com/ccc/index.php?id=19924>

Week 4: 25/1/21 – 31/1/21

This week I decided to focus more on the practical aspect of how my project was going to be built. I had two issues – which programming language to use and getting some intuition on how to begin my engine (the ‘boilerplate code’).

The two languages I was considering were Python and C++, since they were the two that I had some familiarity with. I was very comfortable with Python as a programming language, but recently I had been using C++ more. I was impressed by how, once code had been written and compiled, C++ was incredibly fast at executing said code – often running up to ten times as fast as the interpreted Python code ran.

However, I also realized that, in order to maintain a large codebase over a long period of time, I would have to make sure I had a good fundamental understanding of the structure of what I was doing. As a result, I decided to combine both these potential options and initially use Python to create a very basic first prototype of a chess engine. My aims are to make sure the move generation of this program is valid in traditional chess, before considering whether to extend it to Monster Chess.

I then intend to create my actual Monster Chess engine in the compiled C++, having hopefully learned some lessons via the first prototype in Python. I began to have a clearer shape for the direction my project was going in.

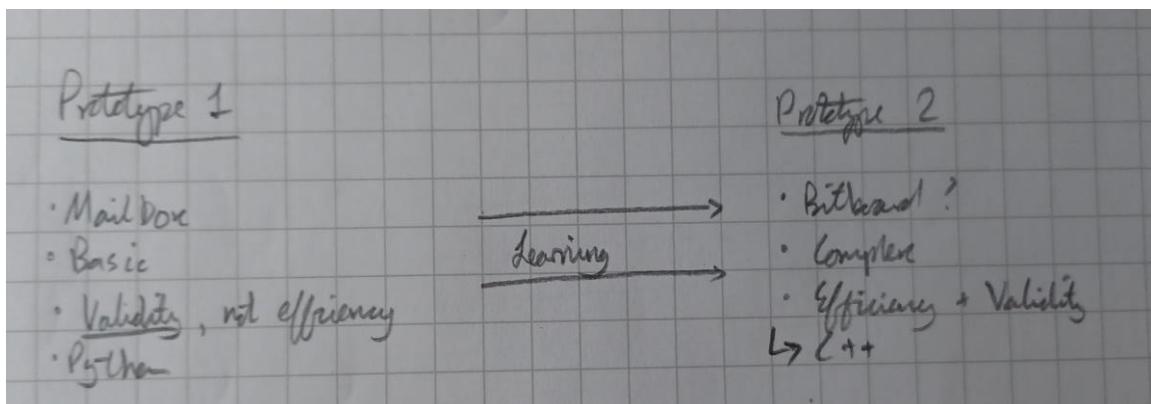


Figure 2 - The path I intend to go on. Prototype 1 will hopefully provide invaluable lessons for Prototype 2.

I realized that I would have to do a lot more research on the specifics of move generation and move selection before beginning either of my prototypes.

Sources:

https://www.chessprogramming.org/Board_Representation

<https://www.chessprogramming.org/Bitboards>

<https://realpython.com/python-vs-cpp/>

Week 5: 1/2/21 – 7/2/21

This week I finalized my ideas for where my project would go by outlining my aims for how testing and evaluation of my eventual final project would go. I am aiming to create a simple User Interface alongside the algorithm itself, so that it is easier for other people to play against my engine.

I also further researched traditional chess algorithms and came across the idea of a ‘mailbox’ chess engine. In mailbox engines, pieces are represented via objects and classes through a coding paradigm called ‘Object Oriented Programming’ (OOP). I have been interested in OOP for a while now as it seems a natural way to further my coding skills, and so I decided to do more research into potentially using it in my first prototype.

I discovered inheritance and how useful it can be when it comes to repeated properties between objects. I visualized this as so:

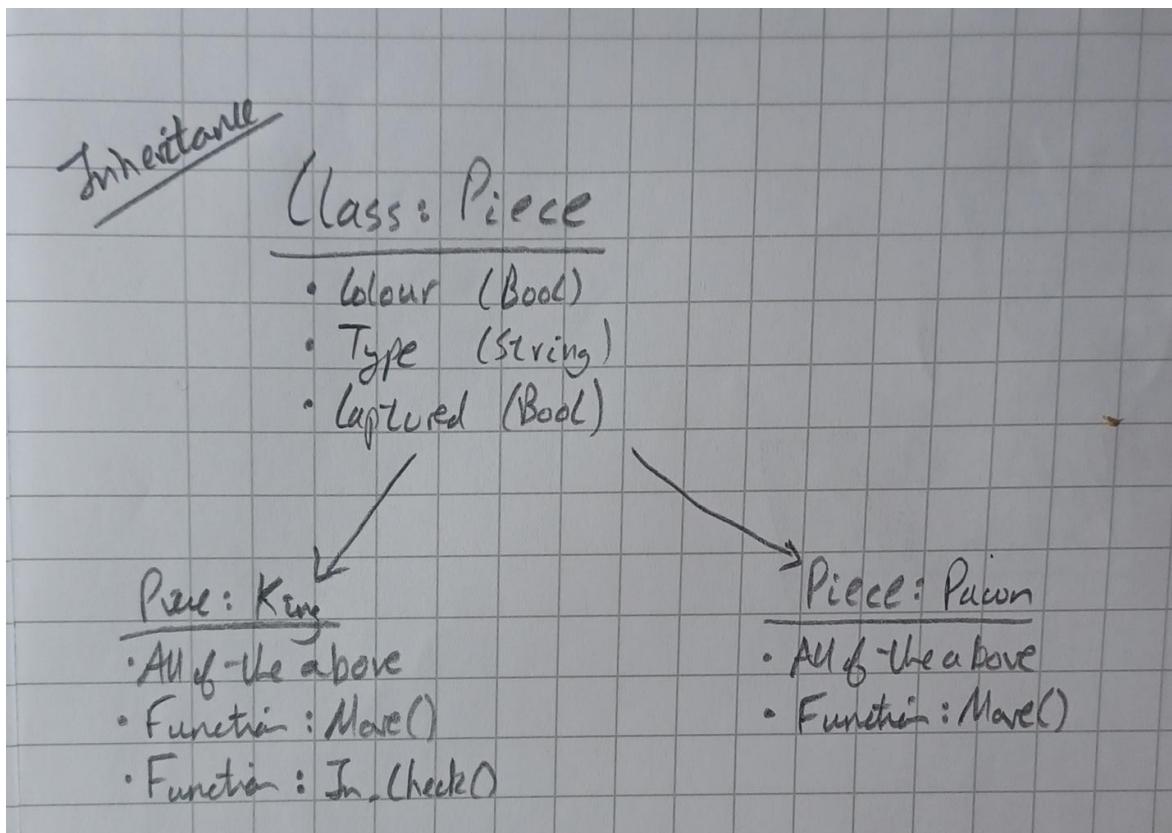


Figure 3 - Visualising inheritance. By exploiting commonalities between different pieces, I can avoid rewriting the same code over and over.

Sources:

<https://www.chessprogramming.org/Mailbox>

Week 6: 8/2/21 – 14/2/21

This week I decided to embark on my first prototype. I had done some more research into other methods of board representations, some of which seemed to be potentially a lot more efficient (such as bitboards, a system in which several 64-bit integers represented various parts of the chessboard). However, they also seemed to be a lot harder to implement and would require more research. As such I decided to leave them for now but first gain some understanding by implementing Prototype 1 with a mailbox board representation.

Using my research on OOP and inheritance, I created 3 major classes – Board, Player, and Piece. The interactions between these three classes and some special functions would be the basis of my mailbox engine in the first prototype. I planned out Prototype 1 to be structured something like this:

There are a few key parts of this plan. First is the crucial ‘Board’ class. Upon initialization, a Board object will create an 8 by 8 2-dimensional array containing the relevant pieces for a game of chess. The Board object will then store this in the current_board variable. Two ‘Player’ objects will be created, whose primary function is to select moves when given a Board variable. In doing this, it must make use of the ‘Piece’ class, to see what potential moves could be made. It must then check which of the moves is legal, and pick one to play out of these,

This selection is then relayed back to the Board object which makes the final move on the board. This should continue until the game ends.

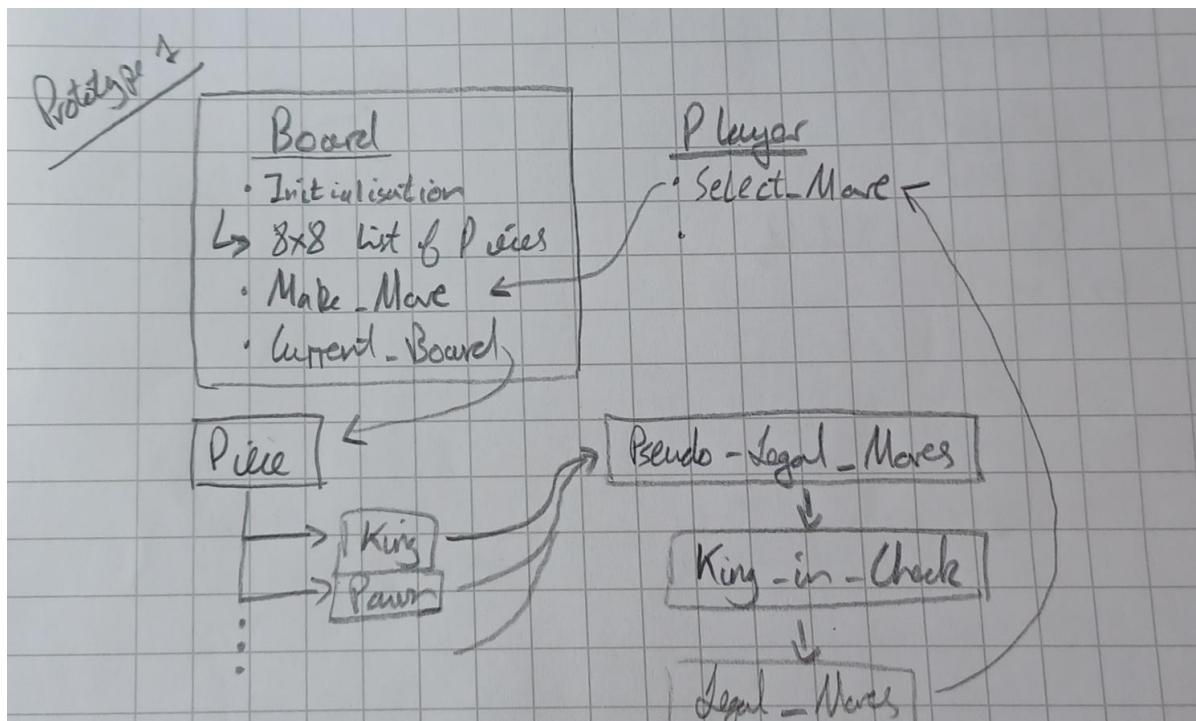


Figure 4 - The model for Prototype 1. The ‘board’ class deals with the board as it is and contains a ‘make_move’ function. A particular ‘player’, with their own ‘select_move’ function, will (by considering all legal moves) relay its preference to the ‘board’, which then updates the list ‘current_board’, containing all the pieces currently on the board.

Sources:

https://www.chessprogramming.org/Main_Page

<https://www.chessprogramming.org/Mailbox>

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/rep.html>

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/index.html>

Week 7: 15/2/21 – 21/2/21

This week I continued to work on Prototype 1. Last week I managed to set up the Board and Player classes rather simply. However, occasionally issues and bugs crept into my implementation – for example, at some points my program would not be able to access the crucial ‘game_state’ variable of the board which described the current position of the board. As such, I decided to create additional helper functions which would help me debug my code. For example, I defined the `__repr__` function of the Board class so that the pieces of the board would be conveniently printed out when I wanted to view the contents of a board. By doing so I was able to fix the bugs that arose with getting my Player objects to talk with my Board object.

However, I still had a few major things to accomplish in order to fully complete Prototype 1. I needed a way to generate and verify moves, and I needed a way for the Player object to select a ‘best’ move.

Sources:

https://www.chessprogramming.org/Main_Page

<https://www.tutorialsteacher.com/python/repr-method>

Week 8: 22/2/21 – 28/2/21

I started this week by completing the ‘Move()’ function for each possible chess piece. This process was quite complicated and involved a lot of debugging and stress testing. For example, it is crucial to

consider literal ‘side-effects’ when trying to move chess pieces a certain number of squares. Consider the diagram below that shows this effect in action:

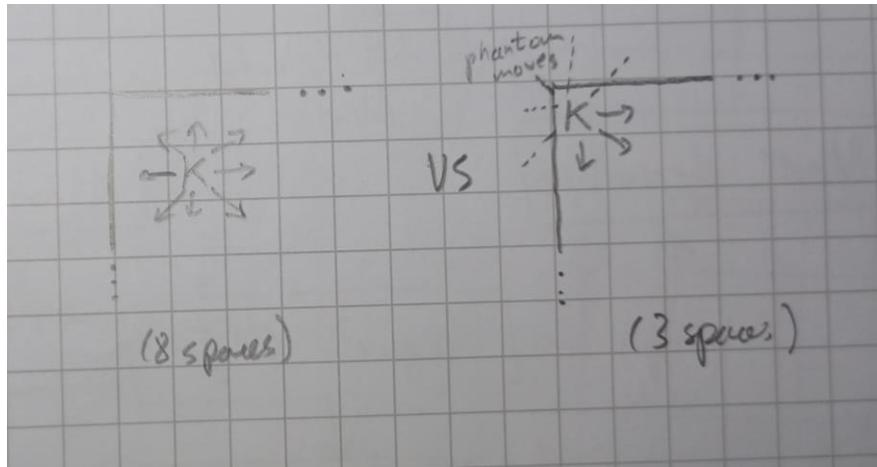


Figure 5 - How 'phantom', illegal moves were accidentally created. In most cases the King has 8 legal moves; however, when it is on the edge of the chessboard, it is much more restricted in its choice.

I needed to make sure I wasn’t accidentally adding ‘phantom’ moves where these moves didn’t exist. This led to some rather amusing bugs in which kings seemed to teleport across the board instead of moving one square! In the end I was able to resolve these bugs by making use of the debugging tools I had created last week. I then used the following structure to resolve ‘pseudo-legal’ moves into ‘legal’ moves:

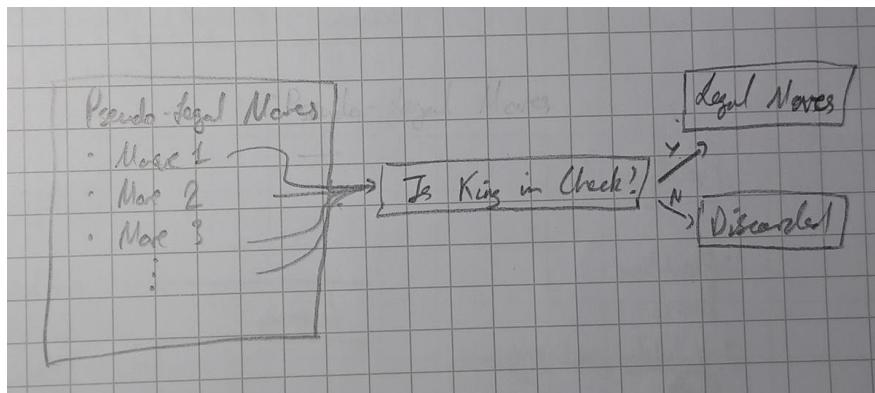


Figure 6 - The decision process for how to check which of the possible moves are legal within the rules of Monster Chess. Illegal moves are discarded.

The next thing I had planned was to have my Player object choose a ‘best’ move for the game to progress. I had done some research into how different selection algorithms work in chess and I realized that the effectiveness of different algorithms can vary wildly with even small changes to the rules of the game. As such I decided to postpone the decision of which exact selection algorithm to use until Prototype 2, as that was when I had planned to implement the specific rules of Monster

Chess. I instead implemented a random move selector for both Player objects in order to verify that Prototype 1 was indeed working. I did come across quite a major issue, however. While the program did successfully work for a couple moves in the opening, when the number of possible moves is quite low, the exponential nature of how many possible sequences of moves are playable combined with the memory-intensive code I had written meant that after just a few moves my program crashed. This was something I had to seriously consider going forwards.

Sources:

https://www.chessprogramming.org/En_passant

https://www.chessprogramming.org/Move_Generation

Major Break: 29/2/21 – 5/9/21

While my project had begun as an outlet for me during the COVID lockdowns, at this point I realized that I could more fully explore this area if I dedicated some time to it within the school system. After talking to the teacher that runs EPQs at my school, the best course of action seemed to be to temporarily pause the project, and continue in September, after my GCSEs had concluded and when I could more properly document my progress. As such I decided to take a break and focus on my school studies, returning to the project in September. Furthermore, I was at quite a natural break-point in my development of the project, having just completed Prototype 1, and it seemed a sensible point in time to put the project on hiatus.

Time Log: Post-Hiatus

Week 1: 6/9/21 – 12/9/21

This week I resumed my EPQ project, having put it on hiatus several months ago. I first had to review exactly what I had already achieved and what goals seemed reasonable yet ambitious for my project. I completed my project form proposal keeping these ideas in mind. I then started out reviewing Prototype 1, as I had planned to before moving on to the main Prototype 2. While I did some things well, there were several key areas in need of improvement before I could move on:

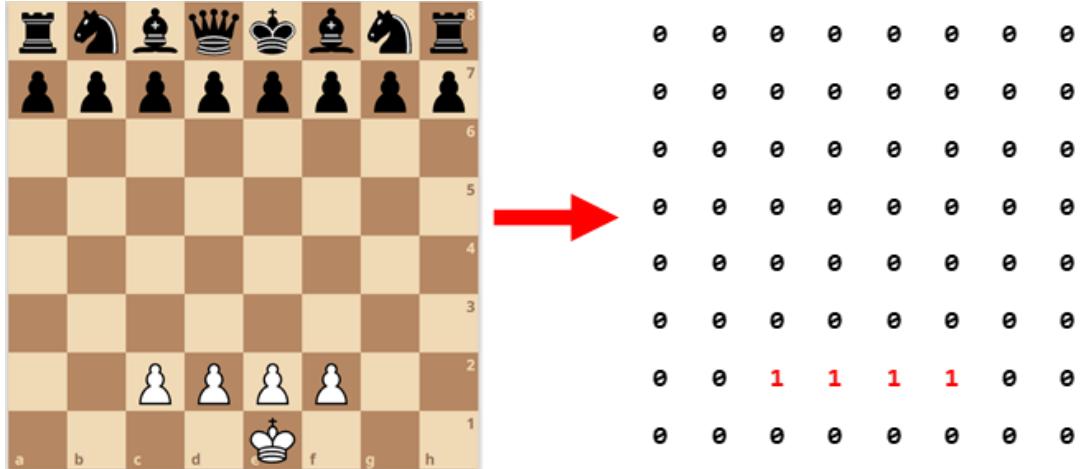
Lessons learned from Prototype 1 – Areas of Efficiency

1. Maintaining a few classes is much better than having to deal with a lot of individual smaller classes, and as such using inheritance is not a bad idea. This is because inheritance simplifies boilerplate code and makes implementing functions for various objects less mentally taxing, in turn leading to fewer bugs.
2. The idea of generating legal moves for white recursively, via a ‘one_move_ignoring_check’ function and a ‘is_player_in_check’ function, is a very good way to avoid rewriting a lot of code.
3. Having a separate driver file to run the entire game made debugging a much easier experience.

Lessons learned from Prototype 1 – Areas in need of Improvement

1. A ‘Mailbox’ approach, where each square of the board (and in my case each piece on the board) is represented by itself as an independent object, is incredibly inefficient, as the computer has to treat each square as a full object (as opposed to as a simple placeholder), leading to a much slower speed of execution.
2. Any computationally efficient approach (that is, any approach that runs quickly) should carefully consider how pseudo-legal moves are verified; this is a massive bottleneck in Prototype 1.
3. Once memory is no longer being used, make sure to free it up, in order to avoid any of the computer-crashing issues I had come across in Prototype 1.
4. It’s unnecessary, and quite inefficient, to have each piece exist as a custom object – I needed to research better ways of representing pieces.
5. I needed to do more type-hinting, especially for complex objects – often I was confused with regards to the format of the parameters my functions were taking in. Python doesn’t provide this by default, so switching languages seemed to make sense.

Having made these observations, I did additional research on other, non-mailbox types of chess engines. In the end I decided to use a coding construct called a ‘bitboard’. Bitboards are essentially 64-bit integers that tell us whether a piece is on a particular square. For example, the ‘WhitePawns’ bitboard looked like this at the beginning of a game:



I had properly digested the concept of the bitboard and had created enough bitboards to represent every piece in the Monster Chess, but I still needed some unifying structure to efficiently deal with different chess positions. As such I decided to reuse some of the ideas from Prototype 1 regarding OOP, and I created a `ChessBoard` class. In the image below I worked out some of the key methods and attributes I wanted the `ChessBoard` class to have:

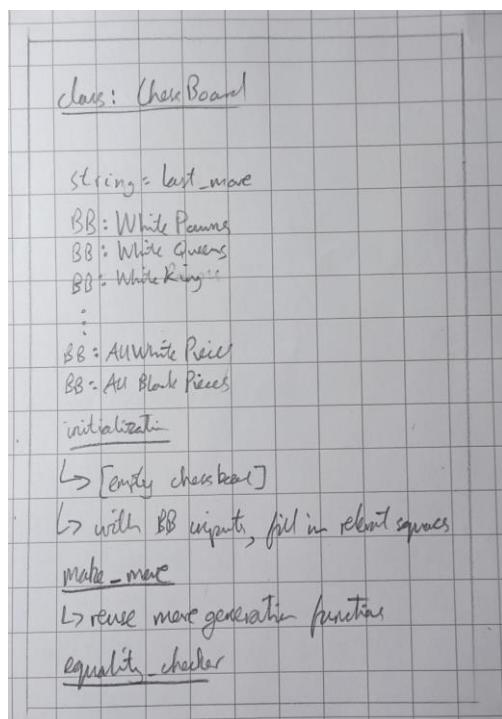


Figure 7- The structure of the class "ChessBoard". Important functions include initialisation, the 'make_move' function, and checking whether two `ChessBoard` objects are equal. Each piece's position is stored in a bitboard.

For next week, I intend to work on bitboard logic and move generation.

Sources:

https://www.chessprogramming.org/General_Setwise_Operations

https://www.chessprogramming.org/Population_Count

<https://www.chessprogramming.org/BitScan>

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/rep.html>

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/index.html>

Week 2: 13/9/21 – 19/9/21

I started out this week exploring how bitboards could be very efficient. Computers are very efficient at working with integers, and there is a happy coincidence in the fact that computers use 64-bit integers while the chessboard has 64 squares. Logic functions, such as OR, AND and NOT, can be cleverly used on bitboards to calculate possible moves in a position. Using the structure from the ChessBoard class I had created last week, I began to implement several key functions which allowed for move generation. However, I soon encountered quite a large issue regarding edge cases once again, as illustrated in the diagram below:

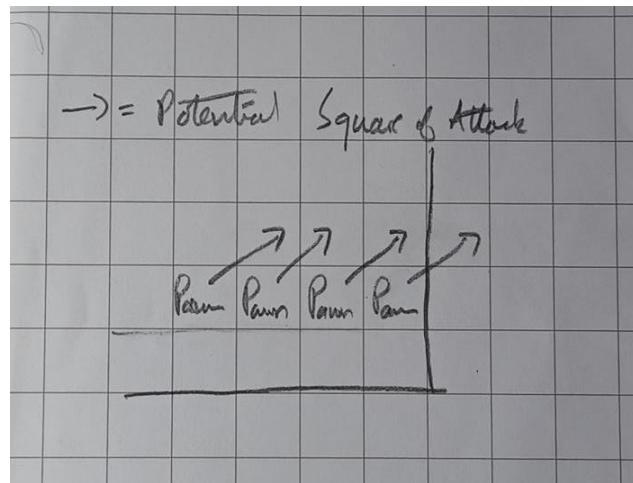


Figure 8 - One example of the so-called 'attack bitboards'. By considering all possible pawns, we can consider all the possible squares they could attack, masking out those which would be beyond the board.

Essentially, I had a similar bug as had occurred in Prototype 1 – I needed an efficient way to check if pieces were on the edges of the board, since this severely restricted their movement. In Prototype 1 I had managed to overcome this bug by utilizing if statements and conditional functions to make sure, for example, that kings weren't escaping the board. However, as I had discovered, this made move generation quite inefficient. I then remembered that, in my research on modern chess engines and

bitboards, I had come across the idea of lookup tables, that stored many often-used bitboards. These bitboards could then be applied to piece bitboards before doing any move generation calculation in order to avoid edge conditions altogether:

Clear File [FILE_H]:							
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
White Pawns							
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
White Pawns N2 On Hfile							
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 9 - By using precomputed standard bitboards (such as, in this case, the clear_file_h bitboard), we improve the system's memory efficiency in creating new bitboards.

Next week I intend to complete pawn, king and knight move generation, which are quite similar in structure to each other.

Sources:

https://www.chessprogramming.org/General_Setwise_Operations

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/physical.html>

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/nonsliding.html>

Week 3: 20/9/21 – 26/9/21

I started the week by finalizing the ‘compute_king’ function, that returns a bitboard with potential squares the king could go to as 1s, and squares the king couldn’t go to as 0s:

```
BB compute_king(const BB &king_loc, const BB &own_side){

    BB king_clip_file_h = king_loc & ClearFile[FILE_H];
    BB king_clip_file_a = king_loc & ClearFile[FILE_A];

    BB spot_1 = king_clip_file_h << 7;
    BB spot_2 = king_loc << 8;
    BB spot_3 = king_clip_file_a << 9;
    BB spot_4 = king_clip_file_a << 1;
    BB spot_5 = king_clip_file_a >> 7;
    BB spot_6 = king_loc >> 8;
    BB spot_7 = king_clip_file_h >> 9;
    BB spot_8 = king_clip_file_h >> 1;

    BB king_moves = spot_1 | spot_2 | spot_3 | spot_4 | spot_5 | spot_6 |
                    spot_7 | spot_8;
```

```

    BB KingValid = king_moves & ~own_side;

    return KingValid;

}

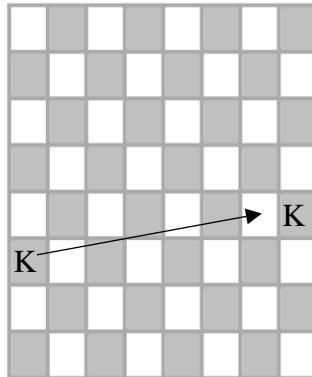
```

My function takes in the position of the king and a bitboard of all the pieces of the king's own colour - either AllWhitePieces or AllBlackPieces.

The first two lines create two additional bitboards which will contain the position of the king as long as it isn't on one of the two edge columns. If it is, these bitboards will just be all 0s.

Now I had to make yet more bitboards to move the king to the various spots. Note that the “<<” and “>>” simply shift the entire bitboards to the left or right, which has the effect of moving the king itself to the left or right. Since shifting by 8 moves the king up or down, shifting by 7 leads the king to spot_1, shifting by 8 to spot_2, etc.

I noticed that when considering spots not directly above or below the king, I had to consider the fact that the king might be on an edge row or column. If so, shifting the king by 1 has the effect of moving the king to the opposite side of the board, like so:



As such, I had to only shift the king 1 to the left if it wasn't in the first column to begin with. A lot of my initial bugs with king movement had to do with this type of error – at times it seemed like the King was teleporting across the board! A similar logic holds for the other edge column. Therefore, I use the ‘king_clip’ bitboards I already made to generate potential spots where the king could end up. I didn't have to worry about the same issue when moving up or down entire rows (i.e. spots 2 and 6) since 64-bit integers automatically discard ones if they move past the 64 bits.

Finally, I used the | operator (logical OR) to combine all the spots into one final bitboard that the King could move to. I then used & (the AND operator) and ~ (the NOT operator) to discard all the spots

where there are pieces of the same colour as the king (pieces of the same army cannot capture each other in Monster Chess, or in Classical Chess).

I used similar processes to create the move generation functions for pawns and knights, since like kings they can't be obstructed by other pieces.

Sources:

https://www.chessprogramming.org/General_Setwise_Operations

https://www.chessprogramming.org/Move_Generation

https://www.chessprogramming.org/Sliding_Piece_Attacks

Week 4: 27/9/21 – 3/10/21

This week I worked on move generation for the so-called ‘slider’ pieces – the bishop, rook and queen. I knew from my research that the queen’s move generation would be relatively easy after I had taken care of the bishop and rook, since its moves are simply a composite of the two others.

While, conceptually, pawn and king movement was not too tough to understand, ‘slider’ pieces are a lot more difficult to efficiently compute. Unlike with the knight or king, we can’t ignore pieces in between our start and end square – we must instead make sure there are no pieces in our way. This problem stumped me for quite a while – there seemed to be no easy way of using the efficiency of bitboards to get around checking, say in the case of the bishop, each diagonal square on the path of the bishop to make sure no other piece was in the way. After doing some more research into the concepts I had come across in my initial research, I looked at several methods that claimed to be able to efficiently compute rook and bishop moves.

Magic Bitboards

This was the first option I came across and seems to have become the industry standard in modern chess engines within the last decade or so. From what I was able to gather, Magic Bitboards come in several different flavors, but they rely on some type of hashing algorithm to index an attack bitboard database. In a sense, magic bitboards rely on even larger Lookup Tables than as discussed before, and then convert real chess positions through a complicated (but efficient) hash function to some index which can be searched on this large database. An added benefit is that the diagonal lines which bishops use can be indexed as easily as the rook’s straight lines of attack.

I spent many hours researching exactly how Magic Bitboards worked – this mostly involved reading a *lot* of pseudocode and staring at many logical operations on 64-bit integers. However, not only did it seem a lot more complex than other potential options, but it also seemed to require a lot of memory as compared to other methods. Given the fact that my computer had already crashed with Prototype 1, most likely due to memory issues, I decided to research other options.

Bit-Twiddling Methods

More research led me to various so-called ‘bit-twiddling’ methods – ‘Hyperbola Quintessence’, ‘Obstruction Difference’, and ‘Exploding Bitboards’. These methods use complicated but efficient techniques to resolve which squares a bishop or rook could move to. All these methods have a memory-speed tradeoff; generally, the more efficient they are, the more memory they take. But in the end, I decided to go with Obstruction Difference, as it seemed to take relatively little memory and had good documentation.

Obstruction Difference

Let's look at the critical function 'lineAttacks'.

```
BB lineAttacks(const BB &const_occ, const int piece_pos, const int direc) {  
    BB occ = const_occ & sliding_piece_masks[piece_pos][direc];  
  
    BB temp = ~((1ULL << piece_pos) - 1);  
    BB upper = (temp) & occ;  
    BB lower = ~temp & occ;  
  
    BB MS1B = -1ULL << bsr(lower | 1ULL);  
    BB LS1B = upper & -upper;  
  
    BB odiff = (2 * LS1B) + MS1B;  
    BB to_return = odiff & sliding_piece_masks[piece_pos][direc];  
  
    return to_return;  
}
```

First, we take in the current occupancy of the board, as well as the position of the piece in consideration (rook, bishop or queen) and an integer which represents a direction. This direction could be horizontal, vertical, or one of the two diagonals.

I've already precomputed a rather large table called 'sliding_piece_attacks'. This 2d array has dimensions 64 by 4 and contains the squares a piece could hit in a particular direction from a square, if the board were empty. For example, sliding_piece_attacks[16][0] refers to the vertical squares a rook on the square h3 could hit:

0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1

As such the bitboard 'occ' contains the pieces which are in the line of sight of a particular piece in a particular direction.

Next, the variable ‘temp’ is a way of splitting the occupancy bitboard into an upper and lower half. Essentially, it has a value of 1 for every position higher than the original position of the rook and a value of 0 for every position lower. For example:

0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

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
0	0	0	0	0	0	0	0	0

Thus we can use this ‘temp’ variable to now generate the ‘upper’ and ‘lower’ variables, by also using the current occupancy map. Say the full occupancy looked like this, with the rook on h3:

1	1	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	R
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	0	0

Then our temp variable creates ‘upper’ and ‘lower’ as such:

temp	upper	lower
1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 R	0 0 0 0 0 0 0 0 R	0 0 0 0 0 0 0 0 R
0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0

Now the next step is to select all the squares in between the lowest bit of the upper bitboard and the highest bit of the lower bitboard. This is done via classic techniques of bit-scanning backwards and an efficient bit trick². Finally, we AND this ‘odiff’ bitboard together with the original array entry for potential squares to get rid of 1s in other directions, giving us a final valid bitboard.

<i>odiff</i>	<i>sliding_piece_masks[piece_pos][direc]</i>	<i>to_return</i>
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0

AND



Simply by varying the integer ‘direc’, we can do the same for the three other directions, thus giving us valid bitboards for bishop, rook and queen moves.

The hard part had now been done; all that was left was sending the input in a nice fashion to combine several directions per piece. For the rook, this looked like this:

```
BB one_rook(const BB &black_rook_loc, const BB &all_pieces, const BB &own_side){

    BB horiz = (lineAttacks(all_pieces, findPosition(black_rook_loc), 0) &
~own_side);
    BB vertic = (lineAttacks(all_pieces, findPosition(black_rook_loc), 1) &
~own_side);

    return horiz | vertic;

}
```

The ‘one_rook’ function just computes the horizontal and vertical potential moves via the ‘lineAttacks’ function, makes sure that no captures are in fact captures of their own side, and then returns both the directions as one valid bitboard.

Sources:

https://www.chessprogramming.org/Obstruction_Difference

https://www.chessprogramming.org/Magic_Bitboards

https://www.chessprogramming.org/Hyperbola_Quintessence

https://www.chessprogramming.org/BitScan#Divide_and_Conquer_2

https://www.chessprogramming.org/General_Setwise_Operations#TheLeastSignificantOneBitLS1B

Small Break: 4/10/21 – 17/10/21

For two weeks, I had to effectively put the EPQ on hold as a result of internal assessments which required my full attention. I returned to the EPQ as soon as I could, when half-term arrived.

Week 5: 18/10/21 – 24/10/21

This week, I planned to finish move generation. To wrap things up I needed to make use of the ChessBoard class I had previously created, alongside ideas involving legal and pseudo-legal moves.

Here I could reuse some of what I had learned while making Prototype 1 – namely, the idea of having a function called ‘one_move_without_check’ and then using this as a building block for white and black’s potential legal moves. Here’s the code for that function, building on the individual component’s I had already built:

```
void one_move_without_check(vector<ChessBoard> &pot_moves, const ChessBoard
&start_pos, const int color) {
    if (color==0) {

        BB black_king_moves = compute_king(start_pos.BlackKing,
start_pos.AllBlackPieces);

        createAllMoves(pot_moves, start_pos, black_king_moves,
findPosition(start_pos.BlackKing), BLACK_KING_INT);

        compute_knight(start_pos.BlackKnights, start_pos.AllBlackPieces,
pot_moves, start_pos, color);
        compute_black_pawns(start_pos.BlackPawns, start_pos.AllBlackPieces,
pot_moves, start_pos, color);
        compute_rook(start_pos.BlackRooks, start_pos.AllBlackPieces,
pot_moves, start_pos, color, BLACK_ROOK_INT);
        compute_bishop(start_pos.BlackBishops, start_pos.AllBlackPieces,
pot_moves, start_pos, color, BLACK_BISHOP_INT);
        compute_rook(start_pos.BlackQueens, start_pos.AllBlackPieces,
pot_moves, start_pos, color, BLACK_QUEEN_INT);
        compute_bishop(start_pos.BlackQueens, start_pos.AllBlackPieces,
pot_moves, start_pos, color, BLACK_QUEEN_INT);

    } else if (color==1) {
```

```

        BB white_king_moves = compute_king(start_pos.WhiteKing,
start_pos.AllWhitePieces);
        createAllMoves(pot_moves, start_pos, white_king_moves,
findPosition(start_pos.WhiteKing), WHITE KING INT);
        compute_white_pawns(start_pos.WhitePawns, start_pos.AllWhitePieces,
pot_moves, start_pos, color);

        compute_rook(start_pos.WhiteQueens, start_pos.AllWhitePieces,
pot_moves, start_pos,color, WHITE QUEEN INT);
        compute_bishop(start_pos.WhiteQueens, start_pos.AllWhitePieces,
pot_moves, start_pos,color, WHITE QUEEN INT);

    }

}

```

I split the function into two cases based on which colour's turn it was. After that, it was just a matter of calculating and appending each type of piece's moves to the 'pot_moves' vector.

I built upon this to build a couple additional functions:

```

void one_move_with_check(vector<ChessBoard> &pot_moves, const ChessBoard
&start_pos, const int color){
    vector<ChessBoard> toBecome;

    one_move_without_check(pot_moves,start_pos,color);
    for (auto x: pot_moves) {
        if (!is_someone_in_check(x,color)){
            toBecome.push_back(x);
        }
    }
    pot_moves = toBecome;
}

```

Here I just checked every resulting chessboard after a pseudo-legal move was made to see whether the colour whose turn has just passed is in check. If it isn't, I added it to a new vector, which then becomes the original vector. In this way I got rid of any illegal positions. A function called 'two_moves_with_check' is similarly defined, with some additional recursion to give white's potential moves.

While implementing these functions, I realized that the 'is_someone_in_check' function was a potential bottleneck. In its current state, I believe this function is what is causing the most inefficiency in my move generation function. I have a somewhat naïve implementation that looks at all potential pseudo-legal moves from the proposed position, checks that the king is not attacked in any of those cases, and then verifies a move as legal. This is computationally inefficient. In my research I did come across better methods, but I realized that these would require major restructuring of my code for

questionable benefit, so I left it to a later date. There were some details that I could relatively easily improve – such as avoiding taking too much memory – by changing parts of my code, and I did so.

In summation, here's my code for the function 'legal_moves':

```
void legal_moves(vector<ChessBoard> &pot_moves, const ChessBoard
&start_pos, const int color, bool to_sort) {

    if (color==0) {
        ChessBoard tempcopy = start_pos;
        one_move_with_check(pot_moves,start_pos,color);

    } else {
        two_moves_with_check(pot_moves,start_pos,color);
    }
}
```

I had to extensive testing to fix remaining bug fixes with my move generation, as laid out in the 'Testing' section. Next week I intend to start the implementation for move selection.

Sources:

https://www.chessprogramming.org/General_Setwise_Operations

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/physical.html>

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/nonsliding.html>

[https://www.chessprogramming.org/Checks_and_Pinned_Pieces_\(Bitboards\)](https://www.chessprogramming.org/Checks_and_Pinned_Pieces_(Bitboards))

Week 6: 25/10/21 – 31/10/21

This week I started by reviewing my research into different search and evaluation algorithms to choose the basic structure of my algorithm. I had to consider and balance a few things when considering which system to use:

- Effectiveness – which algorithm found the better moves?
- Efficiency – how long did each algorithm take?
- Ease of implementation – how feasible was it for me to implement a particular algorithm? For example, it seemed like it would be very difficult to replicate an AlphaZero, massive-scale neural network type algorithm as I don't have access to Google's resources.

I quickly narrowed down my search to focus on so-called ‘deterministic’ algorithms – these have been around since the 1970s when it comes to chess and seemed tried and tested. There are also a lot more resources to study these algorithms. The basic format of how these algorithms work is laid out below:

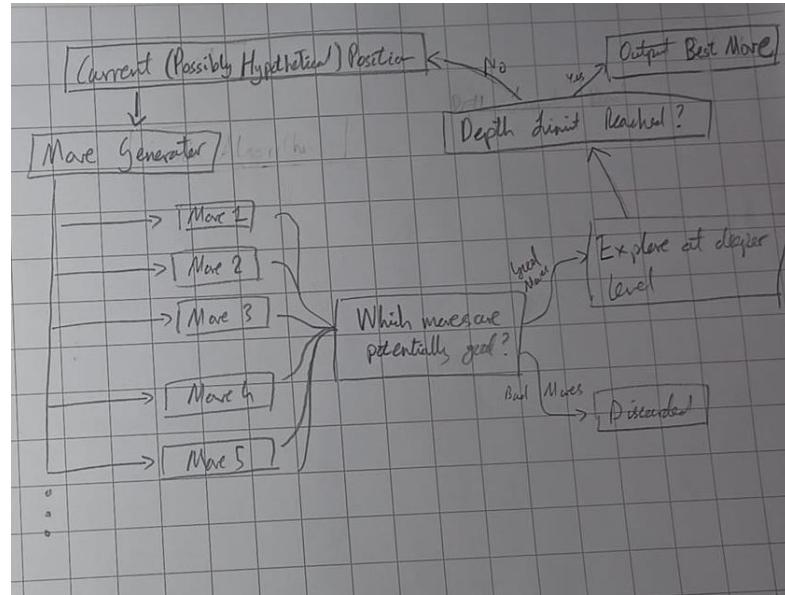


Figure 10 - A deterministic chess algorithm. Key features include the move generator and evaluation functions, as well as a maximum depth limit which ensures the program terminates by selecting a particular move.

The diagram shows how a typical deterministic algorithm would evaluate a chess (or Monster Chess) position. It uses the move generator to consider all moves in some position, classifies these via some heuristic function (normally related to classical chess ideas of assigning points to each type of piece) and then explores each of the ‘interesting’ moves in the same fashion by looping through the whole process again. The number of loops is called ‘depth’ – naturally, a higher depth leads to better search selection. Eventually, when a time or depth limit is hit, the algorithm exits the loop and outputs the best move it has found.

While I had the basic idea of what a deterministic algorithm was, there were still a lot of the details I needed to work out.

Deterministic chess algorithms are classified by “Shannon’s Type”. Claude Shannon was the father of information technology and a prominent mathematician and cryptographer, and was instrumental to early breakthroughs in the field of computer chess. He classified deterministic chess algorithms into two types:

Type A	Type B
Brute-force algorithm, will never ‘miss’ a move.	Selective search, may miss moves.
Can search positions to lower ‘depth’.	Can search positions to higher ‘depth’.
Easier to implement.	Tougher to implement, especially to variants of chess.

Type B algorithms require additional heuristics and strategies which software engineers can provide for chess because that game has been studied for centuries; humans know what the best types of strategies are. But in Monster Chess, there are no experts or grandmasters, and as such I decided to go with Type A algorithms, which are much safer in general.

The next key decision I had to make was between Depth-First and Breadth-First algorithms.

Search algorithms work in a somewhat complicated fashion, which can be illustrated via a search tree diagram (as I have shown in the ‘Research’ section). But principally, there are two types of chess search algorithms – depth first and best first.

A depth first algorithm selects one potential move and evaluates it to a certain ‘depth’ (as described before), and then moves on to the next potential move. Eventually after assessing all possible moves, this depth first algorithm will return whichever move it assesses as the best.

A best first algorithm is a bit more aggressive/ambitious – it will make a cursory judgement of all moves and rank them based on how promising they look. Then, based on how promising a move seems, it will evaluate those moves to a higher depth in the same manner. As a result, with best first algorithms, it is extremely important to have a good ‘judgement’ function, i.e. good heuristics.

Once again, I was faced with the fact that I did not know what good heuristics were in Monster Chess – I could make some guesses, but I didn’t want to potentially bias my algorithm to be worse than it could be. I decided to stick to a depth first algorithm.

Of the depth-first algorithms, the most widely used seemed to be the Minimax algorithm. Minimax algorithms work on the assumption that a good move for one player is a bad move for the other. Thus, say we could statically evaluate every chess position; assign a number to each position which was positive if white was winning, and negative if black was winning. White’s objective is to maximize this function, while black’s is to minimize it.

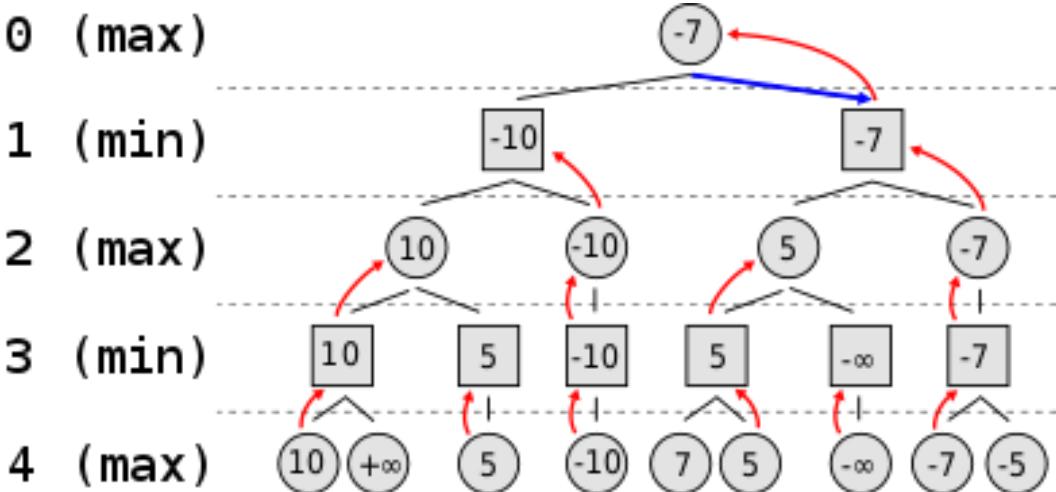
I describe in a bit more detail how minimax algorithms work in the ‘Research’ section, but it’s easiest to think of minimax algorithms as trying to guess what move our opponent will play next, assuming they play the best move, and assessing the resulting position. It goes without saying that a higher depth will mean more accurate position evaluators and thus a better chess engine.

Sources:

https://www.chessprogramming.org/Search#Shannon.27s_Types

<https://www.chessprogramming.org/Depth-First>

<https://www.chessprogramming.org/Best-First>



This week I began to implement a basic minimax algorithm. In my research, I had come across the idea of a ‘negamax’ algorithm, which used essentially the same, zero-sum, recursive ideas in the minimax algorithm, but which was significantly easier to implement as it treated both players (white and black) in the same way.

The pseudocode works like so:

```
int negaMax(int depth) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for (all_moves) {
        score = -negaMax( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
```

The code works recursively, first making sure we are not at a leaf node (the end of the tree in the diagram above), and if so, it returns a static evaluation. If we are somewhere in the middle of the search tree, then we consider all the opponent’s moves, select their best one recursively and define the current node’s value accordingly.

Next, I decided to implement Alpha-Beta pruning. This is a potentially massive efficiency saver that can effectively cross out large branches on the tree of search space. The basic idea is that if we have already explored some part of a position and know that:

- Our opponent can guarantee a decent result if we choose this move, and
- If we choose a different move, our opponent cannot guarantee a decent result,

Then we can skip searching the rest of this branch of the tree search space.

The pseudocode for this function is not too much more complicated, with two extra variables called alpha and beta carrying information for the best moves found so far for both players:

```
int alphaBeta( int alpha, int beta, int depthleft ) {
    if(depth == 0) return evaluate(alpha, beta);
    for (all_moves) {
        score = -alphaBeta( -beta, -alpha, depthleft - 1 );
        if( score >= beta )
            return beta; // fail hard beta-cutoff
        if( score > alpha )
            alpha = score; // alpha acts like max in Minimax
    }
    return alpha;
}
```

While the pseudocode I had made logical sense, I found it quite difficult to embed it into my system. I had various bugs concerning when I should switch around the alpha and beta values to account for the fact that every 2 moves by white is followed by 1 for black. This, combined with the negation of the values for alpha and beta which introduced further practical complexities, meant that I had to revert to the supposedly harder-to-implement minimax function. I had effectively traded simplicity and cleverness for reliability and functionality. This was a tradeoff I had to make several times – to get a functioning, decent prototype of a Monster Chess engine, I would have to make some concessions for ease of maintenance.

In any case, alpha-beta pruning got me almost an entirely functional (albeit quite weak) move selector.

Sources:

<https://www.chessprogramming.org/Minimax>

<https://www.chessprogramming.org/Negamax>

<https://www.chessprogramming.org/Alpha-Beta>

Week 8: 8/11/21 – 14/11/21

This week I wanted to finish implementing my ‘static evaluation’. At the bottom of the search tree (the ‘leaf nodes’), we must make some sort of decision as to how good a position is. This value is known as the static evaluation, and it is positive or negative infinity if the position is checkmate for one side.

While, so far, I had tried to avoid putting in my own thoughts on what ‘good moves’ or ‘bad moves’ were in any position, I realized that I would have to use my own judgement of the game for this aspect of the game.

In classical chess, this value is typically calculated by some combination of the number of pieces for each side with some weight for each piece – the queen is worth more than a rook, for example. The typical matrix goes as such:

Queen: 9 points
Rook: 5 points
Bishop: 3 points
Knight: 3 points
Pawn: 1 point

Most static evaluators, however, will also take additional factors into account, such as ‘passed pawns’ (which could promote and become queens at some point), and sometimes even piece placement matrices.

Of course, I couldn’t directly apply this to Monster Chess, since not only are the sides asymmetric, but the rules are fundamentally different enough that any heuristics that apply in chess don’t work very well in Monster Chess. There has been some study on how to evaluate the static values of pieces for chess variants in general, and I did consider replicating some sort of linear regression model which the papers I read also used. But, just to generate a first working model, I decided on the following values:

Black Queen: -9 points
Black Rook: -5 points
Black Bishop: -3 points
Black Knight: -3 points
Black Pawn: -1 point

White Queen: ∞ points
White Pawn: 3 points
King Distance: -1 per square difference

This is reflective of the fact that white pretty much instantly wins with a queen that can make two moves (if they manage to promote a pawn), and that the closer the two kings are (in general) the easier it is for white to force a checkmate via the Monster King.

There probably is value in applying neural networks to fine-tune these values, but for now these will suffice.

Sources:

<https://www.chessprogramming.org/Evaluation>

<https://www.chessprogramming.org/NNUE>

<http://www.ke.tu-darmstadt.de/publications/reports/tud-ke-2008-07.pdf>

Week 9: 15/11/21 – 21/11/21

The next major step I wanted to take was to combine several potential improvements I had come across while doing research and finish my algorithm. I intend to finish my project by the end of next week, with a basic but functional User Interface.

As I've laid out in the 'Testing' section, apart from just stress-testing my system to iron out bugs, I also needed to make sure my engine was of reasonable strength. Unfortunately, it did not start out very strong. In addition to only searching to a depth of around 5 (in a fixed amount of time which I set at 20 seconds), it looked like the number of nodes per second my search engine was visiting was very low. This meant it was reasonably easy, even for me, to beat my engine at Monster Chess.

I looked back at the research I had done into deterministic, minimax-type algorithms and selected several areas to improve my algorithm. While researching which potential techniques to implement, I also asked for advice in several online forums regarding how best to make my algorithm visit more 'nodes' (possible board positions) more quickly. By cross-referencing various posters' advice with other online reputable sources, I decided to implement several improvements:

Zobrist Hashing

The idea of Zobrist Hashing is important when considering improvements because they allow chess positions to be stored in a lookup index. We use a deterministic 'hashing' algorithm to convert any

position into a number, which is then used as the index for the position in a very large array (on the order of 10^7 on my laptop).

The mathematics behind the hashing function is what makes Zobrist Hashing complicated, but the main idea is to:

- Create an array of pseudo-random numbers corresponding to each piece on each square
- Generate additional random numbers for whose turn it is, castling rights, etc.
- Logically XOR together each piece of relevant position, thus creating an index for a table

```
BB naive_zobrist(const ChessBoard &pos) {
    BB to_return = 0ULL;
    ChessBoard work_with = pos;

    while (work_with.WhitePawns!=0) {
        int position = findAndClearSetBit(work_with.WhitePawns);
        to_return ^= zobrist_table[position][0];
    }
    while (work_with.WhiteQueens!=0) {
        int position = findAndClearSetBit(work_with.WhiteQueens);
        to_return ^= zobrist_table[position][1];
    }
    while (work_with.WhiteKing!=0) {
        int position = findAndClearSetBit(work_with.WhiteKing);
        to_return ^= zobrist_table[position][2];
    }
    while (work_with.BlackPawns!=0) {
        int position = findAndClearSetBit(work_with.BlackPawns);
        to_return ^= zobrist_table[position][3];
    }
    while (work_with.BlackRooks!=0) {
        int position = findAndClearSetBit(work_with.BlackRooks);
        to_return ^= zobrist_table[position][4];
    }
    while (work_with.BlackKnights!=0) {
        int position = findAndClearSetBit(work_with.BlackKnights);
        to_return ^= zobrist_table[position][5];
    }
    while (work_with.BlackBishops!=0) {
        int position = findAndClearSetBit(work_with.BlackBishops);
        to_return ^= zobrist_table[position][6];
    }
    while (work_with.BlackQueens!=0) {
        int position = findAndClearSetBit(work_with.BlackQueens);
        to_return ^= zobrist_table[position][7];
    }
    while (work_with.BlackKing!=0) {
        int position = findAndClearSetBit(work_with.BlackKing);
        to_return ^= zobrist_table[position][8];
    }

    return to_return;
}
```

The code above illustrates a naïve way to find a particular position's Zobrist index, since it recalculates the entire board. There is a slightly smarter way of achieving the same result if given a particular move made and the Zobrist index beforehand, which is to just XOR the values of what has changed (namely the square the piece has moved to and from, and whose turn it is). Here we rely on the fact that XOR is its own inverse, which gives us faster incremental updates than other methods:

```
BB smart_zobrist(BB old_zobrist, const int from_sq, const int to_sq, int type_of_piece, int piece_taken) {

    old_zobrist ^= zobrist_table[from_sq][type_of_piece];
    old_zobrist ^= zobrist_table[to_sq][type_of_piece];

    if (piece_taken != -1) {
        old_zobrist ^= zobrist_table[to_sq][piece_taken];
    }

    return old_zobrist;
}
```

Transposition Tables

The idea behind transposition tables is simple. Some positions can be reached via multiple different sequences of moves, so rather than computing the value of the best move in this repeated position twice, we store the value of the position once and then don't have to work it out again.

In Classical Chess Engines, transposition tables are only very useful in the endgame, when fewer pieces can move and so repetitions happen relatively often. However, I realized that in Monster Chess, because there are so many ways the Monster King can move to the same space twice, transposition tables are even more useful, cutting down on computation and increasing efficiency massively. Thus, implementing transposition tables became a necessity.

```
HASHE * phashe = &hash_table[pos.ZobristValue%HASH_LENGTH];
if (phashe->key==pos.ZobristValue and phashe->hash_depth >=
(depth_to_search-depth)) {

    if (phashe->hash_flag==EXACT) {
        if (depth==0) {
            if (phashe->best!=UNDEFINED) {
                return {phashe->hash_value, phashe->best};
            }
            alpha = max(alpha, phashe->hash_value);
            beta = min(beta, phashe->hash_value);

        } else {
```

```

        return {phashe->hash_value, phashe->best};
    }

} else if (phashe->hash_flag==ALPHA) {
    alpha = max(alpha, phashe->hash_value);
} else if (phashe->hash_flag==BETA) {
    beta = min(beta, phashe->hash_value);
}

```

Here I've already implemented Zobrist Hashing and I have a hash table with entries of type HASHE. This additional code is added to the negamax function before any actual evaluation is done, to avoid doing the same evaluation multiple times. The values alpha and beta are then used to help avoid unnecessary work in the main negamax function.

Iterative Deepening

A further optimization, which also makes the system more user-friendly, is called iterative deepening. One issue that came up at times when I was myself playing against my engine was the idea of time – at times, searching up to, say, depth 6 took less than a second, but at others would take a minute. This is not only inconvenient for me but makes the engine essentially useless in competitive play because it cannot manage its own time.

The solution is iterative deepening. Instead of evaluating at once at depth 6, the engine first evaluates at depth 1, then 2, and so on until a fixed amount of time has passed. When it runs out of time, it returns the best move from whatever the last depth it completed a search for was.

At first glance this may seem inefficient, but there are a couple reason why it is not:

- Transposition tables help avoid a lot of repeated work
- Each successive layer takes exponentially more time than the last, so in fact 90-95% of the time spent is on the last layer that is returned.

The pseudocode is as follows:

```

for (depth = 1;; depth++) {
    val = AlphaBeta(depth, -INFINITY, INFINITY);

    if (TimedOut())
        break;
}

```

We just keep increasing depth until we are timed out.

This system allows the chess engine to take a finite amount of time to give a reasonably decent answer, and so helped improve the engine's gameplay.

Possible Further Improvements to Explore

While I was reasonably happy with performance of my engine after these improvements, there are, as ever, vast multitudes of things I could try out. Some of these I've already mentioned – Quiescence Search, Move Ordering, Principal Search Variation, etc. – but the one I'm most interested in is boosting my static evaluation by implementing some sort of neural network system.

Sources:

https://www.chessprogramming.org/Zobrist_Hashing

https://www.chessprogramming.org/Transposition_Table

<http://mediocrechess.blogspot.com/2007/01/guide-transposition-tables.html>

<http://web.archive.org/web/20070712003703/http://www.seanet.com/~brucemo/topics/>

<http://web.archive.org/web/20070717225513/http://www.seanet.com/~brucemo/topics/hashing.htm>

<http://web.archive.org/web/20070705134347/http://www.seanet.com/~brucemo/topics/iterative.htm>

https://www.chessprogramming.org/Iterative_Deepening

Small Break: 22/11/21 – 12/12/21

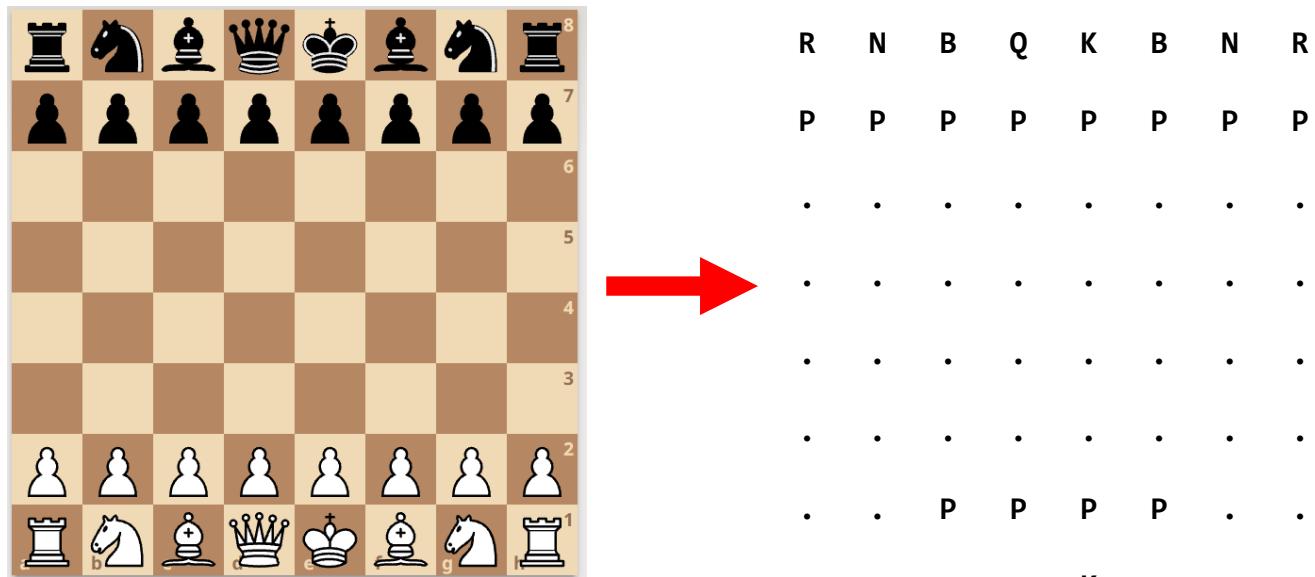
As a result of internal assessments, I once again had to briefly turn my attention away from the EPQ project. I returned to it a couple weeks later.

Week 10: 13/12/21 – 19/12/21

I was satisfied with the finishing touches on my move selection and search algorithm I had implemented last week, but I wanted to make it slightly easier to play against my algorithm. The first thing I had to decide was which Graphical User Interface (GUI) to use.

wxWidgets vs Qt

I had originally used text input which looked like this:



But to make a GUI in C++, I needed to choose a software to work with. The two major options were Qt and wxWidgets, both of which had their advantages and disadvantages:

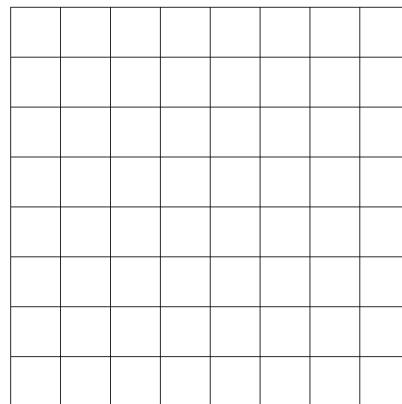
wxWidgets	Qt for C++
Requires less installation; is a ‘light’ library.	Requires a lot of installation; is a ‘heavy’ library.
Works nicely with other libraries.	Is difficult to implement with additional other libraries.
Has relatively few additional functionalities.	Has a lot of extra functionalities that can be used.
Works only on Windows.	Works pretty much on any OS, if additional code supplied.
Can make commercial applications with paying a fee.	Costs a monthly fee to make commercial applications with.
Has relatively few online tutorials.	Has a lot of online tutorials.

I considered all these factors when choosing a library to work with. Since I never intended to make a commercial application, and because I would rather have one framework that contained everything I would need rather than having to constantly download more and more frameworks, I decided to go with Qt.

Week 11: 20/12/21 – 26/12/21

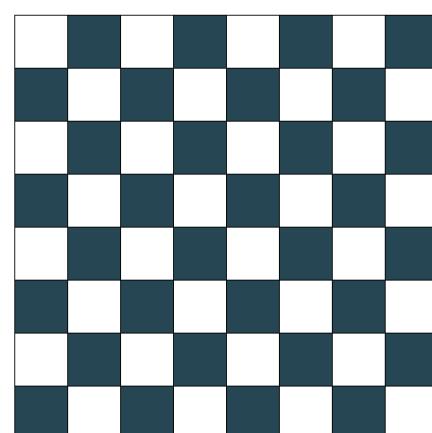
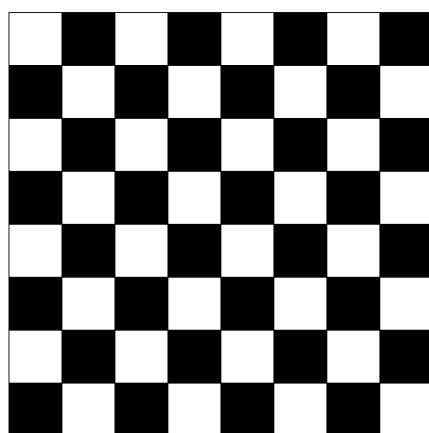
This week I resolved to properly embark on understanding how Qt worked. I started with extremely basic programs, understanding how so-called ‘events’ worked in Qt, and how to draw objects on a screen. I worked my way up to drawing a grid of squares:



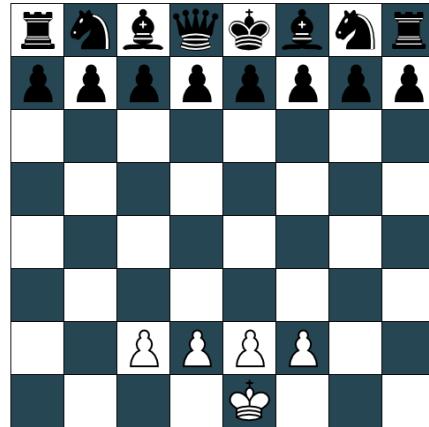


I then managed to alternately color the squares, after which I chose a suitable color palette:

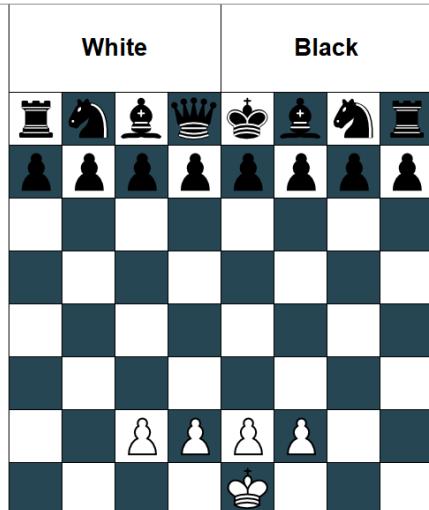




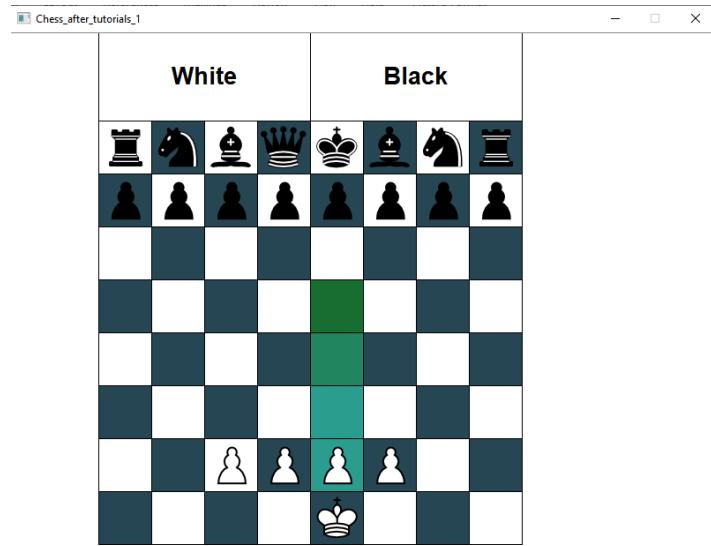
I next added images of pieces to show where the pieces would be:



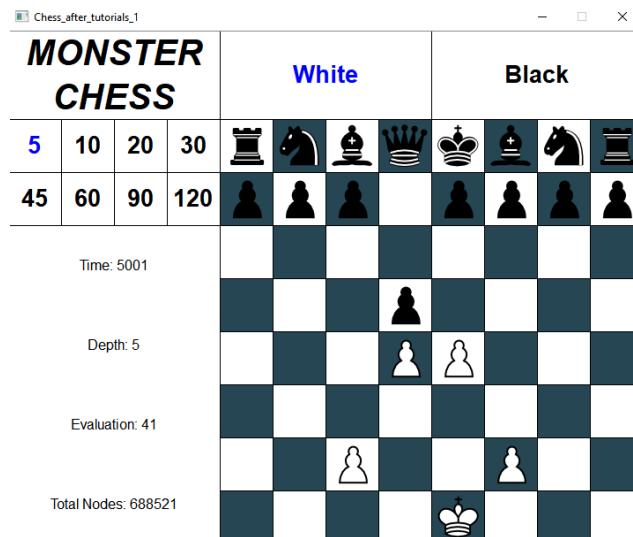
And then added buttons so that you could choose whether to play as black or as white:



Next, I showed all the legal moves you could play when you click on a piece to move it:



Finally, for the finishing touches, I added the ability to choose how much time the engine got to think and displayed some of the engine's thoughts on the position after it played a move.



Sources:

<https://www.qt.io/>

<http://www.wxwidgets.org/>

[Qt Tutorials For Beginners 1 - Introduction](#)

Week 12: 27/12/21 – 2/1/22

This week, having finished my project, I began to stress-test and verify all parts of it worked.

I came across the PyChess library, which proved an invaluable resource. I randomly generated hundreds of chess positions and asked both the PyChess library and my C++ code how many legal moves it saw for the side whose turn it was to play. In this way, I could make sure that there were no discrepancies between my code and the correct move generators:

```
SINGLE (NORMAL CHESS) MOVE GENERATION
Position 1:
Expected moves ..... 19
Python moves..... 19
C++ moves..... 19

Position 2:
Expected moves ..... 15
Python moves..... 15
C++ moves..... 15

Position 3:
Expected moves ..... 12
Python moves..... 12
C++ moves..... 12

Position 4:
Expected moves ..... 6
Python moves..... 6
C++ moves..... 6

Position 5:
Expected moves ..... 18
Python moves..... 18
C++ moves..... 18
```

Initially, I struggled to get these two figures to consistently match up. I had a particular bug where my code would not be able to see certain moves (such as moving a black knight in front of the black king) which would leave a king in check and as such were not legal. I often had positions like so:

```
Position 15:
Expected moves ..... 5
Python moves..... 3
C++ moves..... 5

Position 16:
Expected moves ..... 23
Python moves..... 20
C++ moves..... 23

Position 17:
Expected moves ..... 17
Python moves..... 14
C++ moves..... 17
```

In the end I was able to fix this bug by going into the ‘is_someone_in_check’ function and implementing more stringent checks to make sure no move left the king exposed. Specifically, I made

sure to check after each potential move that the king was not left in check by adding the ‘is_someone_in_check’ function within the other move generation functions.

Week 13: 3/1/22 – 9/1/22

This week was the finale – the matchup of my engine against my father. I implemented some final UI details, and then decided upon the structure of the matchup: 3 matches, with both sides alternating between white and black.

In the first match, my dad played white and played extremely aggressively – pushing pawns forward and going straight for checkmate. My engine, however, diligently defended, blocking pawns with pieces and even sacrificing a few minor pieces to stop white’s progress. In the end, after a grueling 73 moves, my engine managed to grind away and promote enough pawns to checkmate the Monster white king.

In the second game, perhaps somewhat more cautious after defeat in game 1, my dad played much more carefully. As black, he set up a defensive fortress and waited for the engine’s onslaught. This attack, when it came, was close to being successful, but in the end was thwarted by a combination of moves that resulted in long-term compensation for black. My dad won after promoting enough pawns to checkmate.

In the third game, my dad came back as white and played more carefully. He pushed pawns less frequently to avoid them getting blocked easily and was even close to promoting one. In the endgame, however, my engine sacrificed all its remaining pieces except its queen to ensure no possibility of defeat. With the black king in the corner, the game had entered a theoretically drawn position, and so the game ended as a draw.

I’m quite happy with my engine’s performance – it certainly performed much better than I could have, and better than I expected. If anything, this shows the dominance of black in Monster Chess if both players are skilled – setting up a defensive fortress is not too difficult.

Small Break: 10/1/22 – 6/2/22

Having completed the bulk of my project, my attention once again shifted to internal assessments. Thus I had to leave my oral presentation and writeup to the side for about a month before returning to complete my EPQ project.

Weeks 14 and 15: 7/2/22 – 20/2/22

I started and completed writeup.

Week 16: 21/2/22 – 27/2/22

This week I worked on my oral presentation.

Research

My research for this project took a few major forms. Primarily, my research was conducted online, through various chess programming forums and websites. I combined this with research into current state-of-the-art open-source chess engines.

Much of the research was performed as I was planning the project and designing the software, but it also continued into the main build.

Coding Practices

Before embarking on my project, I decided to research what the best practices are for writing code. This was because I knew that this project would end up being quite large and that, in order to facilitate debugging and stress-testing, I would require clean and well-structured code. I realized that if I didn't make an active effort to make my code as clean and well-written as possible, the project could become almost impossible to maintain beyond a point.

The research I did on best coding practices focused on a few key areas. Firstly, many sites stressed the importance of keeping code 'readable' – writing code whose function is clear to whoever is reading it. This is particularly important on projects that are built over long periods of time – I was very likely going to have to edit code that I had initially written months prior.

I also stuck to the advice of keeping line length under 80 characters, which helps make code easier to read. I implemented other software development best practices such as the separation of tasks – making sure different classes and functions are responsible for their own individual tasks. For example, my 'legal_moves' function concerns itself only with generating the legal moves from a chessboard position when given one and does not try to evaluate which one is the best. This helps clearly delineate different tasks from each other and helps in the (inevitable) case that bugs crop up.

While, as in any software development, I came across hundreds (if not thousands) of bugs while creating my product, the best practices I researched and implemented helped me to concentrate on fixing the issues immediately and moving forward rather than fighting my own codebase.

How Chess Engines Fundamentally Work

The first thing I had to grasp with regards to my project was how modern chess engines worked. I then would have to assess how well the techniques and strategies used to make traditional chess engines good could be applied to Monster Chess.

I started by investigating various modern state-of-the-art chess engines such as Stockfish and Leela Chess Zero. Some of the most powerful modern chess engines are open source whose code can be freely viewed.

Through my research into these engines, I was able to get a broad outlook on how chess engines are built. Primarily, chess engines go through two distinct phases – move generation and move selection. I realized I would have to consider both these aspects carefully when translating from traditional chess engines to my own Monster Chess engine.

However, I also realized that current state-of-the-art engines are quite complicated and have very large codebases. It didn't seem feasible or useful to try and analyze the inner workings of these extremely strong engines before first getting a general overview for how various algorithmic techniques have developed over time.

Fortunately, there a lot of resources that detail how move generation and move selection can be algorithmically implemented. Here I had to start making decisions as to what would work best for Monster Chess.

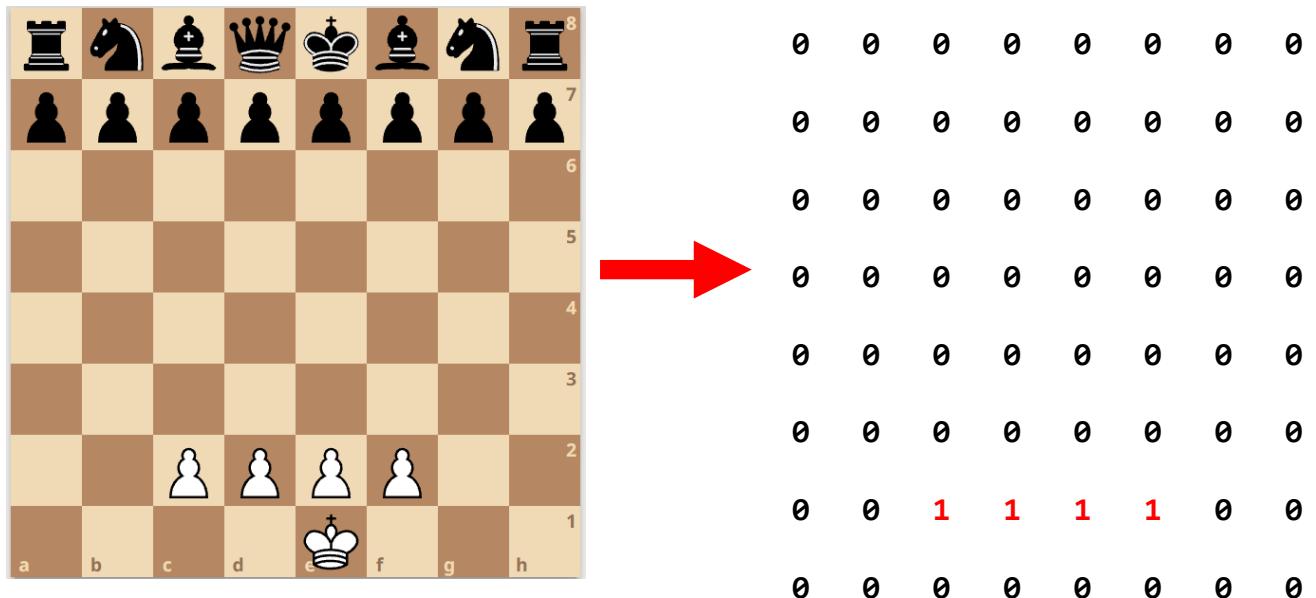
Mailbox vs Bitboards

‘Mailbox’ engines assign an object to each square of the 64 on the chessboard and to each possible piece for both sides. Typically, these engines then loop through every possible piece for each square every move. ‘Mailbox’ chess engines are conceptually easier to work with and are, in a sense, much more naturally thought of (it’s how I guessed chess engines worked before I did any research).

On the other hand, the prevalent technique used in most engines today is called the ‘bitboard’. Bitboards are essentially 64-bit integers that represent 64 boolean (true or false) values, describing whether a piece is on a particular square or not. Bitboards naturally take advantage of the coincidence that the number of squares on a chess board is the same as the number of bits needed to store the typical integer (64).

Most of my research into bitboards was from an online tutorial (referenced at [1]) which gave me the basics, and the chess programming wiki (referenced at [2]), the latter of which was incredibly useful for my entire project.

We can define a chessboard simply by using one ‘bitboard’ (64-bit integer) for each type of piece. For example, the ‘WhitePawns’ bitboard could look like this:



Similarly, we create eight other bitboards for the other types of pieces.

To demonstrate how efficient this bitboard system can be, let me show how easily three additional helper bitboards can be created. While these additional bitboards are not strictly necessary, they make programming a lot easier when dealing with piece movement later.

```
AllWhitePieces = WhitePawns | WhiteQueens | WhiteKing;
AllBlackPieces = BlackBishops | BlackKing | BlackKnights | BlackPawns |
BlackQueens | BlackRooks;
AllPieces = AllBlackPieces | AllWhitePieces;
```

¹Peter Keller, “Chess and Bitboards.” *University of Wisconsin-Madison*, accessed 25th October 2021.

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/index.html>

² Chess Programming Wiki, Wikimedia. “Mailbox.” Last accessed 13 June 2021.

<https://www.chessprogramming.org/Mailbox>

Here the bitboards ‘AllWhitePieces’ can be generated via a logical OR. We combine WhitePawns, WhiteQueens and WhiteKing, putting a 1 if we encounter a 1 in any of the initial bitboards.

<i>WhitePawns</i>	<i>WhiteKing</i>	<i>AllWhitePieces</i>
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
OR	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0	0 0 0 0 0 0 0 0	0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 0	0 0 0 0 1 0 0 0

Thus, by using bitboards, we can represent an entire chessboard with just 9 integers, a significant improvement on the 2d array of strings that is typically used in mailbox engines.

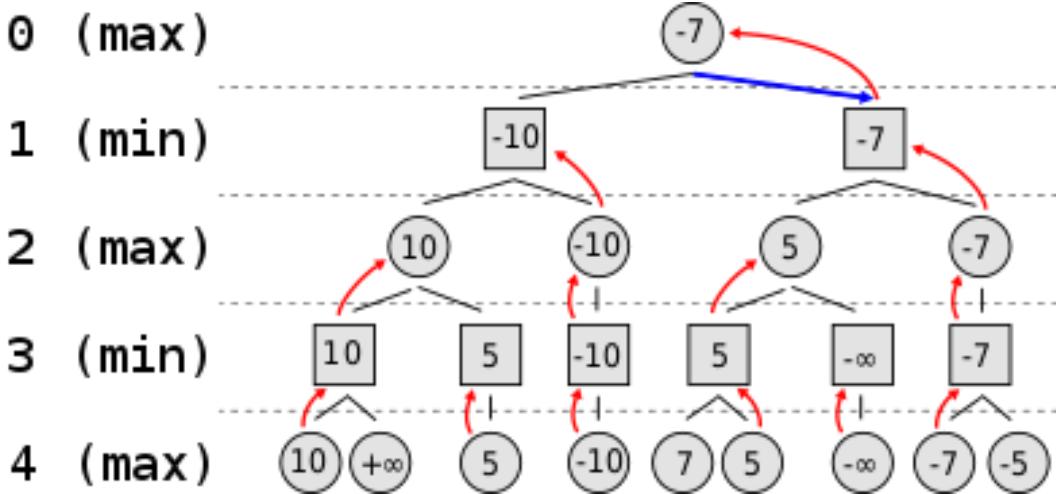
Magic Bitboards vs Obstruction Difference

One of the key decisions I had to make regarding move generation was how to deal with the ‘slider’ pieces (the rook, bishop and queen). There appeared to be two major ways of dealing with these pieces - Magic Bitboards and Obstruction Difference. While Magic Bitboards seemed to be about 20% more efficient at generating sliding legal moves (the legal moves for the bishop, rook and queen) than the Obstruction Difference method, there were a couple of things I was concerned about with them. Firstly, they seemed to require a lot more memory than the Obstruction Difference method; secondly, it is a lot more difficult to implement and debug Magic Bitboard systems in the (inevitable) case of errors. I could also relatively easily conceptually understand Obstruction Difference, so I was reasonably confident that, should bugs arise (as they do), I would be able to resolve them. I was not so confident with Magic Bitboards. I’ve described in some detail how exactly I implemented Obstruction Difference in the Log section of the report.

The Minimax Algorithm

The most widely used algorithm for move selection I encountered was some version of the Minimax algorithm. Minimax algorithms work on the assumption that a good move for one player is a bad

move for the other. Thus, say we could statically evaluate every chess position; assign a number to each position which was positive if white was winning, and negative if black was winning. White's objective is to maximize this function, while black's is to minimize it.



Let's look at an example. Consider the graph above.

Suppose the game being played only has a maximum of two possible moves per player each turn. The algorithm generates the tree on the right, where the circles represent the moves of the player running the algorithm (maximizing player), and squares represent the moves of the opponent (minimizing player). Because of the limitation of computation resources, as explained above, the tree is limited to a look-ahead of 4 moves.

The algorithm evaluates each leaf node using a heuristic static evaluation function, obtaining the values shown. The moves where the maximizing player wins are assigned with positive infinity, while the moves that lead to a win of the minimizing player are assigned with negative infinity. At level 3, the algorithm will choose, for each node, the smallest of the child node values, and assign it to that same node (e.g. the node on the left will choose the minimum between "10" and "+∞", therefore assigning the value "10" to itself).

The next step, in level 2, consists of choosing for each node the largest of the child node values. Once again, the values are assigned to each parent node. The algorithm continues evaluating the maximum and minimum values of the child nodes alternately until it reaches the root node, where it chooses the move with the largest value (represented in the figure with a blue arrow). This is the move that the player should make in order to minimize the maximum possible loss.

In chess, of course, by the fourth step down in any tree search space, we would have a lot more nodes on the tree graph since there are generally a lot of legal moves in any position, but the image above serves as an illustration.

Minimax algorithms essentially try to guess what move the opponent will play next, assuming they play this best move, and assessing the resulting position. It goes without saying that a higher depth will mean more accurate position evaluators and thus a better chess engine.

Shannon's Type A vs Shannon's Type B

The next key area I had to research regarding move selection was which Shannon's Type my algorithm would be: Type A or Type B:

Type A	Type B
Brute-force algorithm, will never 'miss' a move.	Selective search, may miss moves.
Can search positions to lower 'depth'.	Can search positions to higher 'depth'.
Easier to implement.	Tougher to implement, especially to variants of chess.

While Type B algorithms could lead to engines which are stronger than those with Type A algorithms, Type B algorithms require additional heuristics and strategies which software engineers can provide for Classical Chess because that game has been studied for centuries. But in Monster Chess, there are no experts or grandmasters. I didn't want to negatively bias the engine by assuming I knew what types of moves were best, and as such I decided to go with Type A algorithms, which are much safer in general.

Self-Made Static Evaluation vs Neural Static Evaluation

However, I realized that at some point, I would have to use my own judgement to evaluate board positions – the minimax algorithm works by plugging in a static evaluator to the bottom of a search tree algorithm. This is not ideal - in classical chess, humanity has collectively, through millions of games, come up with standard point values for each piece (the queen is worth 9, the rook is worth 5, the bishop and knight 3, and the pawn 1). There are also simple heuristics that classical chess engines use (such as valuing passed pawns) that have similarly been determined to be 'valuable' by strong chess players over the years.

There is no such rich history for Monster Chess – in fact there is very little online presence of the game at all. There is, as far as I could find, just a Wikipedia Page (referenced at [3]) and two other encyclopedias (detailed in [4] and [5]) of chess variants that mention it. In fact, one of these pages claims the game is solved – an idea I’ve explored more in the ‘Evaluation’ and ‘Testing’ sections.

In any case, I came up with what I believe is a reasonable static evaluation of the board – white pawns are worth about as much as a black bishop / knight, and black values distance between kings. But one interesting approach would be to use a linear regression model, or a function-approximating neural network, to generate weights by getting my engine to play itself, like the process outlined in[6].

In order to fully make use of these linear regression models, however, I would have to completely rethink how my project was planned and designed: I would have to plan to make move generation a lot more efficient (I had come across some ideas of ‘pinned bitboards’ and ‘check bitboards’), but this seemed like more trouble than it was worth, considering that the potential benefit did not seem that large. I decided to stick to the Static Evaluation I had come up with.

Python vs C++

The first major structural decision I had to make was what programming language I should implement my engine in. The two languages I was considering were Python and C++, since they were the two that I had some familiarity with. I was very comfortable with Python as a programming language, but recently I had been using C++ more. I was impressed by how, once code had been written and compiled, C++ was incredibly fast at executing said code – often running up to ten times as fast as the interpreted Python code ran. This was a crucial deciding factor for me as there is an exponential nature to how the number of possible legal sequences of moves increases as the computer keeps searching for good moves. This exponential nature means that speed at runtime greatly affects the strength of the final chess engine.

³ Wikipedia, Wikimedia. “Monster chess.” Last accessed 15 September 2021.

https://en.wikipedia.org/wiki/Monster_chess

⁴ The Chess Variant Pages. “Monster Chess.” Last accessed 14 February 2001.

<http://www.chessvariants.org/unequal.dir/monster.html>

⁵ The Chess Variant Pages. “Muenster Chess.” Last accessed 15 January 1997.

<http://www.chessvariants.org/d.betza/chessvar/muenster.html>

⁶ Sacha Drost, Johannes Fürnkranz. “Learning of Piece Values for Chess Variants.” *Technical Report TUD-KE-2008-07*, Knowledge Engineering Group, Technische Universität Darmstadt. <http://www.ke.tu-darmstadt.de/publications/reports/tud-ke-2008-07.pdf>

I also realized that, in order to maintain a large codebase over a long period of time, I would have to make sure I had a good fundamental understanding of the structure of what I was doing. As a result, I decided to combine both these goals and initially use Python to create a very basic first prototype of a chess engine. My aims were to make sure the move generation of this program is valid in traditional chess, before considering whether to extend it to Monster Chess. I then decided to create my actual Monster Chess engine in C++, for the added efficiency it gave.

wxWidgets vs Qt

One of the less algorithmic and mathematically challenging parts of this project was designing the Graphical User Interface to make playing my engine a seamless experience. I had originally used text input which looked like this:

```
[['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r']
 ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p']
 ['.', '.', '.', '.', '.', '.', '.', '.']
 ['.', '.', '.', '.', '.', '.', '.', '.']
 ['.', '.', '.', '.', '.', '.', '.', '.']
 ['.', '.', '.', '.', '.', '.', '.', '.']
 ['.', '.', 'P', 'P', 'P', 'P', '.', '.']
 ['.', '.', '.', 'K', '.', '.', '.']]
```

I wanted to create a way to easily play against my engine. But to make a GUI in C++, I needed to choose a software to work with. The two major options were Qt and wxWidgets, both of which had their advantages and disadvantages:

wxWidgets	Qt for C++
Requires less installation; is a ‘light’ library.	Requires a lot of installation; is a ‘heavy’ library.
Works nicely with other libraries.	Is difficult to implement with additional other libraries.
Has relatively few additional functionalities.	Has a lot of extra functionalities that can be used.
Works only on Windows.	Works pretty much on any OS, if additional code supplied.
Can make commercial applications without paying a fee.	Costs a monthly fee to make commercial applications with.
Has relatively few online tutorials.	Has a lot of online tutorials.

I considered all these factors when choosing a library to work with. Since I never intended to make a commercial application, and because I would rather have one framework that contained everything I would need rather than having to constantly download more and more frameworks, I decided to go with Qt.

Major Restructuring

When I had first got Prototype 2 to a functional state, so my engine could select and play moves, I came across a less severe problem of what I had encountered with Prototype 1. My engine still seemed to be running quite a lot slower than typical C++ engines. My engine was ‘visiting’ or evaluating around 5500 nodes per second, which is extremely slow. I decided to post on some forums that specialize in computer chess, from where I received quite a lot of advice (this can be found in [7]):

What improvements can I make to my naive minimax algorithm? (self.chessprogramming)

submitted 9 months ago by PinkPandaPresident

I have been trying to develop an engine for a variant of chess called Monster Chess, in which white starts with only 4 pawns and a king (black has all their pieces) but white gets two plies for every ply black gets. White is (I believe) favoured to win; the white king can check the black king, for example, and it is very hard to stop white's attack in general (at least at the level I can play, lol). The [Wikipedia link](#) is here.

I am coding in C++ and have used arrays of integers to represent the board state. For my engine I only use a simple minimax algorithm with alpha-beta pruning, with a simple point-system evaluation. I am getting around 5500 nodes/second, and am brute-forcing to a depth of 3.

I would like some advice on how to proceed - is it worth rewriting the whole system with bitboards? Should I try to implement Zobrist hashing? Should I somehow try to intelligently search the tree of possible moves? What kind of depth is it reasonable for me to aim for? Ultimately my goal is to create a system that can beat me while I am playing white.

Any help would be much appreciated.

8 comments edit share save hide delete nsfw spoiler crosspost

⁷ChessProgramming, Reddit. “What improvements can I make to my naive minimax algorithm?” Last accessed 10 January 2021.

https://old.reddit.com/r/chessprogramming/comments/kudw1p/what_improvements_can_i_make_to_my_naive_minimax/

[-] tsojtsjsoj 3 points 9 months ago
5500 nodes per second is really slow, you probably have a operation that is extremely slow. You can have a array with integer based board representation and still get ~200000 nodes per second. Bzzboards though are probably 10 times faster. (This depends on whether you can quiesce nodes or not). You could use a profiler like perf or callgrind, or if your engine is open source I could take a quick look)

If you haven't implemented quiesce you should do it. Otherwise the engine might play weird moves because it stops searching after it reached the maximum depth after a move where it captured a pawn with a queen. Now it thinks it won a pawn but in reality the pawn was defended by another pawn, causing you to lose a queen if you follow the move sequence the engine thought was good.

Move ordering is pretty important. First play the best move you find in the transportation table, then winning captures then losing captures then quiet moves.

For the transposition table to make sense you need to use iterative deepening at the root.

Null move pruning is a very easy to implement technique but can improve performance quite a bit.

Implement check extensions: at the beginning of a node if you see that the position is in check, then you should increment the depth.

permalink embed save report give award reply

[-] PinkPandaPresident [S] 3 points 9 months ago
Thanks a lot for the advice! I have never used GitHub before so apologies if something is set up wrong but you can view my source code here: <https://github.com/PinkPandaPresident/MonsterChess>

This is my second version of the game, I did initially try coding in Python (before realising it was just way too slow).

I think based on what you have suggested I will probably try rewriting the whole thing from scratch. If you do take a look at my code, do tell me if you see any glaring inefficiencies, or something I could improve on for when I redo this (other than what you've already said).

If you could recommend some sort of GUI that I could easily implement that would be very helpful - currently I have a very limited kind of console UI that requires the squares to move from and move to.

And finally I am relatively new to programming as a whole so if you see any best practices I should be observing do tell me.

Thanks a bunch!

permalink embed save parent edit disable inbox replies delete reply

[-] tsojtsjsoj 2 points 9 months ago
practically all chess engines today support a protocol like UCI. If you implement UCI you can use a Gui like Arena or cutechess to play against your engine.

Some tips:

1. don't use `vector<vector<char>>` if it is not necessary, you already know at compile time that your board has $8 \times 8 = 64$ squares, so you could just use `std::array<std::array<char, 8>, 8>` or just `std::array<char, 64>` which will be faster than your nested vectors, because of CPU caching reasons. Writing cache-friendly code is very important when you're writing software that has to be fast.
<https://stackoverflow.com/questions/16699247/what-is-a-cache-friendly-code>
2. Also for performance reasons you should try to initialize or copy "large" types, like `std::vector` or `arrays`, as little as possible. For example in your move generation `possible_moves` you create a vector (that costs a lot of performance), then you return that vector. It could be that most of the time the compiler does a good job at optimizing these but sometimes they can get expensive.
3. If you pass a variable by reference, e.g. `bool is_white_in_check(Board &pos, int color)` and you don't change that variable inside this function you can mark it as `const`, which gives an compiler error if you try to change it inside the function: `bool is_white_in_check(const Board &pos, int color)`
4. Just in case you don't know yet: If you don't mark the variable as `const` then if you change it in the function, then it will also change for the caller of that function.

```
void f(int& a)
    a = 2;
}

const int inf = 10000; /* you can use const also for variable declarations, which gives you an error if you

void main(){
    // inf = 0; would fail because inf is marked 'const'
    int a = inf;
    f(a);
    std::cout << a << std::endl; // prints out 2
}
```

1. It would help if you didn't use non-descriptive types like `char` or `pair<char, char>`. Better would be something like `enum Piece{pawn, knight, bishop, rook, queen, king}`. Just for readability.
2. generally it becomes much easier to work with chess code if you don't have every function for white and black, where both versions are basically the same. Take a look at `Negamax` which is basically min-max but without the duplicate code for black and white. But it may be justified in your case because your version of the game is less symmetric.
3. Are you using windows or linux? If you use windows I can try to explain how to use a profiler like valgrind+callgrind, which lets you see which function of your program is the slowest. On Windows you can probably use the visual studio profiler.

↑ [-] [PinkPandaPresident](#) [S] 1 point 9 months ago
 ↓ Thanks for that! I have implemented a lot of the 'quick fixes' of const and references that you mentioned, and it has pushed the program up to about 7000 nodes per second! I will probably use callgrind later to figure out if there is any function specifically that is being super slow, for when I rewrite the program. Do you recommend anything specific in this case to avoid initializing and copying a vector of type Board a bunch of times in my code? What other methods could I use to generate, for example, the subtree of legal moves from a particular position? Or is it more general advice that I should keep in mind for next time?

In any case thanks for the advice!

[permalink](#) [embed](#) [save](#) [parent](#) [edit](#) [disable inbox](#) [replies](#) [delete](#) [reply](#)

↑ [-] [tsojtsojtsoj](#) 1 point 9 months ago
 ↓ What I do is usually something like this:

```
void movegen(const Board& board, std::vector<Move>& moves)
{
    moves.clear();
    ...
    moves.push_back(some_pawn_move);
    ...

    int negamax(...)
    {

        std::vector<Moves> moves;
        movegen(currentBoard, moves);
        for(const auto& move : moves)
        {
            // do something with 'move'
            ...
        }
    }
}
```

just to be safe, as I said, often the compiler notices if it can optimize the return such that it doesn't get copied all the time. But this way I can be sure that it doesn't get copied.

Just in case you didn't already, you can compile with the option '-O3' with g++ then the compiler does all these optimizations. (e.g. `g++ myprogram.cpp -O3 -o myprogram`)

[permalink](#) [embed](#) [save](#) [parent](#) [report](#) [give award](#) [reply](#)

↑ [-] [PinkPandaPresident](#) [S] 1 point 9 months ago
 ↓ Ah I see, will definitely use this strategy in the future!

[permalink](#) [embed](#) [save](#) [parent](#) [edit](#) [disable inbox](#) [replies](#) [delete](#) [reply](#)

↑ [-] [tsojtsojtsoj](#) 2 points 9 months ago
 ↓ Wait, UCI doesn't work with Monster Chess, you probably have to implement your own GUI. For C++ you could use Qt for your interface. I never used it, I just heard that it was ok.

[permalink](#) [embed](#) [save](#) [parent](#) [report](#) [give award](#) [reply](#)

phpBB® TalkChess.com
 Hosted by Your Move Chess & Games - chessusa.com

[Skip to content](#)

[Quick links](#) [FAQ](#) [Notifications](#) [Private messages](#) [PinkPandaPresident](#)

[ChessUSA - Your Move Chess & Games](#) < [Board index](#) < [Computer Chess Club Forums](#) < [Computer Chess Club: Programming and Technical Discussions](#)

[How to efficiently generate individual moves from bitmasks of pieces?](#)

Moderators: [H.G.Muller](#), [Dann Corbit](#), [Harvey Williamsson](#)

6 posts • Page 1 of 1

[Post Reply](#) [Search this topic...](#) [Search](#)

How to efficiently generate individual moves from bitmasks of pieces?

by [PinkPandaPresident](#) Fri Jan 15, 2021 2:18 pm

I have been trying to build a chess engine for a variant of chess called Monster Chess, in which white starts with just a king and 4 pawns but has two moves for every move black has - I posted about that here, if you are interested ([link](#)).

In any case I have been trying to implement simple bitboards to get some efficiency on my project. I have been following this tutorial: ([link](#)), which is sadly incomplete. As such I have one major question. Having generated a bit mask for, say, all the places any knight could jump to, how do we generate individual moves for each knight independently? I managed to follow this code - ([link](#)) - to do so for the king, but I don't see how we would extend the logic for when we have multiple pieces that can move, especially with pawns.

Any help much appreciated!

Re: How to efficiently generate individual moves from bitmasks of pieces?

by [H.G.Muller](#) Fri Jan 15, 2021 3:39 pm

You have to do that for every Knight separately.

Re: How to efficiently generate individual moves from bitmasks of pieces?

by [PinkPandaPresident](#) » Fri Jan 15, 2021 3:42 pm

From the tutorial I was following. "However, it will be shown later how it is possible to use these algorithms for all the pieces of the same kind and color on the board at the same time."

Such a method exist or is it just a phantom?

Re: How to efficiently generate individual moves from bitmasks of pieces?

by [H.G.Muller](#) » Fri Jan 15, 2021 4:08 pm

Well, I have never used bitboards, but I don't see how doing all Knights at once could ever be better. For a single Knight the set of target squares can be a simple lookup in a 64-entry table indexed by square number. I already don't see how you could get the set of all squares attacked by a Knight more efficiently than doing just that, and OR them together. For getting all squares attacked by Pawns that could still make sense, because each Pawn has few moves, and you can have many Pawns. So you can attack left-forward and right-forward attacks by simple shifts and masking away of the edge wrap-arounds, and the OR together, and you would have done 8 Pawns. But Knights have 8 moves, and you have only 2 Knights. So it is much faster to work per piece than per move direction.

When you start already with bitboards for individual pieces, it seems silly to OR them together, because then you have to figure out later which target square is attacked by which Knight.

Re: How to efficiently generate individual moves from bitmasks of pieces?

by [PinkPandaPresident](#) » Fri Jan 15, 2021 4:18 pm

Right, that makes a lot more sense. Thanks!

I had to weigh up the potential benefit of implementing a particular optimization against the complexity it would add to my project as well as the added memory usage it could take up. I cross-referenced the multitude of advice I had received against many reputable online sources, as detailed in my ‘Sources’ section. By implementing some of the features, commenters on these posts suggested, I was able to clear a lot of confusion I had with some aspects of move generation and general best practices when it came to C++ programming. After my optimizations, I managed to reach around 150,000 nodes per second, a lot higher than the 5,000 I had started at.

Design

I had several iterations of design and refinement, especially when it came to the User Interface. These are detailed below.

Algorithm Design

Most of the design for the code structure itself worked out as the project gradually got bigger. Of course, I kept in mind the principles of good, readable code:

- Keeping line length under 80 characters.
- Separation of tasks – making sure different classes and function are responsible for their own individual tasks.
- Using appropriate naming conventions – I stuck to underscores between words in variable and function names for the entirety of the project.
- Avoiding retyping the same code.
- Reusability – making sure code could be easily extended to multiples aspects of the project, as necessary.

I also worked out the particulars of algorithms through pseudocode and drawing diagrams in my notebook, some of which I have included here:

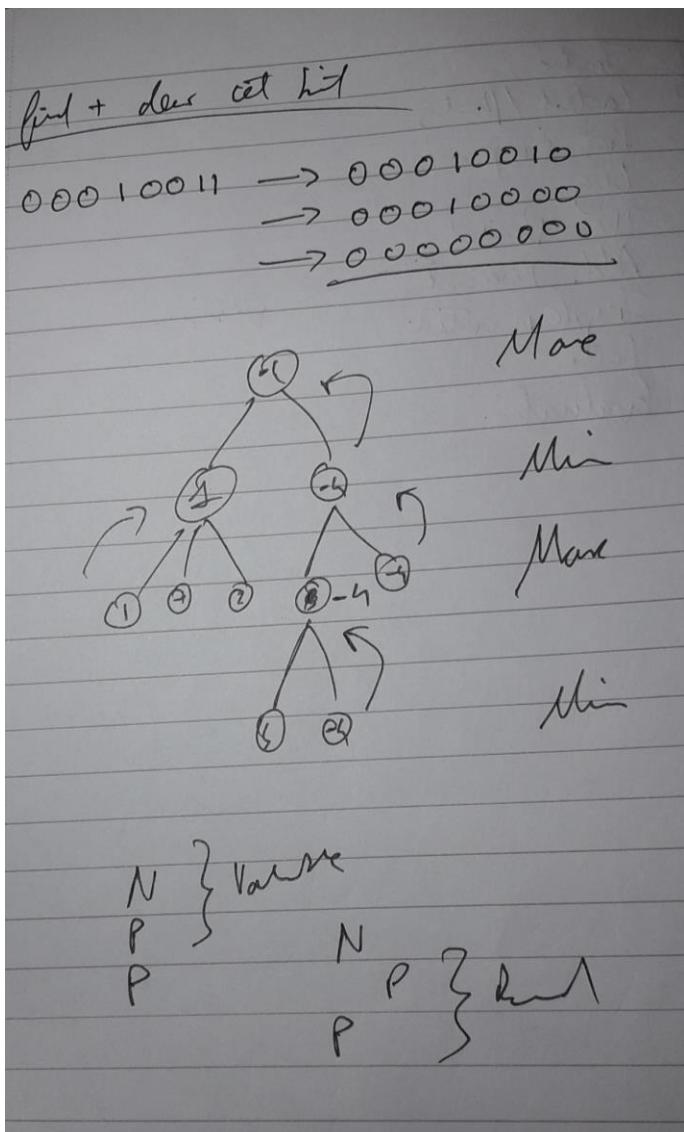


Figure 11 - The initial conceptions of a minimax algorithm. At each stage, we take (alternately) the maximum and minimum value possible for each move, in order to determine the best move possible right now. Described in more detail in the 'Research' section.

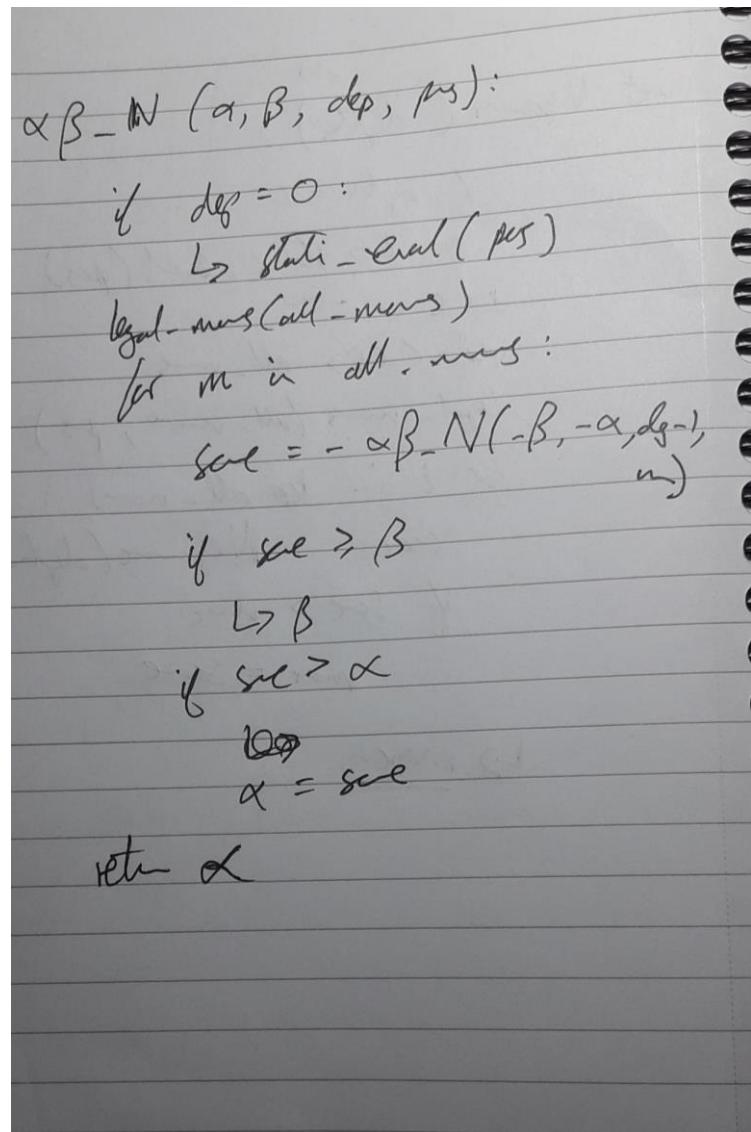


Figure 12 - Exploration of the alpha-beta pruning optimization to the minimax algorithm. While initially conceived of as part of a negamax system, a more standard minimax algorithm was used in the end, as described in the 'Log' section.

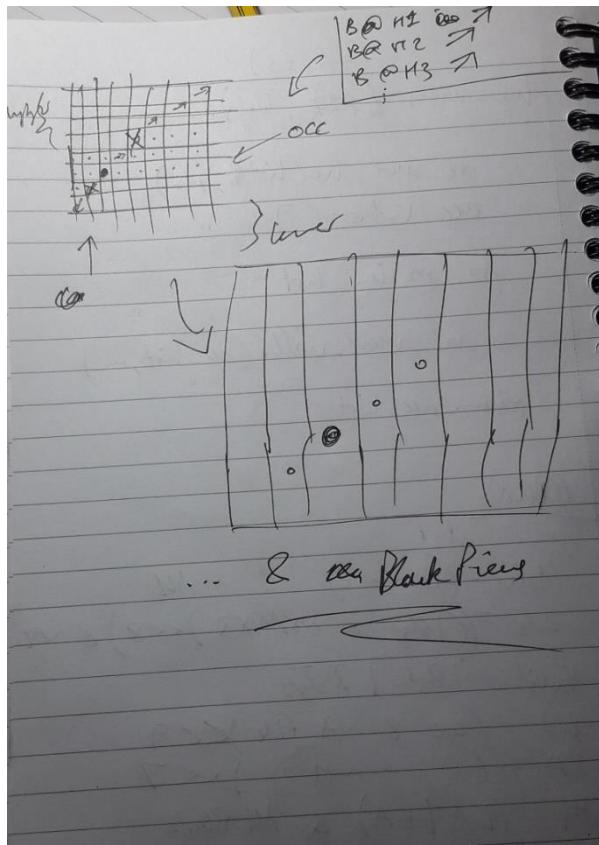


Figure 15 - Exploring possible ways a slider's (rook, bishop, queen) move generation could go.

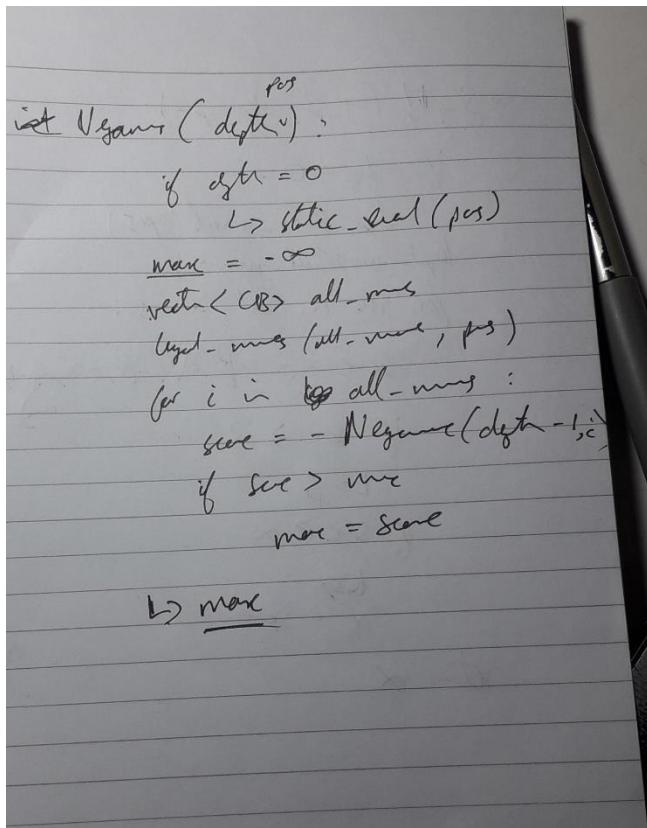


Figure 14 - Reviewing the pseudocode for a negamax algorithm. While similar in spirit to a standard minimax algorithm, the negamax algorithm can avoid some rewritten code.

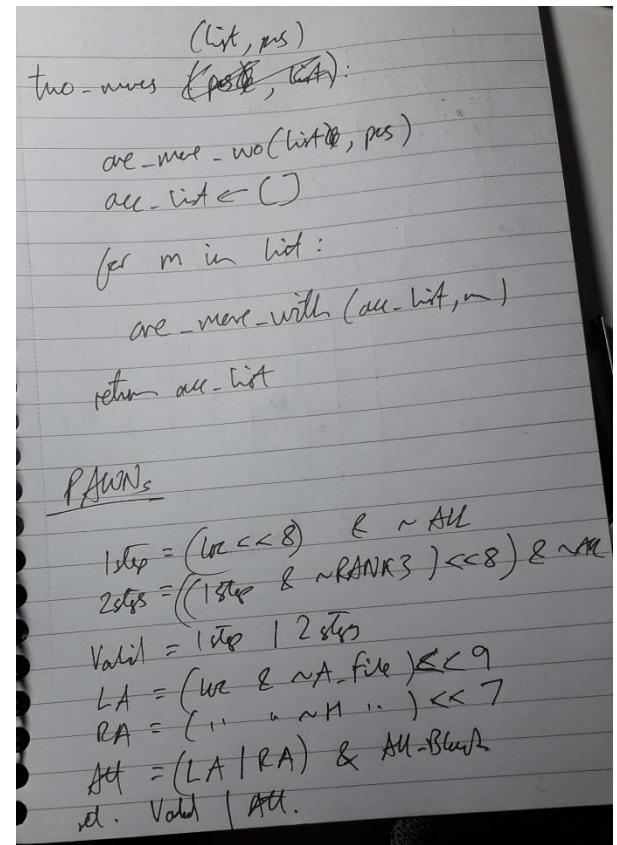


Figure 16 - Investigating the particulars of a pawn's legal moves. Attack and advancement bitboards, as well as bitwise logic, were necessary.

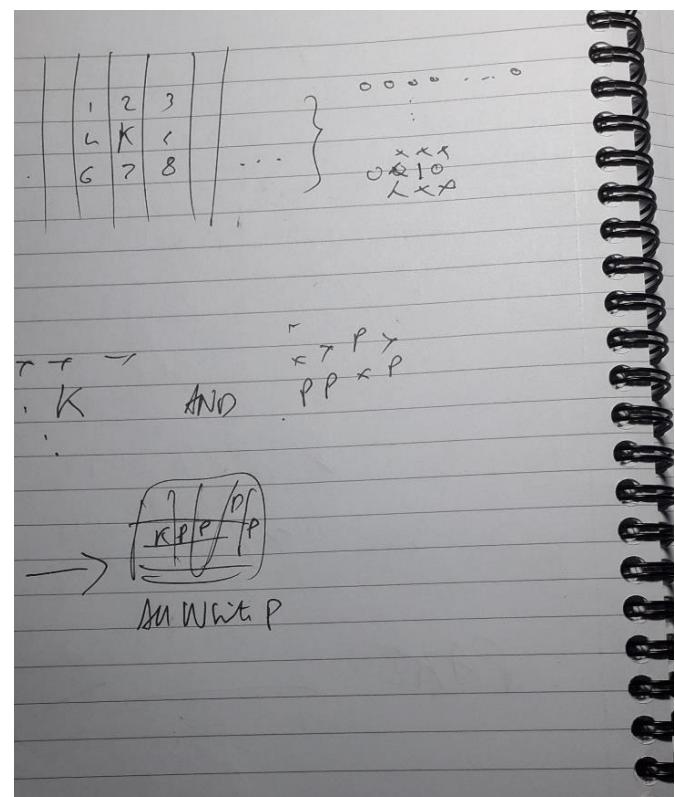
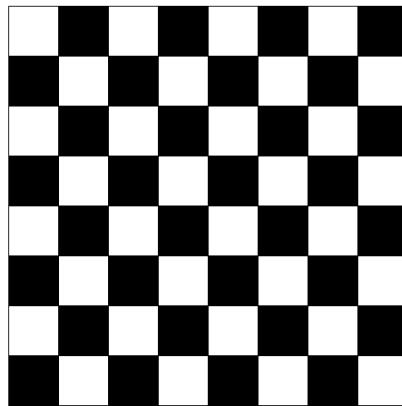
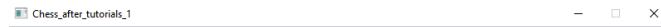


Figure 13 - Debugging the quite literal 'edge cases' that lead to phantom key moves. Described in more detail in the 'Evaluation' section.

User Interface Design

Once I had decided to use Qt for C++, I began to plan out what features my User Interface would have.

I started with extremely basic programs, understanding how so-called ‘events’ worked in Qt, and how to draw objects on a screen. I worked my way up to drawing a grid of squares:



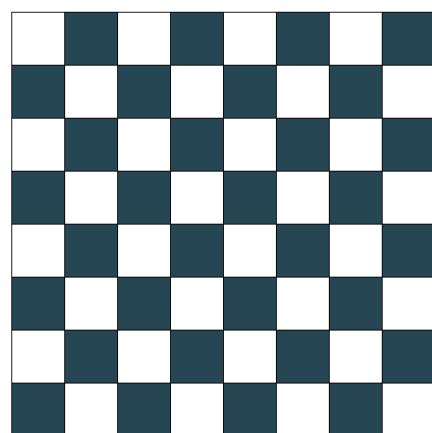
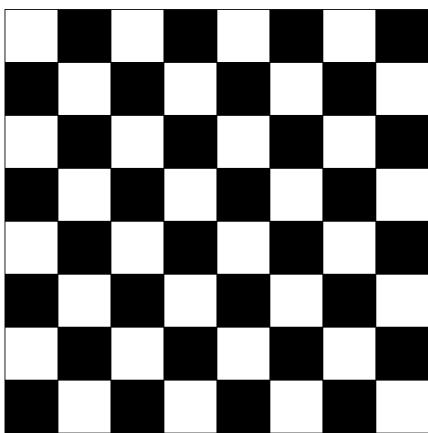
I then managed to alternately color the squares, after which I chose a suitable color palette:

Chess_after_tutorials_1

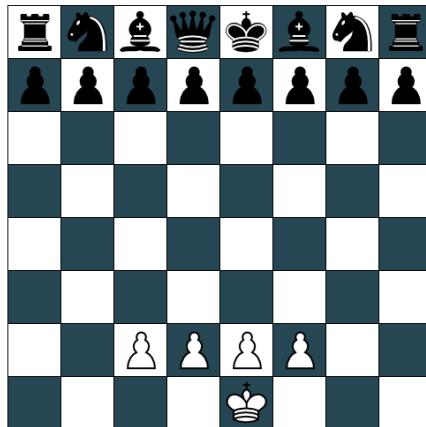
— □ ×

Chess_after_tutorials_1

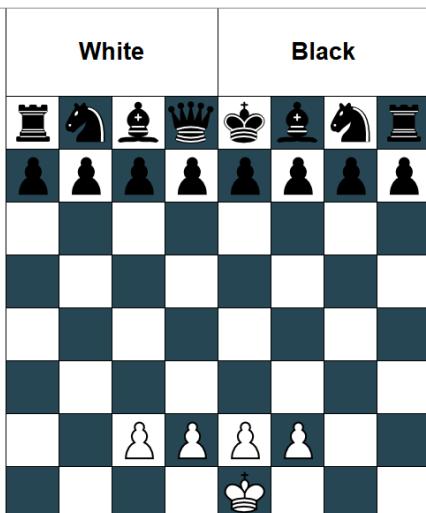
— □ ×



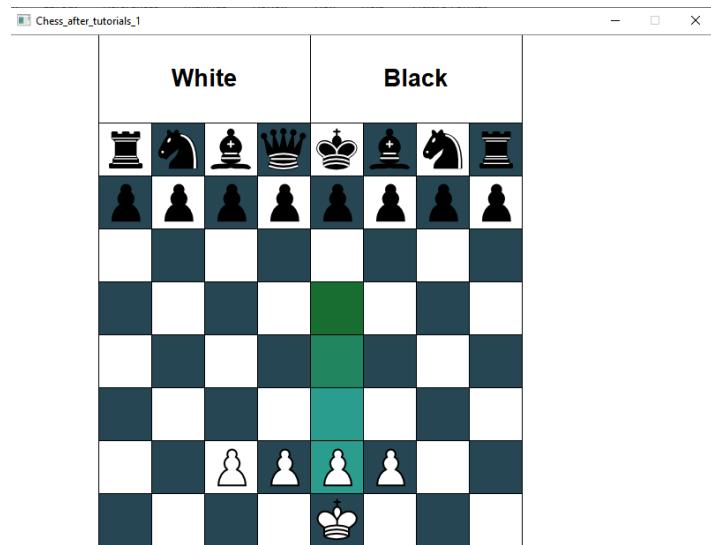
I next added images of pieces to show where the pieces would be:



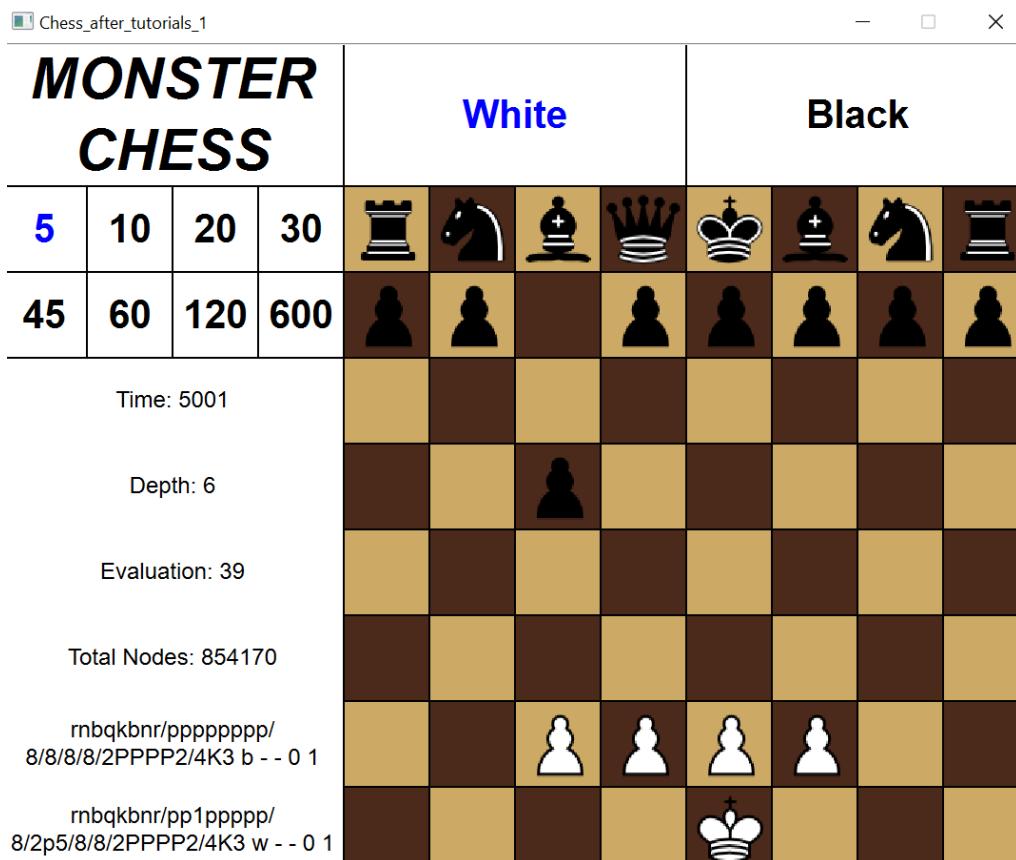
And then added buttons so that you could choose whether to play as black or as white:



Next, I showed all the legal moves you could play when you click on a piece to move it:



Finally, for the finishing touches, I added the ability to choose how much time the engine got to think and displayed some of the engine's thoughts on the position after it played a move. I also had a change of heart regarding the colour scheme.



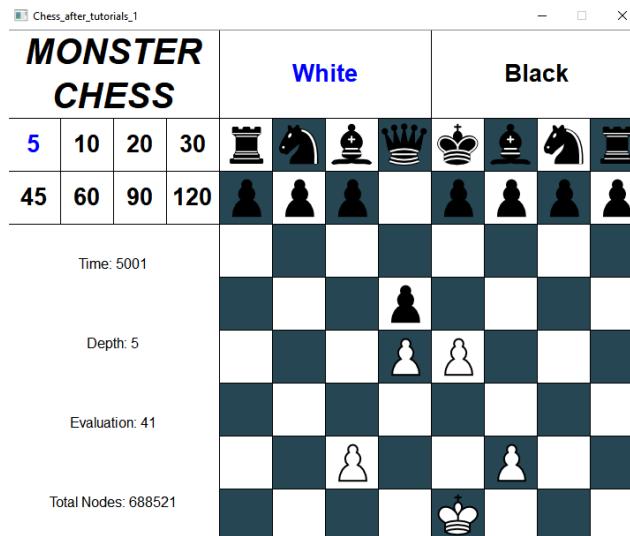
Implementation

The final artefact is split into two main parts. The important theoretical work I have done is to carefully consider how traditional chess engine techniques can be applied to the variant Monster Chess, and to then implement an engine which can play Monster Chess at quite a decent level. Additionally, I have also built a simple-but-functional User Interface in Qt for C++ so that anyone can play against my computer.

Language, User Interface and Portability

I chose to use C++ to code my engine on account of its efficiency at run-time and its built-in Object-Oriented Programming paradigm. I then used Qt for C++ to code a basic UI so that anyone can play against my computer engine. The UI displays several key features while playing a game – the time the computer has spent thinking of a move; the depth it has thought to; the engine’s current evaluation of the board (who it thinks is in the lead currently); and number of total nodes (Monster Chess positions) it has considered (a measure of how much it has been able to evaluate the current position).

I also managed to port all the relevant .exe files and resources onto a USB, so anyone with a computer that runs Windows OS can just plug in my USB and play Monster Chess against my engine.



Move Generation

My engine itself is split into two parts – Move Generation and Move Selection. I had to build my custom move generator within the rules of Monster Chess in order to then build the best Move Selector.

Board Representation

I have used ‘bitboards’ (64-bit integers) to represent various pieces on the chessboard. Bitboards have the advantage of being efficiently and easily manipulated via logic operators such as OR and AND to help quickly generate moves.

Lookup Tables

There are often cases where I had to select all the pieces in the first row, or all pieces not in the last column, or some variation thereof. I again leveraged logical operations on bitboards to do such computations extremely efficiently. But to have maximum efficiency, I pre-computed a lot of extra tables that I often needed to use while generating moves. Specifically, I had two arrays called ‘ClearFile’ and ‘MaskRank’, which contain bitboards which have 0s and 1s in specific rows or columns to either get rid or select these rows. For example:

<i>ClearFile[FILE_A]</i>	<i>MaskRank[RANK_1]</i>
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1

King Movement

I evaluated king movement by first labelling the 8 spots around a King where it could move to as such:



The function that computes the king’s possible moves looks like so:

Input Parameters:

```
BB compute_king(const BB &king_loc, const BB &own_side) {  
}
```

My function takes in the position of the king and a bitboard of all the pieces of the king's own colour - either AllWhitePieces or AllBlackPieces.

Taking care of Edges:

```
BB king_clip_file_h = king_loc & ClearFile[FILE_H];  
BB king_clip_file_a = king_loc & ClearFile[FILE_A];
```

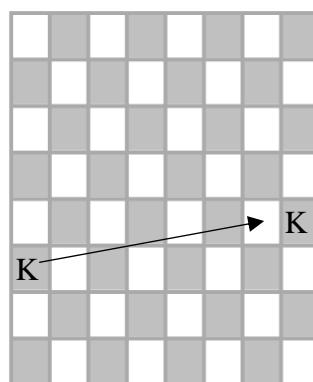
These next two lines create two additional bitboards which will contain the position of the king if it isn't on one of the two edge columns. If it is, these bitboards will just be all 0s.

Calculating potential spots around the King:

```
BB spot_1 = king_clip_file_h << 7;  
BB spot_2 = king_loc << 8;  
BB spot_3 = king_clip_file_a << 9;  
BB spot_4 = king_clip_file_a << 1;  
  
BB spot_5 = king_clip_file_a >> 7;  
BB spot_6 = king_loc >> 8;  
BB spot_7 = king_clip_file_h >> 9;  
BB spot_8 = king_clip_file_h >> 1;
```

Now I made more bitboards to move the king to the various spots. Note that the “<<” and “>>” simply shift the entire bitboards to the left or right, which has the effect of moving the king itself to the left or right. Since shifting by 8 moves the king up or down, shifting by 7 leads the king to spot_1, shifting by 8 to spot_2, etc.

Notice that when considering spots not directly above or below the king, I had to consider the fact that the king might be on an edge row or column. If so, shifting the king by 1 has the effect of moving the king to the opposite side of the board, like so:



As such, I had to make sure to only shift the king 1 to the left if it wasn't in the first column to begin with. A lot of my initial bugs with king movement had to do with this type of error – at times it seemed like the King was teleporting across the board! A similar logic holds for the other edge column. I used the 'king_clip' bitboards I had already made to generate potential spots where the king could end up. I didn't have to worry about the same issue when moving up or down entire rows (i.e. spots 2 and 6) since 64-bit integers automatically discard ones if they move past the 64 bits.

Final Calculation:

```
BB king_moves = spot_1 | spot_2 | spot_3 | spot_4 | spot_5 | spot_6 |
                  spot_7 | spot_8;

BB KingValid = king_moves & ~own_side;
return KingValid;
}
```

Finally, I used the | operator (logical OR) to combine all the spots into one final bitboard that the King could move to. I then used & (the AND operator) and ~ (the NOT operator) to discard all the spots where there are pieces of the same colour as the king (pieces of the same army cannot capture each other in Monster Chess, or in Classical Chess).

Pawn and Knight Movement

Pawns and knights move fundamentally in a similar way to the king – they are not ‘sliders’, like the bishop, rook or queen. As such, their implementation is very similar to that of the king’s, with a few additional complications with the pawns due the asymmetry of how they can move – White and Black pawns can only move in opposite directions. For example, I ended up with this function for the knight:

```
BB one_knight(const BB &knight_loc, const BB &own_side){

    BB spot_1_clip = ClearFile[FILE_A] & ClearFile[FILE_B];
    BB spot_2_clip = ClearFile[FILE_A];
    BB spot_3_clip = ClearFile[FILE_H];
    BB spot_4_clip = ClearFile[FILE_H] & ClearFile[FILE_G];

    BB spot_5_clip = ClearFile[FILE_H] & ClearFile[FILE_G];
    BB spot_6_clip = ClearFile[FILE_H];
    BB spot_7_clip = ClearFile[FILE_A];
    BB spot_8_clip = ClearFile[FILE_A] & ClearFile[FILE_B];

    BB spot_1 = (knight_loc & spot_1_clip) << 10;
    BB spot_2 = (knight_loc & spot_2_clip) << 17;
    BB spot_3 = (knight_loc & spot_3_clip) << 15;
    BB spot_4 = (knight_loc & spot_4_clip) << 6;
```

```

    BB spot_5 = (knight_loc & spot_5_clip) >> 10;
    BB spot_6 = (knight_loc & spot_6_clip) >> 17;
    BB spot_7 = (knight_loc & spot_7_clip) >> 15;
    BB spot_8 = (knight_loc & spot_8_clip) >> 6;

    BB KnightValid = spot_1 | spot_2 | spot_3 | spot_4 | spot_5 | spot_6 |
                      spot_7 | spot_8;

    return (KnightValid & ~own_side);
}

```

Bishop, Rook and Queen Movement

While there were several options for how I should implement ‘slider’ movement (as described in my Log and the Research section), in the end I settled upon Obstruction Difference as my method of choice. This had the added benefit that both the bishop and the rook movement could be implemented with virtually the same code.

Obstruction Difference first takes in the current occupancy of the board, as well as the position of the piece in consideration (rook, bishop or queen) and an integer which represents a direction. This direction could be horizontal, vertical, or one of the two diagonals. A large table called ‘sliding_piece_attacks’ is precomputed. This 2-dimensional array has dimensions 64 by 4 and contains the squares a piece could hit in a particular direction from a square, if the board were empty. For example, `sliding_piece_attacks[16][0]` refers to the vertical squares a rook on the square h3 could hit. As such the bitboard ‘occ’ contains the pieces which are in the line of sight of a particular piece in a particular direction.

Next, the variable ‘temp’ is a way of splitting the occupancy bitboard into an upper and lower half. Essentially, it has a value of 1 for every position higher than the original position of the rook and a value of 0 for every position lower. I used this ‘temp’ variable to now generate the ‘upper’ and ‘lower’ variables, by also using the current occupancy map.

Now the next step was to select all the squares in between the lowest bit of the upper bitboard and the highest bit of the lower bitboard. This was done via classic techniques of bit-scanning backwards (as described in [8]) and an efficient bit trick ([9]). Finally, I had to AND this ‘odiff’ bitboard together

⁸ Chess Programming Wiki, Wikimedia. “BitScan – Divide and Conquer.” Last accessed 16 November 2020.
https://www.chessprogramming.org/BitScan#Divide_and_Conquer_2

⁹ Chess Programming Wiki, Wikimedia. “General Setwise Operations - Least Significant One.” Last accessed 16 November 2020.

https://www.chessprogramming.org/General_Setwise_Operations#TheLeastSignificantOneBitLS1B

with the original array entry for potential squares to get rid of 1s in other directions, giving a final valid bitboard.

Simply by varying the integer ‘direc’, I could do the same for the three other directions, thus giving us valid bitboards for bishop, rook and queen moves.

The hard part had now been done; all that was left was sending the input in a nice fashion to combine several directions per piece. For the rook, this looked like this:

```
BB one_rook(const BB &black_rook_loc, const BB &all_pieces, const BB
&own_side) {

    BB horiz = (lineAttacks(all_pieces, findPosition(black_rook_loc), 0) &
~own_side);
    BB vertic = (lineAttacks(all_pieces, findPosition(black_rook_loc), 1) &
~own_side);

    return horiz | vertic;
}
```

The ‘one_rook’ function just computes the horizontal and vertical potential moves via the ‘lineAttacks’ function, makes sure that no captures are in fact captures of their own side, and then returns both the directions as one valid bitboard. I similarly implemented the bishop and queen move generation function.

Converting Bitboards to Chessboards

I had created bitboards for each piece to show where that piece could end up next move – I had generated all pseudo-legal moves in the form of bitboards. However, two problems remained:

- I needed to be able to separate a bitboard for, say, all “BlackKnights” into its component 1 bits in order to feed each knight’s position to the ‘one_knight’ function.
- I still needed a way to move the pieces - I needed to store chessboards for each potential move where that move had been played.

These two problems are interconnected, and I solved them via ‘compute_piece’ functions like so:

```
void compute_bishop(BB bishop_locs, const BB &own_side, vector<ChessBoard>
&pot_moves, const ChessBoard &start_pos, const int color, const int
type_of_piece){
    while (bishop_locs != 0){

        int temp_bishop_loc = findAndClearSetBit(bishop_locs);
        BB bishop_loc = (ULL << temp_bishop_loc);
        BB bishop_moves =
one_bishop(bishop_loc,start_pos.AllPieces,own_side);

        createAllMoves(pot_moves, start_pos, bishop_moves,
findPosition(bishop_loc), type_of_piece);
    }
}
```

```
}
```

This function takes in a few inputs – a bitboard for the positions of all the bishops, and a bitboard for its own colour’s pieces. It also takes in a vector (C++’s version of an array you can easily add elements to) called ‘pot_moves’ whose elements will be filled up with all the potential moves which the player could end up making. It also needs the current position (in the form of the ChessBoard-type object ‘start_pos’) and the colour that the player currently making a move is.

The function then makes extensive use of the helper function ‘findAndClearSetBit’. This helper function goes through the bitboard ‘bishop_locs’ and finds the positions of each bishop, giving them one by one until no more are found. These positions are then sent to the function ‘one_bishop’, which works similarly to ‘one_rook’, as discussed before. This solves the first of the two issues I had.

For each valid bitboard of potential, pseudo-legal moves created, I now had to make new ChessBoard objects which I would append to the ‘pot_moves’ vector. This is where I had to reckon with the second issue, which in the above code is alluded to via the ‘createAllMoves’ function.

In the end, this is how I coded ‘createAllMoves’:

```
void createAllMoves(vector<ChessBoard> &pot_moves, const ChessBoard
&start_pos, BB &mask, const int from_sq, int type_of_piece) {

    while(mask != 0) {
        int to_sq = findAndClearSetBit(mask);
        createMove(from_sq, to_sq, pot_moves, start_pos, type_of_piece);
    }
}
```

Once again, I used the ‘findAndClearSetBit’ function to separate out individual 1s from a bitboard, and then I send off the relevant information to the function that does the real magic – ‘createMove’. createMove’s pseudocode looks like this:

1. Take in parameters of the squares we’re going to and from; the list of all potential moves we need to append to; the type of piece we’re dealing with; and the initial configuration of the chessboard.
2. Initialise a copy of the ChessBoard object we’ve been given.
3. Check whether a piece exists at the square we want to move the piece to; if it does, store this piece in a variable.
4. Using bit logic, wipe the 1s of the squares we’re moving to and from for each bitboard in the new ChessBoard object.
5. Depending on which type of piece we began with, we need to add back a 1 to the bitboard of this piece in the new ChessBoard object.
 - a. Here I made several rather hilarious bugs where pawns would make it to the last rank and then fail to promote, instead just remaining pawns.

- Add our new ChessBoard object with the moves having been made to the list of potential words.

Checks and Combining Functions

I now had to add together various elements to adapt the move generator I had built from normal chess to Monster Chess. Here I reused some of what I had learned while making Prototype 1 – namely, the idea of having a function called ‘one_move_without_check’ and then using this as a building block for white and black’s potential legal moves:

```
void one_move_without_check(vector<ChessBoard> &pot_moves, const ChessBoard
&start_pos, const int color){
    if (color==0){

        BB black_king_moves = compute_king(start_pos.BlackKing,
start_pos.AllBlackPieces);

        createAllMoves(pot_moves, start_pos, black_king_moves,
findPosition(start_pos.BlackKing), BLACK KING INT);

        compute_knight(start_pos.BlackKnights, start_pos.AllBlackPieces,
pot_moves, start_pos, color);
        compute_black_pawns(start_pos.BlackPawns, start_pos.AllBlackPieces,
pot_moves, start_pos, color);
        compute_rook(start_pos.BlackRooks, start_pos.AllBlackPieces,
pot_moves, start_pos, color, BLACK ROOK INT);
        compute_bishop(start_pos.BlackBishops, start_pos.AllBlackPieces,
pot_moves, start_pos, color, BLACK BISHOP INT);
        compute_rook(start_pos.BlackQueens, start_pos.AllBlackPieces,
pot_moves, start_pos, color, BLACK QUEEN INT);
        compute_bishop(start_pos.BlackQueens, start_pos.AllBlackPieces,
pot_moves, start_pos, color, BLACK QUEEN INT);

    } else if (color==1){

        BB white_king_moves = compute_king(start_pos.WhiteKing,
start_pos.AllWhitePieces);

        createAllMoves(pot_moves, start_pos, white_king_moves,
findPosition(start_pos.WhiteKing), WHITE KING INT);

        compute_white_pawns(start_pos.WhitePawns, start_pos.AllWhitePieces,
pot_moves, start_pos, color);

        compute_rook(start_pos.WhiteQueens, start_pos.AllWhitePieces,
pot_moves, start_pos, color, WHITE QUEEN INT);
        compute_bishop(start_pos.WhiteQueens, start_pos.AllWhitePieces,
pot_moves, start_pos, color, WHITE QUEEN INT);

    }

}
```

I split the function into two cases based on which colour’s turn it was. After that, I had to calculate and append each type of piece’s moves to the ‘pot_moves’ vector.

I built upon this to build a couple additional functions:

```
void one_move_with_check(vector<ChessBoard> &pot_moves, const ChessBoard
&start_pos, const int color) {
    vector<ChessBoard> to_become;

    one_move_without_check(pot_moves, start_pos, color);
    for (auto x: pot_moves) {
        if (!is_someone_in_check(x, color)) {
            to_become.push_back(x);
        }
    }
    pot_moves = to_become;
}
```

Here I check every resulting chessboard after making a pseudo-legal move to see whether the colour whose turn has just passed is in check. If it isn't, I add it to a new vector, which then becomes the original vector. In this way I got rid of any illegal positions. A function called ‘two_moves_with_check’ is similarly defined, with some additional recursion to give white’s potential moves.

It’s worth talking a little about the implementation details when it comes to the ‘is_someone_in_check’ function. In its current state, I believe this function is what is causing the most inefficiency in my move generation function. I have a somewhat naïve implementation that looks at all potential pseudo-legal moves from the proposed position, checks that the king is not attacked in any of those cases, and then verifies a move as legal. This is computationally inefficient. In my research I did come across better methods (such as in [10]), but I realized that these would require major restructuring of my code, so I left it to a later date.

In summation, here’s my code for the function ‘legal_moves’:

```
void legal_moves(vector<ChessBoard> &pot_moves, const ChessBoard
&start_pos, const int color, bool to_sort) {

    if (color==0) {
        ChessBoard tempcopy = start_pos;
        one_move_with_check(pot_moves, start_pos, color);

    } else {
        two_moves_with_check(pot_moves, start_pos, color);
    }
}
```

¹⁰ Chess Programming Wiki, Wikimedia. “Checks and Pinned Pieces (Bitboards).” Last accessed 8 May 2018.
[https://www.chessprogramming.org/Checks_and_Pinned_Pieces_\(Bitboards\)](https://www.chessprogramming.org/Checks_and_Pinned_Pieces_(Bitboards))

Move Selection

Minimax (Negamax) Algorithm

In the end, I settled on a negamax implementation of the classic 2-player combinatorial depth-first search algorithm Minimax.

```
int negaMax(int depth) {
    if (depth == 0) return evaluate();
    int max = -oo;
    for (all_moves) {
        score = -negaMax( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
```

The code works recursively, first making sure we are not at a leaf node (the end of the tree in the diagram above), and if so returning a static evaluation. If we are somewhere in the middle of the search tree, then we consider all the opponent's moves, select their best one recursively and define the current node's value accordingly.

Improvements – Alpha-Beta Pruning

A relatively easy-to-implement improvement I used is called Alpha-Beta pruning, which cuts down on the search tree the algorithm looks through by considering if it has already found a better potential move. This only requires two extra variables, alpha and beta, and is implemented as so:

```
int alphaBeta( int alpha, int beta, int depthleft ) {
    if(depth == 0) return evaluate(alpha, beta);
    for (all_moves) {
        score = -alphaBeta( -beta, -alpha, depthleft - 1 );
        if( score >= beta )
            return beta; // fail hard beta-cutoff
        if( score > alpha )
            alpha = score; // alpha acts like max in MiniMax
    }
}
```

Static Evaluation

I had initially wanted to use the neural network / machine learning algorithm described in [11], but quickly realized this would require too large a restructuring of my code for questionable additional

¹¹ Sacha Droste, Johannes Fürnkranz. “Learning of Piece Values for Chess Variants.” *Technical Report TUD-KE-2008-07*, Knowledge Engineering Group, Technische Universität Darmstadt. <http://www.ke.tu-darmstadt.de/publications/reports/tud-ke-2008-07.pdf>

benefit. I thus stuck to traditional chess evaluations for the black pieces and valued white (Monster) pawns at the worth of a black bishop. I also added a heuristic for the distance between kings, since this was often a deciding factor when playing games of Monster Chess against other humans.

Further Improvements

My final move selection algorithm also makes use of Zobrist Hashing, Transposition Tables and Iterative Deepening, all of which have been previously explained. Altogether, they provided quite a decent efficiency boost, and also made the game more enjoyable to play against the machine, since you could reliably know how long it would take to make a move.

Testing

Most of the testing of my algorithm took place during the build – I was constantly fixing bugs and making sure my code was working as I expected it to. There were two periods when I specifically stress-tested my code: once after completing move generation and once after I had built the User Interface.

Move Generation Testing

I decided to make use of a common library in python which also implements chess moves called PyChess. I then randomly generated hundreds of chess positions and asked both the PyChess library and my C++ code how many legal moves it saw for the side whose turn it was to play. In this way, I

```
SINGLE (NORMAL CHESS) MOVE GENERATION
Position 1:
Expected moves ..... 19
Python moves..... 19
C++ moves..... 19

Position 2:
Expected moves ..... 15
Python moves..... 15
C++ moves..... 15

Position 3:
Expected moves ..... 12
Python moves..... 12
C++ moves..... 12

Position 4:
Expected moves ..... 6
Python moves..... 6
C++ moves..... 6

Position 5:
Expected moves ..... 18
Python moves..... 18
C++ moves..... 18
```

could make sure that there were no discrepancies between my code and the correct move generators:

Initially, I struggled to get these two figures to consistently match up. I had a particular bug where my code would not be able to see certain moves (such as moving a black knight in front of the black king) which would leave a king in check and as such were not legal. I often had positions like so:

```
Position 15:
Expected moves ..... 5
Python moves..... 3
C++ moves..... 5

Position 16:
Expected moves ..... 23
Python moves..... 20
C++ moves..... 23

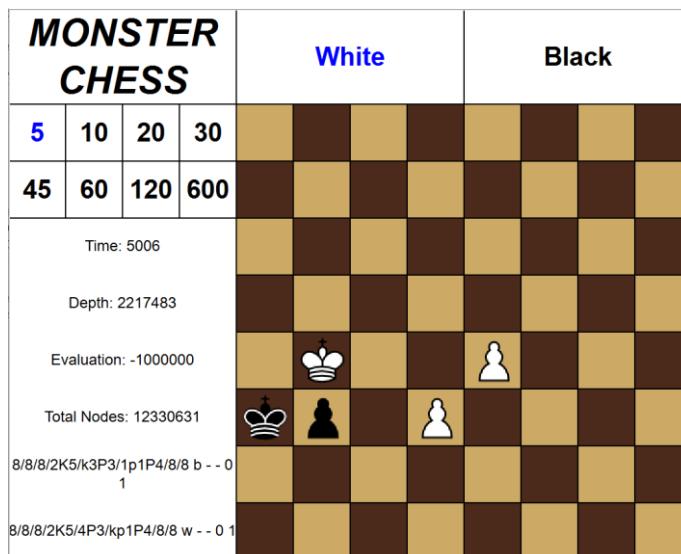
Position 17:
Expected moves ..... 17
Python moves..... 14
C++ moves..... 17
```

In the end I was able to fix this bug by going into the ‘is_someone_in_check’ function and implementing more stringent checks to make sure no move left the king exposed. Specifically, I made sure to check after each potential move that the king was not left in check by adding the ‘is_someone_in_check’ function within the other move generation functions.

User Interface Testing

I had several versions of my User Interface, of gradually increasing complexity. I’ve detailed the design process in the ‘Design’ section. The testing aspect of the User Interface came when I had already implemented some basic features and played many games against my machine to see how it would respond, and to make sure highlighting legal squares and similar features worked.

While doing so, I came across several bugs, both to do with move selection and with the User Interface itself. For example, on the occasions I managed to checkmate the computer, it failed to realize the game was over and instead would let me ‘eat’ its king, leaving the board like so:



These positions were clearly illegal, and I had to go back into the move validator to implement a game-ending function properly.

Evaluation

I had two main goals when starting this project – to learn more about chess engines in general, and to apply this knowledge to Monster Chess; and to beat my father at the game he had taught to me. I have learned a huge amount during this project. I have gained understanding about how complex games are converted into efficient computer code, how minimax and other 2-player zero-sum algorithms are implemented, how OOP works in C++ on a deeper level, and how to implement basic features of a User Interface in C++. My end project plays Monster Chess at the level of an experienced human and has a UI that anyone can use, regardless of their programming knowledge.

Academic Results

I have not just learned more about writing programs and managing large projects over a long period of time, but also about many aspects of algorithm design and tradeoffs between efficiency, memory usage, and ease of implementation / maintainability.

Project Planning and Research

This project was the biggest and most complex I have ever undertaken. As such, I learned a lot about planning my time and using it effectively. For example, I had begun to work on the User Interface through preliminary designs before I had finalized all the improvements on my Move Selection algorithm, since I was waiting for responses regarding the structure of my code before finishing the core project. In the future I think I will be able to parallelize tasks more, which will make me more efficient overall.

I had to do a large amount of research for this project, as I had never built something so complex from scratch before. Throughout the project, I learned better ways of organizing my research and finally settled on collating everything in my project log.

I also had the peculiar task of having to deal with a project that had to be put on a large hiatus for a period. The fact that I had initially begun some work and research before formally starting the EPQ process meant I had to carefully consider what decisions I had consciously and subconsciously made.

Programming Skills

This project was large and involved many interdependent systems. As such, it stretched my programming knowledge, particularly regarding how to maintain large codebases and how to structure and deliver large projects. The size of the project also encouraged me to use and maintain good coding conventions and style, keeping my code readable and easy-to-debug – for the most part. At times, I did have to resort to rather ugly code, as in the following snippet:

```
BB naive_zobrist(const ChessBoard &pos) {
    BB to_return = 0ULL;
    ChessBoard work_with = pos;
```

```

        while (work_with.WhitePawns!=0) {
            int position = findAndClearSetBit(work_with.WhitePawns);
            to_return ^= zobrist_table[position][0];
        }
        while (work_with.WhiteQueens!=0) {
            int position = findAndClearSetBit(work_with.WhiteQueens);
            to_return ^= zobrist_table[position][1];
        }
        while (work_with.WhiteKing!=0) {
            int position = findAndClearSetBit(work_with.WhiteKing);
            to_return ^= zobrist_table[position][2];
        }
        while (work_with.BlackPawns!=0) {
            int position = findAndClearSetBit(work_with.BlackPawns);
            to_return ^= zobrist_table[position][3];
        }
        while (work_with.BlackRooks!=0) {
            int position = findAndClearSetBit(work_with.BlackRooks);
            to_return ^= zobrist_table[position][4];
        }
        while (work_with.BlackKnights!=0) {
            int position = findAndClearSetBit(work_with.BlackKnights);
            to_return ^= zobrist_table[position][5];
        }
        while (work_with.BlackBishops!=0) {
            int position = findAndClearSetBit(work_with.BlackBishops);
            to_return ^= zobrist_table[position][6];
        }
        while (work_with.BlackQueens!=0) {
            int position = findAndClearSetBit(work_with.BlackQueens);
            to_return ^= zobrist_table[position][7];
        }
        while (work_with.BlackKing!=0) {
            int position = findAndClearSetBit(work_with.BlackKing);
            to_return ^= zobrist_table[position][8];
        }

    return to_return;
}

```

Here the constant repetition is a hallmark of code that should be abstracted away, most likely through a some sort of for loop. For example, it would be quite cumbersome to add another piece to this function. However, considering I knew beforehand exactly which pieces could ever possibly be on the chessboard, I did not invest the time required to make this function more readable. This might be a problem if I ever wanted to generalize away from Monster Chess to other variants of chess, and is something I will keep in mind for the future.

That said, I do feel that overall the code is generally clean and well structured. I have split my project into several different files – for example, ‘move_selection.cpp’, ‘debug.cpp’, ‘chessboard.cpp’ and so on – and C++’s default structure of header and source files helped keep my thoughts and code organized.

Development Process

The development process for this project took the vast majority of the duration of the EPQ. In some ways my development process was unusual – the work I did pre-hiatus was a little more haphazard and had to be cleaned up after I decided to formally embark on an EPQ. As such, the time I had allocated to review the first Prototype was simply not enough and took longer than I had expected. This was somewhat concerning at the time, since it suggested that perhaps I had severely underestimated the amount of time required to maintain large projects such as this one. The extra time I had to spend in this phase of the project led, in part, to me not being able to fully explore some large structural changes such as Neural Network static evaluators in the main C++ project. However, I came to realize that this time was well-spent, as I gained a very good idea of the fundamentals of chess engines, that in the end helped speed up the development of my final C++ project. It also meant I had not dramatically misjudged how long the whole project would take me, and that I would be able to fully complete my project as laid out in the project proposal.

Although the length of time spent on Prototype 1 delayed moving on to other sections of the Monster Chess engine, this process was valuable. It allowed me to learn how to deal with unexpected issues and delays in development that will inevitably come up in any project. I learned the importance of doing additional research into what was feasible to implement in a reasonable time period, before jumping to a decision.

I experienced many other bugs in production, but my decision to keep code readable and maintainable meant that I was well-equipped to deal with these issues.

Final Artefact

In the end I organized a 3-match finale between my engine and my father, to see just how strong an engine I had built. Both sides would have black and white alternately. The match ended in two wins for each side and one draw. Interestingly, both sides won with black, which suggests that Monster Chess as a game is perhaps biased towards black. This agrees with some of the research I did, which claimed it was trivial to win with black, but disagrees on the ‘trivial’ part – even as black, my father had to react to the innovative strategies my engine came up with as white.

One of the major issues with my engine is its lack of depth – modern engines can often see to a depth of 30 or 40, while my engine typically thinks to a depth of 7 or 8. This is due in part to hardware differences – modern engines are trained on neural network that are run on hardware orders of magnitude larger than I have access to – but it is also clear that there are several glaring inefficiencies in my project.

The reduced depth means it is not too hard to ‘trick’ my engine as white – if you don’t move your pawns at all, it wastes a lot of computing power move after move calculating the many possible

moves white has. These moves are ones a human would discard immediately, but my engine fails to prune them away efficiently. In the future, I would like to work on more aggressive pruning, perhaps akin to the Monte-Carlo techniques I had come across in my research on neural networks.

The other major improvement I would have loved to explore at more depth is the idea of Neural Networks generating better static evaluators than the ones I (somewhat arbitrarily) assigned to the machine.

Ease of Use

The final application is extremely easily portable to any computer running Windows – the end project fits in my USB. The process of making my program portable was surprisingly easy, since Qt for C++ (the UI software I used) has some built-in support for doing just this. However, my application is still not properly deployed – for example, I cannot easily share it across the internet through common tools like sourceforge.net, nor can I port it to Apple computers. This was a matter of development resources – I focused on getting a stable application to work on the laptops I was using, rather than focusing in detail on user-facing applications.

Conclusions

I feel that through my EPQ project, I have learned a great deal about large-scale programming projects, code maintainability and designing large projects. I have also learned about how modern chess engines work, and how many types of decision-making algorithms work. My final project includes all the features I had set out to include, including a usable User Interface. My engine is of reasonable strength, and despite some clear drawbacks puts up a reasonable fight against my father. I have also learned valuable lessons about time management and planning. As a result of this project, I am a better programmer with a stronger practical and theoretical understanding of large coding projects.

Bibliography

Chess Programming Wiki, Wikimedia. “Mailbox.” Last accessed 13 June 2021.

<https://www.chessprogramming.org/Mailbox>

Chess Programming Wiki, Wikimedia. “Bitboards.” Last accessed 16 September 2021.

<https://www.chessprogramming.org/Bitboards>

Peter Keller, “Physical Location Bitboards.” Last accessed 16 September 2021.

<https://www.chessprogramming.org/Bitboards>

Peter Keller, “Physical Location Bitboards.” *University of Wisconsin-Madison*, accessed 25th October 2021. <http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/physical.html>

Chess Programming Wiki, Wikimedia. “Magic Bitboards.” Last accessed 8 July 2021.

https://www.chessprogramming.org/Magic_Bitboards

Chess Programming Wiki, Wikimedia. “BitScan – Divide and Conquer.” Last accessed 16 November 2020. https://www.chessprogramming.org/BitScan#Divide_and_Conquer_2

Chess Programming Wiki, Wikimedia. “General Setwise Operations - Least Significant One.” Last accessed 16 November 2020.

https://www.chessprogramming.org/General_Setwise_Operations#TheLeastSignificantOneBitLS1B

Chess Programming Wiki, Wikimedia. “Checks and Pinned Pieces (Bitboards).” Last accessed 8 May 2018. [https://www.chessprogramming.org/Checks_and_Pinned_Pieces_\(Bitboards\)](https://www.chessprogramming.org/Checks_and_Pinned_Pieces_(Bitboards))

Chess Programming Wiki, Wikimedia. “Search – Shannon’s Types.” Last accessed 30 April 2021.

https://www.chessprogramming.org/Search#Shannon.27s_Types

Chess Programming Wiki, Wikimedia. “Minimax.” Last accessed 4 April 2021.

<https://www.chessprogramming.org/Minimax>

Janis Griebel, Jos Uiterwijk. “Combining Combinatorial Game Theory with an α - β Solver for Clobber.” *Communications in Computer and Information Science*, November 2016.

https://www.researchgate.net/profile/Jos-Uiterwijk/publication/309668700_Combining_Combinatorial_Game_Theory_with_an_a-b_Solver_for_Clobber/links/581c552408aea429b291add5/Combining-Combinatorial-Game-Theory-with-an-a-b-Solver-for-Clobber.pdf

Sacha Droste, Johannes Fürnkranz. “Learning of Piece Values for Chess Variants.” *Technical Report TUD-KE-2008-07*, Knowledge Engineering Group, Technische Universität Darmstadt.

<http://www.ke.tu-darmstadt.de/publications/reports/tud-ke-2008-07.pdf>

Chess Programming Wiki, Wikimedia. “Negamax.” Last accessed 27 April 2018.

<https://www.chessprogramming.org/Negamax>

Chess Programming Wiki, Wikimedia. “Alpha-Beta.” Last accessed 17 June 2021.

<https://www.chessprogramming.org/Alpha-Beta>

Mediocre Chess. “[Guide] Transposition tables.” Last accessed 1 January 2007.

<http://mediocrechess.blogspot.com/2007/01/guide-transposition-tables.html>

Bruce Moreland, Seanet. “Quiescent Search” Last accessed 4 November 2002.

<https://web.archive.org/web/20071027170528/http://www.brucemo.com/compchess/programming/quiescent.htm>

Chess Programming Wiki, Wikimedia. “Quiescence Search.” Last accessed 15 June 2021.

https://www.chessprogramming.org/Quiescence_Search

Chess Programming Wiki, Wikimedia. “Move Ordering.” Last accessed 4 July 2021.

https://www.chessprogramming.org/Move_Ordering

Chess Programming Wiki, Wikimedia. “Zobrist Hashing.” Last accessed 7 February 2021.

https://www.chessprogramming.org/Zobrist_Hashing

Bruce Moreland, Seanet. “Iterative Deepening.” Last accessed 4 November 2002.

<http://web.archive.org/web/20070705134347/http://www.seanet.com/~brucemo/topics/iterative.htm>

Peter Keller, “Chess and Bitboards.” *University of Wisconsin-Madison*, accessed 25th October 2021.

<http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/index.html>

Wikipedia, Wikimedia. “Monster chess.” Last accessed 15 September 2021.

https://en.wikipedia.org/wiki/Monster_chess

The Chess Variant Pages. “Monster Chess.” Last accessed 14 February 2001.

<http://www.chessvariants.org/unequal.dir/monster.html>

The Chess Variant Pages. “Muenster Chess.” Last accessed 15 January 1997.

<http://www.chessvariants.org/d.betza/chessvar/muenster.html>

ChessProgramming, Reddit. “What improvements can I make to my naive minimax algorithm?” Last accessed 10 January 2021.

https://old.reddit.com/r/chessprogramming/comments/kudw1p/what_improvements_can_i_make_to_my_naive_minimax/

Computer Chess Club, TalkChess.com. “How to efficiently generate individual moves from bitmasks of pieces?” Last accessed 15 January 2021.

<https://talkchess.com/forum3/viewtopic.php?f=7&t=76326&p=879475#p879475>

Appendix 1 – Presentation Slides

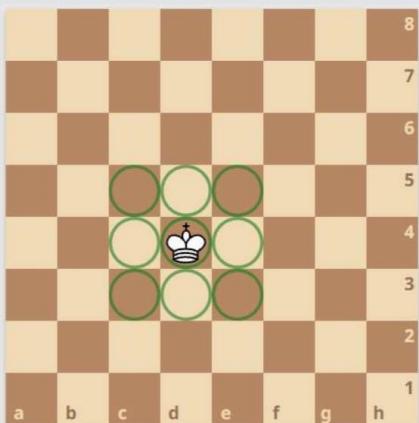


What is Monster Chess?

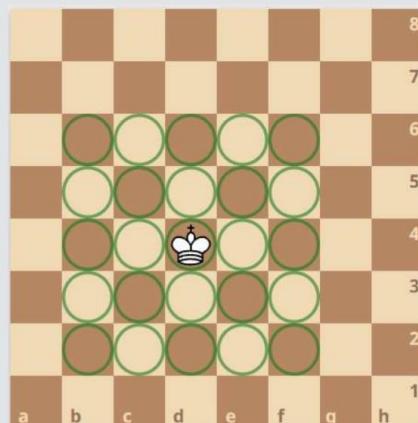
- Variant of Chess
- Material Imbalance
- 2 White Moves : 1 Black Move



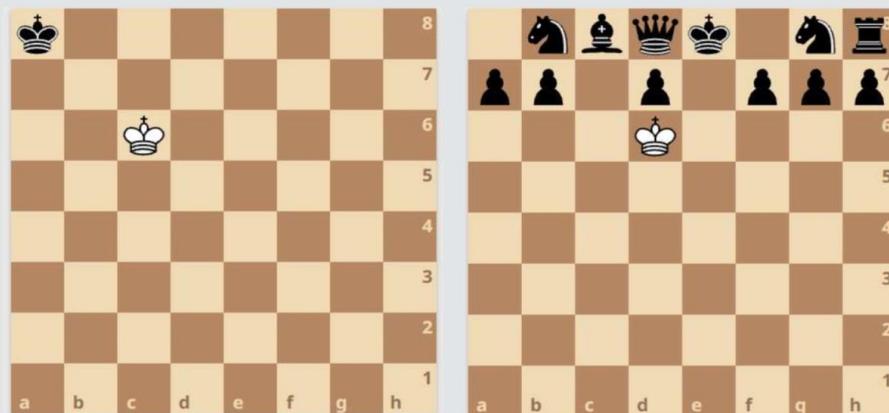
Classical Chess



Monster Chess

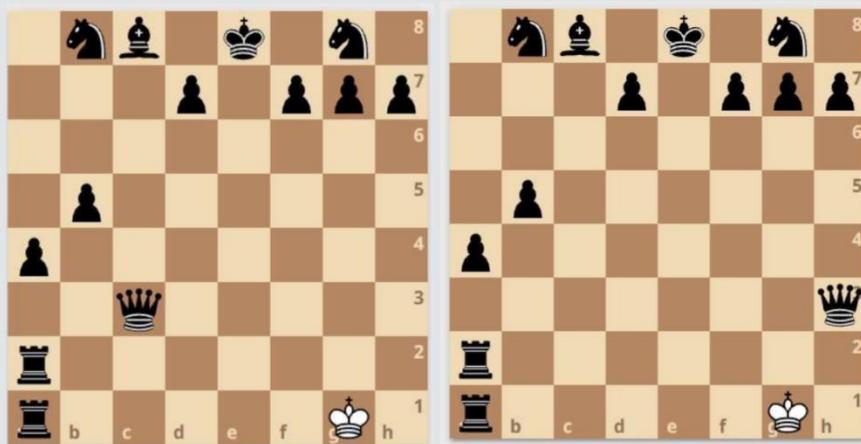


Examples of White Checkmating Black



Monster King Checkmated

Not Checkmate – The Monster can capture the Black Queen



White to Move

*White's 1st Move of their Turn,
Capturing the Knight*

*White Moves back, out of Check,
on their 2nd Move*



Motivation

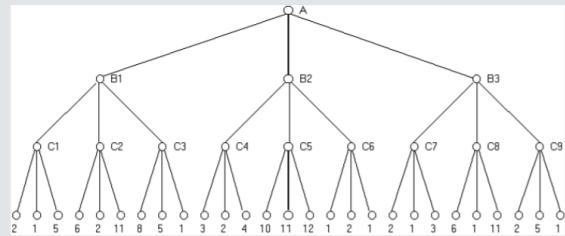
- Explore adversarial programs
- Maintain a large codebase
- Beat my dad!

How do traditional chess algorithms work?

**Move
Generation**

**Move
Selection**

Exponential growth – why speed is important



- Number of possible positions increases very quickly as you search further along in the future
- Speed is essential for a strong engine

Move Generation

- Classical, simple approach:
 - Lots of for loops
 - Represent each piece separately as an object
- Bitboards



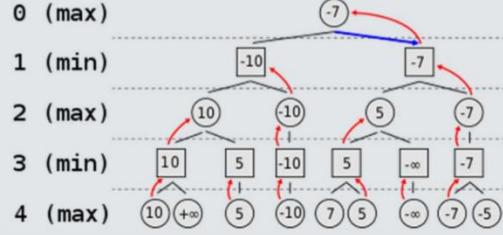
, etc

Move Selection - The Minimax Algorithm

- Static evaluation – who's winning if game was paused?

	1
	3
	3
	5
	9
	+∞
	-1
	-3
	-3
	-5
	-9
	-∞

Move Selection - The Minimax Algorithm



- If we knew opponent's best move, what do we play?
- Work out opponent's best move in a recursive manner
- More depth = stronger engine

Optimizations

- Alpha-beta pruning – hacking branches
- Transposition Tables
- Iterative Deepening

Weirdest Bugs



King teleportation



Horizon effect

Let's Play the Engine!

MONSTER CHESS				White	Black
5	10	20	30		
45	60	120	600		
Time: 5006					
Depth: 2217483					
Evaluation: -1000000					
Total Nodes: 12330631					
8/8/8/2K5/k3P3/1p1P4/8/8 b - - 0 1					
8/8/8/2K5/4P3/kp1P4/8/8 w - - 0 1					





**Any
Questions?**

