WN 2022 Capstone Final Report
Sidney Meade
April 25, 2022

INTRODUCTION

Since the creation of the first chess engines in the early 1950s, the rate at which they have improved is quite astonishing and has been achieved under a number of different algorithmic and hardware-related advancements. For example, many improvements to chess engine heuristics were discovered in the 1960s and 1970s, relying on the use of the Minimax algorithm proven by John von Neumann in 1928, which virtually all modern-day chess engines use some form of. In addition to this, the search time used for the Minimax algorithm significantly decreased due to the doubling of computing power nearly every two years, as Moore's law observes. Regardless of these improvements, the best chess engines remained inferior to top human Grandmasters all the way up until the 1980s. However, beginning in the 1990s, engines began to turn the table. Many companies, such as IBM, began investing in chess engine research and creation, greatly increasing the quality of contemporary chess engines. Finally, in 1996, IBM's Deep Blue chess engine defeated Russian chess player Gary Kasparov in a match, making it the first engine ever to defeat a sitting world chess champion. This marked a turning point in chess engine history, and many chess engines soon began to start dominating even the best chess players humanity had to offer. Today, the highest-rated chess engine, Stockfish, has a chess rating of ~3546 ELO while the highest-rated human player in the world, Magnus Carlsen, has a chess rating of ~2865 ELO, significantly less than Stockfish. While there are many chess engines today that are much stronger than any human, many of them follow the same general procedure when searching for the best move, including Minimax/Negamax pruning with several heuristics to help reduce the number of nodes being searched, thus reducing the time needed to search for the best move. The difference, however, between top engines and other engines largely comes down to the heuristics they employ and the speed at which they operate, making heuristics an important part of move searching for modern-day chess engines.

While many chess engine heuristics are well documented across multiple sources, it is difficult to find comparisons between one heuristic vs. another and the effects of combining multiple heuristics. Such information may be helpful to developers who need to limit the size of the source code being deployed since heuristic implementation generally results in a noticeable increase in source code size. This can have ramifications in embedded entertainment systems that seek to incorporate chess engines, where low source code size is important to the functioning of the application. One example of an embedded entertainment system successfully incorporating a chess engine is Delta's in-flight entertainment system, which anecdotally features a very strong chess engine[1].

The goal of this project was to create a complete chess engine that stores the board state, explores possible moves, uses heuristics to efficiently calculate the best moves, and interacts with a GUI to provide a playable experience for users. Following this, different heuristics could then be compared to better understand which had the greatest reduction in search space from the same engine employing no heuristics. Additionally, this project sought to find combinations of different heuristics that complement each other or may work better together than individually. Ideally, the chess engine should be able to quickly update board state to allow for heuristics to be explored when the engine is searching at greater depths. To achieve this, the chess engine created through this project was designed to update the board state and make moves as quickly as possible, without much concern for memory used. Ideally, an embedded systems engineer would also optimize their engine to reduce space consumed during the search procedure as well as source code size, but for the sake of simplicity, this is ignored in this project. Another thing to consider when comparing chess engine heuristics is the fixed position evaluation method, which can have an effect on the search space, especially with pruning heuristics which depend on fixed position evaluations to decide which branches of the search space should be pruned. The method of fixed position evaluation used for this project will be discussed further in the paper, but largely, the effect of fixed position evaluation on the search space is negligible when a pruning heuristic is employed.

PROBLEMS ADDRESSED

As touched upon in the previous section, the skill level of many modern-day chess engines is largely dependent upon the heuristics they employ. Often times these heuristics noticeably increase the size of the source code, which can be a problem when working with embedded systems[2]. Therefore, one problem this project seeks to address is the tradeoff between heuristic performance and source code size for two common chess engine heuristics: Minimax/Negamax Pruning and Move Ordering. In addition, this project seeks to determine how these heuristics complement each other and how these heuristics, specifically Move Ordering, can be modified to allow for the greatest reduction in search space from a chess engine performing an exhaustive search. Finally, this project sought to address the issue of chess engine design and make recommendations for those just beginning chess engine construction, as it can be overwhelming for many beginners and there are a lot of opportunities to make poor design decisions.

BOARD AND MOVE REPRESENTATION

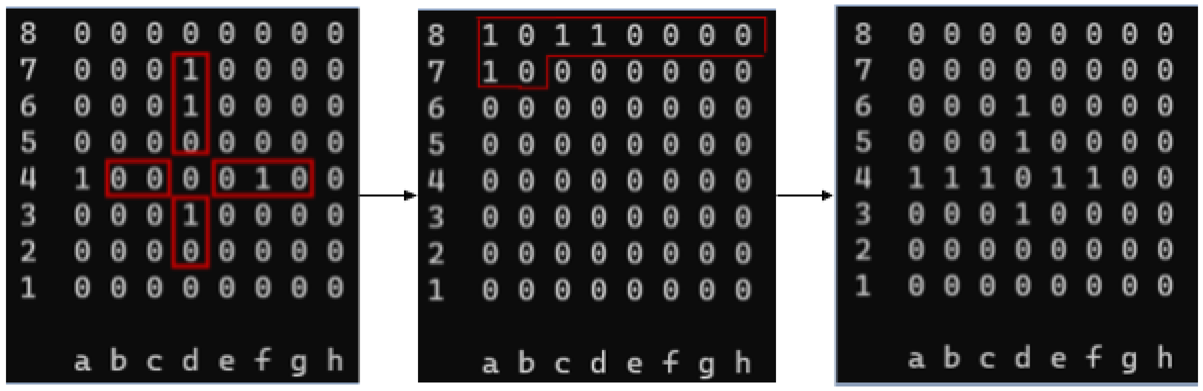
To begin data collection, there were two options available: download and modify an existing chess engine, such as Stockfish, to only employ the Negamax pruning algorithm and Move Ordering heuristics, or create a new chess engine that relies solely upon these heuristics. It was decided to create an entirely new chess engine since this would encourage a deeper understanding of the heuristics as well as how chess engines function and the ability to provide advice for chess engine design. There was also the added benefit of being able to have greater control over the engine since it can be difficult to modify existing open source engines, many of which contain large amounts of legacy code. In other words, using a third party engine may make it difficult to isolate the desired heuristics to explore.

To start construction of a custom-built chess engine, I first started researching chess engine design best practices and searching through the source code of other chess engines to get a better understanding of where to start. One resource I found particularly helpful was the Chess Programming Wiki[3], which provides a plethora of documentation as well as a guide to getting started.

I began constructing the engine by allowing it to calculate potential moves for each piece based on how that piece moves. The main goal was to achieve this both correctly and efficiently. When I began, I quickly realized that the time it took to calculate sliding pieces like rooks, bishops, and queens was approximately 10 times that of other pieces. To speed this up, I used Magic Bitboard[4] sliding piece move generation, which involves

1. For each type of sliding piece and for each possible square, generating all possible blocker configuration bitboards for that piece, where each bitboard represents a possible placement of other pieces that could block the full movement of the sliding piece
2. Multiplying by a "magic number," which is a specialized number used to obtain a unique hash key for a particular blocker mask. Magic numbers can be randomly generated and tested to make sure that multiplying by all possible blocker configurations yields a unique key for each configuration.
3. Calculating the valid moves bitboard given this blocker mask and storing it using the hash key.

This pipeline can be observed in the following Figure 1.



```
8  0 0 0 0 0 0 0 0
7  0 0 0 1 0 0 0 0
6  0 0 0 1 0 0 0 0
5  0 0 0 0 0 0 0 0
4  1 0 0 0 0 1 0 0
3  0 0 0 1 0 0 0 0
2  0 0 0 0 0 0 0 0
1  0 0 0 0 0 0 0 0

   a b c d e f g h
```

```
8  1 0 1 1 0 0 0 0
7  1 0 0 0 0 0 0 0
6  0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0
4  0 0 0 0 0 0 0 0
3  0 0 0 0 0 0 0 0
2  0 0 0 0 0 0 0 0
1  0 0 0 0 0 0 0 0

   a b c d e f g h
```

```
8  0 0 0 0 0 0 0 0
7  0 0 0 0 0 0 0 0
6  0 0 0 1 0 0 0 0
5  0 0 0 1 0 0 0 0
4  1 1 1 0 1 1 0 0
3  0 0 0 1 0 0 0 0
2  0 0 0 0 0 0 0 0
1  0 0 0 0 0 0 0 0

   a b c d e f g h
```

Figure 1: Pipeline for generating efficient sliding piece moves. For each possible blocker mask (left-most image), we multiply by a Magic number to obtain the unique hash key (center image) and store the calculated allowed moves (right-most image) given this blocker mask using the hash key

It should be noted that this occurs before the game starts. When the engine wants to calculate the valid moves given a blocker mask in-game, it simply multiplies the current blocker mask by the corresponding magic number to obtain the correct hash key, which will index the valid moves for that sliding piece given the blockers. In this way, all piece move calculation can be done in-game in O(1) time.

Once I had a method of calculating all possible moves for a given board state, I needed a way to store board position to enable quick pseudolegal move calculation, which involves generating all possible moves before removing invalid moves which leave the friendly king in check. To do this, I used 12 64-bit unsigned integers to represent the board occupation of each type of piece. After implementing this, I realized that it was difficult to find captured pieces when a piece moves, so I created a vector of 64 characters that represents each board square to determine captured pieces in O(1) time. In addition to this, I needed a way of encoding moves to allow for the engine to easily make and unmake moves. To do this, I created a 28-bit unsigned integer with the encoding found in the following figure 2.

Bits	Description
0-5	Represents the square the piece is moving from
6-11	Represents the square the piece is moving to
12-15	Represents the piece being moved [1, 12]
16-19	Represents the piece being captured [0, 12]
20-23	Represents all legal castles remaining
24	Represents whether the move being played is an en passant
25	Represents whether a pawn is being pushed two squares
26	Represents whether a pawn is being promoted on the current move
27	Represents whether the move is a castling move

Figure 2: Move encoding to allow for making and unmaking moves

In this way, board state could be successfully updated by looking at the unsigned integer and making or unmaking the move accordingly.

EVALUATION

Next, the chess engine needed a way to evaluate a fixed position once it reached its maximum depth. It is necessary to have a maximum depth, since in theory chess is a solvable game, yet there are a theorized 10^{120} [5] possible chess games, far too many to evaluate each possible game. To implement the fixed position evaluator, I created a custom evaluation function that takes into account players' material, position, doubled pawns, chained pawns, isolated pawns, passed pawns, and king safety. Additionally, the evaluation function checks to see if the position is either checkmate or stalemate, and provides evaluation accordingly.

Finally, the chess engine needed a way to simulate the Negamax algorithm[6] to find the best move up to a given depth as the engine searches through the space of possible moves. To implement this, I used a depth-first, iterative approach, which involved modifying the best move at a certain depth if the player corresponding to this depth finds a better move to play. I also implemented one of the heuristics I wanted to explore through this project: Negamax pruning. This involves pruning branches that we know no player will want to explore, thus cutting down on the number of nodes searched. An example of Negamax pruning can be seen below in Figure 3.

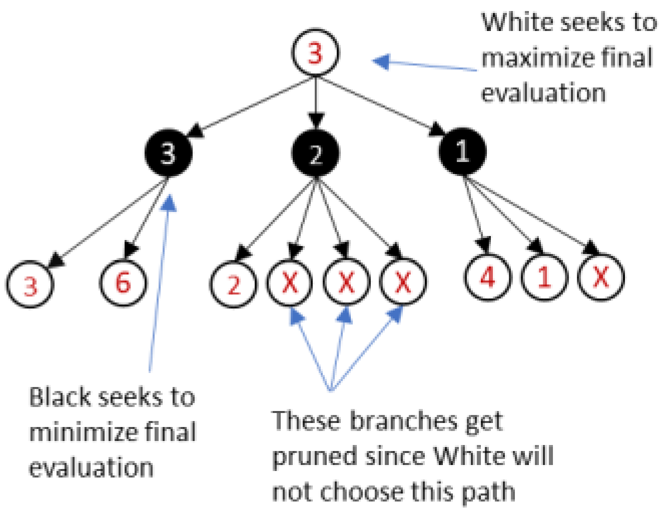

White seeks to maximize final evaluation
3
3
2
1
3
6
2
X
X
X
4
1
X
Black seeks to minimize final evaluation
These branches get pruned since White will not choose this path

Figure 3: Using a depth-first approach, White will find the best move to have an evaluation of 3, then move on to the second branch, where it finds Black can force a move with an evaluation of 2, so it will not take this path. Thus, other sibling branches at a depth of 2 need not be explored.

The second and last heuristic I implemented simply involves ordering the moves to allow for candidate good moves to be explored first. Ideally, a good move to be explored first maximizes the number of branches pruned through Negamax pruning, thus having the greatest reduction in the number of nodes needed to search by the chess engine to find the best move at a given depth. In order to maximize this, it is ideal to look at moves that appear to yield high value for the current player. This will increase the probability that the next move considered for this player will be of lesser quality and thus increase the chances of pruning within this subtree and future subtrees explored. Moves that appear to yield high value to the current player can include types of moves such as checks, captures, and promotions. For the sake of simplicity, the engine implemented for this project only orders captures before passive moves (non-capture moves). In addition to this, capture moves are sorted with the moves involving capture with lower-valued pieces being explored first. The below figure 4 illustrates the order in which moves will be explored by the engine for a sample position.

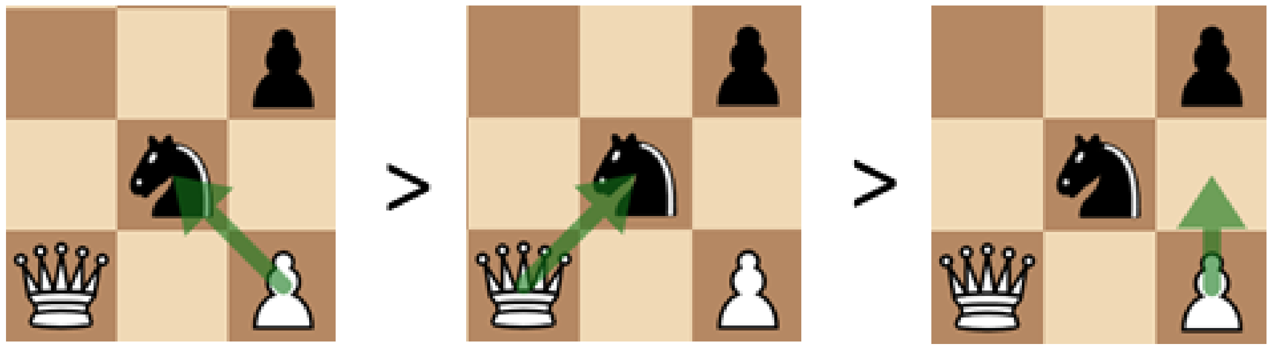


Figure 4: The order in which moves will be explored for the sample position starting with captures made by less valuable pieces, captures made with more valuable pieces, and passive moves.

RESULTS

Following the completion of the chess engine, the effect of the implemented heuristics could then be evaluated. To do this the percent of nodes searched using the heuristics compared to the total nodes searched using an exhaustive search was recorded for the opening position and an obvious capture position as shown below in Figure 5.

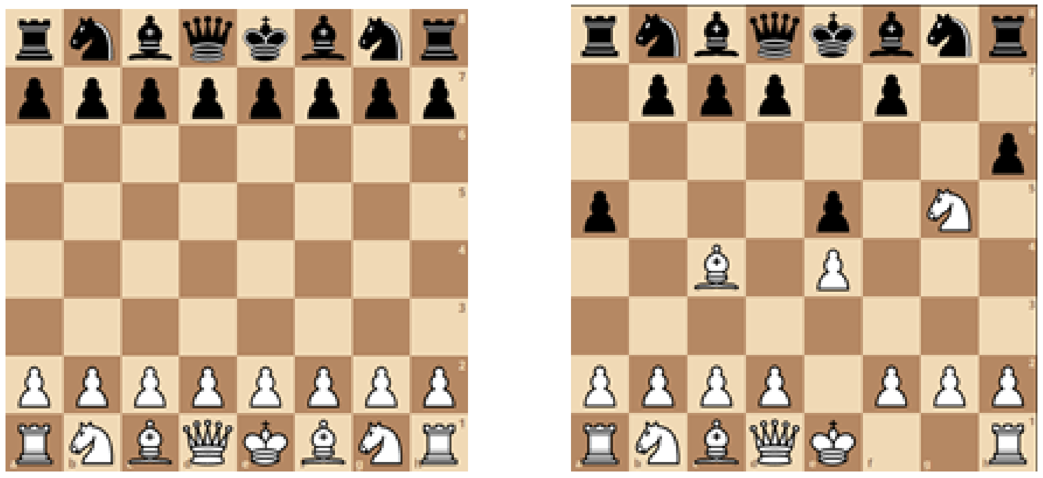


Figure 5: Opening position (left) and obvious capture position (right)

From a depth of 6, the following results were obtained for both of these positions shown in Figure 6.

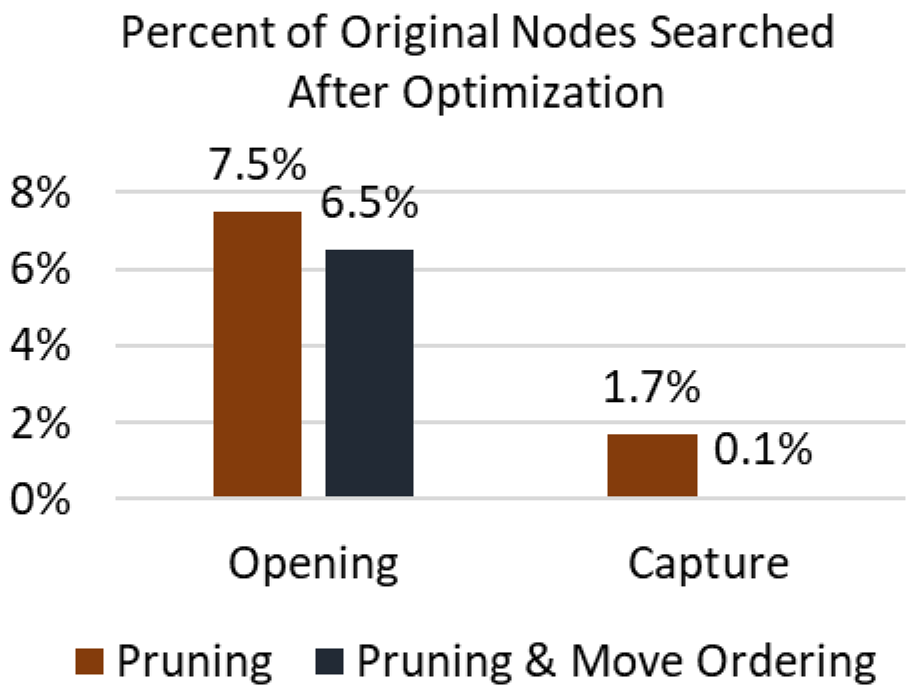


Figure 6: Percent of original nodes searched for different heuristics after optimization for both opening and capture positions from a depth of 6

In Figure 6, it can be observed that Negamax pruning had the largest effect on the nodes searched of the two heuristics used, reducing the nodes searched by 92.5% and 98.3% for the opening and obvious capture positions, respectively. Adding move ordering to the pruning heuristic further reduces the number of nodes searched, especially in the obvious capture position. This is most likely due to the fact that due to move ordering, the engine will evaluate the obvious capture move first. This yields a high evaluation early and makes future branches searched more likely to be pruned, thus reducing the number of nodes searched. Also, move ordering has slightly less of an effect for the opening position since there are not nearly as many captures to explore from a depth of six for the opening position compared to the obvious capture position.

While the findings for the two positions are informative, they do not provide a sense of the effect of move pruning and ordering over the course of an entire game. This is noteworthy, since the effect of these heuristics will have different impacts depending on the phase of the game. For example, during opening play, pieces tend to block each other more so than in the middlegame or endgame. This reduces the number of sibling nodes when calculating possible moves for a given position, which in turn reduces the effectiveness of move pruning since there are fewer sibling branches possible to prune.

To better understand the effect of move ordering and pruning over the course of an entire game, I chose to analyze the chess engine using the famous 1999 game played between Grandmasters Veselin Topalov and Garry Kasparov, noteworthy for its dynamic play and wild piece sacrifices. The percent of nodes searched utilizing the two aforementioned heuristics compared to the nodes searched without utilizing heuristics at a depth of 6 was recorded for every sixth half-move and is presented in the following Figure 7.

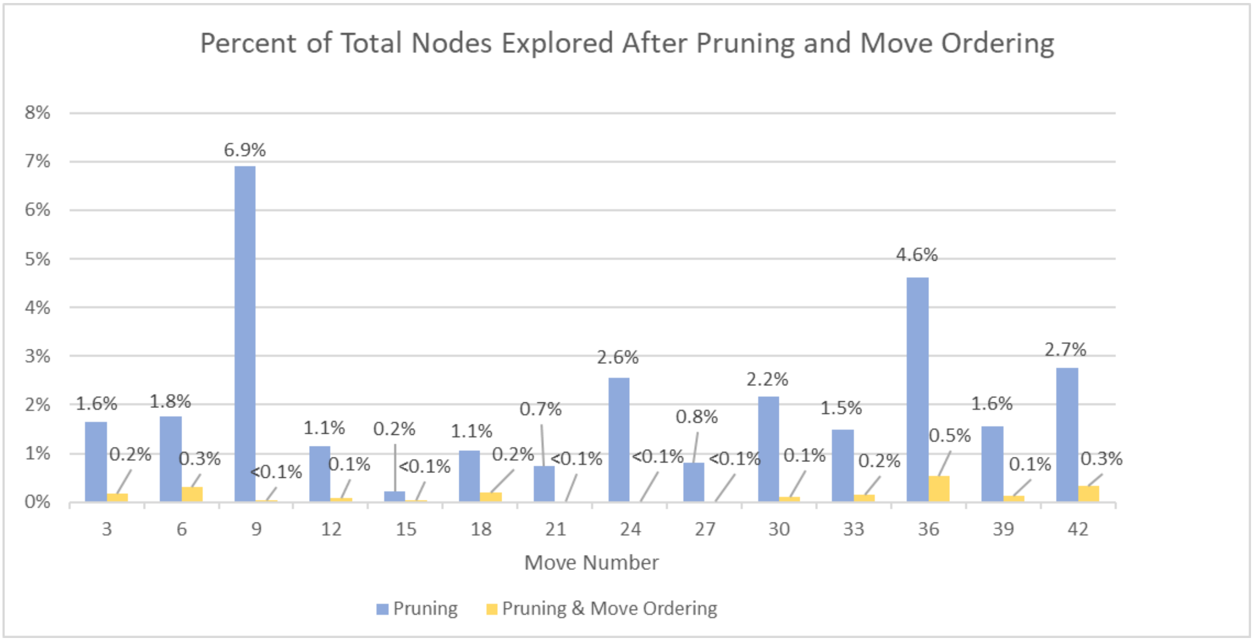


Figure 7: The percent of search space explored utilizing the pruning and pruning with move ordering at a depth of 6

As expected, the number of nodes searched is greatly reduced with pruning in all phases of the game, particularly in the middle game. Adding move ordering with pruning further significantly decreased the percent of the search space needed to search to find the best move for an in-game position.

CONCLUSION

Through the process of designing and implementing the chess engine, a number of questions were answered regarding design principles and heuristics. Namely, using a recursive approach as opposed to an iterative approach is critical to reducing source code size and complexity. An added benefit to this approach is code is much more readable and maintainable. Regarding heuristics, pruning is critical for reducing the number of nodes searched in all phases of the game (opening, middlegame, and endgame) and can be achieved with very little modification to existing code if using a recursive move generation approach. Additionally, move ordering is an effective heuristic that when combined with pruning, can serve to further decrease the number of nodes needed to search by the chess engine to find the best move in a given position as demonstrated in Figure 7. Similarly, implementing move ordering requires little additional modification to source code when using a recursive approach. To summarize, using recursive move generation along with these two heuristics is key to designing an effective chess engine while also maintaining an efficient source code size.

REFERENCES

1. “R/Chess - Chess Game on Delta Airlines.” Reddit, https://www.reddit.com/r/chess/comments/p1d1jf/chess_game_on_delta_airlines/.

2. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G. (2006). Code Size Reduction by Compiler Tuning. In: Vassiliadis, S., Wong, S., Hämäläinen, T.D. (eds) Embedded Computer Systems: Architectures, Modeling, and Simulation. SAMOS 2006. Lecture Notes in Computer Science, vol 4017. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11796435_20

3. “Main Page.” *Chessprogramming Wiki*, https://www.chessprogramming.org/Main_Page.

4. “Magic Bitboards.” *Magic Bitboards - Chessprogramming Wiki*, https://www.chessprogramming.org/Magic_Bitboards.

5. “Shannon Number.” *Wikipedia*, Wikimedia Foundation, 4 Apr. 2022, https://en.wikipedia.org/wiki/Shannon_number.

6. “Negamax.” *Negamax - Chessprogramming Wiki*, https://www.chessprogramming.org/Negamax.