Chess Move Generation Using Bitboards

by

Sophie Elyse Columbia

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science

and The Honors College

in partial fulfillment of the requirements for the degree of

Bachelor of Science

May 2023

APPROVED BY:

Raghuveer Mohan, Ph.D., Thesis Director

Vicky Klima, Ph.D., Second Reader

Raghuveer Mohan, Ph.D., Departmental Honors Director

Mark Hills, Ph.D., Departmental Honors Director

Jefford Vahlbusch, Ph.D., Dean, The Honors College

Copyright © Sophie Elyse Columbia 2023
All Rights Reserved

ABSTRACT

Chess Move Generation Using Bitboards.

(May 2023)

Sophie Elyse Columbia, Appalachian State University

Appalachian State University

Thesis Chairperson: Raghuveer Mohan, Ph.D.

Chess has been a subject of study in artificial intelligence for many decades, and chess agent programs have contributed to many theoretical ideas in the field. It was thought in the early 1960s that no computer would be able to play as well as a human. This idea was squashed when the computer program Deep Blue beat the world chess champion, Gary Kasparov, in 1997. Today, chess engines have surpassed all human abilities.

Many readily available resources for those interested in building their own chess engines are difficult to follow. They tend to either present information in a maze-like fashion or do not provide intuitive explanations for complex functions. This thesis aims to provide an accessible guide to move generation – the first step in building a chess engine that is the basis for all search functions. To make move generation as efficient as possible, we use bitboards as our selected board representation. We capture the state of the board using multiple sequences of 64 bits, where each bit is mapped to a square on the chess board. Remarkably, bitwise operations on these bitboards can generate all possible moves for a given position quickly.

This project also details a rigorous testing approach using crowdsourcing. To encourage interaction from chess experts, we create a user-friendly GUI that allows users to input moves

in a game-like fashion. By explaining our approach to move generation and accuracy testing, we hope to provide a useful resource for those wishing to write their own chess engine.

Contents

1 Introduction **1**

2 Bitboards **5**
2.1 Board Representations . . . 5
2.2 Bitboards In Depth . . . 9

3 Move Generation **19**
3.1 List of Possible Moves . . . 23
3.2 Pawns . . . 25
3.3 Knights . . . 32
3.4 Sliding Pieces . . . 36
3.5 King . . . 41
3.6 Castling . . . 47
3.7 *En Passant* . . . 50
3.8 Check . . . 51
3.9 Pinned Pieces . . . 53

4 Testing **59**
4.1 Perft Testing . . . 59
4.2 Crowdsourcing . . . 59
4.3 Forsyth-Edwards Notation . . . 60
4.4 GUI . . . 61

5 Conclusion **63**
5.1 Future Work . . . 64

Bibliography **66**

Appendices **67**

A Bitboard Masks **68**

B Test Cases **69**

List of Figures

2.1 A piece-list board representation. 6
2.2 An example piece-set representation of a single square. Higher 16 bits represent black pieces, and lower 16 bits represent white pieces. A bit is set to 1 if the corresponding piece attacks the square. 7
2.3 The piece-set representation for the highlighted square. 8
2.4 An example mailbox board representation of two pieces. 9
2.5 Bit values in their associated positions are illustrated. The top left contains the most significant bit. 10
2.6 The two black rooks position. 10
2.7 The bit representation of the two black rooks position. 11
2.8 Hexadecimal bitboard representation of black rooks. 12
2.9 Hexadecimal bitboard representation of white pawns in their starting positions. . . 13
2.10 Hexadecimal bitboard representation of white rooks in their starting positions. . 13
2.11 Hexadecimal bitboard representation of white knights in their starting positions. 14
2.12 Hexadecimal bitboard representation of white bishops in their starting positions. 14
2.13 Hexadecimal bitboard representation of the white queen in its starting position. . 15
2.14 Hexadecimal bitboard representation of the white king in its starting position. . 15
2.15 Hexadecimal bitboard representation of the starting position for white pieces. This results from performing a bitwise OR operation of the above bitboards. . . 16
2.16 Bitboard representation of a possible position with fewer pieces on the board. This is the resulting bitboard from performing a bitwise OR on all piece's bitboards. 17

3.1 Bitboard mask of a diagonal ray. 20
3.2 Bitboard mask of an anti-diagonal ray. 21
3.3 Bitboard mask of rank 1. 21
3.4 Bitboard mask of rank 8. 22
3.5 Bitboard mask of file A. 22
3.6 Bitboard mask of file H. 23
3.7 A position for which the possible moves for white are provided. 24
3.8 Hexadecimal bitboard representation of white pawns in their initial positions. . . 26
3.9 Hexadecimal bitboard representation of white pawns after being shifted left by 8. 27
3.10 Hexadecimal bitboard representation of black pawns after being shifted right by 8. 28
3.11 Hexadecimal bitboard representation of white pawns after being shifted left by 16. 29
3.12 Hexadecimal bitboard representation of white pawns after being shifted left by nine. 30
3.13 Hexadecimal bitboard representation of white pawns after being shifted left by seven. 31

3.14 The bitboard mask used to generate knight moves. All highlighted green squares are 1's, and the knight is a reference for where the focal point of the mask is located. . . . 33
3.15 The *knightMask* wrapped around onto files A and B. . . . 35
3.16 The *knightMask* wrapped around onto files G and H. . . . 35
3.17 The bitboard of desired moves, indicated in green, to be generated for a white rook moving within its rank. . . . 37
3.18 The bitboard mask used to generate king moves. All highlighted green squares are 1's, and the king is a reference for where the focal point of the mask is located. 42
3.19 The bitboard of unsafe squares, colored in red, for a white king to move to. . . . 43
3.20 Safe squares generated when the king remains on the board. Red squares are unsafe while green squares are falsely computed as safe. . . . 44
3.21 The necessary position of the white kingside rook for castling. . . . 48
3.22 The intermediate squares of a white castling kingside. . . . 48
3.23 The overlap of rook and king sliding moves calculates the desired line of sight. . . . 53
3.24 A pinned rook that can only move within its rank, denoted by green squares. . . 54
3.25 The white rook is identified as a pinned piece. . . . 55
3.26 Position which demonstrates the need to check the locations of pieces and not only their line of attack. . . . 56
3.27 Two pinned pawns. Only one can move, and that square is highlighted in green. . . . 58

4.1 GUI for testing via crowdsourcing. . . . 62

B.1 rnbqkbnr/ppppp2p/5p2/6pQ/4P3/3P4/PPP2PPP/RNB1KBNR b KQkq - . . . 69
B.2 4k3/4P3/4K3/8/8/8/8/8 b - - . . . 70
B.3 2Q2Q2/3k2p1/2q2B2/4P3/ B1NPn3/7p/1P6/4K3 b - - . . . 70
B.4 r1q2rk1/pp1bNppp/n1p5/8/ 3PP3/2N3P1/PPP2PBP/R1BQ1RK1 b - - . . . 71
B.5 1krq3r/pp3Npp/1b3n2/8/8/ 2N3B1/PPP2PPP/R4RK1 b - - . . . 71
B.6 4k3/2p1b1p1/1p5p/1q3p2/P5N1/ 4QB2/P1PK3r/8 w - - . . . 72
B.7 8/3KPk2/8/p7/8/8/8/8 w - - . . . 72
B.8 r1bqk2r/pppp1ppp/2n2n2/ 2b1p3/2B1P3/2N2N2/PPPP1PPP/R1BQK2R w KQkq - . 73
B.9 r2qk2r/1pp2ppp/p1npbn2/ 2b1p1B1/2B1P3/2NP1N2/PPPQ1PPP/R3K2R w KQkq - . 73
B.10 r1bqkb1r/ppp2ppp/2n1pn2/ 3p2B1/3P4/2N5/PPPQPPPP/R3KBNR w KQkq - . . . 74
B.11 rnbqkbnr/ppp2ppp/4p3/3pP3/3P4/8/PPP2PPP/RNBQKBNR w KQkq d6 . . . 74
B.12 5k2/3r2pp/5p2/1b6/3NP3/ 1B4P1/7P/R2K4 w - - . . . 75
B.13 2r5/1k3r2/R1p5/1P6/6P1/ 2N2Q2/5P1P/5RK1 b - - . . . 75
B.14 rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - . . . 76
B.15 r1b2rk1/1pq1bppp/p1nppn2/8/ 3NP3/1BN1B3/PPP1QPPP/2KR3R w - - . . . 76
B.16 rn2k1nr/ppp2ppp/3b4/3N4/ 3PPbq/5N2/PPP1K1PP/R1BQ1B1R b - - . . . 77
B.17 r1b1k2r/pp2n1pp/2n1pp2/ q1ppP3/P2P4/2P2N2/2PQ1PPP/R1B1KB1R w - - . . . 77
B.18 r1br2k1/pp1n1pp1/2q1n2p/7Q/ 2P5/1N1BR3/PB3PPP/R5K1 b - - . . . 78
B.19 r2q1rk1/ppp2ppp/2np1n2/ 2b1p3/2P3b1/2NP1NP1/PP2PPBP/R1BQ1RK1 w - - . . . 78
B.20 rn1q1rk1/pbp1bppp/1p2p3/ 3pN3/2PPn3/2N3P1/PP1BPPBP/R2Q1RK1 b - - . . . 79
B.21 1k5r/p5pp/3b4/8/8/6B1/PP6/5RK1 b - - . . . 79
B.22 1k5r/p5pp/3q4/8/8/6B1/PP3PP1/5RK1 b - - . . . 80
B.23 1k5r/p5pp/8/8/8/7B/PP3PP1/R1q1R1K1 w - - . . . 80
B.24 rnbqkbnr/ppp2ppp/8/3p4/3Pp3/4P3/PPP2PPP/RNBQKBNR b KQkq d3 . . . 81
B.25 r1q2rk1/p1p1bppp/bpnp1n2/ 6B1/4P3/1BN2N2/PPP2PPP/R2QK2R w KQ - . . . 81

B.26 r2qk2r/ppp2ppp/8/8/Q7/8/ PP3PPP/R4RK1 b - - 82

Chapter 1

Introduction

Chess has been a subject of study in artificial intelligence for many decades, and chess agent programs have contributed to many theoretical ideas in the field. It was thought in the early 1960s that no computer would be able to play as well as a human. This idea was squashed when the computer program *Deep Blue* famously beat the world chess champion, Gary Kasparov, in 1997 [6]. Since then, chess engines have become increasingly more sophisticated.

Numerous widely available chess engines demonstrate the progress made in the field over the past two decades. Some of the most popular chess engines include *Stockfish*, *Komodo*, and *Houdini* [11]. In particular, Stockfish is generally regarded as the strongest chess engine, with the highest rating amongst its competitors and ability to generate 60 million moves in one second [11]. There is not a well-documented instance of a human defeating Stockfish. However, recent developments in artificial intelligence have begun changing the landscape of competitive chess engines. In 2017, Google claimed that their program, *Alpha Zero*, exceeded Stockfish's abilities after a mere four hours of training [12]. Evidently, chess engines have surpassed all human abilities.

Chess engines also improve humans' understanding of chess by providing analytical commentary and chess theory knowledge. Chess engines provide commentary on games, pinpointing critical mistakes and offering optimal alternatives. For example, on YouTube, there is a popular video by Chess.com where Stockfish offers comedic commentary on the famed Opera Game from the 19th century [10]. Also on Chess.com, there are many functionalities which aim to improve human play [2]. Their chess engine provides feedback in real time and can identify

popular move combinations for player edification. Ultimately, human chess knowledge improves in parallel with chess engines.

Although chess engines have a long history of research, there are very few books written on this subject. These books are typically narrow in scope and necessitate the programmer buying numerous books to cover the entire process of building a chess engine, which is cost prohibitive. Free information is often scattered throughout various online platforms like Chess Programming [1]. These platforms can appear maze-like due to a lack of curation, forcing readers to painfully scour for convoluted information that lacks insights for gaining a deep understanding of chess engine mechanics. They tend to assume that the reader possesses knowledge of chess engine terminology and background, posing difficulties for beginners. Moreover, many platforms lack test cases to use against newly written chess engines.

Chess and artificial intelligence journals that detail building chess engines may hide behind paywalls. These results also tend to be scattered and not curated, or they may simply be outdated. For example, *Chess Compendium* by David Levy offers a good set of chess engine development [14], but its 1988 publication renders it too outdated for modern use. New books tend to have their own shortcomings. For instance, *How to Write a Bitboard Chess Engine* by FM Bill Jordan, published in 2020, uses a confusing representation of the board where it is vertically reflected [13]. Jordan also uses inefficient move generation functions which painstakingly flags each possible square that a piece could move to, regardless of king safety or locations of other pieces.

There is little consensus in these resources about best methods, confusing readers further. How should we represent a board, and why is that the best decision? Should we account for king safety during initial move generation? How do seemingly arbitrary bit manipulations actually give correct results? Through this research, we attempt to consolidate clear instructions and logical explanations to novice programmers wishing to write a chess engine. Note that we do not provide instructions to build the best state-of-the-art chess engine that could challenge AlphaZero or Stockfish. We intend to provide tools for building a rudimentary chess engine, highlighting some of the techniques and computer science principles used. Programmers interested in building a chess engine will find our thesis a good place to start; we provide resources for continuing study.

Chess engines utilize many different areas of computer science, such as user interface design, software design and testing, data structures and algorithms for efficient representation of the chessboard's state, generating a set of valid moves and performing recursive search strategies for finding moves of best play, database connectivity for accessing chess opening theories, and artificial intelligence techniques for computer play. The expansive nature of chess engines makes them an excellent subject for a project-based course, teaching undergraduate students fundamental computer science concepts. Every week, students could focus on building a particular component of a chess engine following intuitive step-by-step instructions.

The first step in building a chess engine is *move generation* – obtaining a list of valid moves. This list is given as input to a heuristic algorithm for deciding optimal moves for computer play. It is arguably the most important step of building a chess engine, as correct, efficiently generated moves are the foundation for a successful engine. Move generation, however, is a complex task.

The complexities of move generation arise from the convoluted rules of chess, including king safety, castling, *en passant*, and promotion. Many replicable chess engines found online generate a list of all pseudo-legal moves, which are moves that ignore king safety and checks. In these engines, making an illegal move is resolved by undoing it and repeatedly picking a move from the list of pseudo-legal moves until one is legal. These engines must test for legality after selecting a move, which further adds to the running time of the move generation algorithm. A function that can quickly return a list of legal moves will be suitable for more complex heuristic search algorithms. Therefore, this thesis focuses on legal move generation, and makes the following contributions.

- We provide an overview of different board representations, focusing on bitboards, which offer a very efficient way to not only represent the different pieces on the board, but also to compute a list of valid moves quickly.
- We present a conceptual understanding of bitboards, which many other resources fail to accomplish. We focus on providing intuitive explanations and visual representations of bitboards and their associated chess board positions.

- We give insight into complex bit manipulation operations for move generation. As move generation entails complex sequences of bit shifts, bit masking, and math operations, we aim to provide readers with detailed comprehension. We utilize images of chess boards to illustrate how bitboards efficiently find possible moves.
- We describe move generation techniques that account for king safety, which many other readily available resources ignore.
- We outline the process for designing an interactive user interface (UI).
- We introduce a rigorous testing approach via crowd sourcing. Generating comprehensive test cases can be accomplished by using expert knowledge and interacting with our UI.

This thesis assumes that the reader is familiar with the rules and objectives of chess, as well as chess algebraic notation used in recording chess games. Readers unfamiliar with these can refer to the following resource [3].

The rest of the thesis is organized as follows. In Chapter 2, we introduce the concept of a bitboard, which allows us to represent bit pieces using 64 bit integers. In Chapter 3, we show how we can use various bitboards to generate chess moves. In Chapter 4, we discuss the elaborate method of testing our move generation algorithm. We also discuss the development of a user interface to capture legal moves from experts to test our chess engine move generation. Finally, in Chapter 5, we summarize our work and comment on future work. If of interest, Appendix A contains the integer or hexadecimal values of bitboard masks used, and Appendix B provides a list of crowdsourced test cases.

Chapter 2

Bitboards

One of the first steps of building a chess engine is deciding how to store the state of a chess board for move generation and search algorithms. In this Chapter, we briefly discuss few of the common board representations, and delve into one particular representation, *the bitboard*, in more detail.

2.1 Board Representations

The most popular choices for board representations are piece-lists, piece-sets, mailbox and bitboards. Each of these representations have their own unique qualities, strengths, and weaknesses.

Piece-Lists

This is one of the simplest ways to represent pieces. For each type of piece, maintain a list, either as a linked list or as arrays, of all locations on the board for that piece. Figure 2.1 shows piece-lists for the Black Bishop and the White Rook. This approach proves inefficient for move generation, as piece-lists are organized by piece type rather than piece location, possibly requiring a scan through all piece-lists to see whether a square is occupied in move generation. Although piece-lists are simple to implement, they may not be an ideal board representation since they require inefficient location searches.

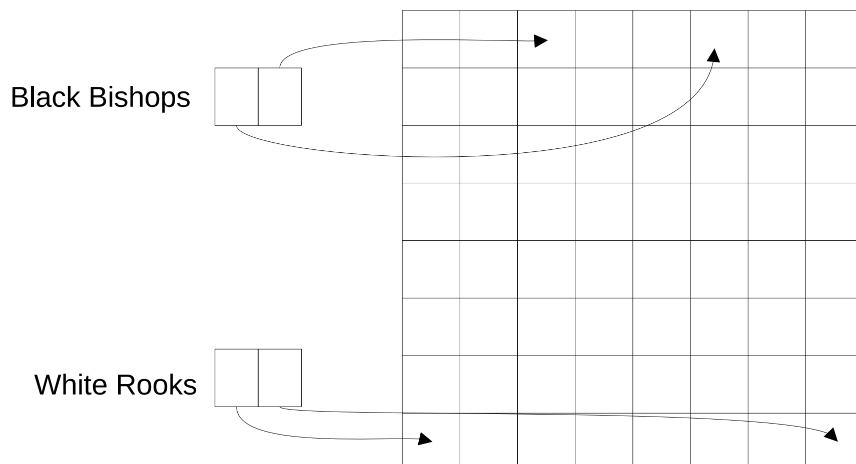


Figure 2.1: A piece-list board representation.

Piece-Sets

Piece-sets are a complex option for board representation. Each square on the board has a 32 bit number, a *bitlist*, where each half represents the white or black pieces. Consider, for instance, that the higher 16 bits represent black pieces and the lower 16 bits represent white pieces. The programmer determines which piece corresponds to a single bit. Within these 16 bits, each of the bits represents a single piece (see Figure 2.2). For example, the bit 0 may represent the white king, bit 1 may represent the white queen, and so on. Figure 2.3 shows an example board position and the bit list for a single highlighted square. A shorthand notation, where each piece is represented by no more than two letters, is used. Black pieces are written in lowercase letters while white pieces are written in uppercase letters. For example, 'K' refers to the white king, 'k' refers to the black king, and 'KB' refers to the white kingside bishop. A bit is set to 1 if the associated piece is attacking that square. This means that we use $64 * 32 = 2048$ bits to represent a board's state. For each move, all bits must be updated if necessary. The update operation is non-intuitive and difficult to debug. Moreover, pawn pushes and castling rights aren't handled by piece-sets. Andrew Tridgell, a professor of Computer Science in Australia, however argues that piece-sets are actually more efficient than bitboards [15]. Though it seems like piece-sets are a powerful board representation selection for advanced programmers, we will

opt for a board representation that can be easily visualized and is still very efficient.

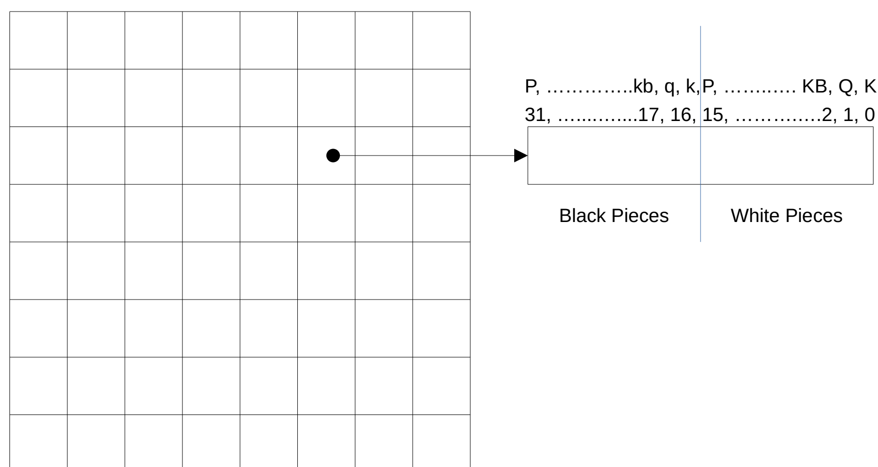


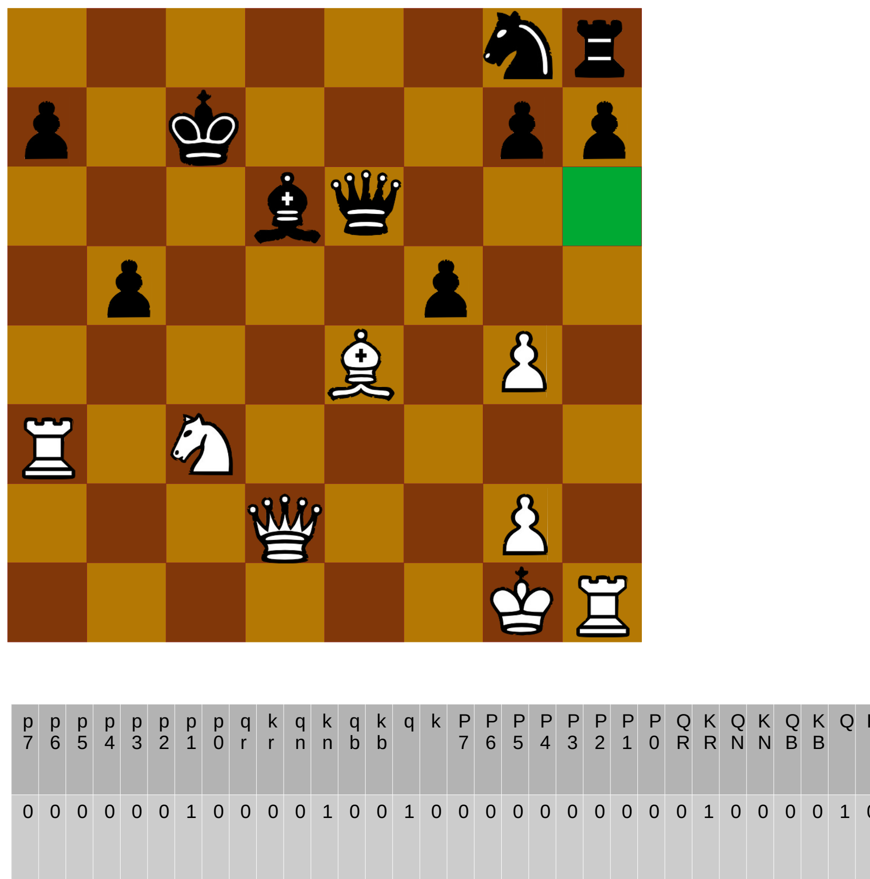
Figure 2.2: An example piece-set representation of a single square. Higher 16 bits represent black pieces, and lower 16 bits represent white pieces. A bit is set to 1 if the corresponding piece attacks the square.

Mailbox

Mailbox is an intuitive board representation that most mimics a physical chess board. Each square on the chess board (which could be represented as an 8x8 two-dimensional array) points to a separate address in memory. This address either contains the piece that occupies that particular square or is empty, indicating that no piece occupies that square (see Figure 2.4). Mailbox is simple to implement and is friendly to beginners. However, during move generation, we would have to use numerous loops to reach all squares, and use different conditional checks depending upon which type of piece, if any, is found at a particular square. Mailbox is appealing due to its clarity, but it does not support incredibly efficient move generation.

Bitboards

Bitboards are an innovative approach to board representation. Each combination of piece type and color is represented by an unsigned 64-bit integer. For example, white pawns and black



p7	p6	p5	p4	p3	p2	p1	p0	qr	kr	qn	kn	qb	kb	q	k	P7	P6	P5	P4	P3	P2	P1	P0	QR	KR	QN	KN	QB	KB	Q	K
0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0

Figure 2.3: The piece-set representation for the highlighted square.

pawns would both have their own bitboard. As a chess board contains 8x8 squares, each bit represents a square. A 0 indicates that its respective square is empty, while a 1 indicates that square is occupied by a piece of that type. Bitboards provide both piece information and location in a single look-up. In addition, we capitalize upon the speed of bit manipulations and simple mathematical operations on integers to alter positions, which will be expanded upon in Chapter 3.

Bitboards have unmatched strength in their quick information accesses, position manipulations, and ease of visualization. For these reasons, we use bitboards as our underlying board representation.

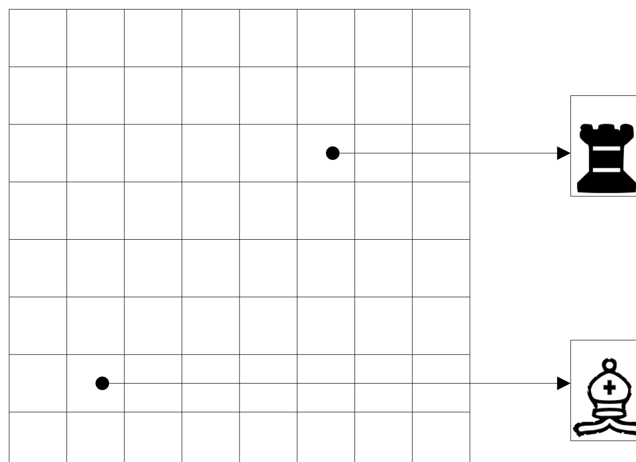


Figure 2.4: An example mailbox board representation of two pieces.

2.2 Bitboards In Depth

Remarkably, we can use multiple 64-bit words to represent the entire state of a chess board. To use bitboards as an underlying board representation, each piece type is represented by a single 64-bit word. Each bit represents a square, hence our need for 64 bits as chess boards are 8x8 squares. For our implementation, the most significant bit is located in the top left corner of the chess board (which is $A8$ in algebraic notation). Then, the least significant bit is in the bottom right of the chessboard ($H1$). See Figure 2.5 for a visualization. Many other chess engine implementations found online do not have the most significant bit in the top left corner, which is not intuitive. These sources tend to present visual examples as if the most significant bit is in the top left corner, expecting readers to understand rearranging the bits on the board for their own implementation.

A bit can be either a 0 or a 1. A 1 denotes that a piece of that type is located in its represented square, while a 0 indicates the square does not contain a piece of that type. Though a bitboard is a sequence of 64 bits, it is easier to visualize the number as you would a chess board. We will split the word into eight rows, where each row contains 8 bits. Consider the following given position in Figure 2.6, with only two black rooks on the chess board, and its

Rank (numbers)

8	2^{63}	2^{62}	2^{61}	2^{60}	2^{59}	2^{58}	2^{57}	2^{56}
7	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}	2^{48}
6	2^{47}	2^{46}	2^{45}	2^{44}	2^{43}	2^{42}	2^{41}	2^{40}
5	2^{39}	2^{38}	2^{37}	2^{36}	2^{35}	2^{34}	2^{33}	2^{32}
4	2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}
3	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}
2	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
1	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
	A	B	C	D	E	F	G	H

File (letters)

Figure 2.5: Bit values in their associated positions are illustrated. The top left contains the most significant bit.

subsequent bitboard representation in Figure 2.7.

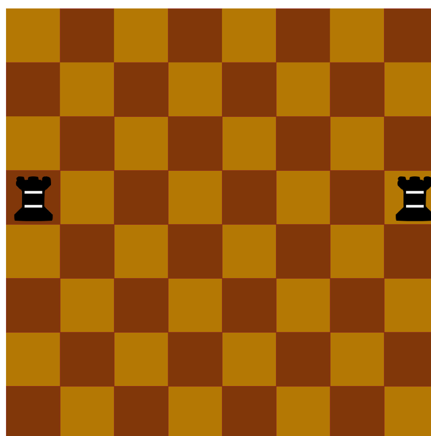


Figure 2.6: The two black rooks position.

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	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	0	0	0	0	0

Figure 2.7: The bit representation of the two black rooks position.

Again, since the top left square is our most significant bit, in binary, this bitboard is `0000000000000000000000001000000100000000000000000000000000000000`. We can further simplify this number's representation by converting it to hexadecimal, where we have 16 digits, 2 on each row, representing the board. In hexadecimal, this number is `0x0000008100000000`. The hexadecimal representation is succinct and easier to understand. From this point on, we will represent a bitboard as a 16 digit hexadecimal number, with 2 digits representing a single row. See Figure 2.8 below for how we will represent our two black rooks position in hexadecimal.

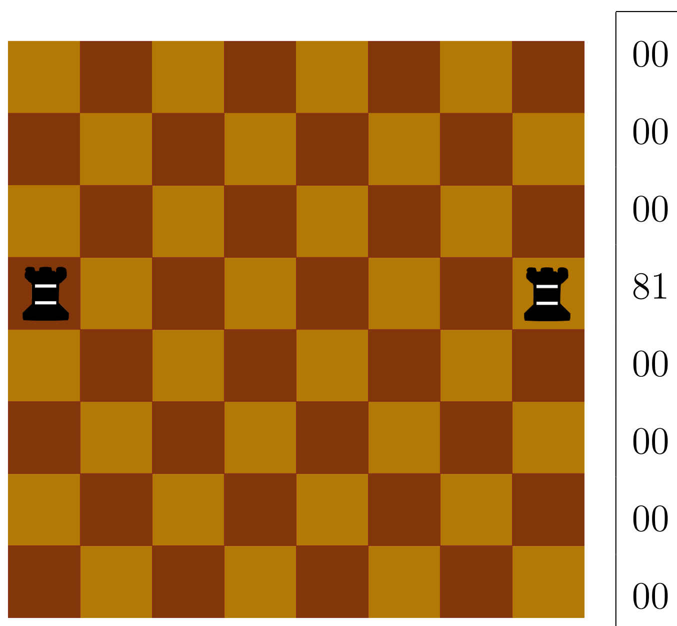


Figure 2.8: Hexadecimal bitboard representation of black rooks.

For further understanding, Figures 2.9 through 2.14 show the bitboards for all pieces in white's initial position. It is easy to see that if we take the bitwise OR operation of all of these bitboards, we get the bitboard that corresponds to the starting positions of the white pieces (see Figure 2.15). In general, the bitwise OR operation of the six bitboards of the white pieces gives the positions of all the white pieces on the board.

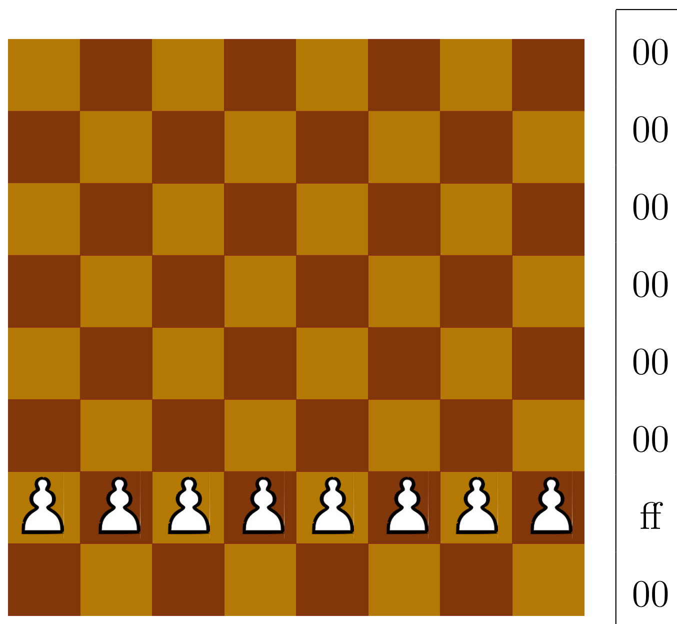


Figure 2.9: Hexadecimal bitboard representation of white pawns in their starting positions.

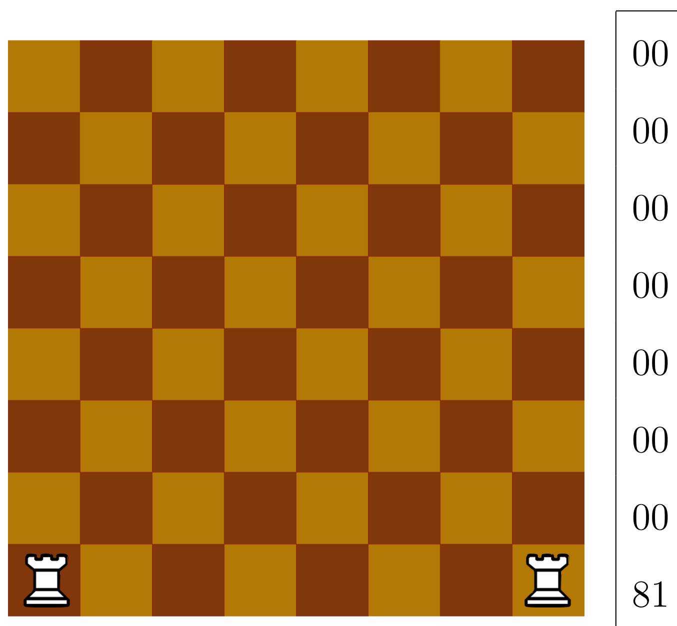


Figure 2.10: Hexadecimal bitboard representation of white rooks in their starting positions.

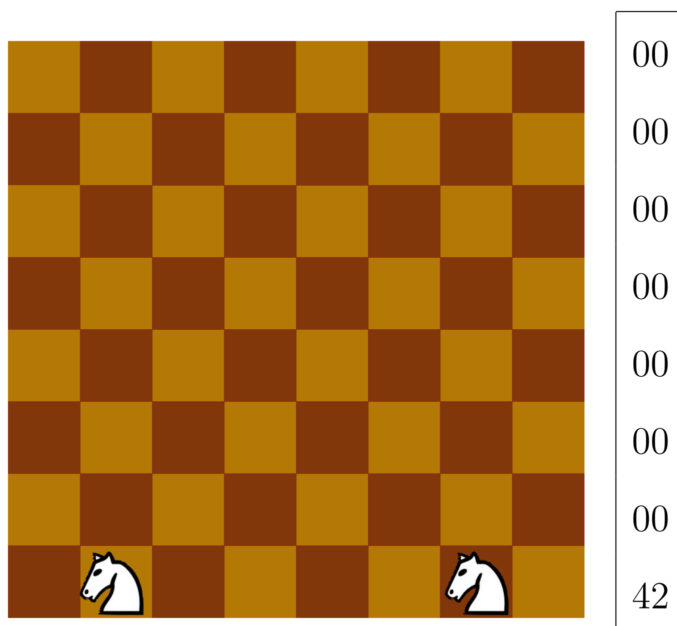


Figure 2.11: Hexadecimal bitboard representation of white knights in their starting positions.

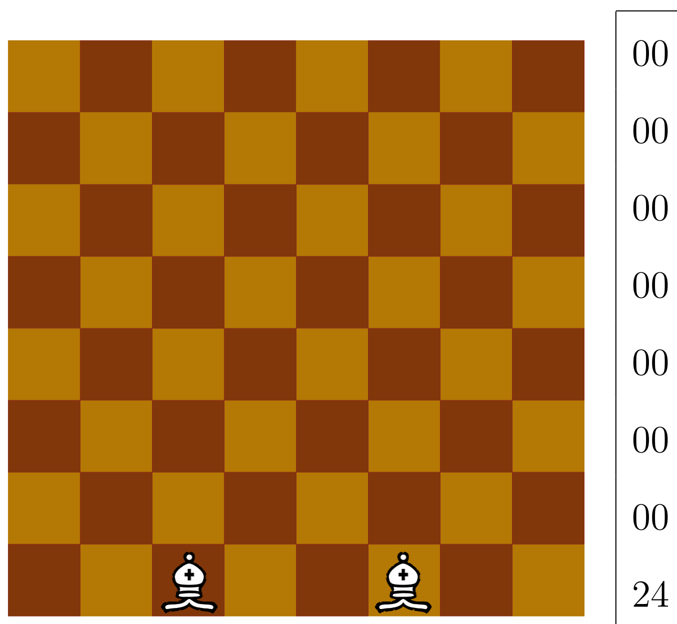


Figure 2.12: Hexadecimal bitboard representation of white bishops in their starting positions.

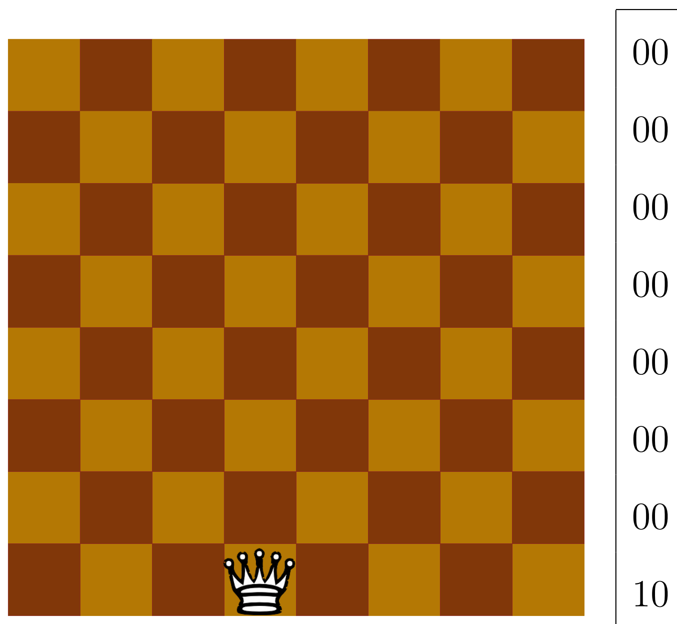


Figure 2.13: Hexadecimal bitboard representation of the white queen in its starting position.

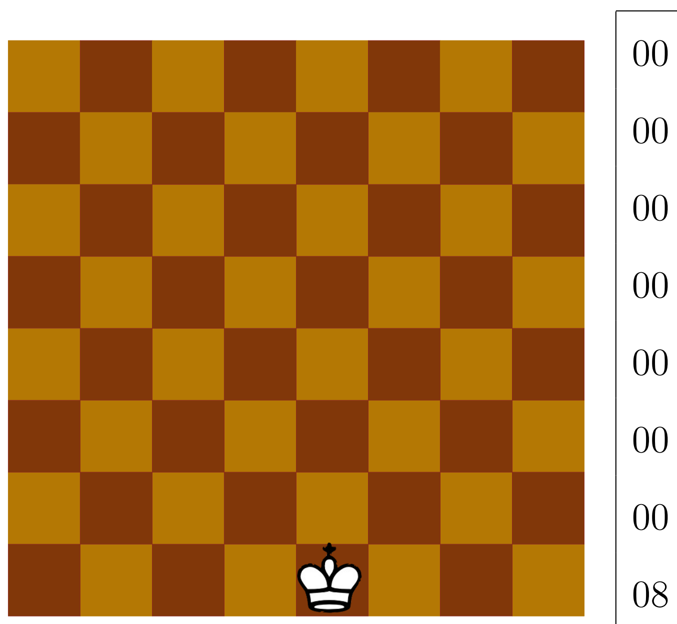


Figure 2.14: Hexadecimal bitboard representation of the white king in its starting position.

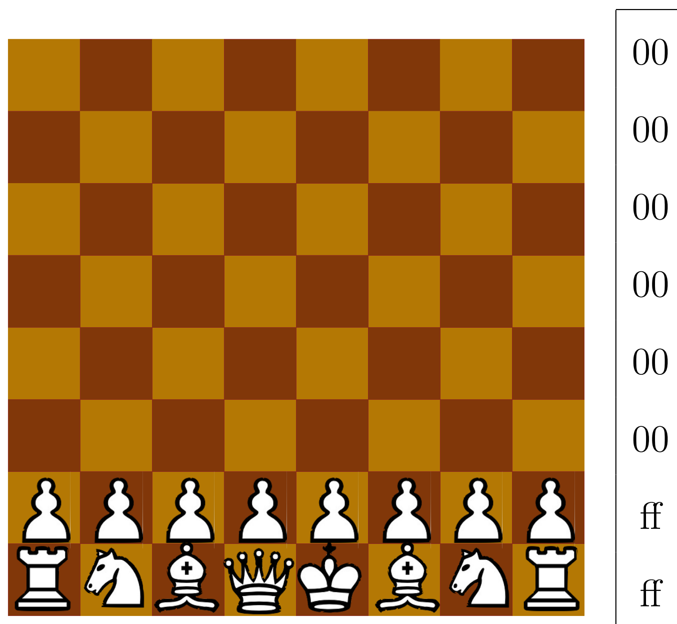


Figure 2.15: Hexadecimal bitboard representation of the starting position for white pieces. This results from performing a bitwise OR operation of the above bitboards.

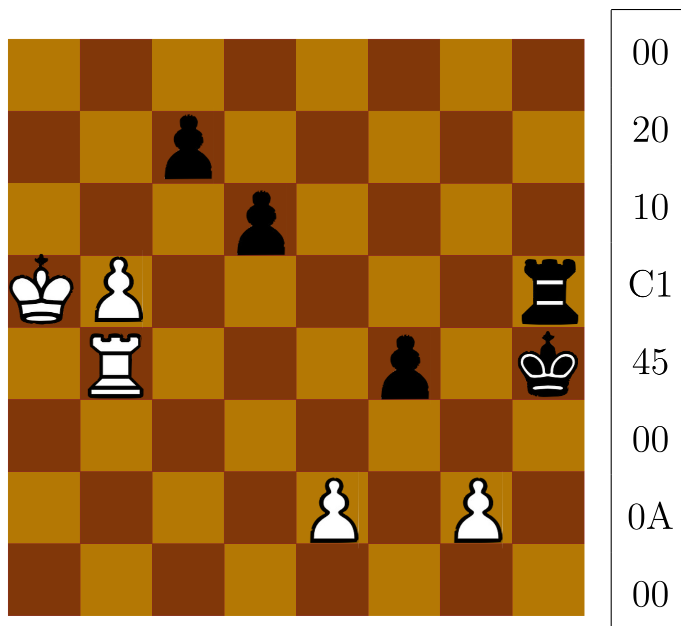


Figure 2.16: Bitboard representation of a possible position with fewer pieces on the board. This is the resulting bitboard from performing a bitwise OR on all piece's bitboards.

The following table contains the bitboards for each piece type that is present on the board shown in Figure 2.16. Again, we bitwise OR these values to calculate the bitboard representing the entire board.

Piece Type	Bitboard
Black rook	0x0000000100000000
Black king	0x0000000001000000
Black pawn	0x0020100004000000
White rook	0x0000000040000000
White king	0x0000008000000000
White pawn	0x0000000000000A00

It is cumbersome to repeatedly write both the color and piece type in code. As alluded to in the introduction of piece-sets, we can use a shorthand notation to refer to each chess piece. Conventionally, we use the first letter of the piece type as its abbreviation. The notable exception is for knights, where we use 'N' instead of 'K' to distinguish it from the king. To

specify the color of the piece, we write the letter in uppercase if it's a white piece and in lowercase if it's a black piece. The following table contains each piece's associated abbreviation.

Piece Type	Shorthand
White rook	R
White knight	N
White bishop	B
White queen	Q
White king	K
White pawn	P
Black rook	r
Black knight	n
Black bishop	b
Black queen	q
Black king	k
Black pawn	p

Now that we can represent our chess board using bitboards and succinctly refer to a piece type, we can proceed to move generation, which we discuss in the next chapter.

Chapter 3

Move Generation

Move generation using bitboards involves lots of bit manipulations, so we will use visualizations for better understanding of these operations on piece positions. As explained in Chapter 2, we will have a bitboard for each unique piece type and color combination. We also need a few other masks to use in these move generation functions.

The bitboards *blackPieces* and *whitePieces* contain a 1 if there is a piece of that respective color in a certain square. We separately bitwise OR all of the bitboards of the respective pieces of each color to calculate these two bitboards. For example;

```
whitePieces = R | N | B | Q | K | P
```

The bitboard *occupied* has a 1 for each square that is currently occupied by a piece. To find *occupied*, we bitwise OR the bitboards of all pieces. We also require *empty*, a bitboard where a 1 denotes an empty square. So;

```
occupied = whitePieces | blackPieces
empty = ~occupied
```

We can also easily write masks for each file, rank, diagonal, and anti-diagonal. The longest diagonal line spans from the bottom left to the top right corner of the board. All other diagonal lines on the board are parallel to this line. Opposite from diagonal rays, the longest anti-diagonal line spans from the top left to the bottom right of the board, and all other anti-diagonal rays on the board are parallel to the longest anti-diagonal. For example, a diagonal

mask is illustrated in Figure 3.1 and an anti-diagonal mask is illustrated in Figure 3.2. Then, rank 1 is `0xff` and rank 8 is `0xff00000000000000` (see Figures 3.3 and 3.4). Similarly, file A is `0x8080808080808080` and file H is `0x0101010101010101` (see Figures 3.5 and 3.6). These masks will be used to filter out invalid moves. Appendix A contains information about all masks as well as bitboards used.

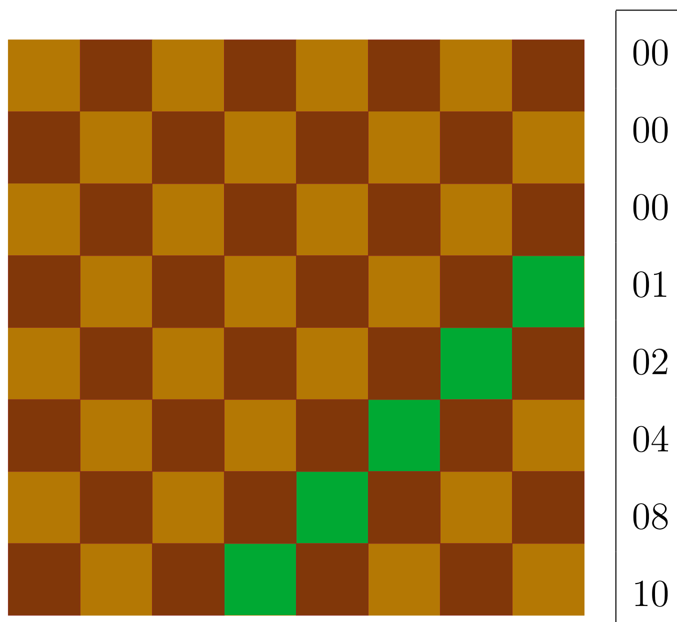


Figure 3.1: Bitboard mask of a diagonal ray.

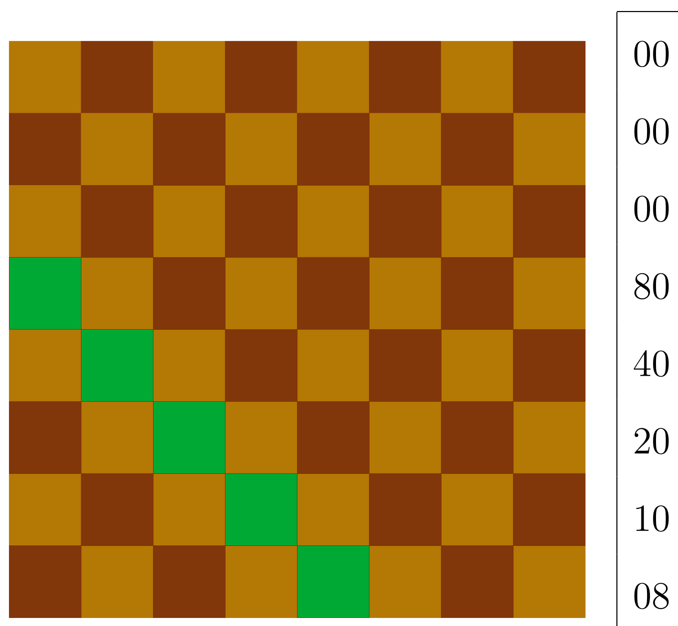


Figure 3.2: Bitboard mask of an anti-diagonal ray.

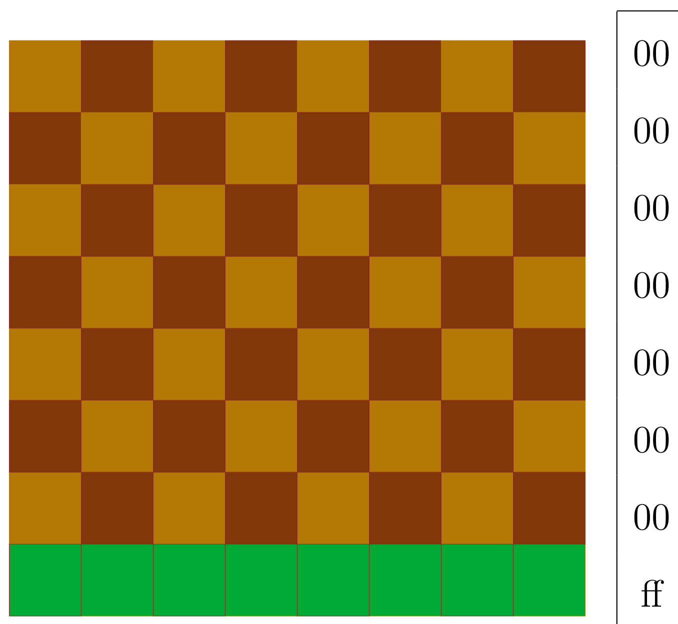


Figure 3.3: Bitboard mask of rank 1.

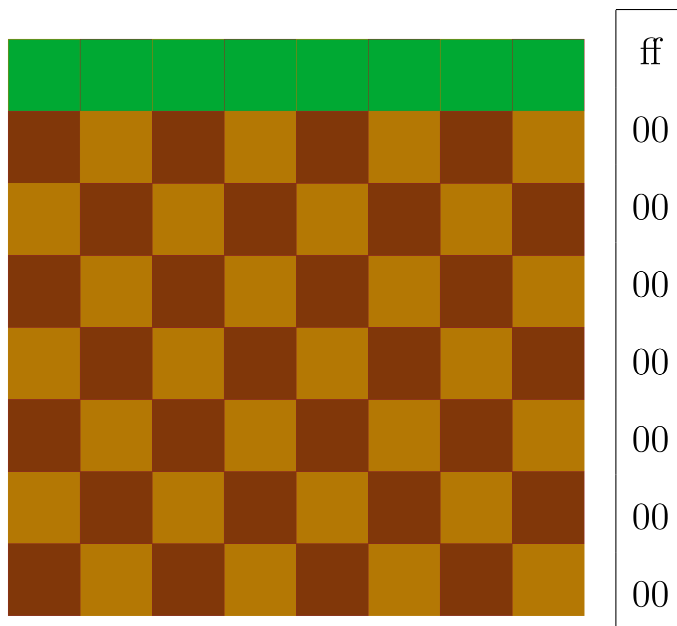


Figure 3.4: Bitboard mask of rank 8.

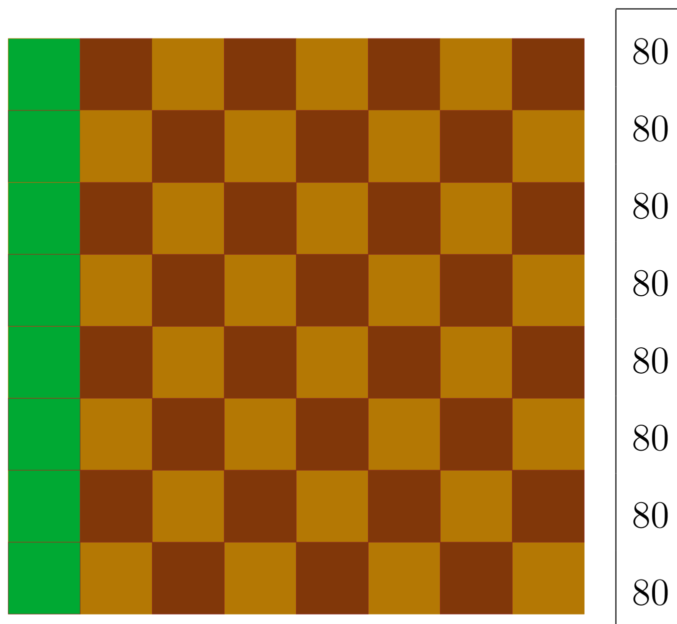


Figure 3.5: Bitboard mask of file A.

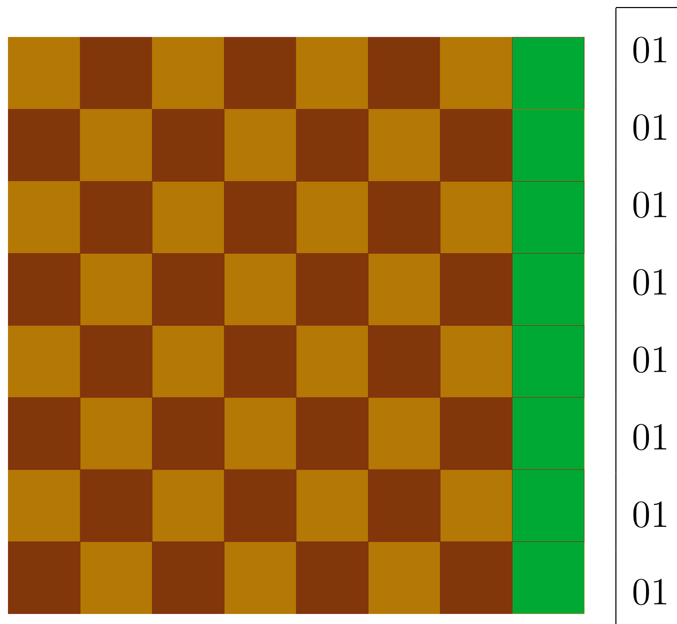


Figure 3.6: Bitboard mask of file H.

In the next section, we explain bitwise operations for identifying possible moves for each piece type, which is based on resources developed by LogicCrazy [8]. Note that in this resource, the bit ordering of each bitboard is reversed and only pseudo-legal move generation is developed. We use a more intuitive bitboard representation in which the most significant bits start from the first row of the board. We also provide a complete move generation that takes into account king safety.

3.1 List of Possible Moves

One of the challenges of move generation is in deciding how to keep track of possible moves. For each move, we need to document which piece is moving where. Difficulties arise once we also consider that there are special move cases, like promotion, where further information also needs to be conveyed, or castling rights that must be maintained. For uniformity, it is best if each move's notation is the same length to easily identify how many possible moves have been generated.

For our implementation, we decided to maintain a single string containing all moves for a given board state, where each move is 4 characters long. We concatenate 4 characters

to this string for each new move we find. In the general case, we have 4 numeric characters representing a move. The first two numbers represent the piece's initial square and the last two numbers represent the square that the piece is moving to, indicated by their row and column combination. The top left square is (0, 0), and the bottom right square is (7, 7). For example, in algebraic notation, a piece moving from B8 to C6 would be notated as '0122'. However, we have a unique representation for special cases like *en passant* and pawn promotion, which will be explained in their respective sections.

Note that the list of possible moves may be extremely shortened if the king is in check. Only legal moves that get the king out of check will be reported as possibilities, as we are strictly performing legal move generation. If the list of possible moves is empty, then we can conclude that the moving color is in checkmate, as no legal moves exist for that color.

Figure 3.7: A position for which the possible moves for white are provided.

See Figure 3.7 and the following list of possible moves for that position if white is to move. Each move within the list has been separated by spaces in this example for ease of identification.

```
6050  6040  7071  6151  6141  4251  4233  4224  4215
4231  4220  4253  4264  4275  5240  5231  5233  5264
5271  6353  6343  7364  7464  7475  7476  5536  5547
5576  5543  5534  6656  6646  6757  6747  7776  7775
```

3.2 Pawns

Moving 1 Square Forward

Pawns have the most intuitive move generation function, so we will introduce them first. Moving one square forward is the simplest move that pawns make. Consider the bitboard in Figure 3.8 that represents all white pawns in their initial positions.

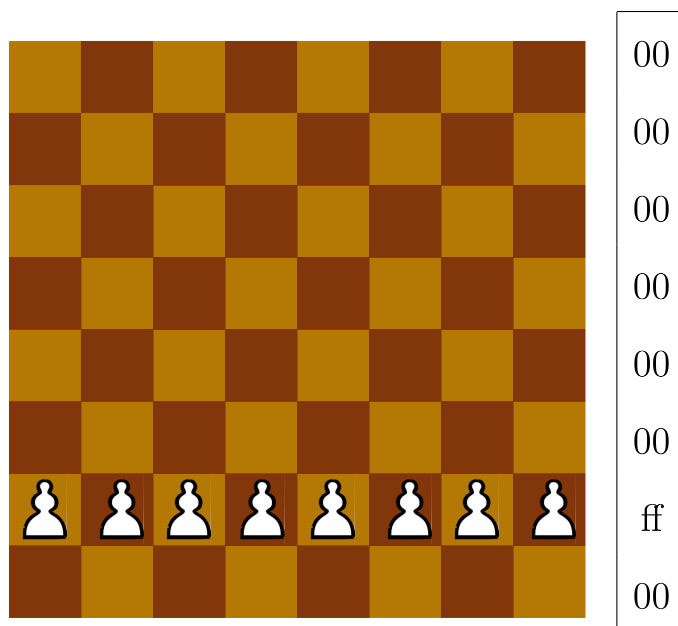


Figure 3.8: Hexadecimal bitboard representation of white pawns in their initial positions.

If we shift the bitboard of white pawns in their initial positions to the left by 8, then we get the result seen in Figure 3.9.

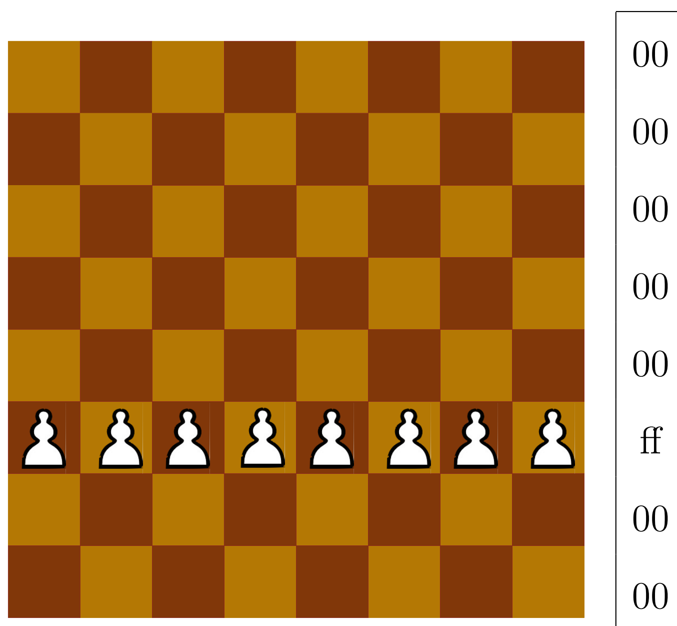


Figure 3.9: Hexadecimal bitboard representation of white pawns after being shifted left by 8.

With this left-shift operation, we have moved all white pawns on the board one square forward at once. We can move all pawns on the board at once in a single bit operation, demonstrating the efficiency and power of bitboards. Now, all we must do is filter out which moves are invalid. Pawns can only move forward into an empty square, so all moves must be to an empty square. Moves which land in rank 8 are also invalid here, as this would be a promotion, which is handled elsewhere as a special case. Thus, we'll bitwise AND our resulting moves with *empty* and NOT *rank*8 to get all of our valid white pawn moves that go one square forward. The following pseudocode is our final bitwise operation.

```
(P << 8) & empty & ~rank8
```

Now, we must convert the bitboard of all possible moves to its string representation. For each bit set to 1, we need to find its row and column position as described in the list of possible moves section. The count of trailing zeros of each set bit will serve as its location from which we will determine its row and column. We find its row using integer division, and we find its column using modular arithmetic. Since the top left square is (0,0), we will subtract each result from 7, the greatest row/column value possible. See the following pseudocode for finding the row and column of the last bit set to 1 within a bitboard.

```
location = trailingZeros(bitboard)

row = 7 - index // 8

column = 7 - index % 8
```

Note that the operation for black pawns is nearly identical. We'll instead shift our black pawn bitboard to the right (also by 8 places) and ensure that none are moved to rank 1. (See Figure 3.10.) Unless otherwise needed, we will exclude explanations for black pieces, as they are nearly identical to the corresponding white pieces, tweaked to account for pieces moving in the opposite direction.

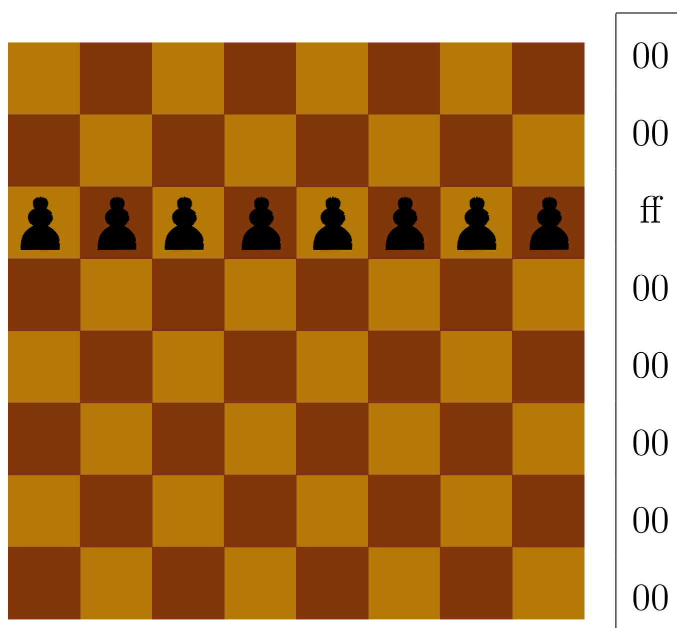


Figure 3.10: Hexadecimal bitboard representation of black pawns after being shifted right by 8.

```
(p >> 8) & empty & ~rank1
```

Moving 2 Squares Forward

To move two squares forward, the pawn must be in its starting position and must not pass through any occupied square. Recall that moving 1 square forward was accomplished by shifting the white pawn bitboard to the left by 8. It then follows that to move 2 squares forward, the

white pawn bitboard must be shifted to the left by 16 (one more rank). Note that moving a pawn two squares forward opens the possibility for capture via *en passant*. We will describe handling this later in the chapter. See Figure 3.11 for the result of shifting white pawns in their initial positions to the left by 16.

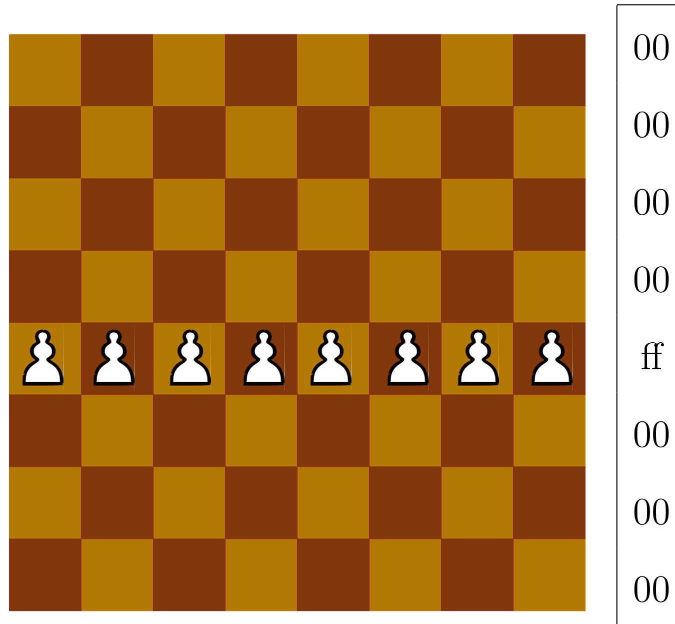


Figure 3.11: Hexadecimal bitboard representation of white pawns after being shifted left by 16.

We must now filter out which pawn moves were illegal. To find which moves are invalid, we bitwise AND the result of the bitshift by 16 with the following conditions:

- *empty*: the pawn must land on an empty square since a pawn can't capture while moving 2 squares forward
- *rank4*: white pawns moved 2 squares forward can only land on rank 4
- the result of *empty* being shifted to the left by 8 (does not pass through an occupied square)

```
(P << 16) & empty & (empty << 8) & rank4
```

Capturing to the Left

We simulate a white pawn capturing to the left by shifting its bitboard to the left by nine bits. Of course, a pawn can only move forward diagonally if it's capturing a piece (i.e. that square is occupied by a piece of the opposite color). See Figure 3.12 for an illustration. Additionally, we need to ensure that the white pawn is not landing on rank 8 or file H. Promotion is a separate case that will be handled elsewhere, so we want to ensure that no pawn captures left and lands on rank 8. We also don't want any pawns capturing left to land on file H, resulting from a pawn on file A being shifted to left by 9 squares. It is clearly impossible for a pawn to capture left and land on the rightmost column of the chess board, yet with bitboard operations, this needs to explicitly be handled. Lastly, we must make sure that the pawn lands on a black piece.

```
(P << 9) & blackPieces & ~rank8 & ~fileH
```

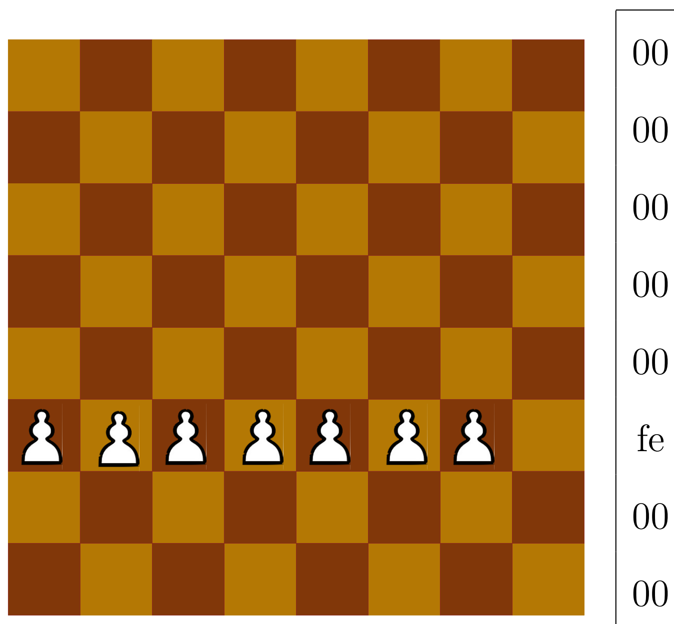


Figure 3.12: Hexadecimal bitboard representation of white pawns after being shifted left by nine.

Capturing to the Right

Capturing to the right is the same idea as capturing to the left. Once we bitshift the white pawn bitboard to the left by 7, we bitwise AND that result with the bitboard for black pieces

and the masks for rank 8 and file A (see Figure 3.13).

```
(P << 7) & blackPieces & ~rank8 & ~fileA
```

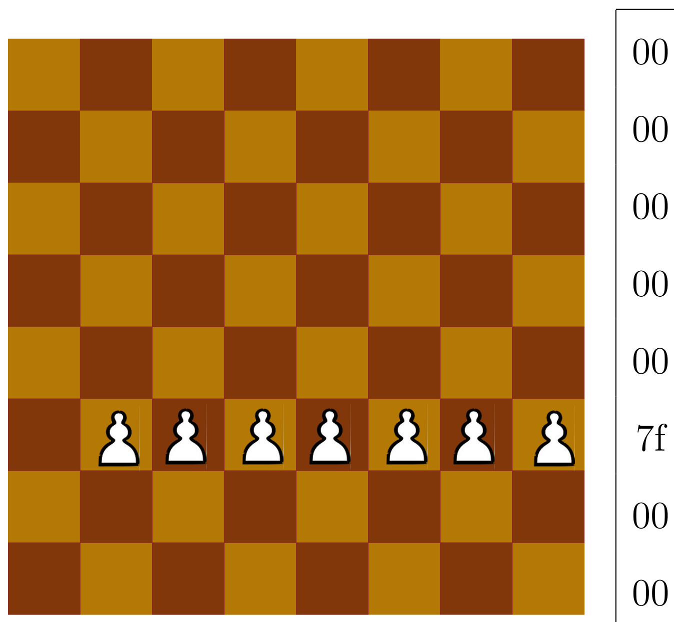


Figure 3.13: Hexadecimal bitboard representation of white pawns after being shifted left by seven.

Pawn Promotion

When promoted, a pawn can become either a rook, a bishop, a knight, or a queen. All pawn moves except for moving two squares forward can result in a promotion. Essentially, we check if moving a white pawn one square forward or a diagonal capture results in it landing on rank 8. We accomplish this by performing a bitwise AND with *rank8* instead of NOT *rank8*. For example, the final bitwise operation for checking if a pawn capturing to the right results in promotion is:

```
(P << 7) & blackPieces & rank8 & ~fileA
```

Performing this slightly modified move generation technique will find all pawn promotion moves. Since promotion is a special move category that will ultimately change the pawn's piece type, it should be differentiated within our list of moves. The notation for a promotion is entirely

up to the designer, but we indicate a white pawn promotion with 2 characters, P(R/B/Q/N), where the second character is the piece type that the pawn is promoted to. For the promotion of a black pawn, we use all lowercase characters to differentiate it from a white pawn promotion. The last two characters are the columns the piece is moving from and then to. We don't need to document the row, for we know that promoted white pawn moves from rank 7 to rank 8. We choose to document each different piece type that a pawn can promote to as a separate possible move. Therefore, promoting a pawn will add 4 moves to our move generation list.

3.3 Knights

Knights are the only pieces in chess that can jump over other pieces. They have 8 possible moving locations, though depending upon the knight's location, some may land off the board and therefore be impossible. It is easiest to calculate a mask containing these 8 target locations and bitshift it to match where a knight is actually located on our board. This mask, called *knightMask*, contains the valid moves for a knight located on F3. Thus, *knightMask* is `0xa1100110a` (see Figure 3.14).

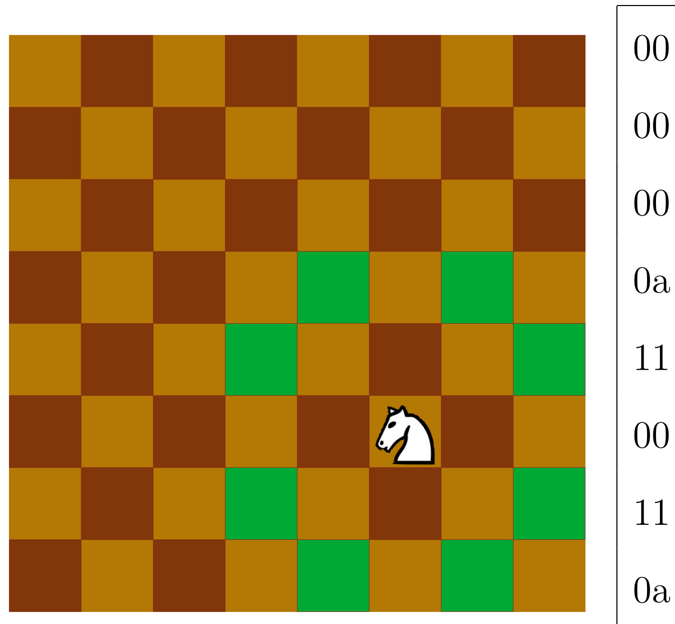


Figure 3.14: The bitboard mask used to generate knight moves. All highlighted green squares are 1's, and the knight is a reference for where the focal point of the mask is located.

We first must isolate each knight within the N bitboard. Ideally, we should be able to perform a quick bitwise manipulation that isolates the last 1 in N. The bitwise operation

```
N & ~(N - 1)
```

leaves only the knight in the least significant bit position as a 1. Once a knight has been isolated, we can generate its moves. We then 0 out the bit of this knight and continue isolating knights until all knights have been handled. The following pseudocode explains how to generate the moves for each knight individually within N.

```
i = N & ~(N - 1)
while (i != 0):
    knightLocation = trailingZeros(i)
    knightMoves(i)
    N = N & ~i
    i = N & ~(N - 1)
```

To figure out how to shift the *knightMask*, we need to identify the location of our moving knight. The count of trailing zeroes in the binary representation of the knight's bitboard is one way to quickly identify its location. The focal point of *knightMask* has 18 trailing zeroes, so we will compare our knight's location to 18 to calculate the necessary shift operation.

Consider the variable *knightLocation*, containing the count of trailing zeroes of the knight for which we are trying to generate moves. If $knightLocation > 18$, then we will need to shift *knightMask* to the left. We calculate the left-shift amount by subtracting 18 from *knightLocation*. Otherwise, we shift the *knightMask* to the right by $18 - knightLocation$.

Now, the bitshift result must be cleaned up, as moves that should have gone off the board instead wrapped around onto the next rank. If *knightLocation* modulo $8 < 4$, then we must bitwise AND the result with NOT (*fileA* OR *fileB*) since no knight on file E through H can move into files A or B, yet the *knightMask* may incorrectly wrap around into these files. (See Figure 3.15.) Otherwise, we bitwise AND the result with NOT (*fileG* OR *fileH*). To visualize how the *knightMask* wraps around to the next rank when the knight is located on file A, see Figure 3.16.

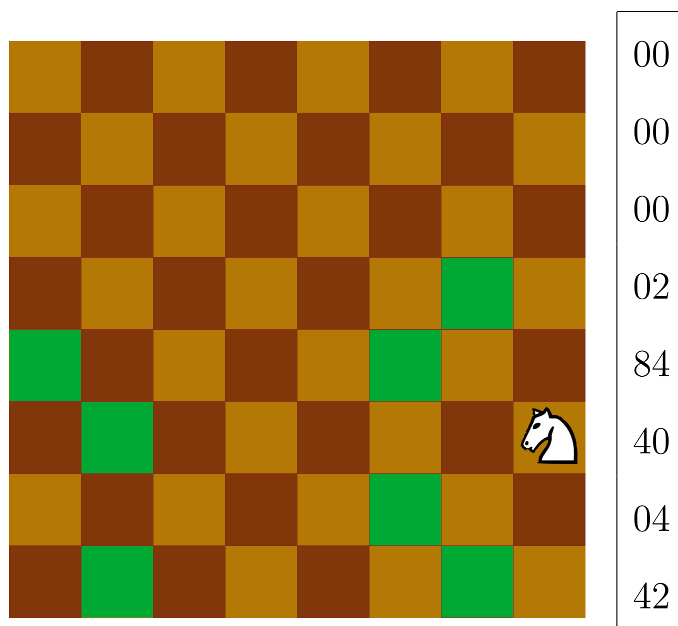


Figure 3.15: The *knightMask* wrapped around onto files A and B.

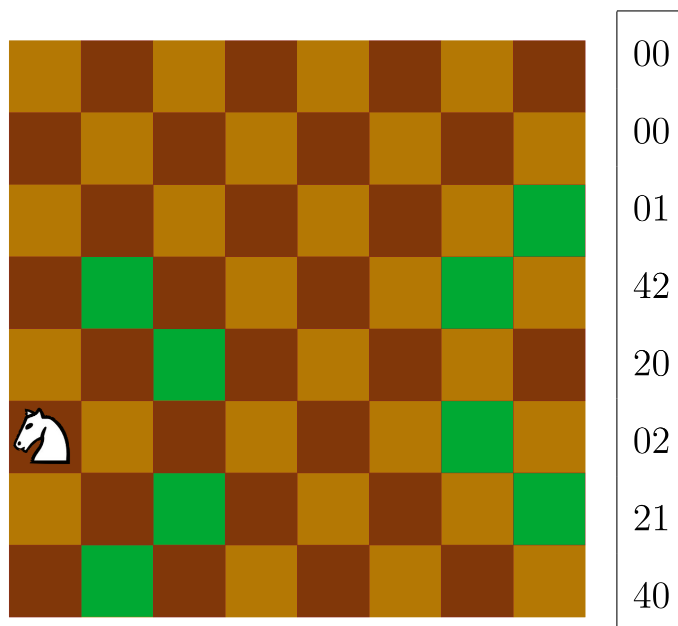


Figure 3.16: The *knightMask* wrapped around onto files G and H.

The following pseudocode can be used to calculate moves for a knight.

```
if knightLocation > 18:
```

```
    result = knightMask << (knightLocation - 18)
else:
    result = knightMask >> (18 - knightLocation)

if knightLocation % 8 < 4:
    result = result & ~(fileA | fileB)
else:
    result = result & ~(fileG | fileH)
return result
```

3.4 Sliding Pieces

Queens, rooks, and bishops are all sliding pieces. Sliding pieces are able to move along their respective rays until they reach the end of the board, encounter a piece of their own, or capture an opposing piece. In our move generation functions for sliding pieces, we aim to generate a bitboard where every square that a sliding piece can move to is marked with a 1. Remarkably, we can efficiently compute this bitboard by utilizing properties of bitwise subtraction. Similarly to knights, the following operations for sliding pieces must be applied to each individual piece of its type. The operation explained for knights can be extended to any piece type's bitboard to isolate each individual piece. Consider the following isolated position given in Figure 3.17, where a white rook is attacking a black pawn. A white pawn is also located on the same rank. We wish to generate all horizontal moves for this white rook on its rank.

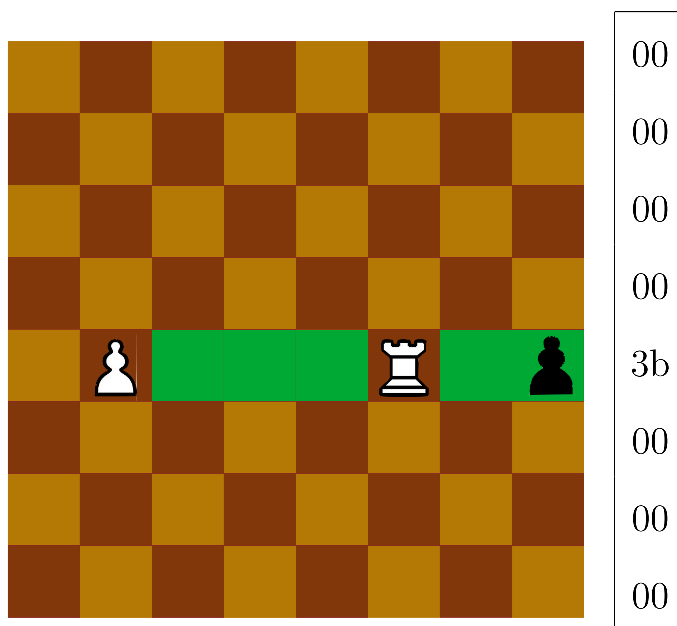
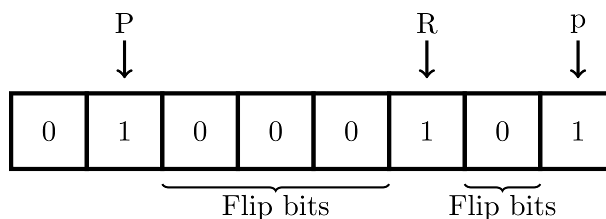
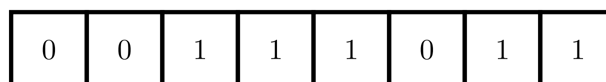


Figure 3.17: The bitboard of desired moves, indicated in green, to be generated for a white rook moving within its rank.

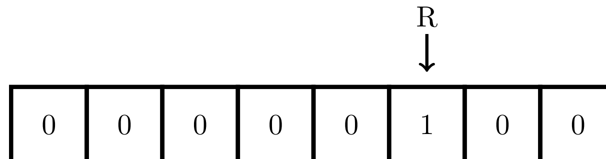
First, we will isolate the rank that the rook resides on. The bit representation of the pieces occupying this rank, referred to as *occ*, is as follows:



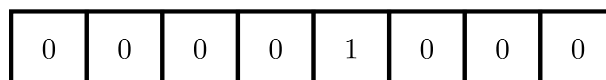
For horizontal rook moves, we wish to flip the bits between the rook and the pawns in *occ* to 1s. For the white pawn, we want its square to be 0, denoting the rook cannot move to that square. If the pawn is black, then the rook can capture it, so that square should be marked as a 1. Our desired bitboard of horizontal moves for the white rook follows. Note that the location of the rook is a 0, as a piece remaining in the same location is not a generated move.



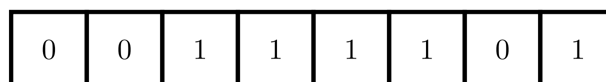
Consider our slider's position (the rook) within this rank, s:



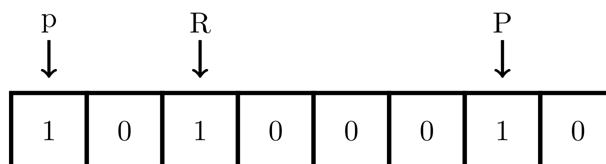
If we multiply s by 2, then the 1 shifts to the left by one square. Consider the resulting bitboard, *2s*:



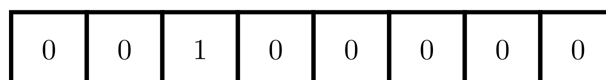
Initially, we'll focus on finding moves to the left of the rook, which is the side where the white pawn resides. The rook is able to move to each of the three squares between itself and the white pawn. Since we wish to flip these three intermediate bits in *occ* to 1's, we can utilize the properties of carrying bits in bitwise subtraction to achieve this. Subtracting *2s* from *occ* will force the flip of the intermediate bits to 0's. See the result of $occ - 2s$:



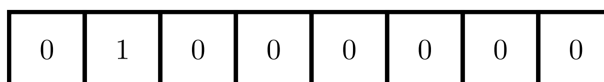
Thus, we've flipped the intermediate bits as desired through the bitwise carry to find the horizontal rook moves to the left. To do the same on the left, we must reverse the involved bitboards and finally reverse the resulting bitboard. We will create a bit reversal function, *reverse*, which reverses the ordering of bits on the bitboard. Specifically, the most significant bit becomes the least significant bit and vice versa. Continuing the above rook example, consider *reverse(occ)*:



The following is the bitboard of $reverse(s)$:

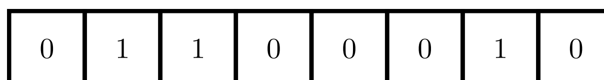


To again achieve the bit flipping that simulates the rook sliding until the first occupied square it encounters, we use $2reverse(s)$:



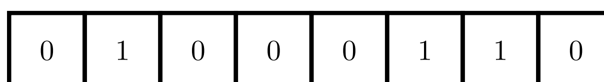
0	1	0	0	0	0	0	0

We subtract $2reverse(s)$ from *reverse(occ)* to calculate the following bitboard:



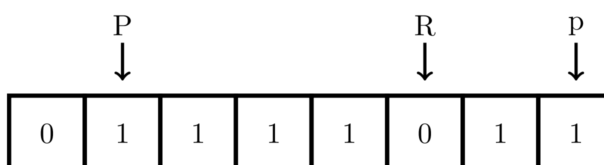
0	1	1	0	0	0	1	0

Lastly, since all involved bitboards have been reversed, we must reverse the resulting bitboard to properly calculate the rook's moves to the right. The following is $reverse(reverse(occ) - 2reverse(s))$:



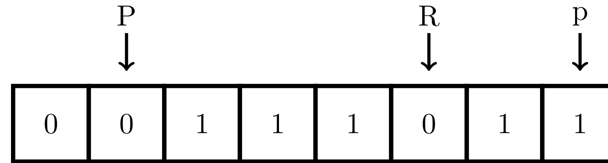
0	1	0	0	0	1	1	0

Notice that in both $occ - 2s$ and $reverse(reverse(occ) - 2reverse(s))$, the bit operation for the opposite side contains a 1 at the location of the first piece the sliding piece encounters within its ray. Performing a bitwise XOR operation will place a 1 at the location of the first piece encountered along a sliding piece's ray. Consider the XOR of the moves to the left and right found for the white rook position:



0	1	1	1	1	0	1	1

After the XOR operation, the possible moves for the white rook have nearly been accurately found. The bits between the sliding piece and the first piece it encounters, including the bit mapped to the location of the first piece encountered, are set to 1. The only incorrect bit is the 1 where the white pawn is located, falsely indicating that a white rook can capture its own pawn. To remove the possibility of a piece capturing another piece of the same color, we bitwise AND the XOR result with NOT *whitePieces*, giving us the following final result for the rook's horizontal moves:



For this example, the appropriate rank mask has been implicitly applied on all above bitboards to isolate our row of interest. The pseudocode of the general operation for finding the horizontal moves of a rook, assuming *mask* is the *rank4* mask since the rook is located on the 4th rank, follows.

```
left = (occ & mask) - 2s
right = reverse(reverse(occ & mask) - 2reverse(s))
horizontal = (left XOR right) & mask
```

This bitwise operation for generating moves, reliant upon predictable bit flipping via bitwise subtraction, works for sliding pieces in all possible directions when we also apply the appropriate mask. The mask ensures sliding pieces remain on their correct respective rays. Depending upon the piece type, we either apply a file and rank mask or a diagonal and anti-diagonal mask.

Rooks

The following pseudocode illustrates how to use the above bitwise operations and masking to find all moves for a rook. Both *rankMask* and *fileMask* are the respective masks containing the rook for which we are generating moves. Note that the following pseudocode contains the helper functions for finding horizontal and vertical moves, and the mask ensuring a piece of its own color cannot be capture must be applied to the function's result. The same is true for the other sliding pieces.

```
#horizontal moves
h0 = (occ & rankMask) - 2s
h1 = reverse(reverse(occ & rankMask) - 2reverse(s))
horizontal = (h0 ^ h1) & rankMask
```

```
#vertical moves
v0 = (occ & fileMask) - 2s
v1 = reverse(reverse(occ & fileMask) - 2reverse(s))
vertical = (v0 ^ v1) & fileMask
```

Bishops

Likewise, the following pseudocode illustrates how to find all moves for a bishop. Both *diagonalMask* and *antidiagonalMask* are the respective masks containing the bishop for which we are generating moves.

```
#diagonal moves
d0 = (occ & diagonalMask) - 2s
d1 = reverse(reverse(occ & diagonalMask) - 2reverse(s))
diagonal = (d0 ^ d1) & diagonalMask

#anti-diagonal moves
a0 = (occ & antidiagonalMask) - 2s
a1 = reverse(reverse(occ & antidiagonalMask) - 2reverse(s))
antidiagonal = (a0 ^ a1) & antidiagonalMask
```

Queens

Once moves can be generated for rooks and bishops, generating moves for a queen is easy. We simply generate both rook and bishop moves for the queen's location, and we return the results of both functions as possibilities.

3.5 King

Similar to generating knight moves, we will bitshift a mask for the king's moves depending upon the location of the king. The mask, *kingMask*, is `0x70507`, with the focal point of the mask being G2 (see Figure 3.18 below). The variable *kingLocation* contains the count of trailing

zeroes of the king we are generating moves for. The count of trailing zeroes for the focal point of *kingMask* is 9, so we will be comparing *kingLocation* to 9 to figure out which direction to bitshift *kingMask*.

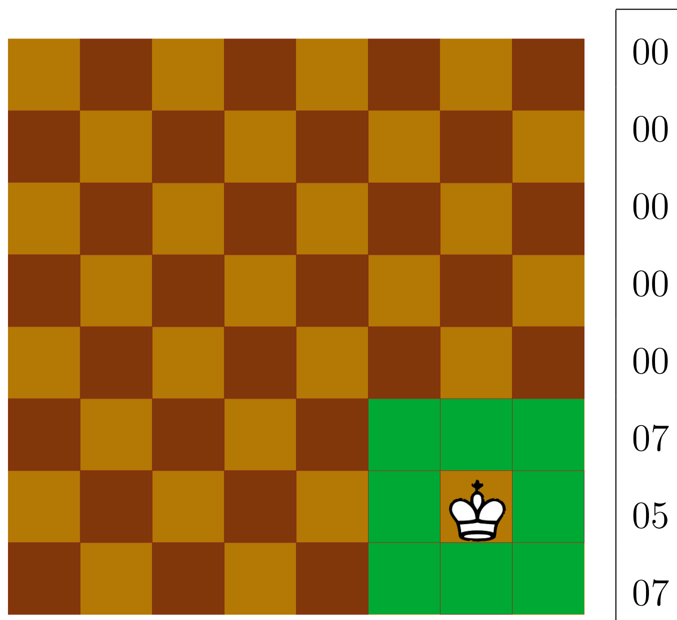


Figure 3.18: The bitboard mask used to generate king moves. All highlighted green squares are 1's, and the king is a reference for where the focal point of the mask is located.

If *kingLocation* is greater than 9, then we need to shift *kingMask* to the left by $kingLocation - 9$. Otherwise, we need to shift *kingMask* to the right by $9 - kingLocation$. Similarly to shifting the *knightMask*, we need to clean up any bits that wrapped onto the next rank instead of moving off the board. To do so, if *kingLocation* modulo $8 < 4$, then we must bitwise AND the result with NOT (*fileA* OR *fileB*). Otherwise, we bitwise AND the result with NOT (*fileG* OR *fileH*).

Finding Safe Squares

It is illegal for a king to move into check. Therefore, when generating possible king moves, we must only generate moves where the king moves to safe squares, or squares that are not attacked by the opposing color. See Figure 3.19 for an example bitboard of unsafe squares. Unfortunately, though it would increase efficiency, the generated moves for the opposite color

from the previous turn cannot be used. Whichever move was made in the previous turn could affect this list (consider if the moved piece now blocks another piece's line of attack). We must newly generate moves for the opposing color to ensure that generated moves are updated to reflect the previous move made.

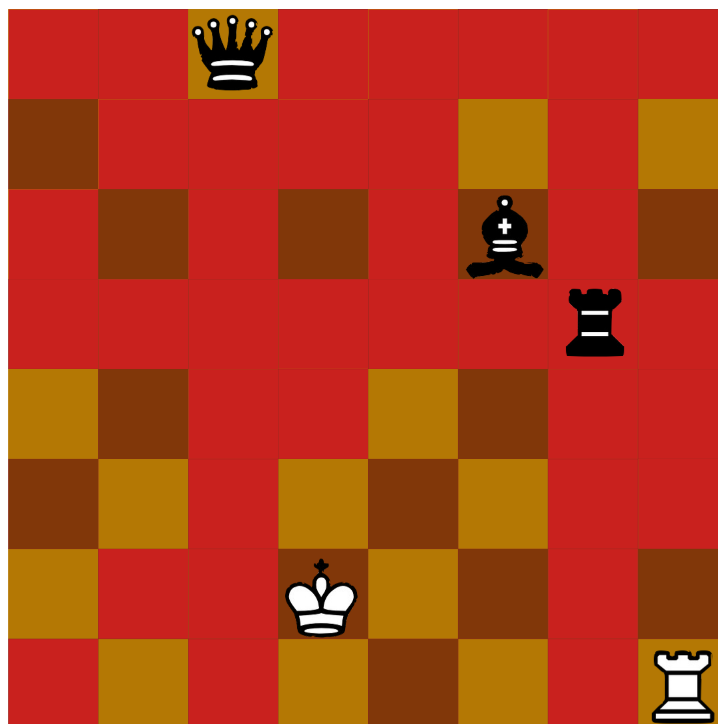


Figure 3.19: The bitboard of unsafe squares, colored in red, for a white king to move to.

We will use a separate method to generate the bitboard of all safe squares for a king to move to, *safe*. Slightly modifying the move generation techniques for each piece of the opposite color as described in this chapter will identify squares that are attacked, *unsafe*. We will simply bitwise NOT *unsafe* to derive *safe*. A key component of ensuring the accuracy of the safe squares found is to temporarily remove the king (of the color moves are being generated for) from the board, as the king's current position may incorrectly block a portion of an opposing sliding piece's attack. Consider a black rook checking a white king. With the king on the board, the squares beyond the king on the side opposite of the rook are falsely found to be safe (see Figure 3.20). Removing the king while generating the opposing color's moves will correctly mark these squares as unsafe.

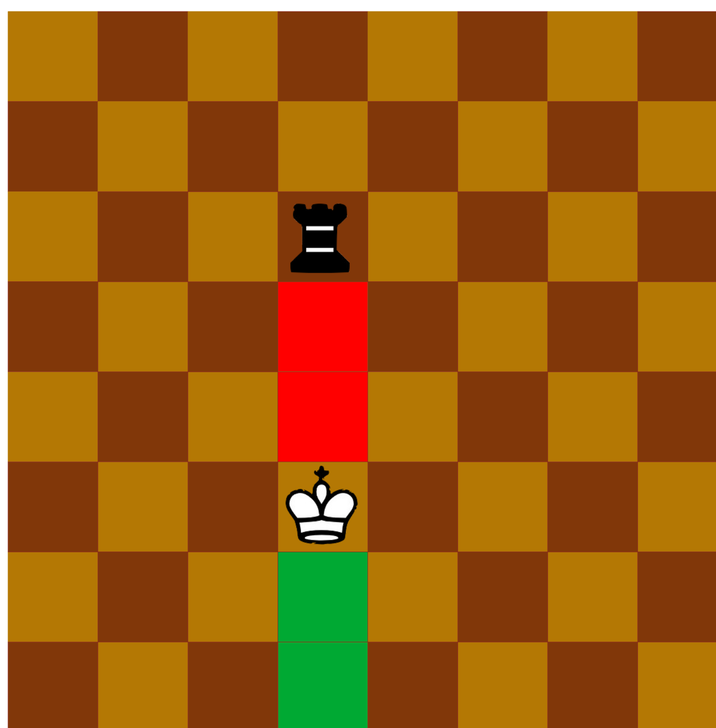


Figure 3.20: Safe squares generated when the king remains on the board. Red squares are unsafe while green squares are falsely computed as safe.

The following list succinctly describes how to calculate *unsafe*. To add the unsafe squares resulting from the attacks of each piece type, we bitwise OR *unsafe* with each operation

described below and assign the result back to *unsafe*.

- For pawns, we only need to consider their diagonal captures, as pawns cannot attack squares that are in the same file.
- For knights, the bitwise operation operation is identical.
- For sliding pieces, the bitwise operation is identical.
- For the king, we do not need to factor in king safety. Kings can never be adjacent to one another; any square that an opposing king attacks is unsafe for the moving king.

The pseudocode for generating *unsafe* for white follows, where we generate moves for black pieces.

```
kingPosition = k
k = 0 #temporarily remove king from board

#pawn diagonal capture moves
unsafe = (p >> 7) & ~fileH
unsafe |= (p >> 9) & ~fileA

for each black knight:
    if knightLocation > 18:
        result = knightMask << (knightLocation - 18)
    else:
        result = knightMask >> (18 - knightLocation)
    if knightLocation % 8 < 4:
        result = result & ~(fileA | fileB)
    else:
        result = result & ~(fileG | fileH)
    unsafe |= result
```

```
for each black bishop and queen:
    unsafe |= diagonal sliding piece moves

for each black rook and queen: #generate horizontal sliding piece moves
    unsafe |= horizontal sliding piece moves

#king moves, with the pseducode provided below
if kingLocation > 9:
    result = kingMask << (kingLocation - 9)
else:
    result = kingMask >> (9 - kingLocation)
if kingLocation % 8 < 4:
    result = result & ~(fileA | fileB)
else:
    result = result & ~(fileG | fileH)
unsafe |= result

k = kingPosition
return unsafe
```

Once we have calculated *safe*, the following pseudocode demonstrates our final bitwise operation for generating king moves:

```
if kingLocation > 9:
    result = kingMask << (kingLocation - 9)
else:
    result = kingMask >> (9 - kingLocation)
if kingLocation % 8 < 4:
    result = result & ~(fileA | fileB)
else:
    result = result & ~(fileG | fileH)
```

```
return result & safe
```

3.6 Castling

Castling has strict rules which complicate identifying if it is a legal move for a given position. For each color, there are two castling possibilities to check for: queenside and kingside. Both the king and the rook involved in castling must be unmoved in their starting positions. All squares between the king and the castling rook must be empty. Castling cannot be utilized to evade check, meaning that the king must not be in check to castle. Moreover, all squares which the king pass over and land on must be safe squares, or squares that are not attacked by an opposing piece.

To help ensure that castling is generated as a possible move only when the involved king and rook are unmoved, we can maintain 4 boolean values indicating the castling rights for a king of each color to castle either kingside or queenside. These boolean values are *castle_wk*, *castle_wq*, *castle_bk*, and *castle_bq*. Initially, castling rights are read in from the FEN, which we discuss in Chapter 4. If castling rights are lost, like when an involved rook moves or becomes captured, then we set the appropriate boolean value(s) to false.

We will first evaluate castling white kingside. The white king cannot be in check when it castles, so calculating *unsafe*, as described in Chapter 3.5, and performing a bitwise AND of *unsafe* and K must result in 0. Then, *castle_wk* should be true and the kingside rook must be in its original position, which is `0x1` (see Figure 3.21). If we bitwise AND R with `1`, then we can check that the white rook is on the bottom right square. To ensure that the king does not pass through or land in check, we use *unsafe*. All squares between the king and the rook must also be empty. To check that both conditions are true for all intermediate squares, we bitwise OR *occupied* and *unsafe*. We bitwise AND the result with all intermediate squares (see Figure 3.22). If the final result is 0, then all intermediate squares are empty and safe. All necessary criteria have been met, so we conclude that white can castle kingside.

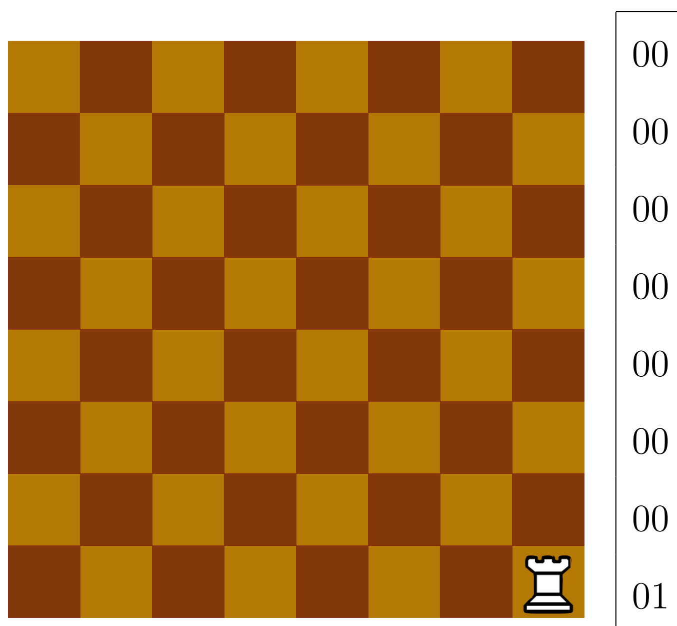


Figure 3.21: The necessary position of the white kingside rook for castling.

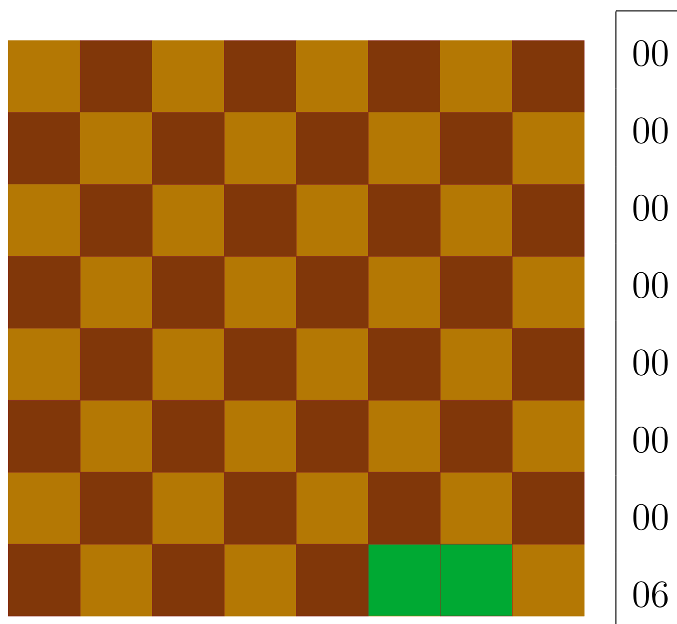


Figure 3.22: The intermediate squares of a white castling kingside.

The operation for identifying the possibility of castling white queenside only needs to be modified to reflect the different intermediate squares and the position of the queenside rook.

To quickly calculate the bit representation of these squares, we can bitshift 1 to the left by the appropriate amount. For example, the bitboard of the white queenside rook is `10000000`, which is equivalent to 1 << 7. Bitshifting will be especially helpful when dealing with castling for black, which is described later.

We must add any castling possibilities found to the list of moves. In our implementation, we choose to represent the movement of the king. Since a king cannot move 2 squares unless it is castling, documenting the king's movement allows us to easily determine that this move is a possible castle. We notate a white kingside castle with '7476'. Likewise, we notate a white queenside castle using '7472'.

The pseudocode for generating possible white castling moves follows.

```
if unsafe & K == 0: #if king is safe
    #check castling white kingside
    if castle_wk and ((1 & R) != 0): # castling rights and R in
                                      # initial position
        if ((occupied | unsafe) & (2 | 4)) == 0:
            # intermediate squares are empty and safe
            castling kingside possible, add '7476' to move list

    #check castling white queenside
    if castle_wq and ((1 << 7 & R) != 0):
        if ((occupied | unsafe)
                & ((1 << 4) | (1 << 5) | (1 << 6)) == 0:
            castling queenside possible, add '7476' to move list
```

Since generating castling moves for black is substantially different from generating castling moves for white, the pseudocode for black castling is provided below. Note that the overall operation is nearly identical, but the main differences are caused by the opposite locations of the squares involved in the castling.

```
if unsafe & k == 0: #if king is safe
    #check castling black kingside
```

```
    if castle_bk and ((1 << 56 & r) != 0):
        if (occupied | unsafe) & ((1 << 57) | (1 << 58)) == 0:
            castling kingside possible, add '0406' to move list

    #check castling black queenside
    if castle_bq and ((1 << 63 & r) != 0):
        if (occupied | unsafe)
                & ((1 << 62) | (1 << 61) | (1 << 60)) == 0:
            castling queenside possible, add '0402' to move list
```

3.7 *En Passant*

If the last move by the opposite color was moving a pawn two squares forward, then the possibility of capturing that pawn via *en passant* must be considered. Only enemy pawns are able to capture a pawn in passing. To account for *en passant*, we need to check for the most recent move, and if it was a pawn moving two squares forward, whether a moving color's pawn can perform a diagonal capture on the *en passant* target square (the square which the previous pawn skipped over).

We will have a helper function to determine if the last move was a pawn moving two squares forward, and if so, which file that pawn resides on to check whether we can add an *en passant*. The helper function gets the previous move information from the FEN, which will be discussed in Chapter 4. If *en passant* is impossible, we return 0. Otherwise, we return the file mask that contains the *en passant* target square. Then, we use the result from the helper function to check whether an *en passant* is a valid move. For a white *en passant* capture to the right, our helper function must return a file mask, a black pawn must be located 1 square to the left of a white pawn, and both pawns must be on rank 5. Similarly to a regular white pawn capture to the right, we ensure that the bitboard result contains no 1s along file A, as they would be the result of an inaccurate bit-shift wraparound.

The pseudocode for a white *en passant* to the right follows:

```
EP = en_passant() #call to helper function
```

```
((P >> 1) & p) & rank5 & ~fileA & EP
```

The pseudocode for a white *en passant* to the left follows:

```
EP = en_passant() #call to helper function
((P << 1) & p) & rank5 & ~fileH & EP
```

3.8 Check

If the king is in check, then the list of possible moves may drastically shorten. A player can escape check in 3 possible ways: by moving the king, by capturing the attacking piece, or by blocking the attack of the attacking piece. However, there are caveats to these simple escape options. If more than one piece is checking the king, then capturing only one of those pieces will not eliminate check. Moreover, if the attacking piece is a knight or a pawn, then its attack cannot be blocked.

First, we must identify if the king is in check. We can efficiently perform this search by pretending that the king is all of the pieces that could be checking it and seeing if it lands on a piece of that type. For example, consider that the white king is in check by a black knight, and we are attempting to identify this attack. If we input the king's location to the knight move generation function, then one of the squares returned will be the location of the checking black knight. We can then conclude that the black knight is checking the king. The same approach will work for pawns, rooks, bishops, and queens. Once we've identified all pieces checking the king, we can keep track of their piece type and bitboard location in a list, *checkingPieces*.

In our main function that generates the moves of all pieces, we must decide whether to perform regular move generation or a modified move generation because the king is in check. If the length of *checkingPieces* is greater than 0, then the king is in check and we must perform modified move generation. The pseudocode for move generation while the king is in check is below. Depending upon the number of pieces and piece type checking the king, we could capture the piece and/or block its attack. Regardless, if the king is in check, moving the king to safety (which includes capturing the checking piece) is always a legal move, so we add these to

the list of possible moves. Since we already ensure the king only moves to safe squares within its move generation operation, we simply generate moves for the king as previously explained.

```
if length(checkingPieces) == 1:
    if piece is not a knight or a pawn:
        generate moves which block or capture the attacking piece
    else:
        generate moves which capture the attacking piece
generate king moves (will be moved to safety)
```

Limiting moves to only those that capture the checking piece is trivial. Since we have the bitboard of the checking piece, we bitwise AND the moves found for pieces other than the king with the checking piece's bitboard. With this bitwise operation, we've filtered out moves that do not land on the checking piece's square.

To identify the moves that block the attack of the sliding checking piece, we must find the line of sight between the king and the checking piece. That is, we wish to generate the bitboard that contains 1s in all of the squares that are on the checking piece's line of attack on the king. Similarly to finding any checking pieces, we can pretend that the king is a sliding piece of the same type checking it. By comparing the ranks and files of the king and the checking piece, we can determine whether the attack is a rook-like maneuver or a bishop-like maneuver. Note that for the purpose of finding the line of sight, we can treat the queen like either a rook or a bishop. We pretend that the king is either a rook or a bishop, depending upon the line of attack on the king. The line of sight is the bitwise AND of the moves of the checking piece and the moves of the king as the correct sliding piece type. See Figure 3.23 for an illustration. We bitwise AND the line of sight with all moves generated to only keep moves that block the attack of the checking piece.

Finally, as mentioned above, we add to the legal moves list all possible king moves. The king itself can only move to safe squares or capture the checking piece. If more than one piece is checking the king, then moving the king out of check is the only way for that color to escape check. Again, if no moves are possible, then it is checkmate.

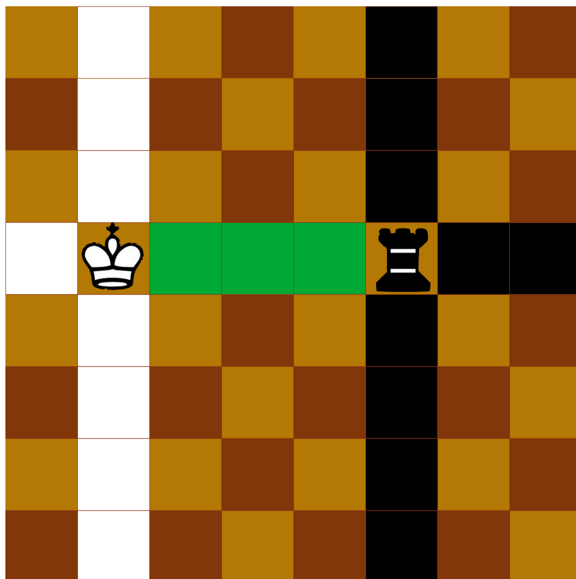


Figure 3.23: The overlap of rook and king sliding moves calculates the desired line of sight.

3.9 Pinned Pieces

Pinned pieces are easy for a human to handle yet difficult for chess engines to recognize. Legal move generation must filter out moves which leave a king in check. A piece is pinned if it is the only piece blocking an opposing piece's attack on its king. The pinning piece of the opposing color must be a sliding piece. This could be, for example, a black queen on the same rank as a white king, where a white rook blocking the attack keeps the king out of check (see Figure 3.24). Pinned pieces can only either stay within the line of attack of the opposing piece or capture the pinning piece, as moving a pinned piece and leaving its own king in check is an illegal move.

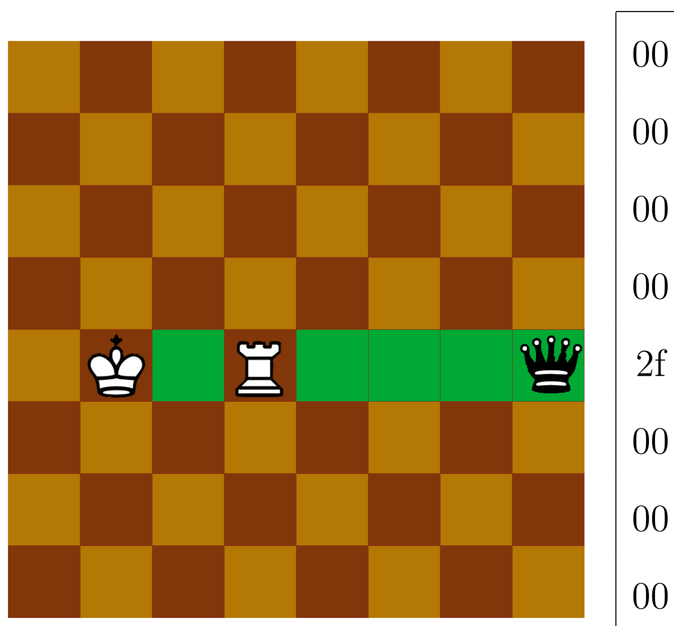


Figure 3.24: A pinned rook that can only move within its rank, denoted by green squares.

We must first identify all pinned pieces and the respective rays they must stay within. Then, we can perform normal move generation for the pinned piece and bitwise AND the resulting possible moves bitboard with the pinned ray. The bitwise AND ensures any move the pinned piece makes will not leave the king in check.

We will use an approach similar to what is described in Chapter 3.8, but it must be modified to accommodate for the fact that we are unsure of the combination of the location and the type of any possible pinned pieces. By pretending that the king is a bishop or a rook (a sliding piece), we can identify any pieces of the same color that could potentially be a pinned piece. Consider the example in Figure 3.24. Generating rook moves for the king and removing the bitwise AND of NOT *whitePieces* (which ensures that a piece of its own color cannot be captured), we find that the white king can land on the white rook on d4. In other words, the white rook is within the white king's line of sight. The white rook is correctly identified as a possible pinned piece. Now, we must check the line of attack for all black rooks, queens, and bishops to see if they attack this white rook. We find that the black queen attacks this rook and all 3 pieces are on the same rank, so we then conclude that the rook is a pinned piece. The

rook must stay within the identified line of attack or capture the attacking black sliding piece, the queen. (See Figure 3.25.)

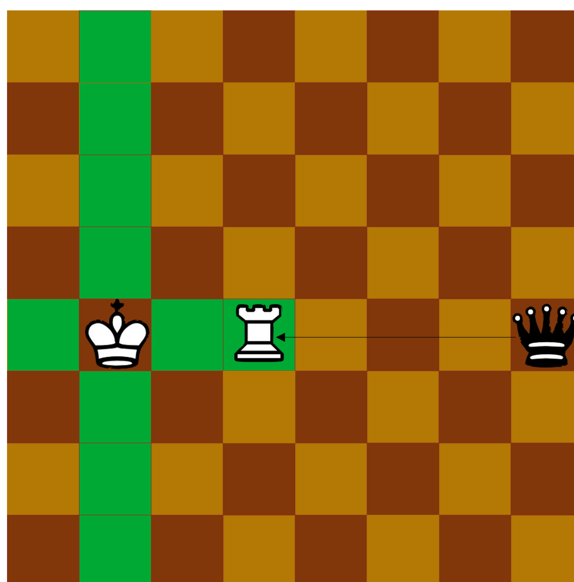


Figure 3.25: The white rook is identified as a pinned piece.

We'll explain how to do this for white to move, as the bitwise operation is symmetrical for black. To find pinned pieces, we first the calculate horizontal and vertical rays (rook-like moves) for the white king, *straight_king_ray*. Again, this is exactly like finding rook moves for the king, but we disregard the bitwise AND of NOT *whitePieces* to see if the king's line of sight includes any white piece. For all black queens and rooks on the same rank or file, we then see if they attack any white piece within the white king's line of sight on the same ray. Note that we must make sure all 3 pieces are on the same ray. It does not suffice to simply see if the rays of the white king and a black sliding piece overlap on a white piece (see Figure 3.26). We do the same operation for diagonals and anti-diagonals, pretending that the king is a bishop.

Lastly, we calculate the ray that any identified pinned piece must remain within. We find the appropriate sliding moves for the pinned piece (regardless of the piece's type). If the pinned piece is on the same rank or file as the king, then we generate rook moves for the pinned piece. Likewise, if the pinned piece is on the same diagonal or anti-diagonal as the king, we

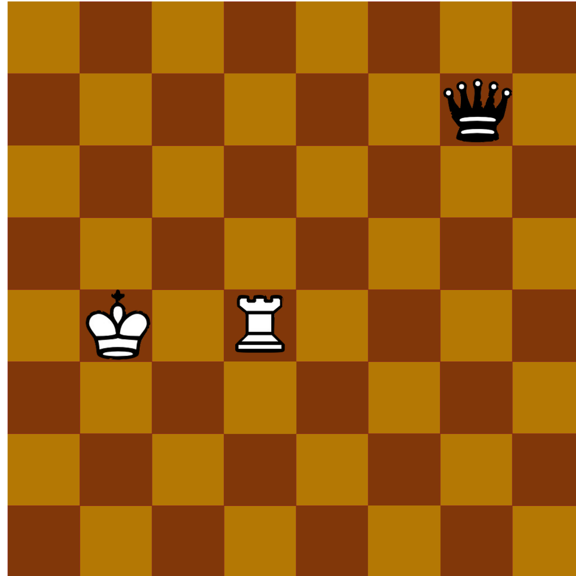


Figure 3.26: Position which demonstrates the need to check the locations of pieces and not only their line of attack.

generate bishop moves for the pinned piece. We bitwise AND the result with NOT *occupied* and the rank/file/diagonal/anti-diagonal mask that the pinned piece and king reside on. We finally bitwise OR this result with the black attacking piece, as the pinned piece should still be able to capture the attacking piece.

Below is the pseudocode for finding any white pinned pieces and their respective pinned rays.

```
straight_king_ray = horizontal and vertical moves for king
diagonal_king_ray = diagonal and antidiagonal moves for king
rank = rank of king's location
file = file of king's location
diag = diagonal of king's location
anti = antidiagonal of king's location
```

```
for each black sliding piece:
    B = bitboard of current black sliding piece
    if piece is not a bishop and (B & rank != 0 or B & file != 0)
        poss = horizontal and vertical moves of B
        poss = poss & straight_king_ray & whitePieces
        if poss != 0: # pinned piece identified
            if poss & rank != 0: # pinned piece on same rank as king
                ray = straight_pinned_ray(poss, rank) | B
            else: # pinned piece on same file as king
                ray = straight_pinned_ray(poss, file) | B

    else if piece is not a rook and (B & diag != 0 or B & anti != 0):
        poss = diagonal and antidiagonal moves of B
        poss = poss & diagonal_king_ray & whitePieces
        if poss != 0:
            if poss & anti != 0: # pinned piece on same antidiagonal
                ray = diagonal_pinned_ray(poss, anti) | B
            else: # pinned piece on same diagonal
                ray = diagonal_pinned_ray(poss, diag) | B

straight_pinned_ray(poss, mask):
    ray = horizontal and vertical moves of poss
    ray = ray & ˜occupied & mask
    return ray

diagonal_pinned_ray(poss, mask):
    ray = diagonal and antidiagonal moves of poss
    ray = ray & ˜occupied & mask
    return ray
```

In our implementation, we add all identified pinned pieces and the bitboard of the squares each must remain within to a dictionary for quick access. We also create a bitboard of all pinned pieces (where a 1 denotes that a pinned piece resides in that square). Then, when generating moves for pieces, we can easily identify if a piece is pinned and restrict its moves appropriately by bitwise ANDing its moves with its associated pinned ray. For sliding pieces, since we generate the moves for each piece separately, we can easily ensure that the pinned piece we are generating moves for has illegal moves filtered out.

Unfortunately, our efficient move generation technique for pawns where all pawns are manipulated in one bit shift operation makes it difficult to account for pinned pawns. A pinned pawn must be considered individually, as it should be allowed to move within the ray in which it is pinned. See Figure 3.27 for an example chess position with two pinned pawns, yet only one has any possible moves. To keep the general pawn move generation efficient, we bitwise NOT the pawn bitboard with the pinned piece bitboard, temporarily removing all pinned pawns. We then do the general pawn move generation as normal with all unpinned pawns, and for each pinned pawn, we generate its moves separately as we do with sliding pieces.

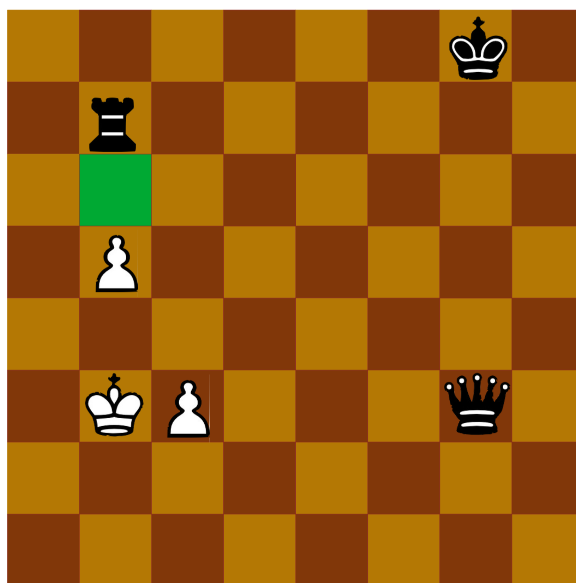


Figure 3.27: Two pinned pawns. Only one can move, and that square is highlighted in green.

Chapter 4

Testing

This chapter presents our rigorous testing framework. We introduce a popular testing approach, and then we propose an alternative user-friendly framework that utilizes crowdsourcing to generate test cases.

4.1 Perft Testing

Perft is a popular testing function for chess engines. For a given position, Perft traverses its tree of possible moves. By using move generation and making/unmaking moves, it explores all possible sequences of moves. This is a robust testing approach that we recommend further research into for advanced programmers, yet we believe that it is too complex for most undergraduate students to implement themselves. With the complexity of building a chess engine, we would rather like to explore a more structured and visual testing approach. However, if interested in leaning about Perft testing, we recommend visiting the following resource for more information [7].

4.2 Crowdsourcing

Perhaps the most straightforward approach to testing move generation is to feed in positions and check that the list of all possible moves from that given position is accurate. If we test using comprehensive positions that cover edge cases we know that our engine is particularly

susceptible to failing, then we can be reasonably assured that our move generation is correct. Unfortunately, it is difficult and time-consuming for one person to generate these test cases and all of the possible legal moves for the position. That individual would also be more prone to making errors when trying to list all legal moves, a burdensome process. Therefore, we propose a crowdsourcing approach to involve others in the rigorous testing process.

There are many examples of crowdsourcing, but a particularly notable crowdsourcing campaign was Google's *The ESP Game*, designed to create labels for Google Images. With participation open to all, users were paired and provided images which they had to describe using keywords within a set time limit [5]. The catch was that these users had to agree upon an answer, yet they had no way of communicating with one another. If two users described an image using the same keyword(s), then Google could assume it to be a good label for the image, as two independent sources agreed upon the label. With a points system and a competitive gaming atmosphere, users enjoyed contributing to the crowdsourcing campaign. Some users even had consecutive playing times of over 15 hours, attesting to the fun and addictive nature of *The ESP Game* [5]. As evidenced by Google's campaign, crowdsourcing can both be beneficial for the developer and enjoyable for the contributor.

Inspired by *The ESP Game*, we aimed to make crowdsourcing test cases fun to encourage user engagement. We seek the participation of chess experts, individuals who understand chess and are able to identify all legal moves for a given chess position. Since each move is a four digit number (with the exception of a few special cases), it is unreasonable to ask contributors to provide the full list of possible moves for a position in this cumbersome format. To make crowdsourcing as user-friendly as possible, we designed a GUI where moves can be inputted in a game-like fashion, and these moves are translated to their appropriate digit and character representation. We use Forsyth-Edwards Notation to input chess positions to the GUI for the chess expert to work with.

4.3 Forsyth-Edwards Notation

Forsyth-Edwards Notation (FEN) is a concise string representation of a chess position. Each piece on the board is represented by a single character as described in chapter 2, beginning in

the top left corner of the board. Each rank is separated by the '/' symbol. If a square is empty, then the count of the consecutive spaces within the same rank is used instead. For example, the FEN of the starting position for a chess board is:

`rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR`

Along with the positions of pieces on a chess board, there is other information that we must have for a given position including color to move, castling rights, and an *en passant* target square. FEN provides an intuitive way to document this information as well, which is included after the piece positions. Each field is separated by a space. Consider the complete FEN of the starting position in chess:

(rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -)

The above FEN has the following components after the position of the chess board:

- 'w': This character indicates the color to move, white. If the character is 'b', then black is to move.
- 'KQkq': This four character sequence indicates castling rights. 'K' is for white castling kingside, 'Q' is for white castling queenside, and the same for black's castling rights. To indicate the loss of castling rights, the associated character is replaced with '-'.
- '-': The last character(s) indicates the (en passant) target square. If the last move was a pawn moving two squares forward, then the '-' is replaced by the algebraic notation of the square that the pawn passed over.

For more information on FEN, we recommend visiting the following resource [4].

4.4 GUI

We used *Pygame* to design our GUI. See the following YouTube playlist for a tutorial on the *Pygame* Python module [9]. For the correct identification of possible moves, the GUI must include color to move, any *en passant* target square if applicable, and the castling rights for the position. Our design requires the FEN of the position for which we wish to display to our chess expert. We input the FEN as a .txt file and document the moves identified by the chess expert also as a .txt file.

Simulating playing a game of chess on a chess engine, the chess expert can move pieces to each possible square they can legally move to, writing the move to the output text file when the user clicks the 'Submit Move' button. After submitting a move, the GUI returns the piece to its original position so that the chess expert can continue submitting other moves. Once all possible moves have been identified, the user clicks the 'Finish' button, completing the desired output for their given test case. See Figure 4.1 for our GUI design. Finally, we compare the output text file derived from the chess expert's interaction to the output of our move generation functions, testing for correctness and completeness.

Figure 4.1: GUI for testing via crowdsourcing.

By designing an engaging, user-friendly GUI, we hope to encourage chess experts to assist identifying all legal moves from a given position. Google's *The ESP Game* proves that crowdsourcing can be effective when participants enjoy partaking. The GUI provides an accessible, interactive way for all chess experts to participate in contributing test cases.

Chapter 5

Conclusion

The majority of online resources that explain move generation are either locked behind paywalls or assume the reader will be able to follow along with sparse and unclear explanations of complex topics. Moreover, pseudo-legal move generation dominates these resources, which forces search functions to unmake illegal moves made that do not consider king safety. Making and unmaking moves greatly adds to the time complexity of these searches. Therefore, we detail move generation in an accessible manner that strictly identifies all possible legal moves.

Move generation functions must be efficient and accurate to serve as a strong foundation for a chess engine. Bitboards provide a desirable balance between efficiency and comprehensibility, owing in large part to their ease of visualization, making them an ideal selection for board representation. With multiple bitboards, we can store the entire state of a chessboard using only sequences of 0s and 1s. We can manipulate these bits via bit shifting, masking, and bitwise mathematical operations to quickly generate possible moves. Bitboards can also account for special moving cases like castling and *en passant*.

Inspired by Google's *The ESP Game* [5], we propose utilizing crowdsourcing to test the accuracy of our move generation functions. An interactive GUI encourages chess experts to identify all possible moves from a given chess position. We can then compare the output of our move generation functions to the provided test cases. In this manner, generating test cases becomes a fun community effort.

When only looking at final bitwise operations for each piece's move generation function, it can be extremely difficult to understand how these seemingly arbitrary steps correctly

identify each of their possible moves. Using numerous visualizations of bitboards, we show how bitboard manipulations are used to generate possible moves. Moreover, we provide an intuitive, straightforward testing framework whose GUI can be used as the basis for a fully functioning chess engine. We hope that through our work, move generation using bitboards morphs from being a nearly impossible concept to understand to one that introductory computer science students can grasp.

5.1 Future Work

Move generation is the first step in building a chess engine. With efficient and accurate move generation functions, a well-designed search function can identify the strongest move to play. Due to time constraints, our testing of the move generation functions was limited. Though we tested extensively along the way, we were only able to test our completed results against approximately 30 test cases. Some of these test cases can be found in Appendix B. We manually compared our output with these provided test cases. With more time, we wish to automate testing so that we may run hundreds of test cases.

In the future, we hope to build upon our crowdsourcing testing methods. Though the GUI mimics moving pieces as if one were playing a game of chess, we hope to make it a bit more engaging. We could, perhaps, add in more competitive elements. By challenging users to identify as many legal moves possible within a set time limit, we would divide the work for each test case amongst chess experts and give inputting moves another game-like quality.

We hope to build an entire chess engine utilizing the move generation techniques described in this work. We would like to explore search techniques, like the alpha-beta algorithm, or use advanced neural networks, to best select the next move for a computer to play. We could explore some of the state of the art chess engines like Stockfish and Alpha Zero, and elucidate on the techniques used for computer play.

In addition, we would also like to create a sophisticated GUI similar to popular chess engines online. We wish to include a game log, where past moves made within the game are documented. Then, users could review their games and analyze which particular moves greatly affected the game's outcome. Additionally, we would need to more option buttons for undoing

moves and beginning a new game. Some users may also appreciate a timer that mimics a chess clock. With these features, our chess engine could rival the experience of playing against popular chess engines.

Ultimately, we hope this thesis serves as the framework for a future undergraduate course designed around students building their own chess engine. Since chess engines utilize many different fundamental topics covered in introductory computer science courses, building a chess engine could make for a remarkably engaging project-based course. We hope to design weekly labs where we guide students through each step of building their own chess engine.

Bibliography

[1] Chess programming wiki. https://www.chessprogramming.org/Main_Page. Accessed: 2023-05-01.

[2] Chess.com. https://www.chess.com. Accessed: 2023-05-01.

[3] Chess.com lessons. https://www.chess.com/lessons. Accessed: 2023-05-01.

[4] Forsyth-edwards notation chess programming wiki. https://www.chessprogramming.org/Forsyth-Edwards_Notationt, journal=ChessProgramming WIKI. Accessed: 2023-05-01.

[5] *Human Computation*. Google TechTalks. Accessed: 2023-05-01.

[6] Kasparov vs. deep blue: The rematch. https://researcher.watson.ibm.com/researcher/view_group.php?id=2942. Accessed: 2023-05-01.

[7] Perft chess programming wiki. https://www.chessprogramming.org/Perft. Accessed: 2023-05-01.

[8] *PyGame with Python 3 Game Development*. Logic Crazy Chess. Accessed: 2023-05-01.

[9] Pygame with python 3 game development. https://youtube.com/playlist?list=PLQVvvaa0QuDdLkP8MrOXLe_rKuf6r80KO. Accessed: 2023-05-01.

[10] Stockfish chess engine explains most famous chess game. https://www.youtube.com/watch?v=lP36_hLMHIc. Accessed: 2023-05-01.

[11] Doug Barlow. What is the strongest chess engine? https://ocfchess.org/what-is-the-strongest-chess-engine/. Accessed: 2023-05-01.

[12] BBC. Google's 'superhuman' deepmind ai claims chess crown. https://www.bbc.com/news/technology-42251535. Accessed: 2023-05-01.

[13] F.B. Jordan. *How to Write a Bitboard Chess Engine: How Chess Programs Work*. Amazon Digital Services LLC - Kdp, 2020.

[14] David Levy. *Computer Chess Compendium*. Springer New York, 2013.

[15] Andrew Tridgell. Bit board bonkers??, Aug 1997.

Appendices

Appendix A

Bitboard Masks

```
knightMask = 43234889994
kingMask = 460039
fileA = 9259542123273814144
fileH = 72340172838076673
fileAB = 13889313184910721216
fileGH = 217020518514230019
rank1 = 255
rank4 = 4278190080
rank5 = 1095216660480
rank8 = 18374686479671623680
boardMask = 0xffffffffffffffff
rankMasks = [0xff, 0xff00, 0xff0000,
                   0xff000000, 0xff00000000, 0xff0000000000,
                   0xff000000000000, 0xff00000000000000]
fileMasks = [0x8080808080808080, 0x4040404040404040,
                   0x2020202020202020, 0x1010101010101010,
                   0x0808080808080808, 0x0404040404040404,
                   0x0202020202020202, 0x0101010101010101]
diagonalMasks = [0x1, 0x102, 0x10204, 0x1020408, 0x102040810,
                   0x10204081020, 0x1020408102040,
                       0x102040810204080, 0x204081020408000,
                       0x408102040800000, 0x810204080000000,
                       0x1020408000000000, 0x2040800000000000,
                       0x4080000000000000, 0x8000000000000000]
antiDiagonalMasks = [0x80, 0x8040, 0x804020, 0x80402010, 0x8040201008,
                       0x804020100804, 0x80402010080402,
                        0x8040201008040201, 0x4020100804020100,
                        0x2010080402010000, 0x1008040201000000,
                        0x804020100000000, 0x402010000000000,
                        0x201000000000000, 0x100000000000000]
```

Appendix B

Test Cases

See Figures B.1 through B.26 for example test cases and their associated FEN.

Figure B.1: rnbqkbn-r/ppppp2p/5p2/6pQ/4P3/3P4/PPP2PPP/RNB1KBNR b KQkq -

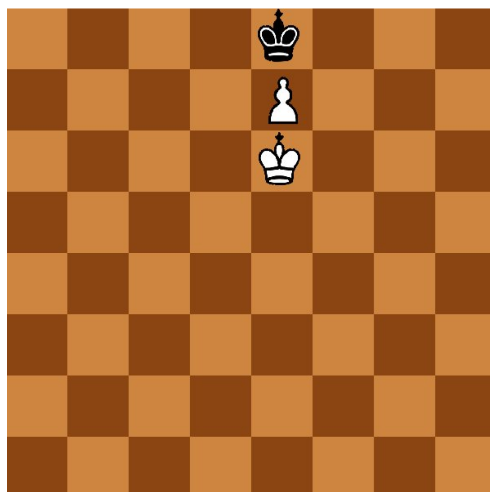


Figure B.2: 4k3/4P3/4K3/8/8/8/8/8 b - -

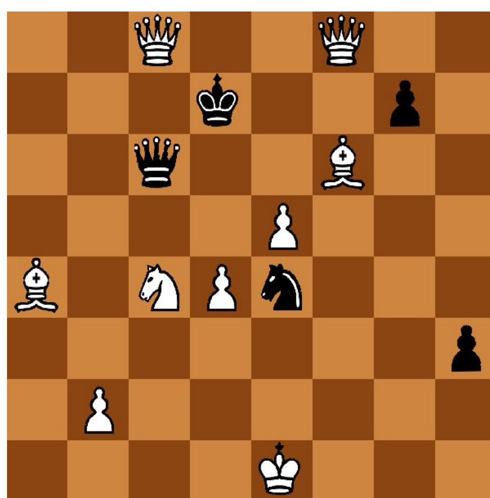


Figure B.3: 2Q2Q2/3k2p1/2q2B2/4P3/
B1NPn3/7p/1P6/4K3 b - -

Figure B.4: r1q2rk1/pp1bNppp/n1p5/8/
3PP3/2N3P1/PPP2PBP/R1BQ1RK1 b - -

Figure B.5: 1krq3r/pp3Npp/1b3n2/8/8/
2N3B1/PPP2PPP/R4RK1 b - -

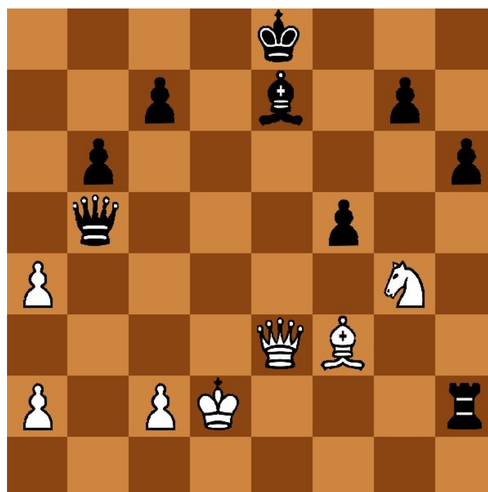


Figure B.6: 4k3/2p1b1p1/1p5p/1q3p2/P5N1/4QB2/P1PK3r/8 w - -

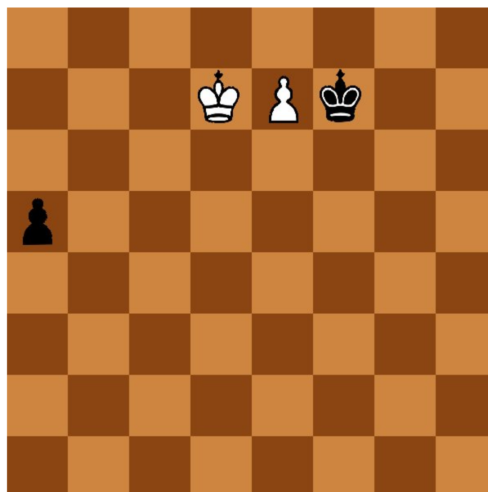


Figure B.7: 8/3KPk2/8/p7/8/8/8/8 w - -

Figure B.8: r1bqk2r/pppp1ppp/2n2n2/2b1p3/2B1P3/2N2N2/PPPP1PPP/R1BQK2R w KQkq -

Figure B.9: r2qk2r/1pp2ppp/p1npbn2/2b1p1B1/2B1P3/2NP1N2/PPPQ1PPP/R3K2R w KQkq -

Figure B.10: r1bqkb1r/ppp2ppp/2n1pn2/3p2B1/3P4/2N5/PPPQPPPP/R3KBNR w KQkq -

Figure B.11: rnbqkbn-r/ppp2ppp/4p3/3pP3/3P4/8/PPP2PPP/RNBQKBNR w KQkq d6

Figure B.12: 5k2/3r2pp/5p2/1b6/3NP3/
1B4P1/7P/R2K4 w - -

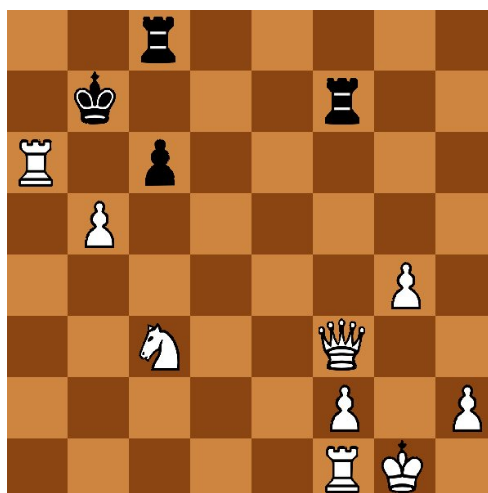


Figure B.13: 2r5/1k3r2/R1p5/1P6/6P1/
2N2Q2/5P1P/5RK1 b - -

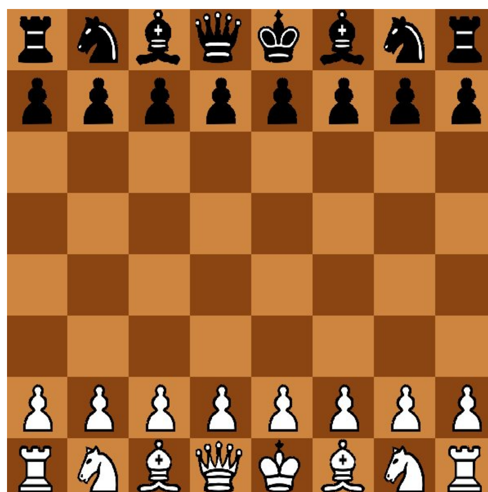


Figure B.14: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -

Figure B.15: r1b2rk1/1pq1bppp/p1nppn2/8/3NP3/1BN1B3/PPP1QPPP/2KR3R w - -

Figure B.16: rn2k1nr/ppp2ppp/3b4/3N4/
3PPpbq/5N2/PPP1K1PP/R1BQ1B1R b - -

Figure B.17: r1b1k2r/pp2n1pp/2n1pp2/
q1ppP3/P2P4/2P2N2/2PQ1PPP/R1B1KB1R w
- -

Figure B.18: r1br2k1/pp1n1pp1/2q1n2p/7Q/2P5/1N1BR3/PB3PPP/R5K1 b - -

Figure B.19: r2q1rk1/ppp2ppp/2np1n2/2b1p3/2P3b1/2NP1NP1/PP2PPBP/R1BQ1RK1 w - -

Figure B.20: rn1q1rk1/pbp1bppp/1p2p3/3pN3/2PPn3/2N3P1/PP1BPPBP/R2Q1RK1 b - -

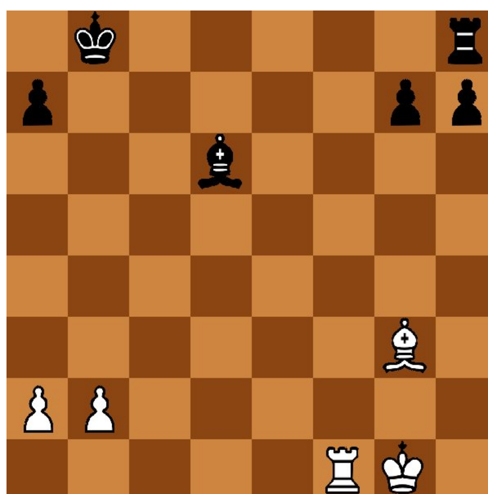


Figure B.21: 1k5r/p5pp/3b4/8/8/6B1/PP6/5RK1 b - -

Figure B.22: 1k5r/p5pp/3q4/8/8/6B1/PP3PP1/5RK1 b - -

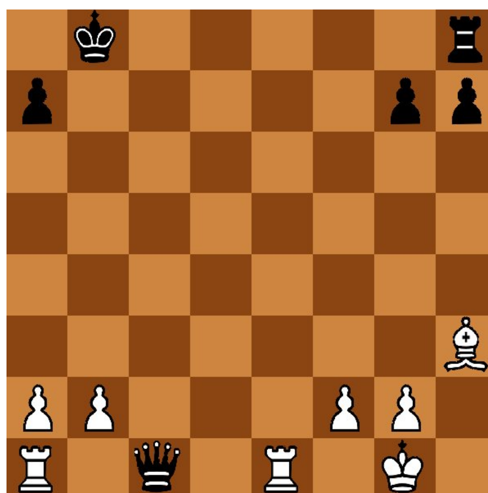


Figure B.23: 1k5r/p5pp/8/8/8/7B/PP3PP1/R1q1R1K1 w - -

Figure B.24: rnbqkbn-
r/ppp2ppp/8/3p4/3Pp3/4P3/PPP2PPP/RNBQKBNR
b KQkq d3

Figure B.25: r1q2rk1/p1p1bppp/bpnp1n2/
6B1/4P3/1BN2N2/PPP2PPP/R2QK2R w KQ -

Figure B.26: r2qk2r/ppp2ppp/8/8/Q7/8/
PP3PPP/R4RK1 b - -