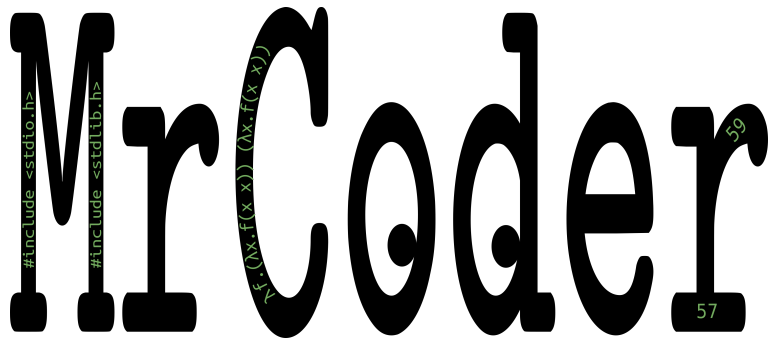
Yolah Board Game

Building a Two-Player Perfect-Information Game with AI Players

Pascal Garcia

December 25, 2025


MrCoder
#include <stdio.h>
#include <stdlib.h>
λf.(λx.f(x x)) (λx.f(x x))
59
57

C’est en forgeant qu’on devient forgeron

*À Sar**ah**, **H**ug**o** et C**élya** ❤️*
ahholy
aholy
yolah

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Yolah Game

I created the Yolah game to illustrate effective techniques for implementing board games and artificial intelligences for my students. I was inspired by the Pingouins game, whose box you can see in Figure 1.1 (I highly recommend it ☺)

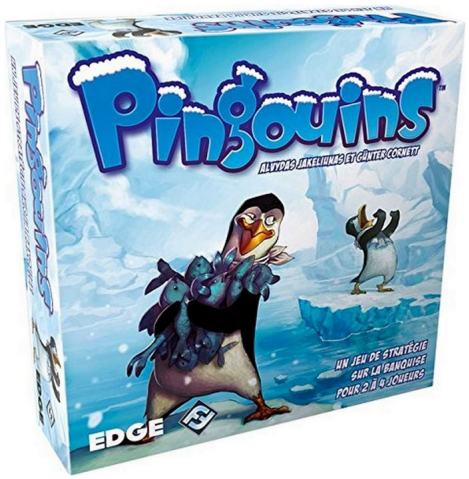


Figure 1.1: The Pingouins game box

Important

I have done my best with my current knowledge (*ars longa, vita brevis*) to implement my game and the associated AIs. But like any good scientist, you should look at my work with a critical eye. I wrote the book in French (easier for me) and asked an AI assistant (Claude [1]) to translate it for me.

I will now describe the rules of the game, then I will explain why I chose these rules, I will give an example of a game between two AIs and then I will present the rest of the book.

1.1.1 Game Rules

The Yolah game board is shown in Figure 1.2. You can see four black pieces and four white pieces placed symmetrically. Black starts by choosing one of their four

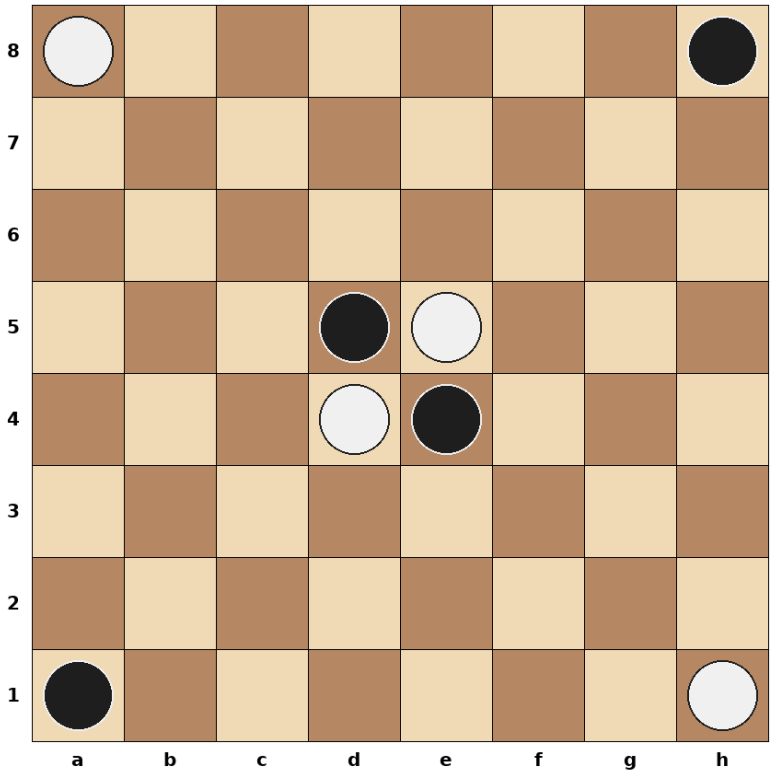


Figure 1.2: The initial configuration of the Yolah game

pieces. A piece can never disappear from the board because Yolah is a game without captures. A piece moves in all eight directions as far as it wishes as long as it is not blocked by another piece or a hole (a concept we will soon discuss). For example, if black chooses to move their piece located at `d5`, the squares where it can land are indicated by small black crosses in Figure 1.3.

Now, if the black piece at `d5` moves to `b7`, which we will denote as `d5:b7`, we get the configuration shown in Figure 1.4. Notice that the starting square of the black piece disappears and becomes a hole! This square (this hole) becomes inaccessible and impassable for the rest of the game! This will create opportunities to block the opponent and try to create areas where the opponent cannot go.

A move earns one point for the player who just moved. For example, in the configuration of Figure 1.4, the black player has one point and the white player who has not yet moved has zero points. The goal of the game is quite simple to summarize: you must move longer than your opponent!

Now it is white's turn to play. They must decide which white piece they will move. Suppose it is the piece at `e5`. The possible moves for this white piece are shown in Figure 1.5. If white decides to make the move `e5:f5`, we end up in the configuration of Figure 1.6 and the score is one point each (each player has played one move).

To summarize, the rules of Yolah are as follows:

- The game is a two-player game (black and white) played in turns.

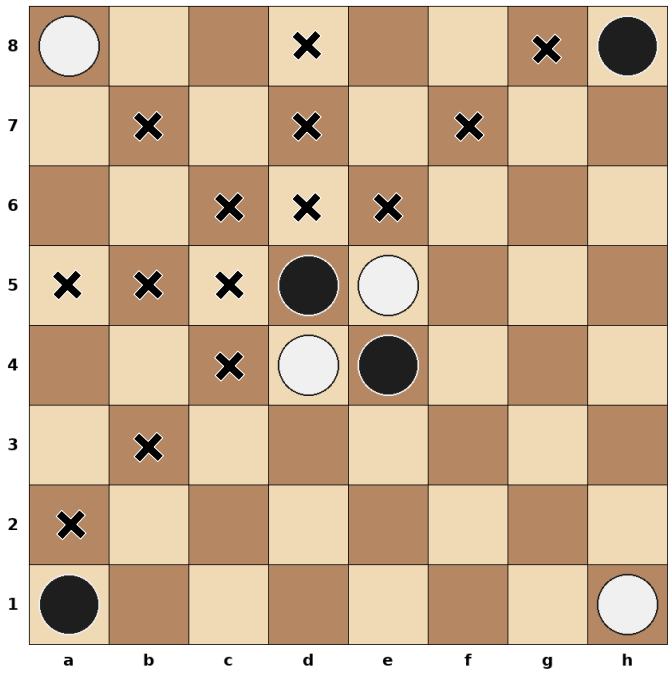


Figure 1.3: Possible moves (small black crosses) for the black piece located on square `d5`

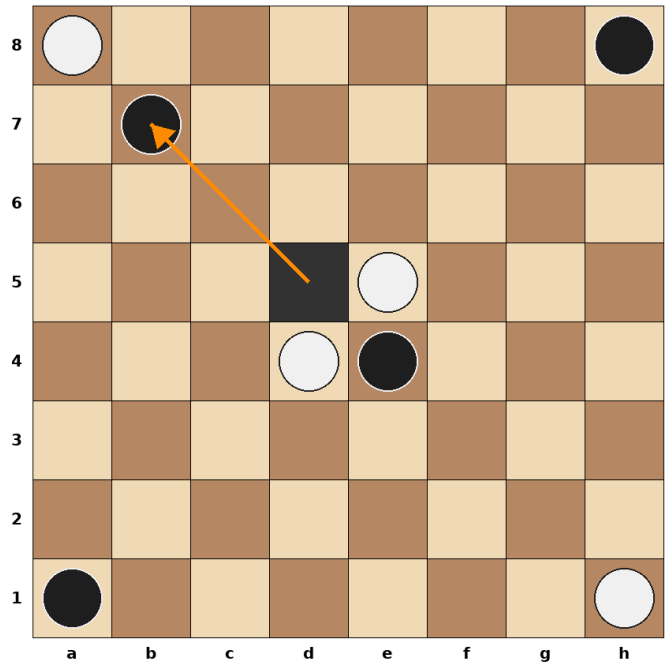


Figure 1.4: Black just moved from `d5` to `b7`. The starting square `d5` becomes inaccessible and impassable for the rest of the game

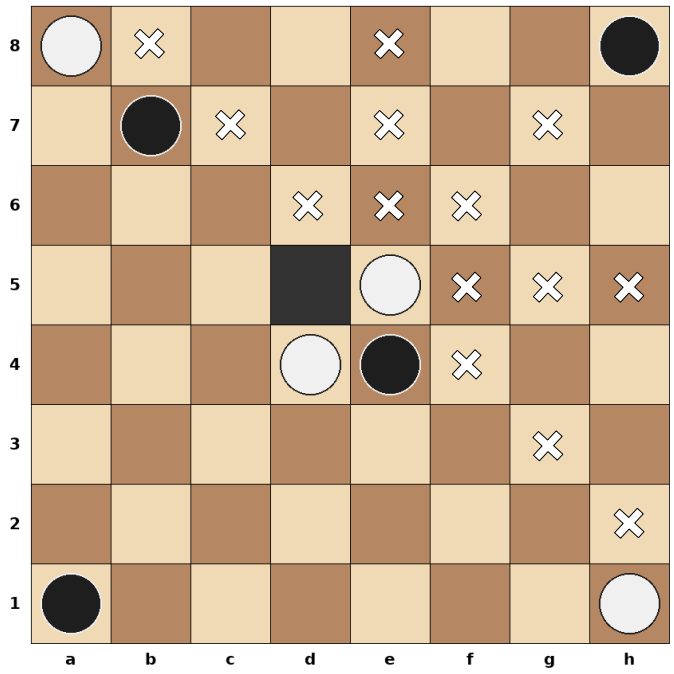


Figure 1.5: Possible moves (small white crosses) for the white piece located on square `e5`

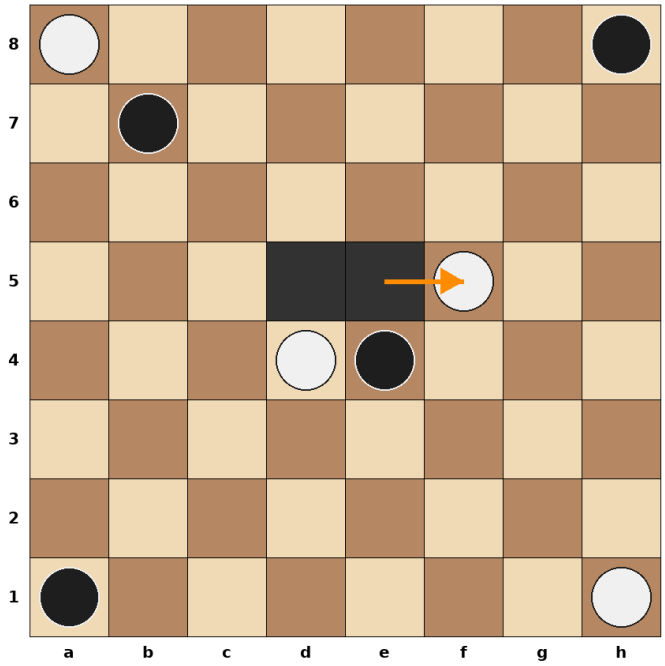


Figure 1.6: White just moved from `e5` to `f5`. The starting square `e5` becomes inaccessible and impassable for the rest of the game. The score is one point each (each player has moved once)

- Each player has four pieces.
- On their turn, the player chooses one of the pieces that can still move; if no piece can move, they pass their turn (we will denote by the move `a1:a1` to skip one's turn).
- They must move the chosen piece in one of the eight directions, as many squares as desired, but must not land on or be blocked by a piece or a hole.
- After moving the chosen piece, the starting square of the move becomes a hole and can no longer be crossed or landed on.
- After each move, the player earns one point.
- The game ends when both players can no longer move.
- The player with the most points wins the game.
- If both players have the same number of points, the game is declared a draw.

1.1.2 Interesting Characteristics of Yolah for Developing AIs

I chose the above rules for Yolah, on the one hand because I liked the dynamics of the Penguin game, but also because there are no cycles in the game, so there is no need for special rules to prevent a game from never ending. The number of moves available to each player is quite large at the beginning (but reasonable)[1], but it will decrease little by little with the appearance of holes[2]. This allows the AIs to look ahead a fairly large number of moves.

I also wanted to be able to reuse concepts used for the efficient implementation of chess, and the size of the board and the movement of the pieces (same way of moving as a Queen in chess) allow me to do that.

1.1.3 Game Example

To get an idea of how a Yolah game unfolds, we will have two artificial intelligences play against each other. The first AI will be based on Monte Carlo Tree Search and the second will be based on Minimax with a neural network. We will study both of these AIs later in the book. The second AI is stronger and you will see its zone isolation strategy in action!

The progression of the game is described in Figures 1.7, 1.8 and 1.9.

The white AI estimates that it is winning starting from move 10 (see Figure 1.7k). We can see at move 30 (see Figure 1.8k) that it has successfully isolated a zone where black can no longer access. At move 32 (see Figure 1.8m) it moves one of its pieces out of the isolated zone because the other piece will be able to collect all the points from

[1] 56 possible moves for the black player at the start of the game.

[2] The number of possible moves does not necessarily decrease after each move; there are configurations where the player has more than 56 available moves.

that zone. It is more useful to use the other piece to gather points elsewhere. Note that starting from move 47 (see Figure 1.9h onward), black has no more available moves and must therefore pass their turn.

The game is won by the white player 32 to 24, which is a very good score because the Yolah game seems to favor black.

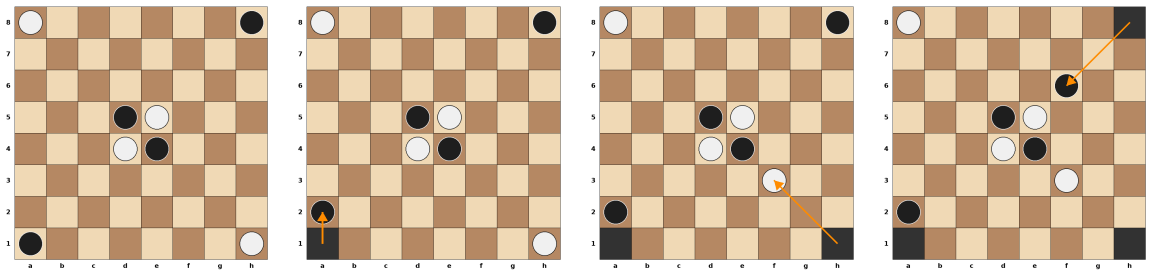
1.2 What's Next

In the next chapter, we will study the implementation of the Yolah game in [2]. This implementation is designed to be efficient because it will be important for the AIs to be able to play many games per second; their level of play will depend on it.

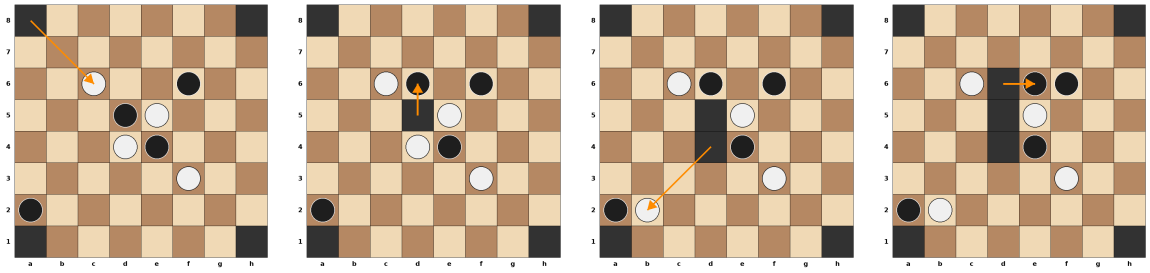
Chapter 3 describes the common interface for our different AIs. Chapter 4 presents a very simple AI based on Monte Carlo search. The next AI, described in Chapter **??**, is an evolution of the previous one and will allow, unlike the Monte Carlo AI, the development of a game tree. Note that these two AIs will not require heuristics provided by humans and only need the rules of the game. Chapter 6 presents an AI based on minimax tree search with heuristics provided by humans. The heuristic used by the AI is a linear combination of the heuristics provided by humans; the weights of each heuristic in this linear combination are learned using a genetic algorithm. Our last and strongest AI is presented in Chapter 7. A neural network is used instead of heuristics. This neural network is trained on a set of games played by the previous AI.

Chapter 8 then evaluates all these AIs by having them compete in a tournament. We will then conclude and propose different directions for creating other artificial players.

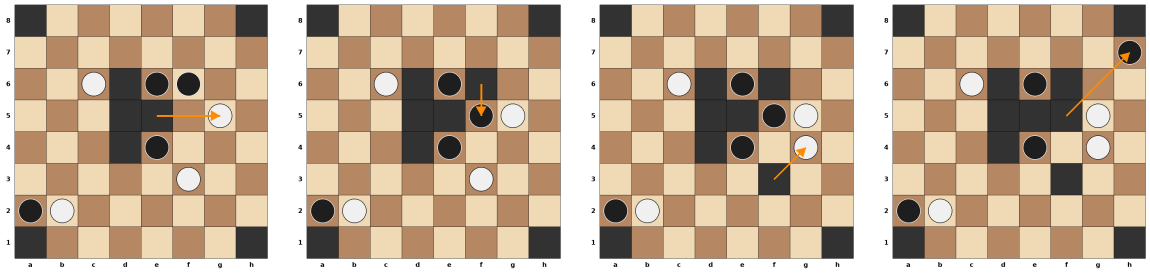
Happy reading!



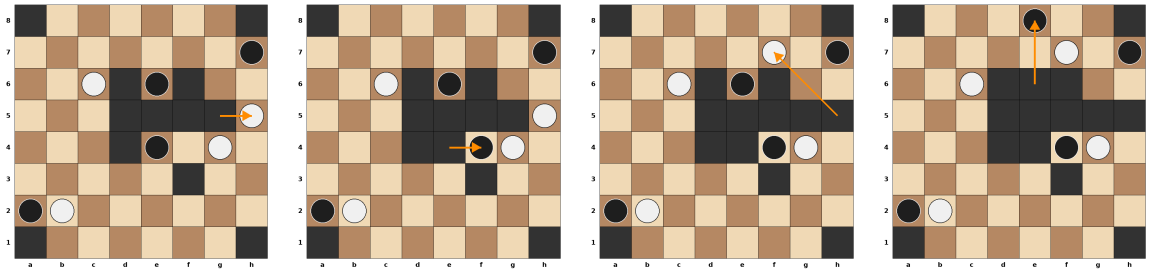
(a) Initial position (b) Move 1 (Black) (c) Move 2 (White) (d) Move 3 (Black)



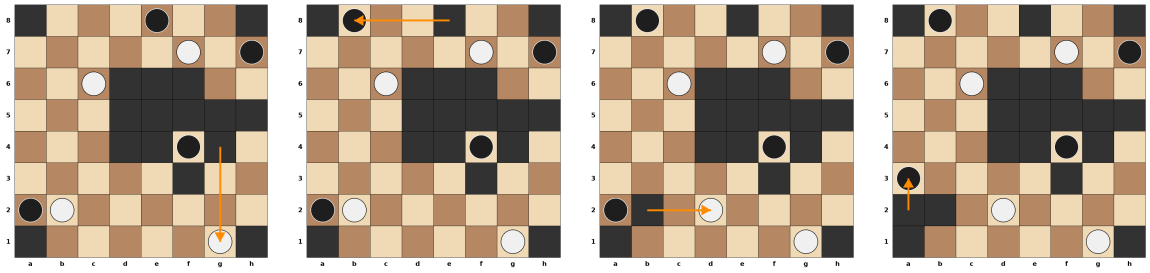
(e) Move 4 (White) (f) Move 5 (Black) (g) Move 6 (White) (h) Move 7 (Black)



(i) Move 8 (White) (j) Move 9 (Black) (k) Move 10 (White) (l) Move 11 (Black)

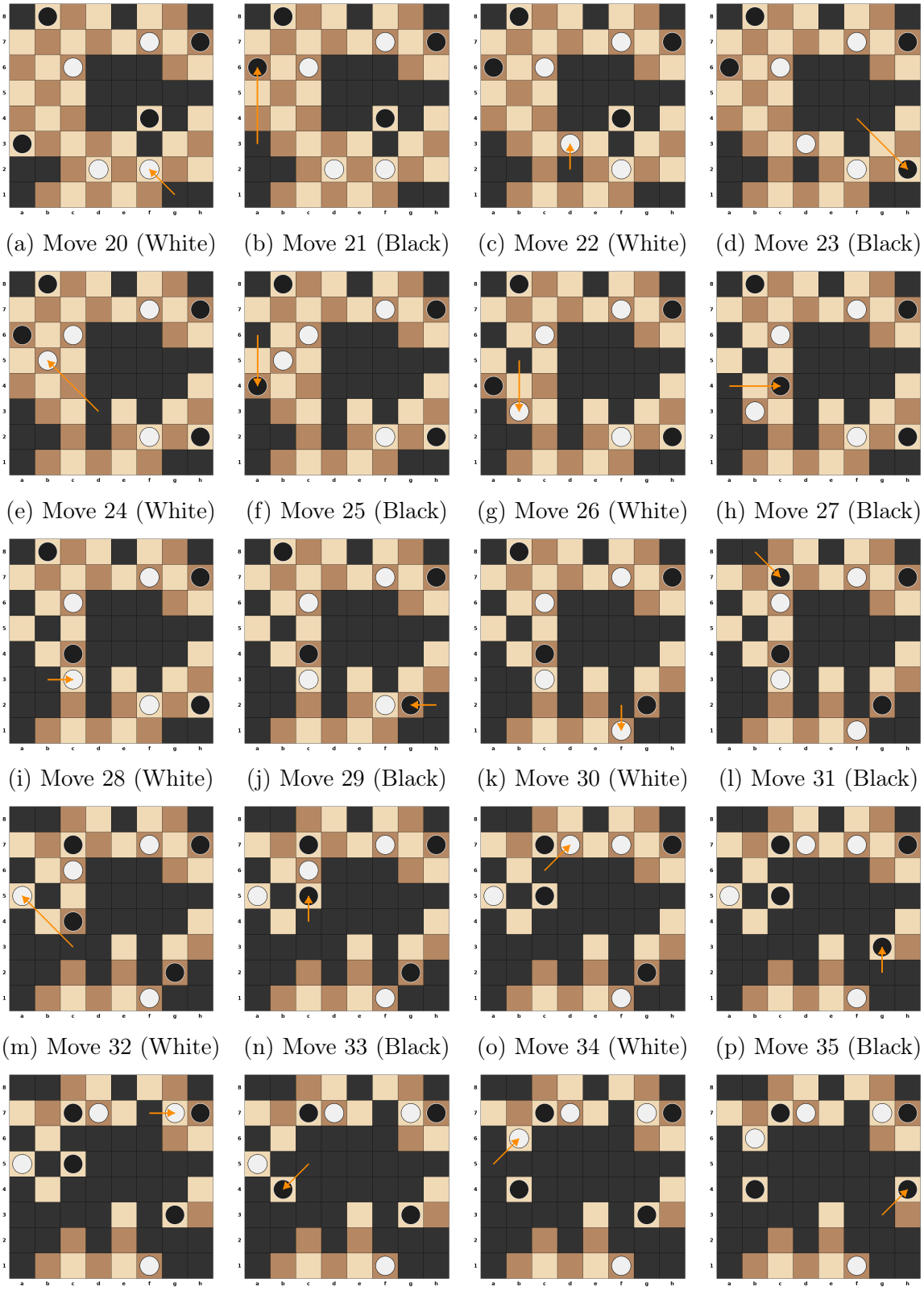


(m) Move 12 (White) (n) Move 13 (Black) (o) Move 14 (White) (p) Move 15 (Black)



(q) Move 16 (White) (r) Move 17 (Black) (s) Move 18 (White) (t) Move 19 (Black)

Figure 1.7: Game example between two AIs - moves 1 to 19



(a) Move 20 (White) (b) Move 21 (Black) (c) Move 22 (White) (d) Move 23 (Black)

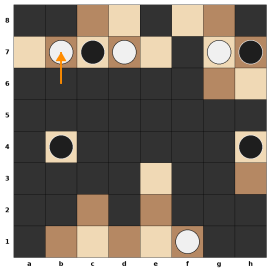
(e) Move 24 (White) (f) Move 25 (Black) (g) Move 26 (White) (h) Move 27 (Black)

(i) Move 28 (White) (j) Move 29 (Black) (k) Move 30 (White) (l) Move 31 (Black)

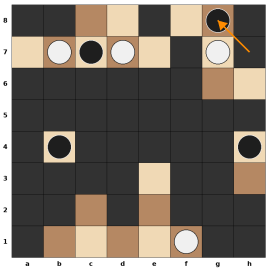
(m) Move 32 (White) (n) Move 33 (Black) (o) Move 34 (White) (p) Move 35 (Black)

(q) Move 36 (White) (r) Move 37 (Black) (s) Move 38 (White) (t) Move 39 (Black)

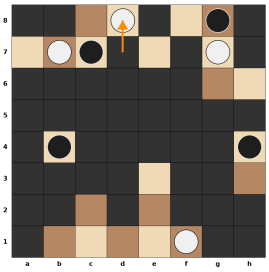
Figure 1.8: Game example between two AIs - moves 20 to 39



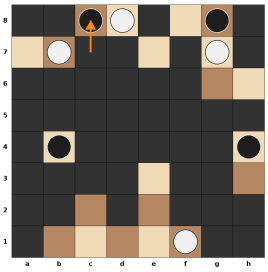
(a) Move 40 (White)



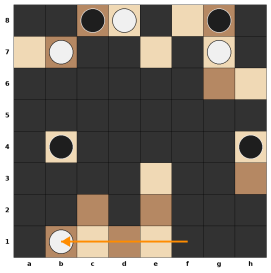
(b) Move 41 (Black)



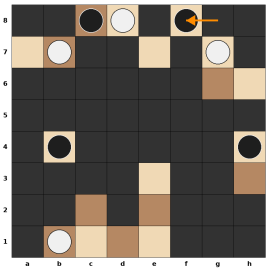
(c) Move 42 (White)



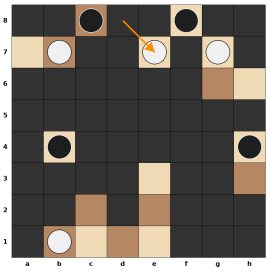
(d) Move 43 (Black)



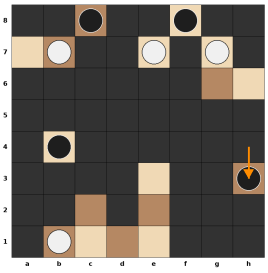
(e) Move 44 (White)



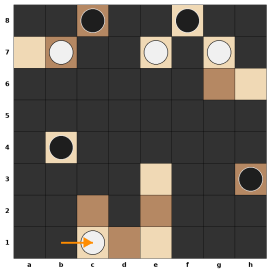
(f) Move 45 (Black)



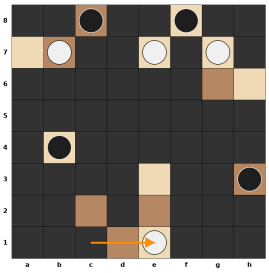
(g) Move 46 (White)



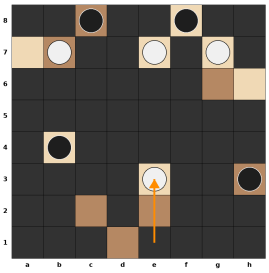
(h) Move 47 (Black)



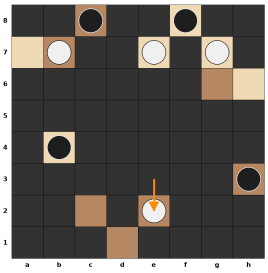
(i) Move 48 (White)



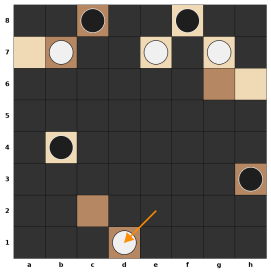
(j) Move 49 (White)



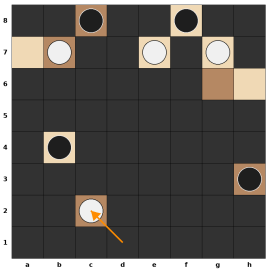
(k) Move 50 (White)



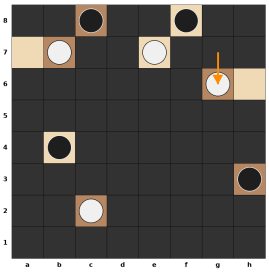
(l) Move 51 (White)



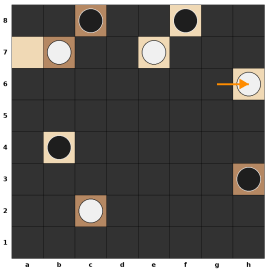
(m) Move 52 (White)



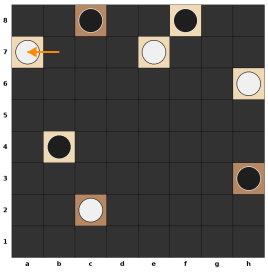
(n) Move 53 (White)



(o) Move 54 (White)



(p) Move 55 (White)



(q) Move 56 (White)

Figure 1.9: Game example between two AIs - moves 40 to 56. White wins 32 to 24

Chapter 2

Game Engine

We will implement game management in C++, striving to create an efficient implementation capable of running as many games per second as possible—this will be crucial for developing high-level AIs. We will test our implementation by generating as many random games as possible within a given time at the end of this chapter. We will draw inspiration from the excellent Stockfish chess engine [3] for Yolah's data structures.

2.1 Data Structures

We will represent the positions of black pieces, white pieces, and destroyed squares using 64-bit unsigned integers: `uint64_t`. We call these integers bitboards. We use `uint64_t` because the Yolah board contains 64 squares, giving us one bit per square. Table 2.1 shows the position of each board square in the bitboard. This information is represented in code by the enumeration in Listing 1[1].

Table 2.1: Position of each board square in the bitboard

8	bit_{56}	bit_{57}	bit_{58}	bit_{59}	bit_{60}	bit_{61}	bit_{62}	bit_{63}
7	bit_{48}	bit_{49}	bit_{50}	bit_{51}	bit_{52}	bit_{53}	bit_{54}	bit_{55}
6	bit_{40}	bit_{41}	bit_{42}	bit_{43}	bit_{44}	bit_{45}	bit_{46}	bit_{47}
5	bit_{32}	bit_{33}	bit_{34}	bit_{35}	bit_{36}	bit_{37}	bit_{38}	bit_{39}
4	bit_{24}	bit_{25}	bit_{26}	bit_{27}	bit_{28}	bit_{29}	bit_{30}	bit_{31}
3	bit_{16}	bit_{17}	bit_{18}	bit_{19}	bit_{20}	bit_{21}	bit_{22}	bit_{23}
2	bit_{8}	bit_{9}	bit_{10}	bit_{11}	bit_{12}	bit_{13}	bit_{14}	bit_{15}
1	bit_{0}	bit_{1}	bit_{2}	bit_{3}	bit_{4}	bit_{5}	bit_{6}	bit_{7}
	a	b	c	d	e	f	g	h

```
1 enum Square : int8_t {
2     SQ_A1, SQ_B1, SQ_C1, SQ_D1, SQ_E1, SQ_F1, SQ_G1, SQ_H1,
3     SQ_A2, SQ_B2, SQ_C2, SQ_D2, SQ_E2, SQ_F2, SQ_G2, SQ_H2,
```

[1] By default, the enumeration starts at value 0, so `SQ_A1` equals 0, `SQ_B1` equals 1, ...

```
4       SQ_A3, SQ_B3, SQ_C3, SQ_D3, SQ_E3, SQ_F3, SQ_G3, SQ_H3,
5       SQ_A4, SQ_B4, SQ_C4, SQ_D4, SQ_E4, SQ_F4, SQ_G4, SQ_H4,
6       SQ_A5, SQ_B5, SQ_C5, SQ_D5, SQ_E5, SQ_F5, SQ_G5, SQ_H5,
7       SQ_A6, SQ_B6, SQ_C6, SQ_D6, SQ_E6, SQ_F6, SQ_G6, SQ_H6,
8       SQ_A7, SQ_B7, SQ_C7, SQ_D7, SQ_E7, SQ_F7, SQ_G7, SQ_H7,
9       SQ_A8, SQ_B8, SQ_C8, SQ_D8, SQ_E8, SQ_F8, SQ_G8, SQ_H8,
10      SQ_NONE,
11      SQUARE_ZERO = 0,
12      SQUARE_NB   = 64
13  };
```

Listing 1: Board squares

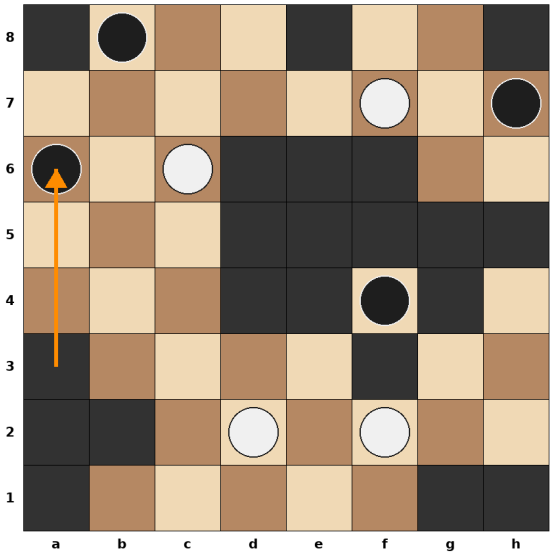


Figure 2.1: Board configuration corresponding to move 21 of the game given as an example in the previous chapter (Figure 1.8b)

Let us take as an example the board shown in Figure 2.1.

- The positions of black pieces on the board (Table 2.2) are represented by the bitboard: `0b0000001010000000000000010000000001000000000000000000000000000000`.

- The positions of white pieces on the board (Table 2.3) are represented by the bitboard: `0b0000000000010000000000100000000000000000000000000001010000000000`.

- The positions of holes (Table 2.4) are represented by the bitboard: `0b1001000100000000001110001111100001011000001000010000001111000001`.

Table 2.2: Black piece positions

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

Table 2.3: White piece positions

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

Table 2.4: Destroyed square positions (holes)

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

Bitboards allow us to efficiently manipulate the board using bitwise operations while requiring minimal memory to represent it.

We represent the game board with the `Yolah` class, an excerpt of which is given in Listing 2.

```
1  constexpr uint64_t BLACK_INITIAL_POSITION =
2  0b10000000'00000000'00000000'00001000'00010000'00000000'00000000'00000001;
3  constexpr uint64_t WHITE_INITIAL_POSITION =
4  0b00000001'00000000'00000000'00010000'00001000'00000000'00000000'10000000;
5  class Yolah {
6      uint64_t black = BLACK_INITIAL_POSITION;
7      uint64_t white = WHITE_INITIAL_POSITION;
8      uint64_t holes = 0;
9      uint8_t black_score = 0;
10     uint8_t white_score = 0;
11     uint8_t ply = 0;
12 public:
13     // ...
14 };
```

Listing 2: Attributes of the Yolah class representing the game board

The attributes of the `Yolah` class are:

- Line 6, `black`: the bitboard for black pieces.
- Line 7, `white`: the bitboard for white pieces.
- Line 8, `holes`: the bitboard for holes (destroyed squares).
- Line 9, `black_score`: the score, or number of moves, of the black player.
- Line 10, `white_score`: the score, or number of moves, of the white player.
- Line 11, `ply`: the number of moves played by both players since the start of the game.

A `Yolah` object will occupy `sizeof(Yolah) == 32` bytes in memory. Note that due to padding[2], it would not have been wise to write, for example,

```
1  class Yolah {
2      uint64_t black = BLACK_INITIAL_POSITION;
3      uint8_t black_score = 0;
```

[2] https://en.wikipedia.org/wiki/Data_structure_alignment.

```
4       uint64_t white = WHITE_INITIAL_POSITION;
5       uint8_t white_score = 0;
6       uint64_t holes = 0;
7       uint8_t ply = 0;
8   public:
9       // ...
10  };
```

because with this implementation, we would have `sizeof(Yolah) == 48` bytes.

The bitboard representation allows for efficiently obtaining game information. For example, to see the squares occupied by black and white pieces, simply perform: `black | white`. For the positions in Tables 2.2 and 2.3, we would obtain in a single highly efficient operation[3] the black and white piece positions shown in Table 2.5. In binary notation, we get:

```
black | white
== 0000001010000000000000010000000000100000000000000000000000000000 |
   0000000000100000000001000000000000000000000000000010100000000000
== 0000001010100000000001010000000001000000000000000010100000000000
```

Table 2.5: Black and white piece positions obtained by computing: `black | white`

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

We will continue using bitwise operations throughout this chapter, particularly for testing game over conditions and generating possible moves.

2.2 Game Over Test

To test for game over, we will need some enumerations and constants (Listings 3 and 5) and the `shift` function (Listing 4). The `NORTH` constant in the `Direction` enumeration (Listing 3) equals 8. Why this value? Let us revisit below, see Table 2.6, the table from the previous chapter showing the position of each bit in a bitboard on the board.

[3] You can find information on instruction latency and throughput for various processors at: https://agner.org/optimize/instruction_tables.pdf

We can see that adding 8 to the bit number in any square gives us the bit number of the square to the north (unless we exit the board). The same applies to the other constant values in the `Direction` enumeration (Listing 3).

Table 2.6: Position of each board square in the bitboard. Note that for a square not in rank 8, bit_{i+8} corresponds to the square north of bit_i

8	bit_{56}	bit_{57}	bit_{58}	bit_{59}	bit_{60}	bit_{61}	bit_{62}	bit_{63}
7	bit_{48}	bit_{49}	bit_{50}	bit_{51}	bit_{52}	bit_{53}	bit_{54}	bit_{55}
6	bit_{40}	bit_{41}	bit_{42}	bit_{43}	bit_{44}	bit_{45}	bit_{46}	bit_{47}
5	bit_{32}	bit_{33}	bit_{34}	bit_{35}	bit_{36}	bit_{37}	bit_{38}	bit_{39}
4	bit_{24}	bit_{25}	bit_{26}	bit_{27}	bit_{28}	bit_{29}	bit_{30}	bit_{31}
3	bit_{16}	bit_{17}	bit_{18}	bit_{19}	bit_{20}	bit_{21}	bit_{22}	bit_{23}
2	bit_{8}	bit_{9}	bit_{10}	bit_{11}	bit_{12}	bit_{13}	bit_{14}	bit_{15}
1	bit_{0}	bit_{1}	bit_{2}	bit_{3}	bit_{4}	bit_{5}	bit_{6}	bit_{7}
	a	b	c	d	e	f	g	h

```
enum Direction : int8_t {
    NORTH = 8,
    EAST  = 1,
    SOUTH = -NORTH,
    WEST  = -EAST,
    NORTH_EAST = NORTH + EAST,
    SOUTH_EAST = SOUTH + EAST,
    SOUTH_WEST = SOUTH + WEST,
    NORTH_WEST = NORTH + WEST
};
```

Listing 3: Directions

```
template<Direction D>
constexpr uint64_t shift(uint64_t b) {
    if constexpr (D == NORTH)
        return b << NORTH;
    else if constexpr (D == SOUTH)
        return b >> -SOUTH;
    else if constexpr (D == EAST)
        return (b & ~FileHBB) << EAST;
    else if constexpr (D == WEST)
        return (b & ~FileABB) >> -WEST;
    else if constexpr (D == NORTH_EAST)
```

```
12          return (b & ~FileHBB) << NORTH_EAST;
13      else if constexpr (D == NORTH_WEST)
14          return (b & ~FileABB) << NORTH_WEST;
15      else if constexpr (D == SOUTH_EAST)
16          return (b & ~FileHBB) >> -SOUTH_EAST;
17      else if constexpr (D == SOUTH_WEST)
18          return (b & ~FileABB) >> -SOUTH_WEST;
19      else return 0;
20  }
```

Listing 4: Shift by direction

The `shift(uint64_t b)` function given in Listing 4 shifts all 1-bits in the bitboard parameter in direction `D`[4]. Given

`black` == 1000000000000000000000000000010000010000000000000000000000000001

the bitboard shown in Table 2.7. To shift the black pieces one square north, we perform: `shift<NORTH>(black)`. We then get the bitboard

1000000000000000000000000000010000010000000000000000000000000001

shown in Table 2.8. Note that the piece on `h8` is no longer on the board.

Table 2.7: Game board before applying `shift<NORTH>`

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

Table 2.8: Game board after applying `shift<NORTH>`

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

We can thus easily obtain all squares directly adjacent to the pieces on a given `board` by performing the following operation:

```
1 shift<NORTH>(board) | shift<SOUTH>(board) | shift<EAST>(board) |
2 shift<WEST>(board) | shift<NORTH_EAST>(board) | shift<NORTH_WEST>(board) |
3 shift<SOUTH_EAST>(board) | shift<SOUTH_WEST>(board)
```

[4]Thanks to `if constexpr`, when we call `shift<NORTH>(b)` for example, the compiler will transform the code to: `return b << NORTH;`.

For the board in Table 2.9, we get the positions represented by stars in Table 2.10.

Table 2.9: Game board

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

Table 2.10: Positions around the pieces

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

In the `shift` function code (Listing 4), we can see the use of constants `FileHBB` and `FileABB` (`BB` for bitboard). These constants allow us to mask certain bits that would end up in incorrect positions after shifting. For example, if we shift the bitboard shown in Table 2.9 in the `EAST` direction, the white pawn on `h1` would end up on `a2`. To avoid this, before shifting, we eliminate elements from column `h` so they don't end up in column `a` (line 8 of Listing 4). The constants `FileABB` and `FileHBB` along with the rank and file enumerations are defined in Listing 5.

```
1  enum File : uint8_t {
2      FILE_A, FILE_B, FILE_C, FILE_D, FILE_E, FILE_F, FILE_G, FILE_H,
3      FILE_NB
4  };
5  enum Rank : uint8_t {
6      RANK_1, RANK_2, RANK_3, RANK_4, RANK_5, RANK_6, RANK_7, RANK_8,
7      RANK_NB
8  };
9  constexpr uint64_t FileABB = 0x0101010101010101;
10 //   0x0101010101010101
11 //== 0b0000000100000001000000010000000100000001000000010000000100000001
12 //             a8       a7       a6       a5       a4       a3       a2       a1
13
14 constexpr uint64_t FileHBB = FileABB << 7;
15 //   0x8080808080808080
16 //== 0b1000000010000000100000001000000010000000100000001000000010000000
17 //     h8       h7       h6       h5       h4       h3       h2       h1
```

Listing 5: Files and ranks

The function that tests for game over is called `game_over` and is described in Listing 6. Its implementation is straightforward:

- On line 2, we retrieve in `possible` the bitboard with 1s in free positions.
- On line 3, we create the `players` bitboard containing 1s in positions occupied by either player.
- Lines 4 to 8 create the `around_players` bitboard containing 1s at positions around each player's pieces.
- Finally, on line 9, we test whether no free position exists adjacent to either player. The bitwise AND will keep a 1 in a result bit if and only if there is a 1 at that position in both the `possible` and `around_players` bitboards—meaning the corresponding square is both free and accessible by a player. If `around_players & possible` equals 0, it means no position is both adjacent to a player and free.

```
1  bool Yolah::game_over() const {
2      uint64_t possible = ~holes & ~black & ~white;
3      uint64_t players  = black | white;
4      uint64_t around_players = shift<NORTH>(players) |
5          shift<SOUTH>(players) | shift<EAST>(players) |
6          shift<WEST>(players) | shift<NORTH_EAST>(players) |
7          shift<NORTH_WEST>(players) | shift<SOUTH_EAST>(players) |
8          shift<SOUTH_WEST>(players);
9      return (around_players & possible) == 0;
10 }
```

Listing 6: Game over test

Now that we know how to test for game over, we will study how to efficiently generate possible moves in a given position.

2.3 Generating Possible Piece Moves

Move generation will use a technique called *magic bitboards*, which is a perfect hash function that maps blocker occupancy patterns to unique lookup table indices without collisions (we will see all this in detail later in this chapter). To understand this technique, I will first present perfect hashing through a simple example. Throughout this chapter, I will use the terms *perfect hashing* and *magic bitboards* interchangeably, as they refer to the same technique in this context. I drew inspiration from [4] for this example.

2.3.1 Magic Perfect Hashing Function

Suppose we need to find the position of the only 1-bit in an unsigned long integer (`uint64_t`). For example, if we call `position` the function that calculates this position, we get the following results:

```
position(1)           == 0
position(0b1000)      == 3
position(0b10000000)  == 7
position(0x8000000000000000) == 63
```

Note that processors have instructions to efficiently compute this function[5], and the `gcc` compiler allows efficient calculation of this position with the function `__builtin_ctzll(unsigned long long x)`[6], for example,

```
position(0b1000) == __builtin_ctzll(0b1000)
```

However, we will code this `position` function using a magic perfect hashing function. This function will be useful for move generation later. Using a standard hash table, we could first fill it with 64 values and then simply look up this table, as in the following code:

```
unordered_map<uint64_t, uint8_t> table {
    {1, 0}, {0b10, 1}, {0b100, 2}, {0b1000, 3}, //...
}
int position(uint64_t x) {
    return table[x];
}
```

Looking up values in this hash table is far more costly than indexing a simple array[7]. However, we cannot directly use an array because the indexing values span too large a range—for example, the value `0x8000000000000000` ($9,223,372,036,854,775,808$!). The idea is to find an efficient hash function that transforms each of the 64 values[8]—which we call bitboards—into the range $[0, 2^6 - 1 = 63]$. Moreover, we want no collisions—this is called perfect hashing. If there were collisions, positions would be incorrect. For example, if keys `0b100` and `0b1000000` were both transformed to index 42, we would need to store value 2 and 6 in `table[42]`, but we cannot store more than one value at the same location.
The magic perfect hashing function will have the following form (listing 7):

```
constexpr uint64_t MAGIC = //...
constexpr int K = 6;
int magic_perfect_hashing(uint64_t bitboard) {
    return bitboard * MAGIC >> (64 - K);
}
```

[5] `tzcnt` on Intel.

[6] Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined.

[7] We will measure the performance gain from magic perfect hashing compared to using a standard hash table at the end of this section.

[8] `1, 0b10, 0b100, 0b1000, 0b10000, ....`

Listing 7: Magic bitboard perfect hashing

In this function:

- On line 2, the constant `K` gives the number of bits for our index. Here with `K == 6`, this gives us a maximum of $2^K = 2^6 = 64$ possible values in the table.
- On line 4, we see the formula for calculating the table index from the `uint64_t key` parameter. We multiply the key by the `MAGIC` constant and then right-shift to keep only the `K` most significant bits of the result. This operation is very inexpensive, but how do we find this magic constant?

A simple method to find the `MAGIC` constant is to generate it randomly until we find a value that produces no collisions! This approach is used in Stockfish [3] to generate magic bitboards. Listing 8 shows this approach.

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  int main() {
6      random_device rd;
7      mt19937_64 mt(rd());
8      uniform_int_distribution<uint64_t> d;
9      unordered_map<uint64_t, int> positions;
10     for (int i = 0; i < 64; i++) {
11         positions[1ULL << i] = i;
12     }
13     constexpr int K = 7;
14     auto index = [](uint64_t magic, int k, uint64_t bitboard) {
15         return bitboard * magic >> (64 - k);
16     };
17     while (true) {
18         uint64_t MAGIC = d(mt);
19         bool found = true;
20         set<uint64_t> seen;
21         for (const auto [bitboard, pos] : positions) {
22             int64_t i = index(MAGIC, K, bitboard);
23             if (seen.contains(i)) {
24                 found = false;
25                 break;
26             }
27             seen.insert(i);
28         }
```

```
29          if (found) {
30              cout << format("found magic for K = {}: {:#x}\n", K, MAGIC);
31              int size = 1 << K;
32              vector<int> table(size, -1);
33              for (const auto [bitboard, pos] : positions) {
34                  table[index(MAGIC, K, bitboard)] = pos;
35              }
36              cout << format("uint8_t positions[{}] = {{", size);
37              for (int i = 0; i < size; i++) {
38                  cout << table[i] << ',';
39              }
40              cout << "};\n";
41              break;
42          }
43      }
44  }
```

Listing 8: Random search for the `MAGIC` constant

- On lines 6 to 8, we set up a random generator to randomly generate `uint64_t` values.
- On line 9, the lookup table `positions` associates each long integer containing only one 1-bit with its position. This table is initialized on lines 10 to 12.
- On line 13, the constant `K` is the number of bits in our key obtained by magic perfect hashing. `K` must be large enough to represent all values we need to cover. For example, here we have 64 possible values since we have 64 possible positions for the 1-bit in a 64-bit long integer. With `K == 7`, we can cover $2^7 = 128$ different values. Note that this is more than the 64 values we need, and `K == 6` would suffice (`K == 5` would be too small). However, when trying to find the `MAGIC` constant randomly for `K == 6`, this program could not find one in reasonable time ☹ (we will see another approach to succeed with `K == 6` ☺).
- On lines 14 to 16, the `index` function calculates the index for key `bitboard` (this bitboard contains only one 1-bit) using the magic perfect hashing formula.
- The `while` loop, lines 17 to 43, loops until it finds a `MAGIC` constant that achieves magic perfect hashing (this loop could potentially run forever).
- On line 18, we create the `MAGIC` value randomly.
- The `set<uint64_t>` on line 20 allows us to remember the indices (obtained via the `index` function) we have already used, to verify we have no collisions.
- On lines 21 to 28, the `for` loop tests all bitboards, finds the index for each using the hash (`index` function), and verifies there are no collisions (line 23) with indices obtained for previous bitboards.

- Finally, on lines 29 to 42, if we found a `MAGIC` constant that creates magic perfect hashing, we display it and then display a *C++* array containing the position of the 1-bit for each bitboard. The output of this program for a given execution is shown below.

```
found magic for K = 7: 0x65e4d4ee86638416
uint8_t positions[128] = {
    63, -1, 54, -1, 49, 55, 33, -1, 50, -1, -1, 56, 34, -1, 43, -1,
    51, -1, -1, 11, -1, -1, 57,  3, 39, 35, 14, -1, 44, 22, -1, -1,
    52, 31, -1, -1, -1, -1, 12, 20, -1, 18, -1, -1, 58, -1, -1,  4,
    60, 40,  0, 36, -1, 15, -1, -1, 45, -1, 27, 23,  6, -1, -1, -1,
    62, 53, 48, 32, -1, -1, -1, 42, -1, 10, -1,  2, 38, 13, 21, -1,
    30, -1, -1, 19, 17, -1, -1, -1, 59, -1, -1, -1, -1, 26,  5, -1,
    61, 47, -1, 41,  9,  1, 37, -1, 29, -1, 16, -1, -1, -1, 25, -1,
    46, -1,  8, -1, 28, -1, -1, 24, -1,  7, -1, -1, -1, -1, -1, -1
};
```

To get the position of the 1-bit for bitboard `0b1000` for example, we need to look up the following value:

```
   positions[0b1000 * 0x65e4d4ee86638416 >> (64 - 7)]
== positions[0x2f26a774331c20b0 >> 57]
== positions[0x17]
== positions[23]
== 3
```

Notice that there are vacant slots in the `positions` array—those containing -1—so we wasted space. We would not have wasted any space if we had found a `MAGIC` constant for `K == 6`. But is this possible?

To answer this question, we will proceed differently. Obviously, we do not want to scan through all 2^{64} $(9,223,372,036,854,775,808)$ possible values and test each one for magic perfect hashing. We will use an Satisfiability Modulo Theories (SMT) solver that allows us to set constraints before searching the entire space. Setting constraints first will prune the search space. Note that we have no guarantee the search will be fast, but for this problem, finding a `MAGIC` constant for `K == 6` will be very quick. If you are interested in how an SMT solver works, you can consult [5] and [6].

The code in Listing 9 describes the approach using the Z3 Theorem Prover (Z3) solver [7].

```python
from z3 import *

positions = {}
for i in range(64):
    positions[1 << i] = i

solver = Solver()
MAGIC  = BitVec('magic', 64)
K = 6
bitboards = list(positions.keys())

def index(magic, k, bitboard):
    return magic * bitboard >> (64 - k)

for i in range(64):
    index1 = index(MAGIC, K, bitboards[i])
    for j in range(i + 1, 64):
        index2 = index(MAGIC, K, bitboards[j])
        solver.add(index1 != index2)

if solver.check() == sat:
    model = solver.model()
    m = model[MAGIC].as_long()
    print(f'found magic for K = {K}: {m:#x}')
    size = 1 << K
    table = [-1] * size
    for bitboard, pos in positions.items():
        table[index(m, K, bitboard) & size - 1] = pos
    print(f'constexpr uint8_t positions[{size}] = {{')
    for i in range(size):
        print(f'{table[i]},', end='')
    print('\n};')
```

Listing 9: Searching for the `MAGIC` constant with an SMT solver

- Lines 3 to 5 associate each bitboard containing only one 1-bit with the corresponding position of that bit. We use the `positions` dictionary (line 3) for this.
- On line 7, we instantiate the Z3 solver.
- On line 8, we declare that the `MAGIC` constant is of type `z3.BitVec`, a type provided by the Z3 solver for representing bit vectors—here we use a 64-bit vector.

- The `index` function defined on lines 12 and 13 calculates the hash based on the `MAGIC` constant and `K`.
- On lines 15 to 19, we give the solver the constraints that hashed values from the `index` function for two different bitboards must not map to the same location. This constraint ensures perfect hashing.
- On line 21, we run the solver, which returns `sat` if it found a `MAGIC` value satisfying the constraints. Note that if the solver returns `unsat` instead of `sat`, it means no 64-bit integer allows perfect hashing. Even though the search space is very large, on my machine, finding a solution takes less than a second!
- Lines 22 to 32 print to standard output the `MAGIC` value and the *C++* array of obtained values. This array is shown below.

```
found magic for K = 6: 0x2643c51ab9dfa5b
constexpr uint8_t positions[64] = {
    0,  1,  2, 14,  3, 22, 28, 15, 11,  4, 23, 55,  7, 29, 41, 16,
   12, 26, 53,  5, 24, 33, 56, 35, 61,  8, 30, 58, 37, 42, 17, 46,
   63, 13, 21, 27, 10, 54,  6, 40, 25, 52, 32, 34, 60, 57, 36, 45,
   62, 20,  9, 39, 51, 31, 59, 44, 19, 38, 50, 43, 18, 49, 48, 47
};
```

Note that in *Python* it was important to perform the operation `index(m, K, bitboard) & size - 1` on line 28 because Python integers have arbitrary precision. For example, we have:

```
   0x2643c51ab9dfa5b * 0x8000000000000000 >> 58
== 0x4c878a3573bf4b60

   (0x2643c51ab9dfa5b * 0x8000000000000000 >> 58) & size - 1
== 0x4c878a3573bf4b60 & 0b111111
== 32
```

In *C++* we would have:

```
   0x2643c51ab9dfa5b * 0x8000000000000000 >> 58
== 32
```

Note that we don't have the same problem on line 18 because `MAGIC` is of type `BitVec(64)`.

To test the efficiency of our magic perfect hashing technique, we set up the micro-benchmark shown in Listing 10. We use the *Google Benchmark* library for these measurements [8]. The benchmark execution results are shown in Listing 11. The `BM_magic_positions` function using magic perfect hashing is much faster than the `BM_unordered_map_positions` function using a standard hash table. Of course, the `BM_builtin_ctz_positions` function using a dedicated instruction is the most efficient (but will be of no use for move generation).

```
#include <benchmark/benchmark.h>
#include <unordered_map>
#include <vector>

std::vector<uint64_t> generate_isolated_bits() {
    std::vector<uint64_t> samples;
    for (int i = 0; i < 64; i++) {
        samples.push_back(1ULL << i);
    }
    return samples;
}
static void BM_magic_positions(benchmark::State& state) {
    constexpr uint64_t MAGIC = 0x2643c51ab9dfa5b;
    constexpr int K = 6;
    constexpr uint8_t positions[64] = {
         0,  1,  2, 14,  3, 22, 28, 15, 11,  4, 23, 55,  7, 29, 41, 16,
        12, 26, 53,  5, 24, 33, 56, 35, 61,  8, 30, 58, 37, 42, 17, 46,
        63, 13, 21, 27, 10, 54,  6, 40, 25, 52, 32, 34, 60, 57, 36, 45,
        62, 20,  9, 39, 51, 31, 59, 44, 19, 38, 50, 43, 18, 49, 48, 47
    };
    auto samples = generate_isolated_bits();
    size_t idx = 0;
    for (auto _ : state) {
     uint64_t bitboard = samples[idx++ & 63];
     benchmark::DoNotOptimize(positions[bitboard * MAGIC >> (64 - K)]);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_magic_positions);
static void BM_unordered_map_positions(benchmark::State& state) {
    std::unordered_map<uint64_t, uint8_t> map;
    for (uint8_t i = 0; i < 64; i++) {
        map[1ULL << i] = i;
```

```
    }
    auto samples = generate_isolated_bits();
    size_t idx = 0;
    for (auto _ : state) {
        uint64_t bitboard = samples[idx++ & 63];
        benchmark::DoNotOptimize(map[bitboard]);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_unordered_map_positions);
static void BM_builtin_ctz_positions(benchmark::State& state) {
    auto samples = generate_isolated_bits();
    size_t idx = 0;
    for (auto _ : state) {
        uint64_t bitboard = samples[idx++ & 63];
        benchmark::DoNotOptimize(__builtin_ctzll(bitboard));
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_builtin_ctz_positions);
BENCHMARK_MAIN();
```

Listing 10: Micro-benchmark comparing execution times of magic perfect hashing, a standard hash table, and a dedicated machine instruction.

```
Run on (12 X 4400 MHz CPU s)
CPU Caches: L1 Data 32 KiB (x6), L1 Instruction 32 KiB (x6),
            L2 Unified 256 KiB (x6), L3 Unified 12288 KiB (x1)
Load Average: 0.62, 1.33, 2.92

Benchmark                        Time        CPU    Iterations  Throughput
-------------------------------------------------------------------------
BM_magic_positions           0.542 ns   0.542 ns   1000000000  1.85 G/s
BM_unordered_map_positions    30.2 ns    30.2 ns     23361655  33.1 M/s
BM_builtin_ctz_positions     0.480 ns   0.480 ns   1000000000  2.08 G/s
```

Listing 11: Micro-benchmark execution results from Listing 10

Now that we have seen how the magic perfect hashing technique works through a simple example, we can study generating possible piece moves using this technique.

2.3.2 Magic Bitboards for Piece Moves

For each board configuration, we want to be able to provide, for a given square, the list of possible moves for that game configuration. For example, for the board in figure 2.2 where the possible moves for the piece at `d3` are represented by stars in table 2.11, we would like to very efficiently obtain the bitboard shown in table 2.12 indicating the different possible moves.

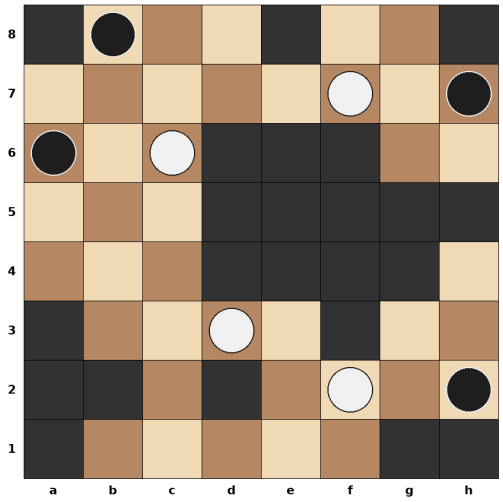


Figure 2.2: Board position example (after move 23 in figure 1.8)

Table 2.11: Possible moves for white piece at d3

8	■	●	·	·	■	·	·	■
7	·	·	·	·	·	○	·	●
6	●	·	○	■	■	■	·	·
5	·	*	·	■	■	■	■	■
4	·	·	*	■	■	■	■	·
3	■	*	*	○	*	■	·	·
2	■	■	*	■	*	○	·	●
1	■	*	·	·	·	*	■	■
	a	b	c	d	e	f	g	h

Let's take square `d3` as an example. We first need a mask that will allow us to isolate the board squares reachable by the piece at `d3` without considering obstacles. This mask is represented in table 2.13. Let `mask` be the mask from table 2.13 and `board` be the bitboard from table 2.14 representing the game board from figure 2.2. We obtain the bitboard `occupancies`, giving the obstacles on the possible trajectories of the piece at `d3` by doing: `occupancies = mask & board`. This bitboard is given in table 2.15. There is a 1 for each square containing a piece or a hole. What we need is to be able to index a table, let's call it `uint64_t MOVES_D3[]`, using the bitboard `occupancies`, to obtain the bitboard of possible moves from square `d3`. This bitboard is represented in table 2.12. Then we only need to iterate through each 1 bit in this bitboard to know which squares we can move to and create the corresponding move.

Table 2.12: Bitboard of possible moves for the piece at d3 (see table 2.11)

8	56 0	57 0	58 0	59 0	60 0	61 0	62 0	63 0
7	48 0	49 0	50 0	51 0	52 0	53 0	54 0	55 0
6	40 0	41 0	42 0	43 0	44 0	45 0	46 0	47 0
5	32 0	33 **1**	34 0	35 0	36 0	37 0	38 0	39 0
4	24 0	25 0	26 **1**	27 0	28 0	29 0	30 0	31 0
3	16 0	17 **1**	18 **1**	19 0	20 **1**	21 0	22 0	23 0
2	8 0	9 0	10 **1**	11 0	12 **1**	13 0	14 0	15 0
1	0 0	1 **1**	2 0	3 0	4 0	5 **1**	6 0	7 0
	a	b	c	d	e	f	g	h

Table 2.13: Masque pour les cases atteignables par la pièce en d3 sans considérer les éventuels obstacles

8	56 0	57 0	58 0	59 **1**	60 0	61 0	62 0	63 0
7	48 0	49 0	50 0	51 **1**	52 0	53 0	54 0	55 **1**
6	40 **1**	41 0	42 0	43 **1**	44 0	45 0	46 **1**	47 0
5	32 0	33 **1**	34 0	35 **1**	36 0	37 **1**	38 0	39 0
4	24 0	25 0	26 **1**	27 **1**	28 **1**	29 0	30 0	31 0
3	16 **1**	17 **1**	18 **1**	19 **0**	20 **1**	21 **1**	22 **1**	23 **1**
2	8 0	9 0	10 **1**	11 **1**	12 **1**	13 0	14 0	15 0
1	0 0	1 **1**	2 0	3 **1**	4 0	5 **1**	6 0	7 0
	a	b	c	d	e	f	g	h

Table 2.14: Board bitboard

8	56 **1**	57 **1**	58 0	59 0	60 **1**	61 0	62 0	63 **1**
7	48 0	49 0	50 0	51 0	52 0	53 **1**	54 0	55 **1**
6	40 **1**	41 0	42 **1**	43 **1**	44 **1**	45 **1**	46 0	47 0
5	32 0	33 0	34 0	35 **1**	36 **1**	37 **1**	38 **1**	39 **1**
4	24 0	25 0	26 0	27 **1**	28 **1**	29 **1**	30 **1**	31 0
3	16 **1**	17 0	18 0	19 **1**	20 0	21 **1**	22 0	23 0
2	8 **1**	9 **1**	10 0	11 **1**	12 0	13 **1**	14 0	15 **1**
1	0 **1**	1 0	2 0	3 0	4 0	5 0	6 **1**	7 **1**
	a	b	c	d	e	f	g	h

Table 2.15: Occupancies bitboard (`mask & board`) for d3

8	56 0	57 0	58 0	59 0	60 0	61 0	62 0	63 0
7	48 0	49 0	50 0	51 0	52 0	53 0	54 0	55 **1**
6	40 **1**	41 0	42 0	43 **1**	44 0	45 0	46 0	47 0
5	32 0	33 0	34 0	35 **1**	36 0	37 **1**	38 0	39 0
4	24 0	25 0	26 0	27 **1**	28 **1**	29 0	30 0	31 0
3	16 **1**	17 0	18 0	19 **0**	20 0	21 **1**	22 0	23 0
2	8 0	9 0	10 0	11 **1**	12 0	13 0	14 0	15 0
1	0 0	1 0	2 0	3 0	4 0	5 0	6 0	7 0
	a	b	c	d	e	f	g	h

To obtain the index in `MOVES_D3`, we will use the magic bitboard technique we described in the previous section. For square `d3`, we need to find a value `K` and a value `MAGIC` (see listing 7) to be able to obtain the bitboard of possible moves by doing: `MOVES_D3[occupancies * MAGIC >> (64 - K)]`.

To find a `MAGIC` constant for square `d3`, we will need to enumerate all possible obstacle placements on the squares reachable by the piece at `d3`. This means we must enumerate all subsets of 1 bits of the bitboard in table 2.13. Eight subsets from this enumeration are presented in figure 2.3. There are 25 squares on the trajectory of the piece at `d3` (see table 2.13), which means there are $2^{25} = 33,554,432$ subsets to enumerate!

We would like to reduce this number of possible obstacle configurations on the piece's trajectory. Indeed, as we will see when we detail the code to determine the value of the `MAGIC` constant, the memory space to store the possible moves for a given square will depend on this number of configurations.

To reduce this number of configurations, we will not consider the squares on the board edges to remove the 1 bits on ranks 1 and 8 and files a and h from the mask in table 2.13. We obtain the bitboard in table 2.16. We will be missing information about the edges, but we will assume that if it's possible to play to `d7` for example, it will also be possible to play to `d8`. Of course there might be an obstacle at `d8`, but we can easily eliminate this impossible move as we will see later in this chapter (section 2.3.4). With this reduction, we are left with $2^{17} = 131,072$ subsets to enumerate. That's much better, but we can still reduce this number further.

Table 2.16: Mask for reachable squares for the piece at `d3` excluding board edges

8	56 0	57 0	58 0	59 0	60 0	61 0	62 0	63 0
7	48 0	49 0	50 0	51 **1**	52 0	53 0	54 0	55 0
6	40 0	41 0	42 0	43 **1**	44 0	45 0	46 **1**	47 0
5	32 0	33 **1**	34 0	35 **1**	36 0	37 **1**	38 0	39 0
4	24 0	25 0	26 **1**	27 **1**	28 **1**	29 0	30 0	31 0
3	16 0	17 **1**	18 **1**	19 **0**	20 **1**	21 **1**	22 **1**	23 0
2	8 0	9 0	10 **1**	11 **1**	12 **1**	13 0	14 0	15 0
1	0 0	1 0	2 0	3 0	4 0	5 0	6 0	7 0
	a	b	c	d	e	f	g	h

Indeed, we can split a piece's moves into diagonal and orthogonal moves. We will obtain two tables: `MOVES_D3_DIAG` and `MOVES_D3_ORTHO`. We will need to consult two

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

(a) Empty (0 bits)

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	**1**	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

(b) 1 bit set

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	**1**	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	**1**	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

(c) 2 bits set

8	0	0	0	**1**	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	**1**	0	0	0	0	0
3	0	0	0	0	0	**1**	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

(d) 3 bits set

8	0	0	0	0	0	0	0	0
7	0	0	0	**1**	0	0	0	**1**
6	**1**	0	0	0	0	0	**1**	0
5	0	**1**	0	0	0	0	0	0
4	0	0	0	**1**	**1**	0	0	0
3	0	**1**	0	0	0	0	0	**1**
2	0	0	0	0	0	0	0	0
1	0	**1**	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

(e) 9 bits set

8	0	0	0	**1**	0	0	0	0
7	0	0	0	**1**	0	0	0	**1**
6	**1**	0	0	**1**	0	0	**1**	0
5	0	**1**	0	**1**	0	**1**	0	0
4	0	0	**1**	**1**	**1**	0	0	0
3	**1**	**1**	**1**	0	**1**	**1**	**1**	**1**
2	0	0	**1**	**1**	**1**	0	0	0
1	0	**1**	0	**1**	0	**1**	0	0
	a	b	c	d	e	f	g	h

(f) 25 bits set (full)

Figure 2.3: Six subsets of the mask for d3 showing different occupancy patterns (0 to 25 bits set) from the 2^{25} possible ones

tables instead of one to obtain the possible moves at `d3` (or any other square), but we will save a lot of memory space. We will see how to combine the values from these two tables later in this chapter. Instead of the mask from table 2.16, we will now have the two masks from figure 2.4.

For the orthogonal mask in figure 2.4a, we have $2^{10} = 1,024$ subsets and for the diagonal mask in figure 2.4b, we have $2^7 = 128$ subsets, so we go from $2^{17} = 131,072$ to $2^{10} + 2^7 = 1,152$ obstacle configurations to consider for the piece at `d3`!

8	56 0	57 0	58 0	59 0	60 0	61 0	62 0	63 0
7	48 0	49 0	50 0	51 **1**	52 0	53 0	54 0	55 0
6	40 0	41 0	42 0	43 **1**	44 0	45 0	46 0	47 0
5	32 0	33 0	34 0	35 **1**	36 0	37 0	38 0	39 0
4	24 0	25 0	26 0	27 **1**	28 0	29 0	30 0	31 0
3	16 0	17 **1**	18 **1**	19 **0**	20 **1**	21 **1**	22 **1**	23 0
2	8 0	9 0	10 0	11 **1**	12 0	13 0	14 0	15 0
1	0 0	1 0	2 0	3 0	4 0	5 0	6 0	7 0
	a	b	c	d	e	f	g	h

(a) Orthogonal mask

8	56 0	57 0	58 0	59 0	60 0	61 0	62 0	63 0
7	48 0	49 0	50 0	51 0	52 0	53 0	54 0	55 0
6	40 0	41 0	42 0	43 0	44 0	45 0	46 **1**	47 0
5	32 0	33 **1**	34 0	35 0	36 0	37 **1**	38 0	39 0
4	24 0	25 0	26 **1**	27 0	28 **1**	29 0	30 0	31 0
3	16 0	17 0	18 0	19 **0**	20 0	21 0	22 0	23 0
2	8 0	9 0	10 **1**	11 0	12 **1**	13 0	14 0	15 0
1	0 0	1 0	2 0	3 0	4 0	5 0	6 0	7 0
	a	b	c	d	e	f	g	h

(b) Diagonal mask

Figure 2.4: Orthogonal and diagonal masks for d3 without edge squares

We can now study how to find the `MAGIC` and `K` constants for the orthogonal and diagonal moves of each square on the board. The code is given in listing 12[9]. An execution of this program[10] is given in listing 13[11].

```
1  constexpr uint64_t FileABB = 0x0101010101010101;
2  constexpr uint64_t FileHBB = FileABB << 7;
3  constexpr uint64_t Rank1BB = 0xFF;
4  constexpr uint64_t Rank8BB = Rank1BB << (8 * 7);
5
6  constexpr bool is_ok(Square s) { return s >= SQ_A1 && s <= SQ_H8; }
7
8  constexpr File file_of(Square s) { return File(s & 7); }
9
```

[9] I tried to use the Z3 solver to obtain these constants but without success.

[10] Given the random generator and its initialization, the program may find different constants for different executions.

[11] The output is reformatted for readability.

```
10  constexpr Rank rank_of(Square s) { return Rank(s >> 3); }
11
12  constexpr uint64_t rank_bb(Rank r) { return Rank1BB << (8 * r); }
13
14  constexpr uint64_t rank_bb(Square s) { return rank_bb(rank_of(s)); }
15
16  constexpr uint64_t file_bb(File f) { return FileABB << f; }
17
18  constexpr uint64_t file_bb(Square s) { return file_bb(file_of(s)); }
19
20  constexpr uint64_t square_bb(Square s) {
21      return uint64_t(1) << s;
22  }
23
24  constexpr Square operator+(Square s, Direction d) {
25      return Square(int(s) + int(d));
26  }
27
28  int manhattan_distance(Square sq1, Square sq2) {
29      int d_rank = std::abs(rank_of(sq1) - rank_of(sq2));
30      int d_file = std::abs(file_of(sq1) - file_of(sq2));
31      return d_rank + d_file;
32  }
33
34  enum MoveType {
35      ORTHOGONAL,
36      DIAGONAL
37  };
38
39  uint64_t reachable_squares(MoveType mt, Square sq, uint64_t occupied) {
40      uint64_t  moves    = 0;
41      Direction o_dir[4] = {NORTH, SOUTH, EAST, WEST};
42      Direction d_dir[4] = {NORTH_EAST,SOUTH_EAST,SOUTH_WEST,NORTH_WEST};
43      for (Direction d : (mt == ORTHOGONAL ? o_dir : d_dir)) {
44          Square s = sq;
45          while (true) {
46              Square to = s + d;
47              if (!is_ok(to) || manhattan_distance(s, to) > 2) break;
48              uint64_t bb = square_bb(to);
49              if ((square_bb(to) & occupied) != 0) break;
50              moves |= bb;
51              s = to;
```

```
52          }
53      }
54      return moves;
55  }
56
57  std::pair<int, uint64_t> magic_for_square(MoveType mt, Square sq) {
58      using namespace std;
59      uint64_t edges = ((Rank1BB | Rank8BB) & ~rank_bb(sq)) |
60                           ((FileABB | FileHBB) & ~file_bb(sq));
61      uint64_t moves_bb = reachable_squares(mt, sq, 0) & ~edges;
62      vector<uint64_t> occupancies;
63      vector<uint64_t> possible_moves;
64      uint64_t b = 0;
65      int size = 0;
66      do {
67          occupancies.push_back(b);
68          possible_moves.push_back(sliding_moves(mt, sq, b));
69          size++;
70          b = (b - moves_bb) & moves_bb;
71      } while (b);
72      int k = popcount(moves_bb);
73      int shift = 64 - k;
74      random_device rd;
75      mt19937_64 twister(rd());
76      uniform_int_distribution<uint64_t> d;
77      vector<uint32_t> seen(1 << k);
78      vector<uint64_t> moves(1 << k);
79      for (uint32_t cnt = 0;; cnt++) {
80          uint64_t magic = d(twister) & d(twister) & d(twister);
81          bool found = true;
82          for (size_t j = 0; j < occupancies.size(); j++) {
83              uint64_t occ = occupancies[j];
84              int index = magic * occ >> shift;
85              if (seen[index] == cnt && moves[index] != possible_moves[j])
86              {
87                  found = false;
88                  break;
89              }
90              seen[index] = cnt;
91              moves[index] = possible_moves[j];
92          }
93          if (found) {
```

```
94              return {k, magic};
95          }
96      }
97      unreachable();
98  }
99
100 int main() {
101     using namespace std;
102     for (MoveType mt : {ORTHOGONAL, DIAGONAL}) {
103         stringstream ss_k, ss_magic;
104         ss_k << format("int {}_K[64] = {{",
105                        mt == ORTHOGONAL ? "H" : "D");
106         ss_magic << format("uint64_t {}_MAGIC[64] = {{",
107                            mt == ORTHOGONAL ? "H" : "D");
108         for (int sq = SQ_A1; sq <= SQ_H8; sq++) {
109             const auto [k, magic] = magic_for_square(mt, Square(sq));
110             ss_k << dec << k << ',';
111             ss_magic << showbase << hex << magic << ',';
112         }
113         cout << ss_k.str() << "};\n";
114         cout << ss_magic.str() << "};\n\n";
115     }
116 }
```

Listing 12: Program to find the `K` and `MAGIC` constants from listing 7 for the obstacle configurations of a given move type and square

In listing 12,

- At line 39, the function `uint64_t reachable_squares(MoveType mt, Square sq, uint64_t occupied)` will allow us to create the mask of squares reachable from square `sq` for the move type `mt`, which will be either orthogonal (`ORTHOGONAL` at line 35) or diagonal (`DIAGONAL` at line 36). The created mask will be similar to the masks in figure 2.4.
  - At lines 43 to 53, we iterate through each direction according to the move type, starting from the starting square (`sq`) until we encounter an obstacle.
  - At line 47, we test whether we go off the board or wrap around, for example, if we are at square `SQ_A8` and the direction is `EAST`, `SQ_H8 + EAST == SQ_B1`, the test with Manhattan distance will allow us to exclude this kind of case (see line 28 for the definition of Manhattan distance and listing 5 for the definitions of `File` and `Rank`).
  - At line 48, we transform the destination square `to` into a bitboard that contains a single 1, on the bit corresponding to the square.

– At line 49, we test whether there is an obstacle on the square we want to move to using the bitboard we just created and the bitboard `occupied` containing the obstacles on the board.

– At line 50, we add the new square as a possible move.

– At line 51, we move one square in direction `d`.

- At line 57, the function `pair<int, uint64_t> magic_for_square(MoveType mt, Square sq)` will return a pair whose first element will be the value of `K`, and whose second element will be the `MAGIC` constant (see listing 7), for the move type `mt` (`ORTHOGONAL` or `DIAGONAL`) and square `sq`.

  – At lines 59 to 60, we create a mask that will cover the board edges but being careful not to exclude squares that must remain accessible by the piece. For example, if `sq == SQ_A1`, we obtain the bitboard from figure 2.5a and if `sq == SQ_C2`, we obtain the one from figure 2.5b.

8	56 **1**	57 **1**	58 **1**	59 **1**	60 **1**	61 **1**	62 **1**	63 **1**
7	48 0	49 0	50 0	51 0	52 0	53 0	54 0	55 **1**
6	40 0	41 0	42 0	43 0	44 0	45 0	46 0	47 **1**
5	32 0	33 0	34 0	35 0	36 0	37 0	38 0	39 **1**
4	24 0	25 0	26 0	27 0	28 0	29 0	30 0	31 **1**
3	16 0	17 0	18 0	19 0	20 0	21 0	22 0	23 **1**
2	8 0	9 0	10 0	11 0	12 0	13 0	14 0	15 **1**
1	0 **0**	1 0	2 0	3 0	4 0	5 0	6 0	7 **1**
	a	b	c	d	e	f	g	h

(a) Edge mask for square a1

8	56 **1**	57 **1**	58 **1**	59 **1**	60 **1**	61 **1**	62 **1**	63 **1**
7	48 **1**	49 0	50 0	51 0	52 0	53 0	54 0	55 **1**
6	40 **1**	41 0	42 0	43 0	44 0	45 0	46 0	47 **1**
5	32 **1**	33 0	34 0	35 0	36 0	37 0	38 0	39 **1**
4	24 **1**	25 0	26 0	27 0	28 0	29 0	30 0	31 **1**
3	16 **1**	17 0	18 0	19 0	20 0	21 0	22 0	23 **1**
2	8 **1**	9 0	10 **0**	11 0	12 0	13 0	14 0	15 **1**
1	0 **1**	1 **1**	2 **1**	3 **1**	4 **1**	5 **1**	6 **1**	7 **1**
	a	b	c	d	e	f	g	h

(b) Edge mask for square c2

Figure 2.5: Edge masks excluding the rank and file of the piece

– At line 61, we create the bitboard `moves_bb` of squares accessible by the piece from square `sq`. The parameter `occupied` is equal to zero to indicate that there are no obstacles. We remove the edges with `& ~edges` as we saw in paragraph 2.3.2.

– At lines 62 to 71, we will create all obstacle configurations on the squares accessible by the piece at `sq`, without considering the squares in `edge`, by enumerating all subsets of the bitboard `moves_bb`.

  * At lines 62 and 63, the vector `occupancies` will contain all obstacle configurations on the piece's trajectory, without considering the squares in `edge`, and for each of these configurations, the vector `possible_moves` will contain the bitboard of possible moves for that configuration. For

example, we could have as an obstacle configuration the bitboard from figure 2.6a and the associated possible moves would be represented by the bitboard from figure 2.6b.

8	56 0	57 0	58 0	59 0	60 0	61 0	62 0	63 0
7	48 0	49 0	50 0	51 0	52 0	53 0	54 0	55 0
6	40 0	41 0	42 0	43 **1**	44 0	45 0	46 0	47 0
5	32 0	33 0	34 0	35 **1**	36 0	37 **1**	38 0	39 0
4	24 0	25 0	26 0	27 **1**	28 **1**	29 0	30 0	31 0
3	16 0	17 0	18 0	19 **0**	20 0	21 **1**	22 0	23 0
2	8 0	9 0	10 0	11 **1**	12 0	13 0	14 0	15 0
1	0 0	1 0	2 0	3 0	4 0	5 0	6 0	7 0
	a	b	c	d	e	f	g	h

(a) Occupancy configuration example

8	56 0	57 0	58 0	59 0	60 0	61 0	62 0	63 0
7	48 0	49 0	50 0	51 0	52 0	53 0	54 0	55 0
6	40 **1**	41 0	42 0	43 0	44 0	45 0	46 0	47 0
5	32 0	33 **1**	34 0	35 0	36 0	37 0	38 0	39 0
4	24 0	25 0	26 **1**	27 0	28 0	29 0	30 0	31 0
3	16 **1**	17 **1**	18 **1**	19 **0**	20 **1**	21 0	22 0	23 0
2	8 0	9 0	10 **1**	11 0	12 **1**	13 0	14 0	15 0
1	0 0	1 **1**	2 0	3 0	4 0	5 **1**	6 0	7 0
	a	b	c	d	e	f	g	h

(b) Possible moves for this occupancy

Figure 2.6: Example of occupancy configuration and corresponding possible moves

* The loop between lines 66 and 71 will allow us to enumerate all subsets of the bitboard `moves_bb`. To do this, we use the technique called *Carry-Rippler*[12] at line 70. Let's take a fictitious example with `moves_bb = 0b1011`. We then obtain

```
1 uint64_t b = 0;                  // b == 0
2 b = (0 - 0b1011) & 0b1011;       // b == 0b0001
3 b = (0b0001 - 0b1011) & 0b1011; // b == 0b0010
4 b = (0b0010 - 0b1011) & 0b1011; // b == 0b0011
5 b = (0b0011 - 0b1011) & 0b1011; // b == 0b1000
6 b = (0b1000 - 0b1011) & 0b1011; // b == 0b1001
7 b = (0b1001 - 0b1011) & 0b1011; // b == 0b1010
8 b = (0b1010 - 0b1011) & 0b1011; // b == 0b1011
9 b = (0b1011 - 0b1011) & 0b1011; // b == 0
```

* At line 68, we create the possible moves for the obstacle configuration `b`. Note that since `b` does not cover the squares in `edge`, if no obstacle in `b` blocks the piece's moves to a square in `edge`, it will appear in the possible moves.

[12] If you are interested in this kind of bit manipulation tricks, you will love the book *Hacker's Delight*[9].

- At line 72, we count the number of 1 bits in the bitboard `moves_bb`. We set `k` to this value, which means that for square `sq` and move type `mt`, the space occupied by our perfect hash table[13] will be 2^k. Note that some obstacle configurations produce the same possible moves, so it might be possible to obtain a smaller value of `k`. But this approach allows us to find the different `MAGIC` constants very quickly and the memory consumed is relatively small (less than 110 KiB in total).
- At lines 74 to 76, we set up the random generator that will allow us to create the `MAGIC` constants.
- At line 77, the array `seen` will allow us to memorize the indices, obtained by the formula at line 84, that we have already encountered.
- At line 78, the array `moves` will allow us to memorize the possible moves for a given index. It may happen that two different obstacle configurations lead to the same possible moves, this array will allow us not to consider as collisions these two different configurations leading to the same index but having the same possible moves.
- The loop from line 79 to 96 will search for the `MAGIC` constant that allows creating the perfect hash for square `sq` and value `k`. Note that this loop is potentially infinite. We inform the compiler, at line 97, that this part of the code will never be reached thanks to the function `std::unreachable`.
- At line 80, we generate a `MAGIC` candidate. We generate three random numbers and perform a bitwise AND between them to obtain a random number with fewer 1 bits. This trick is very important because without it we couldn't find the `MAGIC` constants in a reasonable time![14]
- At lines 82 to 92, we verify that there are no collisions for the constant being considered. If this is the case, we return the value `k` and the magic constant found.
  - At line 84, for a given obstacle configuration, we calculate its index `index` using the perfect hash function: `(magic * occ >> (64 - k))`.
  - At line 85, we test whether there is a collision. We use a classic little trick to avoid having to reinitialize `seen` to zero each time we test a new `magic` candidate. The variable `cnt` allows us to know whether the value contained in seen for position index is a value updated for an old `magic` candidate or is for the current candidate. Indeed, if `seen[index]` is less than `cnt`, the value is no longer current. Now, the part `(moves[index] != possible_moves[j])` allows us to ignore a collision that produces the same possible moves.

- At lines 100 to 116, the `main` will produce and display on standard output all values of `k` and all `MAGIC` constants for each square on the board and for both move types. These constants will soon be used to initialize the table of all possible moves for each of the obstacle configurations. We can see a possible output in listing 13.

[13] This is the approach used in the Stockfish software[3].
[14] Note that using this trick for listing 8 slowed down obtaining the `MAGIC` constant.

```
int O_K[64] = {
    12, 11, 11, 11, 11, 11, 11, 12,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    12, 11, 11, 11, 11, 11, 11, 12,
};
uint64_t O_MAGIC[64] = {
    0x80011040002082,     0x40022002100040,     0x1880200081181000,
    0x2080240800100080,   0x8080024400800800,   0x4100080400024100,
    0xc080028001000a00,   0x80146043000080,
    0x8120802080034004,   0x8401000200240,      0x202001282002044,
    0x81010021000b1000,   0x808044000800,       0x300080800c000200,
    0x8c000268411004,     0x810080058020c100,
    0xc248608010400080,   0x30024040002000,     0x9001010042102000,
    0x210009001002,       0xa0061d0018001100,   0x2410808004000600,
    0x6400240008025001,   0xc10600010340a4,
    0x628080044011,       0x4810014040002000,   0x380200080801000,
    0x10018580080010,     0x101040080180180,    0x9208020080040080,
    0x10400a21008,        0x6800104200010484,
    0x21400280800020,     0x9400402008401001,   0x8430006800200400,
    0x8104411202000820,   0x8010171000408,      0x1202000402001008,
    0x881100904002208,    0x15a0800a49802100,
    0x224001808004,       0x4420201002424000,   0xc04500020008080,
    0x2503009004210008,   0x42801010010,        0x2000400090100,
    0x8080011810040002,   0x44401c008046000d,
    0x4000800521104100,   0x82000b080400080,    0x10821022420200,
    0x9488a82104100100,   0x1004800041100,      0x81600a0034008080,
    0xa00056210280400,    0x5124088200,
    0x4210410010228202,   0x1802230840001081,   0x1002102000400901,
    0x1100c46010000901,   0x281000408001003,    0xc001001c00028809,
    0x10020008008c4102,   0x280005008c014222,
};

int D_K[64] = {
    6, 5, 5, 5, 5, 5, 5, 6,
    5, 5, 5, 5, 5, 5, 5, 5,
    5, 5, 7, 7, 7, 7, 5, 5,
```

```
    5, 5, 7, 9, 9, 7, 5, 5,
    5, 5, 7, 9, 9, 7, 5, 5,
    5, 5, 7, 7, 7, 7, 5, 5,
    5, 5, 5, 5, 5, 5, 5, 5,
    6, 5, 5, 5, 5, 5, 5, 6,
};
uint64_t D_MAGIC[64] = {
    0x811100100408200,    0x412100401044020,    0x404044c00408002,
    0xa0c070200010102,    0x104042001400008,    0x8802013008080000,
    0x1001008860080080,   0x20220044202800,
    0x2002610802080160,   0x4080800808610,      0x91c2800a10a0132,
    0x400242401822000,    0x8530040420040001,   0x142010c210048,
    0x8841820801241004,   0x804212084108801,
    0x2032402094100484,   0x40202110010210a2,   0x8010000800202020,
    0x800240421a800,      0x62200401a00444,     0x224082200820845,
    0x106021492012000,    0x8481020082849000,
    0x40a110c59602800,    0x10020108020400,     0x208c020844080010,
    0x2000480004012020,   0x8001004004044000,   0xa044104128080200,
    0x1108008015cc1400,   0x8284004801844400,
    0x8180a020c2004,      0x9101004080100,      0x8840264108800c0,
    0xc004200900200900,   0x8040008020020020,   0x20010802e1920200,
    0x80204000480a0,      0xc0a80a100008400,
    0x4018808114000,      0x90092200b9000,      0x80020c0048000400,
    0x6018005500,         0x80a0204110a00,      0x4018808407201,
    0x6050040806500280,   0x108208400c40180,
    0x803081210840480,    0x201210402200200,    0x200010400920042,
    0x902000a884110010,   0x851002021004,       0x43c08020120,
    0x6140500501010044,   0x200a04440400c028,
    0x14a002084046000,    0x10002409041040,     0x100022020500880b,
    0x1000000000460802,   0x21084104410,        0x8000001053300104,
    0x4000182008c20048,   0x112088105020200,
};
```

Listing 13: Example output from one execution of the program from listing 12

```
uint64_t orthogonalTable[102400];
uint64_t diagonalTable[5248];
Magic orthogonalMagics[SQUARE_NB];
Magic diagonalMagics[SQUARE_NB];
```

Listing 14: The tables allowing us to generate possible moves from a given square and an obstacle configuration

Now that we have the `MAGIC` constants, we will be able to use them to initialize the tables described in listing 14, which will store the necessary information allowing us to generate possible moves from a given square and an obstacle configuration.

- At line 1, the `orthogonalTable` array will contain the bitboards of possible moves for all orthogonal moves, for all squares and all obstacle configurations. The value 102400 for the size of this table is obtained by calculating

```
size_t size = 0;
for (int i = 0; i < 64; i++) {
    size += 1 << O_K[i];
}
```

- At line 2, the `diagonalTable` array does the same but for diagonal moves.
- At line 3, the `orthogonalMagics` array will define for each square of the board the different information allowing us to apply the perfect hashing and retrieve the possible moves in the `orthogonalTable` array.
- At line 4, the `diagonalMagics` array does the same for diagonal moves.

```
1  struct Magic {
2      uint64_t  mask;
3      uint64_t  magic;
4      uint64_t* moves;
5      uint32_t  shift;
6
7      uint32_t index(uint64_t occupied) const {
8          return uint32_t( ((occupied & mask) * magic) >> shift );
9      }
10 };
```

Listing 15: Data structure allowing us to store the useful information to find possible moves for a given square and obstacle configuration

Listing 15 describes the structure allowing us to store the information for applying the perfect hashing for a given square.

- At line 2, the `mask` attribute is the bitboard allowing us to isolate the part of the board that contains the squares affected by the piece's movement for the considered square and movement type. This mask corresponds to the `moves_bb` variable at line 61 of listing 12.

- At line 3, the `magic` attribute is the `MAGIC` constant that we found for the considered square and movement. This attribute will take the value of one of the constants from the `O_MAGIC` or `D_MAGIC` tables from listing 13.
- At line 4, the `moves` attribute is a pointer to the part of the `orthogonalTable` or `diagonalTable`, depending on the movement type, that will contain the possible moves for the considered square.
- At line 5, the `shift` attribute is the shift `(64 - K)` in the formula from listing 7.
- At lines 7 to 9, the function `uint32_t index(uint64_t occupied) const` calculates the position in `moves` where to find the bitboard of possible moves. We find the formula from listing 7 with the mask applied to the obstacles to only consider those in the piece's trajectory.

```
1 uint64_t moves_bb(Square sq, uint64_t occupied) {
2     uint32_t idx_omoves = orthogonalMagics[sq].index(occupied);
3     uint32_t idx_dmoves = diagonalMagics[sq].index(occupied);
4     return orthogonalMagics[sq].moves[idx_omoves] |
5             diagonalMagics[sq].moves[idx_dmoves];
6 }
```

Listing 16: The `moves_bb` function uses the elements from listings 14 and 15 to efficiently compute the bitboard of possible moves for a given square and obstacle configuration

The function `uint64_t moves_bb(Square sq, uint64_t occupied)` from listing 16 will use the tables from listing 14 to compute the bitboard of possible moves, both orthogonal and diagonal, for the square `sq` and the obstacle configuration `occupied`. This function will be used for move generation in section 2.3.4. To do this,

- At line 2, we compute the index `idx_omoves` using the perfect hashing, by using the `Magic` structure for the square `sq` and orthogonal moves.
- At line 3, we compute the index `idx_dmoves` using the perfect hashing, by using the `Magic` structure for the square `sq` and diagonal moves.
- At lines 4 and 5, we can retrieve the bitboard of possible orthogonal moves (line 4) and create the union of these with the diagonal moves by combining this bitboard using a bitwise OR with the bitboard of possible diagonal moves (line 5). We thus obtain the bitboard of all possible moves for the piece at `sq`, taking into consideration all obstacles from the bitboard `occupied`.

```
1 void init() {
2     init_magics(ORTHOGONAL, orthogonalTable, orthogonalMagics);
3     init_magics(DIAGONAL, diagonalTable, diagonalMagics);
4 }
```

Listing 17

The function `void init()` from listing 17 will initialize once and for all, at the beginning of the program, the different tables. We simply call at lines 2 and 3 the function `void init_magics(MoveType mt, uint64_t table[], Magic magics[])` described in listing 18 for orthogonal and diagonal moves with their associated tables.

```
1  Square& operator++(Square& d) { return d = Square(int(d) + 1); }
2
3  void init_magics(MoveType mt, uint64_t table[], Magic magics[]) {
4      static constexpr uint64_t O_MAGIC[64] = { 0x80011040002082, // ...
5      static constexpr uint64_t D_MAGIC[64] = { 0x811100100408200,// ...
6      using namespace std;
7      int32_t size = 0;
8      vector<uint64_t> occupancies;
9      vector<uint64_t> possible_moves;
10     for (Square sq = SQ_A1; sq <= SQ_H8; ++sq) {
11         occupancies.clear();
12         possible_moves.clear();
13         Magic& m = magics[sq];
14         uint64_t edges = ((Rank1BB | Rank8BB) & ~rank_bb(sq)) |
15                          ((FileABB | FileHBB) & ~file_bb(sq));
16         uint64_t moves_bb = sliding_moves(mt, sq, 0) & ~edges;
17         m.mask = moves_bb;
18         m.shift = 64 - popcount(m.mask);
19         m.magic = (mt == ORTHOGONAL ? O_MAGIC : D_MAGIC)[sq];
20         m.moves = table + size;
21         uint64_t b = 0;
22         do {
23             occupancies.push_back(b);
24             possible_moves.push_back(sliding_moves(mt, sq, b));
25             b = (b - moves_bb) & moves_bb;
26             size++;
27         } while (b);
28         for (size_t j = 0; j < occupancies.size(); j++) {
29             int32_t index = m.index(occupancies[j]);
30             m.moves[index] = possible_moves[j];
31         }
32     }
33 }
```

Listing 18

The function `void init_magics(MoveType mt, uint64_t table[], Magic magics[])` will initialize the different tables for the considered movement type `mt`. It is very similar to the function from listing 12 which allowed us to find the `MAGIC` constants.

- Lines 4 and 5 use the magic constants that we found by running the program from listing 12, whose execution output was given in listing 13.

- At line 8, the `occupancies` array will contain all obstacle configurations for the considered square for each iteration of the loop at line 10. The `possible_moves` array will contain the bitboard of possible moves for a given obstacle configuration. Note that we could have placed these two arrays starting from line 11, since they must be reinitialized at each iteration of the loop at line 10. However, it is more efficient to call the `clear` method at lines 11 and 12, which will avoid unnecessary allocations.

- The loop comprising lines 10 to 32 will update the `magics` parameter for all squares of the board and store the possible moves in a region of the `table` parameter.

- We can recognize, from lines 14 to 27, the enumeration of obstacle subsets that we studied in listing 12. We will update the `Magic` structure `m` from line 13, corresponding to the square `sq`, with

  - At line 17, the mask of squares to consider for the movement of the piece at `sq` and the movement type `mt`.
  - At line 18, the right shift to perform after the multiplication: `(occupied & mask) * magic` (see listing 15).
  - At line 19, the magic constant to use for the movement type `mt` and the considered square `sq`.

- At line 20, we compute the address of the beginning of the location in the `table` array that will allow us to store the possible moves for each obstacle configuration. To do this, we use the `size` variable which counts the number of obstacle configurations we have processed since the beginning of the `for` loop.

- Finally, at lines 28 to 31, for each obstacle configuration, we compute its index using the perfect hashing and we store the possible moves in the region of the `table` array reserved for this purpose.

Now that we have at our disposal the function `moves_bb` from listing 16, it will be quite easy to generate the list of possible moves in a given position of the game. But before doing that, we will first describe how to represent a move.

Chapter 3

AI Players

Chapter 4

Monte Carlo Player

Chapter 5

MCTS Player

Chapter 6

Minmax Player

Chapter 7

Minmax with Neural Network Player

Chapter 8

AI Tournament

Chapter 9

Conclusion

Acronymes

SMT Satisfiability Modulo Theories. 23, 24

Z3 Z3 Theorem Prover. 23, 24, 33

Bibliography

[1] Anthropic. *Claude Sonnet 4.5*. https://www.anthropic.com/claude. Large Language Model. 2025.

[2] cppreference.com. *C++ Reference*. Accessed: 2025-01-28. 2025. URL: https://en.cppreference.com/.

[3] Stockfish Team. *Stockfish: Open Source Chess Engine*. https://stockfishchess.org/. Accessed: 2025-01-28. 2025.

[4] Pradyumna Kannan. *Magic Move-Bitboard Generation in Computer Chess*. http://pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf. Accessed: 2025-01-28. 2008.

[5] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. 2nd. Texts in Theoretical Computer Science. An EATCS Series. Berlin Heidelberg: Springer-Verlag, 2016. ISBN: 978-3-662-50496-3.

[6] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. 2nd. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021. ISBN: 978-1-64368-160-3.

[7] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 4963. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[8] Google. *Google Benchmark: A microbenchmark support library*. https://github.com/google/benchmark. C++ microbenchmarking library. 2025.

[9] Henry S. Warren. *Hacker's Delight*. 2nd. Addison-Wesley Professional, 2012. ISBN: 978-0321842688.