Yolah Board Game

Building a Two-Player Perfect-Information Game and Artificial Players

Pascal Garcia

February 12, 2026

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 Sarah, Hugo et Célya ❤️

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1   The Yolah Game

We designed Yolah, a variant of the Pingouins game, to illustrate effective techniques for implementing board games and artificial players for our students. You can see the Pingouins game box in Figure 1.1.
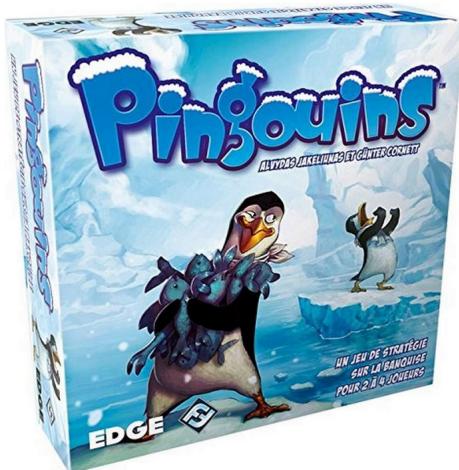
Figure 1.1: The Pingouins game box

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

We now describe the rules of the game, then explain why we chose these rules, give an example of a game between two artificial players and then present the rest of the book.
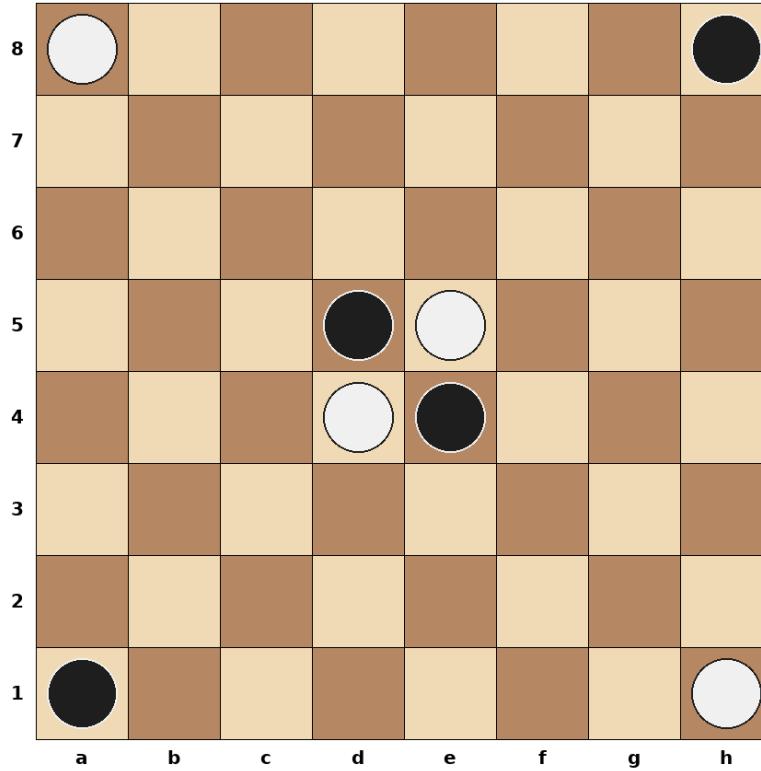
Figure 1.2: The initial configuration of the Yolah game

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 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 soon discuss). For example, if black chooses to move their piece located at d5, the squares where it can land are indicated by black crosses in Figure 1.3.

Now, if the black piece at d5 moves to b7, which we 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 creates 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 make more moves 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
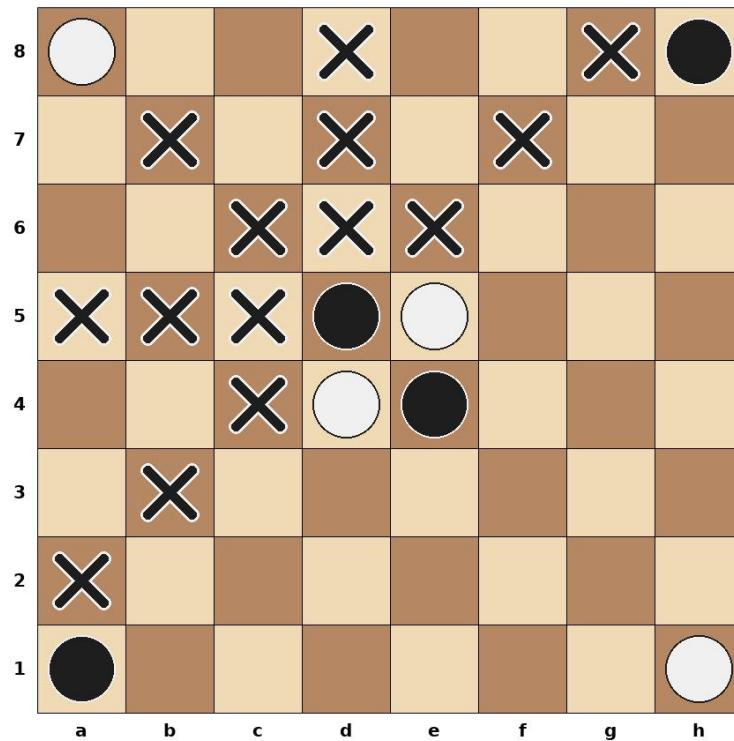
Figure 1.3: Possible moves (black crosses) for the black piece located on square d5
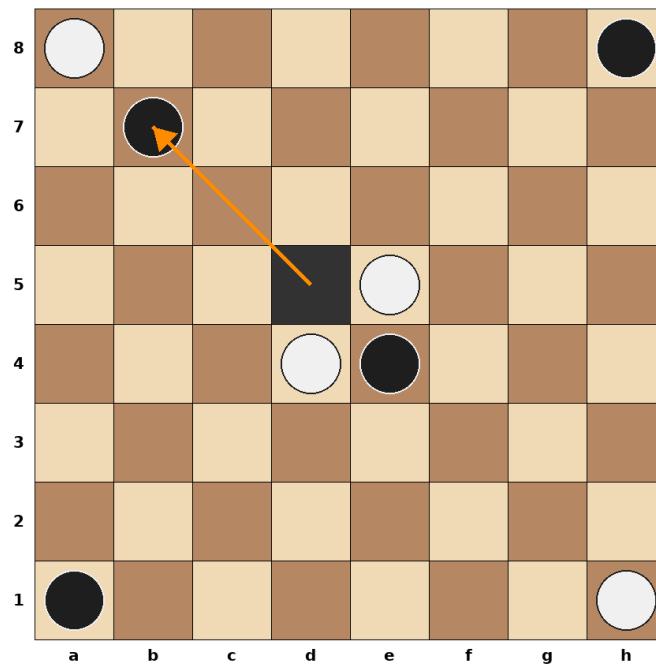
Figure 1.4: Black just moved from d5 to b7. The starting square d5 becomes inaccessible and impassable for the rest of the game

of Figure 1.6 and the score is one point each (each player has played one move).
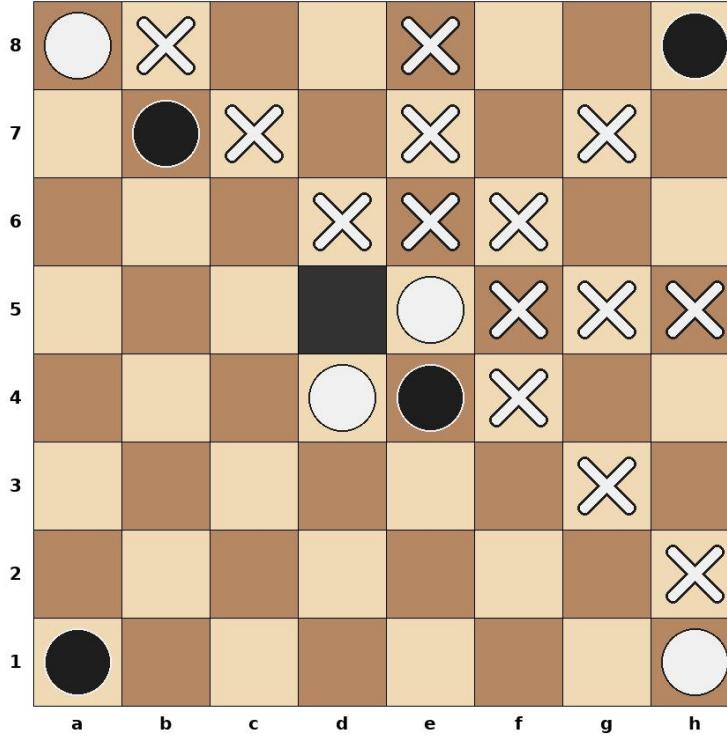
Figure 1.5: Possible moves (white crosses) for the white piece located on square `e5`

To summarize, the rules of Yolah are as follows:

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

- 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 (denoted by the move `a1:a1`).

- 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.

4
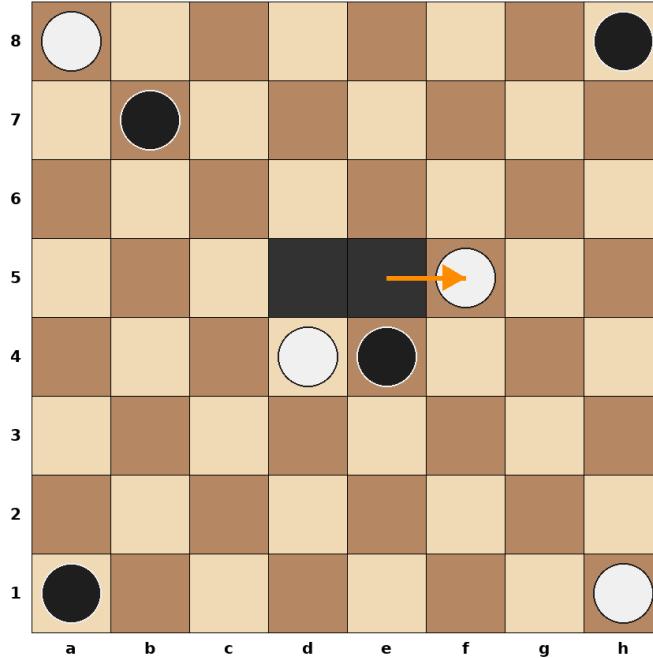
Figure 1.6: White just moved from `e5` to `f5`. The starting square `e5` becomes inaccessible and impassable for the rest of the game. The score is one point each (each player has moved once)

1.1.2 Interesting Characteristics of Yolah for Developing AIs

We chose to modify the rules of the Pingouins game to create our Yolah variant, allowing us to reuse concepts from chess implementation. This is why we use a board identical to that of chess, rather than the hexagonal tiles of the Pingouins game. Piece movement thus becomes identical to that of a Queen in chess. As in the Pingouins game, there are no cycles in the game, so there is no need for special rules to ensure a game terminates. The number of moves available to each player is quite large at the start of the game (larger than for the Pingouins game), yet reasonable [1], but it gradually decreases as holes appear [2]. This allows artificial players to look ahead a fairly large number of moves. Compared to the Pingouins game, we removed the varying point values of tiles and defined a fixed starting configuration to simplify the game and make it symmetric.

1.1.3 Game Example

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

---

[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 a player has more than 56 moves available.

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 that 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 is able to collect all the points from 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 study the implementation of the Yolah game in C++ [2]. This implementation is designed to be efficient because it is important for the artificial players to be able to play many games per second; their level of play depends on it. We develop on a Linux operating system, and occasionally use instructions specific to the x86-64 architecture.

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, unlike the Monte Carlo AI, allows the development of a game tree. Note that these two artificial players do not require human-provided heuristics 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 artificial player 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 artificial player.

Chapter 8 then evaluates all these AIs by having them compete in a tournament. We then conclude and propose various 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)
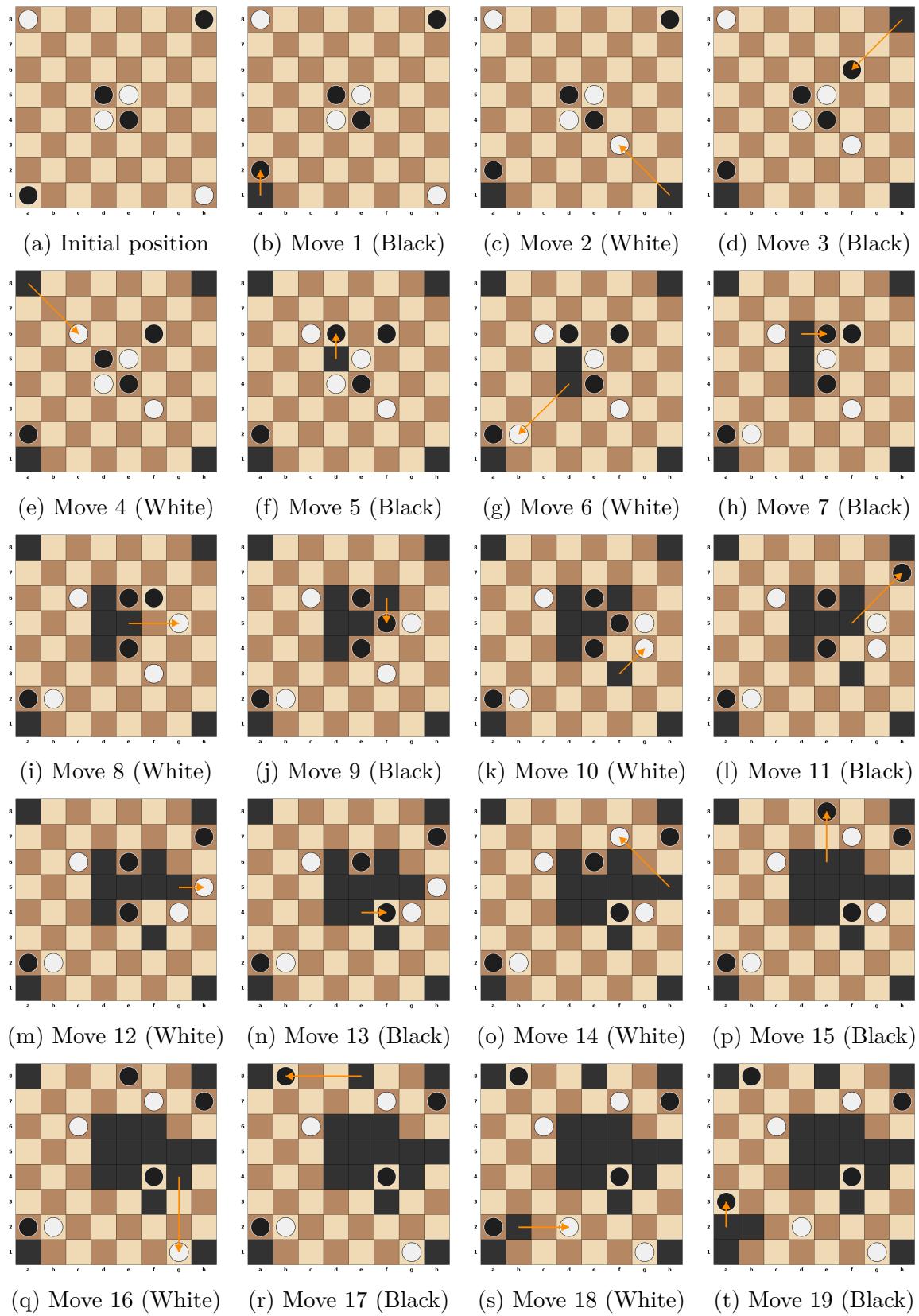
Figure 1.7: Game example between two AIs - moves 1 to 19

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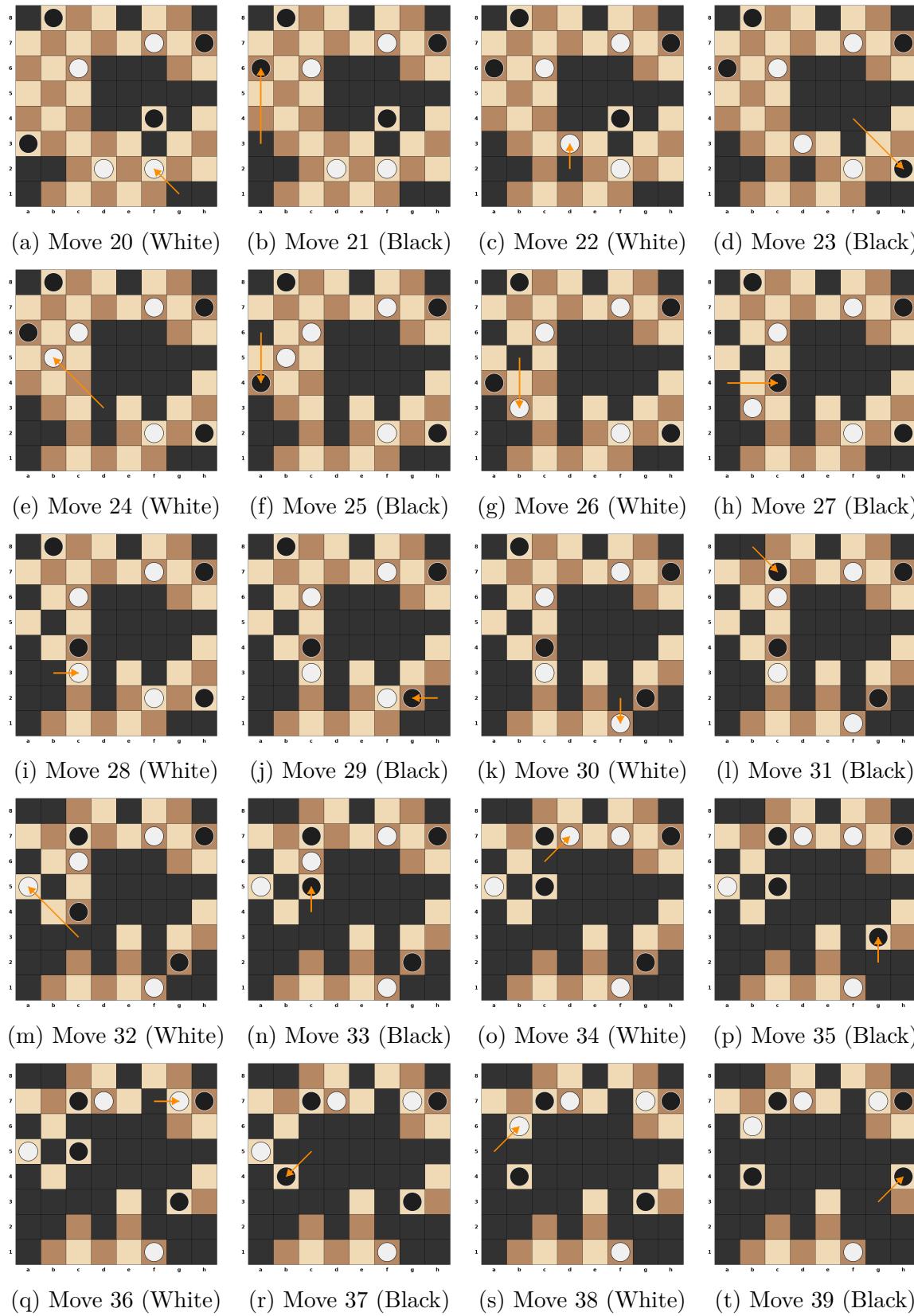
Figure 1.8: Game example between two AIs - moves 20 to 39

(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)
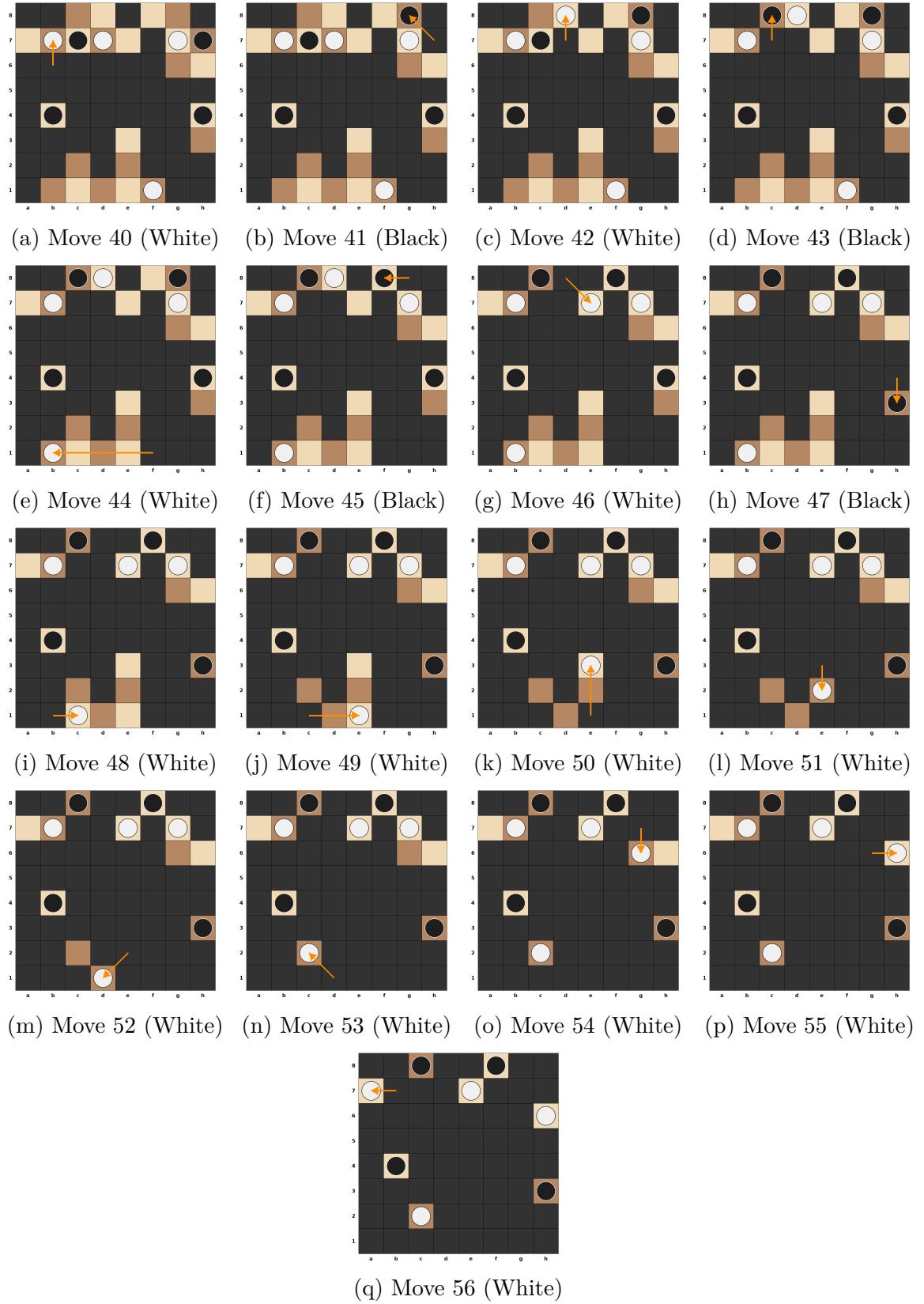
Figure 1.9: Example game between two artificial players – moves 40 to 56. White wins 32 to 24

Chapter 2

Game Engine

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

2.1 Data Structures

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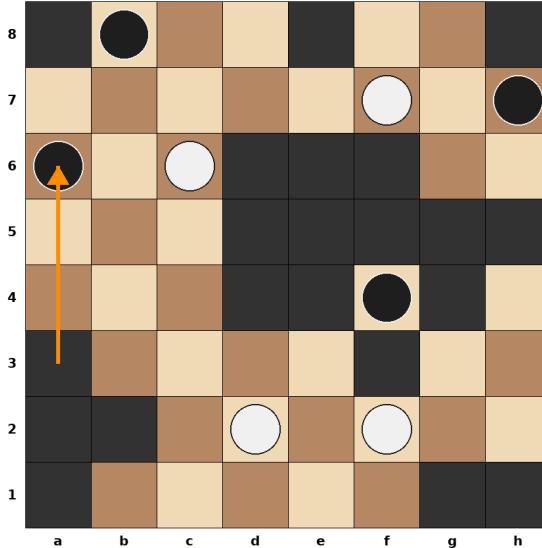
Figure 2.1: Board configuration corresponding to move 21 of the game given as an example in the previous chapter (Figure 1.8b)

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: `0b00000010'10000000'00000001'00000000'00100000'00000000'00000000'00000000`.

- The positions of white pieces on the board (Table 2.3) are represented by the bitboard: `0b00000000'00100000'00000100'00000000'00000000'00000000'00101000'00000000`.

- The positions of holes (Table 2.4) are represented by the bitboard: `0b10010001'00000000'00111000'11111000'01011000'00100001'00000011'11000001`.

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

Table 2.2: Black piece positions

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

Table 2.3: White piece positions

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

Table 2.4: Destroyed square positions (holes)

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

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

```cpp
constexpr uint64_t BLACK_INITIAL_POSITION =
0b10000000'00000000'00000000'00001000'00010000'00000000'00000000'00000001;
constexpr uint64_t WHITE_INITIAL_POSITION =
0b00000001'00000000'00000000'00010000'00001000'00000000'00000000'10000000;
class Yolah {
    uint64_t black = BLACK_INITIAL_POSITION;
    uint64_t white = WHITE_INITIAL_POSITION;
    uint64_t holes = 0;
    uint8_t black_score = 0;
    uint8_t white_score = 0;
    uint8_t ply = 0;
public:
    // ...
};
```

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

```cpp
class Yolah {
    uint64_t black = BLACK_INITIAL_POSITION;
    uint8_t black_score = 0;
    uint64_t white = WHITE_INITIAL_POSITION;
    uint8_t white_score = 0;
    uint64_t holes = 0;
```

---

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

```
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

==

00000010'10000000'00000001'00000000'00100000'00000000'00000000'00000000 |
00000000'00100000'00000100'00000000'00000000'00000000'00101000'00000000

==

00000010'10100000'00000101'00000000'00100000'00000000'00101000'00000000
```

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 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 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. 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).

---

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

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

```cpp
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

```cpp
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)
        return (b & ~FileHBB) << NORTH_EAST;
    else if constexpr (D == NORTH_WEST)
        return (b & ~FileABB) << NORTH_WEST;
    else if constexpr (D == SOUTH_EAST)
```

16

```
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 set bits in the bitboard parameter in direction D[4]. Given

black
==
10000000'00000000'00000000'00001000'00100000'00000000'00000000'00000001

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

00000000'00000000'00001000'00100000'00000000'00000000'00000001'00000000

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 bits set at free positions.

- On line 3, we create the `players` bitboard with bits set at positions occupied by either player.

- Lines 4 to 8 create the `around_players` bitboard with bits set 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 keeps a bit set in the result if and only if the corresponding bit is set 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  constexpr 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, let us now study how to efficiently generate possible moves in a given position.

2.3 Generating Possible Piece Moves

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

2.3.1 Magic Perfect Hashing Function

Suppose we need to find the position of the only set 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 in C++ we can efficiently compute this position with the function std::countr_zero(un signed long long x)[6], for example,

```
position(0b1000) == std::countr_zero(0b1000)
```

However, we code this position function using a magic perfect hashing function. This kind of function is 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:

```
1  unordered_map<uint64_t, uint8_t> table {
2      {1, 0}, {0b10, 1}, {0b100, 2}, {0b1000, 3}, //...
3  }
4  int position(uint64_t x) {
5      return table[x];
6  }
```

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 values 2 and 6 in table[42], but we cannot store more than one value at the same location.

The magic perfect hashing function has the following form (listing 7):

```
1  constexpr uint64_t MAGIC = //...
2  constexpr int K = 6;
3  int magic_perfect_hashing(uint64_t bitboard) {
4      return bitboard * MAGIC >> (64 - K);
5  }
```

---

[5]tzcnt on Intel.

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

[7]We 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.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    random_device rd;
    mt19937_64 mt(rd());
    uniform_int_distribution<uint64_t> d;
    unordered_map<uint64_t, int> positions;
    for (int i = 0; i < 64; i++) {
        positions[1ULL << i] = i;
    }
    constexpr int K = 7;
    auto index = [](uint64_t magic, int k, uint64_t bitboard) {
        return bitboard * magic >> (64 - k);
    };
    while (true) {
        uint64_t MAGIC = d(mt);
        bool found = true;
        set<uint64_t> seen;
        for (const auto [bitboard, pos] : positions) {
            int64_t i = index(MAGIC, K, bitboard);
            if (seen.contains(i)) {
                found = false;
                break;
            }
            seen.insert(i);
```

21

```
28              }
29          if (found) {
30              cout << format("found magic for K = {}: {:#x}\n",
31                             K, MAGIC);
32              int size = 1 << K;
33              vector<int> table(size, -1);
34              for (const auto [bitboard, pos] : positions) {
35                  table[index(MAGIC, K, bitboard)] = pos;
36              }
37              cout << format("uint8_t positions[{}] = {{", size);
38              for (int i = 0; i < size; i++) {
39                  cout << table[i] << ',';
40              }
41              cout << "};\n";
42              break;
43          }
44      }
45  }
```

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 set 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 set 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 shall see another approach to succeed with `K == 6` later in this chapter).

- On lines 14 to 16, the `index` function calculates the index for key `bitboard` (this bitboard contains only one set bit) using the magic perfect hashing formula.

- The **`while`** loop, lines 17 to 44, 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 43, 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 set bit for each bitboard. The output of this program for a given execution is shown below.

```
$ g++ -std=c++20 -O3 -Wall -Wpedantic -march=native\
find_magic.cpp -o find_magic
$ ./find_magic
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 set 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 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 use an Satisfiability Modulo Theories (SMT) solver that allows us to set constraints before searching the entire space. Setting constraints first prunes the search space. Note that we have no guarantee the search is fast, but for this problem, finding a `MAGIC` constant for `K == 6` is 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 set 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.

```
$ python find_magic.py
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 microbenchmark 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_countr_zero_positions` function using a dedicated instruction is the most efficient (but is of no use for move generation).

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

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);
```

26

```
31  static void BM_unordered_map_positions(benchmark::State& state) {
32      std::unordered_map<uint64_t, uint8_t> map;
33      for (uint8_t i = 0; i < 64; i++) {
34          map[1ULL << i] = i;
35      }
36      auto samples = generate_isolated_bits();
37      size_t idx = 0;
38      for (auto _ : state) {
39          uint64_t bitboard = samples[idx++ & 63];
40          benchmark::DoNotOptimize(map[bitboard]);
41      }
42      state.SetItemsProcessed(state.iterations());
43  }
44  BENCHMARK(BM_unordered_map_positions);
45  static void BM_countr_zero_positions(benchmark::State& state) {
46      auto samples = generate_isolated_bits();
47      size_t idx = 0;
48      for (auto _ : state) {
49          uint64_t bitboard = samples[idx++ & 63];
50          benchmark::DoNotOptimize(std::countr_zero(bitboard));
51      }
52      state.SetItemsProcessed(state.iterations());
53  }
54  BENCHMARK(BM_countr_zero_positions);
55  BENCHMARK_MAIN();
```

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

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic\
positions_bench.cpp -o positions -lbenchmark
$ ./positions
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: 1.59, 1.78, 1.63
--------------------------------------------------------------
Benchmark                        Time         CPU      Iterations
```

```
    -------------------------------------------------------------
    BM_magic_positions              0.552 ns   0.552 ns    1000000000
    BM_unordered_map_positions       31.1 ns    31.1 ns      22584016
    BM_countr_zero_positions        0.491 ns   0.491 ns    1000000000
```

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 white crosses in figure 2.3, we would like to very efficiently obtain the bitboard shown in table 2.11 indicating the different possible moves.
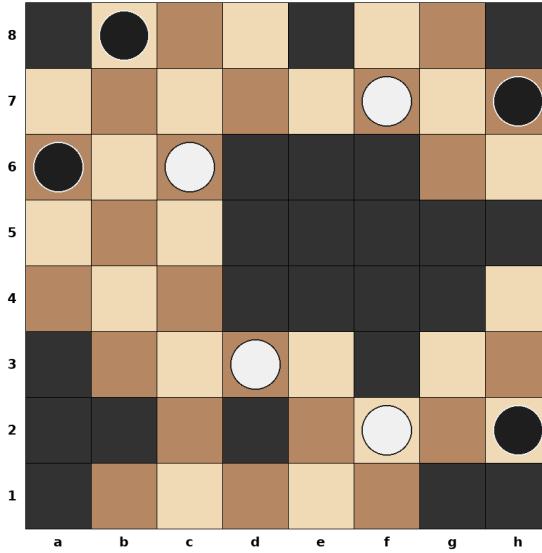
Figure 2.2: Board position example (after move 23 in figure 1.8)

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 2.12. Let `mask` be the mask from table 2.12 and `board` be the bitboard from table 2.13 representing the game board from figure 2.2 where set bits represent the position of black or white pieces and holes. 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 2.14. There is a set bit 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 2.11. Then we only need to iterate through each set bit in
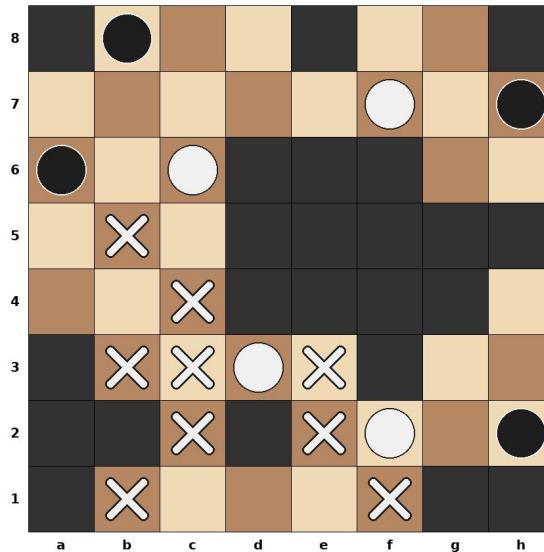
Figure 2.3: Possible moves (white crosses) for the white piece at d3

Table 2.11: Bitboard of possible moves for the piece at d3 (see figure 2.3)

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

29

Table 2.12: Mask for reachable squares for the piece at d3 without considering obstacles

	a	b	c	d	e	f	g	h
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 2.13: Board bitboard

	a	b	c	d	e	f	g	h
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 2.14: 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

this bitboard to know which squares we can move to and create the corresponding move. To obtain the index in `MOVES_D3`, we 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 need to enumerate all possible obstacle placements on the squares reachable by the piece at `d3`. This means we must enumerate all subsets of set bits of the bitboard in table 2.12. Eight subsets from this enumeration are presented in figure 2.4. There are 25 squares on the trajectory of the piece at `d3` (see table 2.12), 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 shall 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 depends on this number of configurations.

To reduce this number of configurations, we do not consider the squares on the board edges to remove the set bits on ranks 1 and 8 and files a and h from the mask in table 2.12. We obtain the bitboard in table 2.15. We are missing information about the edges, but we assume that if it is possible to play to `d7` for example, it is also possible to play to `d8`. Of course there might be an obstacle at `d8`, but we can easily eliminate this impossible move as we shall 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.

Indeed, we can split a piece's moves into diagonal and orthogonal moves. We

	a	b	c	d	e	f	g	h
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	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) Empty (0 bits)

	a	b	c	d	e	f	g	h
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	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

(b) 1 bit set

	a	b	c	d	e	f	g	h
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	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

(c) 2 bits set

	a	b	c	d	e	f	g	h
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

(d) 3 bits set

	a	b	c	d	e	f	g	h
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

(e) 9 bits set

	a	b	c	d	e	f	g	h
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

(f) 25 bits set (full)

Figure 2.4: Six subsets of the mask for d3 showing different occupancy patterns (0 to 25 bits set) from the 2^{25} possible ones

Table 2.15: Mask for reachable squares for the piece at **d3** excluding board edges

	a	b	c	d	e	f	g	h
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

obtain two tables: `MOVES_D3_DIAG` and `MOVES_D3_ORTHO`. We need to consult two tables instead of one to obtain the possible moves at **d3** (or any other square), but we save a lot of memory space. We shall see how to combine the values from these two tables later in this chapter. Instead of the mask from table 2.15, we now have the two masks from figure 2.5.

For the orthogonal mask in figure 2.5a, we have $2^{10} = 1,024$ subsets and for the diagonal mask in figure 2.5b, 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**!

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]We 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.

(a) Orthogonal mask

	a	b	c	d	e	f	g	h
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

(b) Diagonal mask

	a	b	c	d	e	f	g	h
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

Figure 2.5: Orthogonal and diagonal masks for d3 without edge squares

```cpp
9
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 1ULL << 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(reachable_squares(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,

- On line 39, the function **`uint64_t`** `reachable_squares(MoveType mt, Square sq,` **`uint64_t`** `occupied)` allows us to create the mask of squares reachable from square `sq` for the move type `mt`, which is either orthogonal (`ORTHOGONAL` on line 35) or diagonal (`DIAGONAL` on line 36). The created mask is similar to the masks in figure 2.5.

  - On 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.

  - On 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 allows 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`).

  - On line 48, we transform the destination square `to` into a bitboard that contains a single 1, on the bit corresponding to the square.

  - On 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.

  - On line 50, we add the new square as a possible move.

  - On line 51, we move one square in direction `d`.

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

  - On lines 59 to 60, we create a mask that covers 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 2.6a and if `sq == SQ_C2`, we obtain the one from figure 2.6b.

  - On 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.

  - On lines 62 to 71, we 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`.

    * On lines 62 and 63, the vector `occupancies` contains all obstacle configurations on the piece's trajectory, without considering the squares

37

(a) Edge mask for square a1

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

(b) Edge mask for square c2

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

Figure 2.6: Edge masks excluding the rank and file of the piece

in `edge`, and for each of these configurations, the vector `possible_moves` contains the bitboard of possible moves for that configuration. For example, we could have as an obstacle configuration the bitboard from figure 2.7a and the associated possible moves would be represented by the bitboard from figure 2.7b.

(a) Occupancy configuration example

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

(b) Possible moves for this occupancy

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

Figure 2.7: Example of occupancy configuration and corresponding possible moves

* The loop between lines 66 and 71 allows us to enumerate all subsets of the bitboard `moves_bb`. To do this, we use the technique called *Carry-*

Rippler[12] on line 70. Let's take a fictitious example with `moves_bb = 0b1011`. We then obtain

```
uint64_t b = 0;                      // b == 0
b = (0 - 0b1011) & 0b1011;           // b == 0b0001
b = (0b0001 - 0b1011) & 0b1011;      // b == 0b0010
b = (0b0010 - 0b1011) & 0b1011;      // b == 0b0011
b = (0b0011 - 0b1011) & 0b1011;      // b == 0b1000
b = (0b1000 - 0b1011) & 0b1011;      // b == 0b1001
b = (0b1001 - 0b1011) & 0b1011;      // b == 0b1010
b = (0b1010 - 0b1011) & 0b1011;      // b == 0b1011
b = (0b1011 - 0b1011) & 0b1011;      // b == 0
```

  * On 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 appears in the possible moves.

– On line 72, we count the number of set 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] is 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 lets us find the different `MAGIC` constants very quickly and the memory consumed is relatively small (less than 110 KiB in total).

– On lines 74 to 76, we set up the random generator that allows us to create the `MAGIC` constants.

– On line 77, the array `seen` allows us to memorize the indices, obtained by the formula on line 84, that we have already encountered.

– On line 78, the array `moves` allows 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 lets us avoid treating as collisions two different configurations that lead to the same index and have the same possible moves.

– The loop from line 79 to 96 searches 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, on line 97, that this part of the code is never reached thanks to the function `std::unreachable`.

– On 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 set bits. This trick is very important because without it we couldn't find the `MAGIC` constants in a reasonable time![14]

---

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

[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.

- On 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.

  * On line 84, for a given obstacle configuration, we calculate its index index using the perfect hash function: (magic * occ >> (64 - k)).
  * On 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]) lets us ignore a collision that produces the same possible moves.

- On lines 100 to 116, the main produces and displays 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.

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic chapter02.cpp\
-o chapter02
$ ./chapter02
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,
    0xa000056210280400,   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,   0x828004801844400,
    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,      0x1000022020500880b,
    0x1000000000460802,    0x21084104410,         0x8000001053300104,
    0x4000182008c20048,    0x112088105020200,
};
```

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

```
1  uint64_t orthogonalTable[102400];
2  uint64_t diagonalTable[5248];
3  Magic orthogonalMagics[SQUARE_NB];
4  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 can use them to initialize the tables described in listing 14, which store the necessary information allowing us to generate possible moves from a given square and an obstacle configuration.

- On line 1, the `orthogonalTable` array contains 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];
}
```

- On line 2, the `diagonalTable` array does the same but for diagonal moves.

- On line 3, the `orthogonalMagics` array defines 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.

- On 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 used to store the information for applying the perfect hashing for a given square.

- On line 2, the `mask` attribute is the bitboard used 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 on line 61 of listing 12.

- On line 3, the `magic` attribute is the `MAGIC` constant that we found for the considered square and movement. This attribute takes the value of one of the constants from the `O_MAGIC` or `D_MAGIC` tables from listing 13.

- On line 4, the `moves` attribute is a pointer to the part of the `orthogonalTable` or `diagonalTable`, depending on the movement type, that contains the possible moves for the considered square.

- On line 5, the `shift` attribute is the shift (`64 - K`) in the formula from listing 7.

- On 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 uses 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 is used for move generation in section 2.3.4. To do this,

- On line 2, we compute the index `idx_omoves` using the perfect hashing, by using the `Magic` structure for the square `sq` and orthogonal moves.

- On line 3, we compute the index `idx_dmoves` using the perfect hashing, by using the `Magic` structure for the square `sq` and diagonal moves.

- On 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_all_magics() {
2      init_magics(ORTHOGONAL, orthogonalTable, orthogonalMagics);
3      init_magics(DIAGONAL, diagonalTable, diagonalMagics);
4  }
```

Listing 17: Initialisation of all the magic bitboards

The function **void** `init_all_magics()` from listing 17 initializes once and for all, at the beginning of the program, the different tables. We simply call on 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 = reachable_squares(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(reachable_squares(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: Initialization of the various tables used to obtain the bitboard of possible moves based on the obstacle configuration

The function **void** init_magics(MoveType mt,**uint64_t** table[],Magic magics[]) initializes 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.

- On line 8, the occupancies array contains all obstacle configurations for the considered square for each iteration of the loop on line 10. The possible_moves array contains 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 on line 10. However, it is more efficient to call the clear method on lines 11 and 12, which avoids unnecessary allocations.

- The loop comprising lines 10 to 32 updates 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 update the Magic structure m from line 13, corresponding to the square sq, with

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

- On line 20, we compute the address of the beginning of the region in the `table` array reserved for storing 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, on 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 is quite easy to generate the list of possible moves in a given position of the game. But before doing that, we first describe how to represent a move.

2.3.3 Move Representation

We represent a move by a pair, whose first element is the departure square and the second element is the arrival square. For example, the game shown in figures 1.7, 1.8, and 1.9 is textually represented by the following moves:

```
a1:a2  h1:f3  h8:f6  a8:c6  d5:d6  d4:b2  d6:e6  e5:g5  f6:f5  f3:g4
f5:h7  g5:h5  e4:f4  h5:f7  e6:e8  g4:g1  e8:b8  b2:d2  a2:a3  g1:f2
a3:a6  d2:d3  f4:h2  d3:b5  a6:a4  b5:b3  a4:c4  b3:c3  h2:g2  f2:f1
b8:c7  c3:a5  c4:c5  c6:d7  g2:g3  f7:g7  c5:b4  a5:b6  g3:h4  b6:b7
h7:g8  d7:d8  c7:c8  f1:b1  g8:f8  d8:e7  h4:h3  a1:a1  b1:c1  a1:a1
c1:e1  a1:a1  e1:e3  a1:a1  e3:e2  a1:a1  e2:d1  a1:a1  d1:c2  a1:a1
g7:g6  a1:a1  g6:h6  a1:a1  b7:a7
```

```
1  class Move {
2      uint16_t data;
3  public:
4      constexpr Move() = default;
5      constexpr explicit Move(uint16_t d) : data(d) {}
6      constexpr explicit Move(Square from,
7                              Square to)
8          : data((to << 6) + from) {}
9      constexpr Square from_sq() const {
10         return Square(data & 0x3F);
11     }
```

```cpp
12    constexpr Square to_sq() const {
13        return Square((data >> 6) & 0x3F);
14    }
15    static constexpr Move none() { return Move(0); }
16    constexpr bool operator==(const Move& m) const {
17        return data == m.data;
18    }
19    constexpr bool operator!=(const Move& m) const {
20        return data != m.data;
21    }
22    constexpr explicit operator bool() const {
23        return data != 0;
24    }
25    constexpr uint16_t raw() const { return data; }
26 };
27
28 std::ostream& operator<<(std::ostream& os, const Move& m) {
29    static constexpr std::string_view square2string[SQUARE_NB] = {
30        "a1", "b1", "c1", "d1", "e1", "f1", "g1", "h1",
31        "a2", "b2", "c2", "d2", "e2", "f2", "g2", "h2",
32        "a3", "b3", "c3", "d3", "e3", "f3", "g3", "h3",
33        "a4", "b4", "c4", "d4", "e4", "f4", "g4", "h4",
34        "a5", "b5", "c5", "d5", "e5", "f5", "g5", "h5",
35        "a6", "b6", "c6", "d6", "e6", "f6", "g6", "h6",
36        "a7", "b7", "c7", "d7", "e7", "f7", "g7", "h7",
37        "a8", "b8", "c8", "d8", "e8", "f8", "g8", "h8",
38    };
39    os << square2string[m.from_sq()] << ':'
40        << square2string[m.to_sq()];
41    return os;
42 }
```

Listing 19: Move representation, the departure and arrival squares are each stored on 6 bits

The `Move` class, presented in listing 19, represents a move using an unsigned 16-bit integer: **uint16_t** data. We need 6 bits to represent the departure square and 6 other bits to represent the arrival square.

- On lines 6-8, we construct the move from the departure square `from` and the arrival square `to`. We place the departure square number in the 6 least significant bits of `data` and the arrival square starting from bit 6.

- On line 9, the function `Square from_sq()` retrieves the square whose number is located in bits 0 to 5 of `data`. This corresponds to the departure square.

47

- On line 12, the function `Square to_sq()` retrieves the square whose number is located in bits 6 to 11 of `data`. This corresponds to the arrival square.

- We represent passing one's turn, which occurs when a player can no longer move but the opponent still has moves available, by the fictitious move `a1:a1`. The function `Move none()`, described on line 15, returns this move. The departure square and arrival square are both `SQ_A1 == 0`.

- On line 28, the function `ostream& operator<<(std::ostream& os, const Move& m)` displays a move in the format: `departure square:arrival square`.

2.3.4 List of Moves for a Given Board Configuration

The list of possible moves in a given game configuration is represented by the `MoveList` class shown in listing 20.

- On line 1, the constant `MAX_NB_MOVES` is an upper bound on the maximum number of possible moves for the players. In the starting configuration, the black player has a choice of $14 \times 4 = 56$ moves. Note that this number of possible moves does not necessarily decrease; for example, in the configuration shown in figure 2.8, the black player has a choice of 59 moves. To obtain this upper bound, we ran many random games, observed the maximum number of available moves for the players during these games, and added two to it.
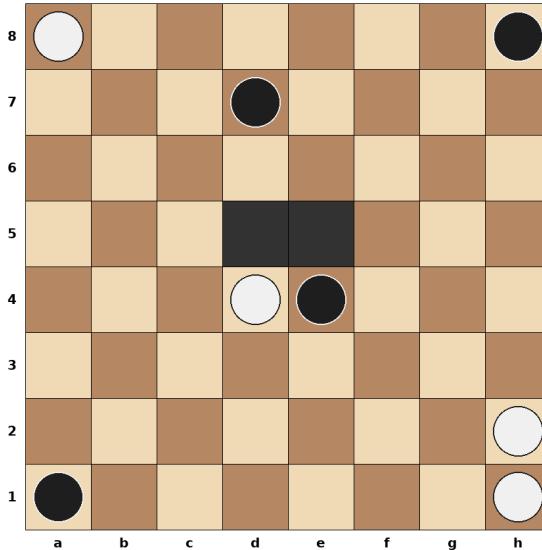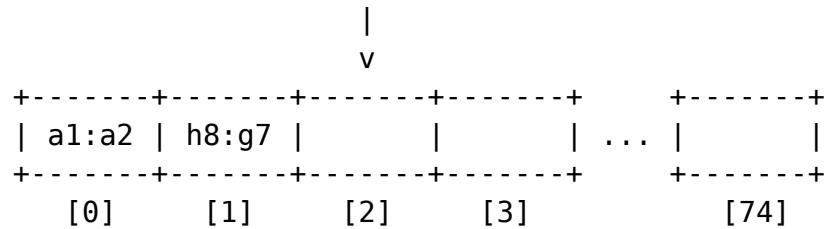

Figure 2.8: In this position, the black player has a choice of 60 moves

- On line 3, moves are stored in the `move_list` array which, in our usage as we shall see later, is placed on the stack. The `last` pointer points to the cell just after the last move (which is more convenient for implementation). For example, if the current player only has the moves `Move(SQ_A1, SQ_A2)` and `Move(SQ_H8, SQ_G7)`, we have the following configuration:

`last`

```
                        |
                        v
    +-------+-------+-------+-------+     +-------+
    | a1:a2 | h8:g7 |       |       | ... |       |
    +-------+-------+-------+-------+     +-------+
       [0]     [1]     [2]     [3]           [74]
```

- On lines 6 to 11, the `begin` and `end` functions let us iterate over a `MoveList` object. For example, we can write:

  ```
  MoveList list;
  // ...
  for (Move m : list) {
      // ...
  }
  ```

- The remaining functions allow us to retrieve the size (`size` on lines 12 to 14), index the list like an array (**operator[]** on lines 15 to 18), and directly obtain the pointer to the beginning of the `move_list` array (`data` on line 19).

```cpp
static constexpr uint16_t MAX_NB_MOVES = 75;
class MoveList {
    Move move_list[MAX_NB_MOVES], *last;
public:
    constexpr MoveList() : last(move_list) {}
    constexpr const Move* begin() const {
        return move_list;
    }
    constexpr const Move* end() const { return last; }
    constexpr Move* begin() { return move_list; }
    constexpr Move* end() { return last; }
    constexpr size_t size() const {
        return last - move_list;
    }
    constexpr const Move& operator[](size_t i) const {
        return move_list[i];
    }
    constexpr Move& operator[](size_t i) { return move_list[i]; }
    constexpr Move* data() { return move_list; }
    friend class Yolah;
};
```

Listing 20: Class representing a list of moves

The function **void** Yolah::moves(**uint8_t** player, MoveList& moves) presented on line 19 of listing 21, stores in the moves parameter of type MoveList the possible moves in the current game configuration for the player passed as parameter (player) (which is either BLACK or WHITE)[15]. In this function,

- The loop between lines 23 and 29 iterates over each piece of player player, whose positions are given in the bitboard bb, and generates all possible moves for each of them.

  - On line 24, we retrieve the square corresponding to the position of the rightmost set bit in the bitboard bb. We use the function Square pop_lsb(**uint64_t**& b) defined on line 4 for this purpose. This function returns the square corresponding to the position of the rightmost set bit in the bitboard b and, additionally, clears that bit. For example,

    ```
    uint64_t bb = 0b1000100100010000;
    Square sq = pop_lsb(bb);
    // sq == SQ_E1
    // bb ==      0b1000100100000000
    ```

  - On line 25, we use our magic bitboard function moves_bb to generate the bitboard containing all possible moves for this position and obstacle configuration. We mentioned in paragraph 2.3.2 on page 31 that we would need to handle potential obstacles on the edges that we had ignored. Remember that to limit the number of obstacle configurations to consider, we had ignored the board edges. As a consequence, we might allow moves to edge squares even when there are obstacles there. It is very easy to eliminate these potentially illegal moves by removing moves that land on occupied squares using: & ~occupied.

  - On lines 26 to 28, we iterate through the bitboard b of possible moves and again use the pop_lsb function to retrieve the destination squares of the various moves (the starting position being the from position of the piece under consideration), in order to construct a Move object that we place in the moves.move_list array.

- On lines 30 to 32, we check whether the player had at least one move available, by checking if the move_list pointer still points to the beginning of the moves.move_list array. If so, which is not the most frequent occurrence[16], we add the pass move (Move::none()) as the only possible move for the player.

- Finally, on line 33, we set the end pointer moves.last to the cell just after the one containing the last move.

```
1  constexpr Square lsb(uint64_t b) {
2      return Square(std::countr_zero(b));
```

---

[15]It proves useful later in this book to be able to retrieve a player's possible moves even when it is not their turn to play.

[16]We indicate this using the C++ [[unlikely]] attribute.

```
3  }
4  Square pop_lsb(uint64_t& b) {
5      const Square s = lsb(b);
6      b &= b - 1;
7      return s;
8  }
9  class Yolah {
10     uint64_t black = BLACK_INITIAL_POSITION;
11     uint64_t white = WHITE_INITIAL_POSITION;
12     uint64_t holes = 0;
13     uint8_t black_score = 0;
14     uint8_t white_score = 0;
15     uint8_t ply = 0;
16 public:
17     // ...
18 };
19 void Yolah::moves(uint8_t player, MoveList& moves) const {
20     Move* move_list = moves.move_list;
21     uint64_t occupied = black | white | holes;
22     uint64_t bb = player == BLACK ? black : white;
23     while (bb) {
24         Square   from = pop_lsb(bb);
25         uint64_t b    = moves_bb(from, occupied) & ~occupied;
26         while (b) {
27             *move_list++ = Move(from, pop_lsb(b));
28         }
29     }
30     if (move_list == moves.move_list) [[unlikely]] {
31         *move_list++ = Move::none();
32     }
33     moves.last = move_list;
34 }
35 constexpr uint8_t Yolah::current_player() const {
36     return ply & 1;
37 }
38 void Yolah::moves(MoveList& moves) const {
39     this->moves(current_player(), moves);
40 }
```

Listing 21: Generating possible moves for a player in the current game position

The function **uint8_t** current_player() on line 35 of listing 21 returns the current player. To do this, it tests whether the number of moves played since the begin-

ning of the game is even or odd. The black player plays in positions where `ply` equals
0, 2, 4, 6, …, while the white player plays in positions where `ply` equals 1, 3, 5, 7, ….
We test parity by examining the least significant bit of `ply`; using (`ply & 1`) yields:
0 when even (which corresponds to `BLACK`) and 1 when odd (which corresponds to
`WHITE`).

2.3.5   Making and Unmaking Moves

To complete the Yolah implementation, we still need to be able to apply a move to
the game. The function **void** `play(Move m)` in listing 22 modifies the board state by
playing the move `m` passed as parameter.

- On line 2, we check that the player is not passing their turn[17]; we use the
  `[[likely]]` attribute because it is much more common for the player to be able
  to play. Whether they pass or not, we increment, on line 14, the `ply` attribute
  which we use to count the number of moves played in the game and to determine
  the current player.

- On line 3, `pos1` is the bitboard corresponding to the move's source square.

- On line 4, `pos2` is the bitboard corresponding to the move's destination square.

- On line 5, we test if it is white's turn to play.

  - If so, we toggle, on line 6, using XOR, the bits corresponding to the source
    and destination squares of the move in the `white` bitboard which gives the
    position of the white player's pieces. For example,

    ```
    uint64_t white = 0b1000100100010000;
    Move m = Move(SQ_E1, SQ_G1);
    uint64_t pos1 = square_bb(m.from_sq());
    // pos1 ==            0b010000;
    uint64_t pos2 = square_bb(m.to_sq());
    // pos2 ==            0b100000;
    // pos1 | pos2 ==     0b110000;
    // white == 0b1000100100010000;
    white ^= pos1 | pos2;
    // white == 0b1000100100100000;
    ```

    Then on line 7, we increase the white player's score because they have just
    made a move.

  - If it is black's turn to play, we proceed similarly, but for the black player's
    `black` bitboard on line 9, and we increase their score on line 10.

- On line 12, we eliminate the source square of the move by adding a hole to the
  `holes` bitboard at the position corresponding to the move's source square.

---

[17]Which they must do if and only if they have no valid move available.

```
1  void Yolah::play(Move m) {
2      if (m != Move::none()) [[likely]] {
3          uint64_t pos1 = square_bb(m.from_sq());
4          uint64_t pos2 = square_bb(m.to_sq());
5          if (ply & 1) {
6              white ^= pos1 | pos2;
7              white_score++;
8          } else {
9              black ^= pos1 | pos2;
10             black_score++;
11         }
12         holes |= pos1;
13     }
14     ply++;
15 }
```

Listing 22: The move `m` is played in the current game position

It proves useful for several of our AIs to be able to undo the previously played move. The function **void** undo(Move m) in listing 23 performs the inverse operation of that performed by the function **void** play(Move m) in order to restore the game board to the state it was in before playing move m. Typically, we use undo as follows:

```
/* return type */ recursive(Yolah& yolah, /* other parameters */) {
    // ...
    MoveList moves;
    yolah.moves(moves);
    for (Move m : moves) {
        yolah.play(m);
        recursive(yolah, /* ... */);
        yolah.undo(m);
    }
    // ...
    return // ...
}
```

In the previous listing, it is more efficient to use the undo function than to save the complete game state before the recursive call and then restore it afterward. The benchmark in listing 24 tests this assertion. The function count_nodes_with_undo on line 1 tests state restoration using the undo function and makes a number of recursive calls controlled by the depth parameter.
The function count_nodes_with_restore on line 15 tests restoration by saving the complete game state on line 23. The previous state is restored via the assignment on line 28.

53

The results are given in listing 25. The performance gain would be even more pronounced for a game with a larger state than Yolah's.

```
1  void Yolah::undo(Move m) {
2      ply--;
3      if (m != Move::none()) [[likely]] {
4          uint64_t pos1 = square_bb(m.from_sq());
5          uint64_t pos2 = square_bb(m.to_sq());
6          if (ply & 1) {
7              white ^= pos1 | pos2;
8              white_score--;
9          } else {
10             black ^= pos1 | pos2;
11             black_score--;
12         }
13         holes ^= pos1;
14     }
15 }
```

Listing 23: The last played move m is undone

```
1  uint64_t count_nodes_with_undo(Yolah& yolah, int depth) {
2      if (depth == 0 || yolah.game_over()) {
3          return 1;
4      }
5      uint64_t count = 0;
6      MoveList moves;
7      yolah.moves(moves);
8      for (const Move& m : moves) {
9          yolah.play(m);
10         count += count_nodes_with_undo(yolah, depth - 1);
11         yolah.undo(m);
12     }
13     return count;
14 }
15 uint64_t count_nodes_with_restore(Yolah& yolah, int depth) {
16     if (depth == 0 || yolah.game_over()) {
17         return 1;
18     }
19     uint64_t count = 0;
```

```
20        MoveList moves;
21        yolah.moves(moves);
22        // Save the complete state
23        Yolah saved = yolah;
24        for (const Move& m : moves) {
25            yolah.play(m);
26            count += count_nodes_with_restore(yolah, depth - 1);
27            // Restore the complete state
28            yolah = saved;
29        }
30        return count;
31    }
32    static void BM_recursive_with_undo(benchmark::State& state) {
33        int depth = state.range(0);
34        for (auto _ : state) {
35            Yolah yolah;
36            uint64_t nodes = count_nodes_with_undo(yolah, depth);
37            benchmark::DoNotOptimize(nodes);
38        }
39        // Calculate nodes for reporting
40        Yolah yolah;
41        uint64_t nodes = count_nodes_with_undo(yolah, depth);
42        state.SetItemsProcessed(state.iterations() * nodes);
43        state.counters["nodes"] = nodes;
44    }
45    static void BM_recursive_with_restore(benchmark::State& state) {
46        int depth = state.range(0);
47        for (auto _ : state) {
48            Yolah yolah;
49            uint64_t nodes = count_nodes_with_restore(yolah, depth);
50            benchmark::DoNotOptimize(nodes);
51        }
52        Yolah yolah;
53        uint64_t nodes = count_nodes_with_restore(yolah, depth);
54        state.SetItemsProcessed(state.iterations() * nodes);
55        state.counters["nodes"] = nodes;
56    }
57    // Test at various depths
58    BENCHMARK(BM_recursive_with_undo)->Arg(3)->Arg(4)->Arg(5);
59    BENCHMARK(BM_recursive_with_restore)->Arg(3)->Arg(4)->Arg(5);
60    int main(int argc, char** argv) {
61        init_all_magics();
```

```
62      benchmark::Initialize(&argc, argv);
63      benchmark::RunSpecifiedBenchmarks();
64  }
```

Listing 24: Benchmark to test the assertion that the `undo` function provides a performance gain over saving the complete game state

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic\
undo_vs_restore_bench.cpp -o undo_vs_restore_bench -lbenchmark
$ ./undo_vs_restore_bench
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)

Benchmark                       Time (ms)   Throughput
----------------------------------------------------------
BM_recursive_with_undo/3            1.09     150.4 M/s
BM_recursive_with_undo/4           56.50     153.1 M/s
BM_recursive_with_undo/5         3059.00     149.3 M/s
BM_recursive_with_restore/3         1.24     132.3 M/s
BM_recursive_with_restore/4        70.00     123.6 M/s
BM_recursive_with_restore/5      3492.00     130.8 M/s
```

Listing 25: Benchmark results from listing 24

2.4   Testing with Random Games

We present two ways of testing our implementation of Yolah. Both approaches are based on playing random games. First, we visually inspect whether everything appears correct, and once any "easy" bugs have been detected, we move on to a large battery of tests, again by generating random games, but this time by checking game properties that should be respected by our implementation and by comparing the move generation based on magic bitboards with a much simpler (but far less efficient) implementation, to verify that both different implementations produce the same results. Once this battery of tests has been passed successfully, we profile the code by generating a set of random games, and based on this analysis, we improve the program's execution time.

2.4.1   Testing by Observing Random Games

The code for visually testing our implementation is given in Listing 26. The beginning of an execution of this code is shown in Listing 27.

- On lines 4 and 5, we initialize a pseudo-random generator, either with a seed given by the user (parameter `seed`) to make the test reproducible, or with a random number using `random_device`[18].

- The loop on lines 6 to 25 plays `nb_games` random games.

  - On line 7, we initialize a new game.
  - On lines 8 to 23, we play the random game.

    * On lines 9 and 10, we retrieve the list of possible moves in the current game position and sort them to make the display of possible moves more readable.
    * On lines 11 to 14, we display the number of possible moves and then the list of all moves. You can see the result in Listing 27.
    * On line 16, we display the board with the moves represented by crosses. This makes it easy to see if any moves are missing or if there are extra moves.
    * On lines 17 to 19, we randomly choose a move and display it.
    * On lines 20 and 21, we wait for the Enter key to be pressed before continuing the loop. This gives us time to analyze the outputs.

```cpp
void play_random_games(size_t nb_games,
                       optional<uint64_t> seed = nullopt) {
    MoveList moves;
    random_device rd;
    mt19937 mt(seed.value_or(rd()));
    for (size_t i = 0; i < nb_games; i++) {
        Yolah yolah;
        while (!yolah.game_over()) {
            yolah.moves(moves);
            sort(begin(moves), end(moves));
            cout << format("# moves: {}\n", moves.size());
            for (const auto& m : moves) {
                cout << m << ' ';
            }
            cout << "\n\n";
            cout << YolahWithMoves(yolah, moves) << '\n';
            uniform_int_distribution<uint64_t> d(0, moves.size()-1);
            Move m = moves[d(mt)];
            cout << m << '\n';
            string _;
```

---

[18]This generator produces high-quality random numbers, using for example `/dev/urandom` on Linux. However, obtaining the random number is rather slow compared to using a pseudo-random generator such as `mt19937`. A typical use of `random_device` is to seed a pseudo-random generator, as we do here.

```
21              getline(cin, _);
22              yolah.play(m);
23          }
24          cout << yolah << '\n';
25      }
26 }
27 int main() {
28      init_all_magics();
29      play_random_games(42);
30 }
```

Listing 26: Playing random games, displaying the list of moves and the board to check the engine visually

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic chapter02.cpp\
-o chapter02
$ ./chapter02
Black player

8 | ○ | . | . | . | . | . | . | ● |

7 | . | . | . | . | . | . | . | . |

6 | . | . | . | . | . | . | . | . |

5 | . | . | . | ● | ○ | . | . | . |

4 | . | . | . | ○ | ● | . | . | . |

3 | . | . | . | . | . | . | . | . |

2 | . | . | . | . | . | . | . | . |

1 | ● | . | . | . | . | . | . | ○ |

    a   b   c   d   e   f   g   h
score: 0/0

# moves: 56
a1:b1 a1:c1 a1:d1 a1:e1 a1:f1 a1:g1 a1:a2 a1:b2 a1:a3 a1:c3 a1:a4 a1:a5
a1:a6 a1:a7 e4:b1 e4:e1 e4:c2 e4:e2 e4:g2 e4:d3 e4:e3 e4:f3 e4:f4 e4:g4
e4:h4 e4:f5 e4:g6 e4:h7 d5:a2 d5:b3 d5:c4 d5:a5 d5:b5 d5:c5 d5:c6 d5:d6
d5:e6 d5:b7 d5:d7 d5:f7 d5:d8 d5:g8 h8:h2 h8:h3 h8:h4 h8:h5 h8:f6 h8:h6
h8:g7 h8:h7 h8:b8 h8:c8 h8:d8 h8:e8 h8:f8 h8:g8


Black player
```

```
8 | ○ | X | X | X | X | X | X | ● |
7 | X | X | · | X | · | X | X | X |
6 | X | · | X | X | X | X | X | X |
5 | X | X | X | ● | ○ | X | · | X |
4 | X | · | X | ○ | ● | X | X | X |
3 | X | X | X | X | X | X | · | X |
2 | X | X | X | · | X | · | X | X |
1 | ● | X | X | X | X | X | X | ○ |
    a   b   c   d   e   f   g   h
score: 0/0
```

d5:a2

White player

```
8 | ○ | · | · | · | · | · | · | ● |
7 | · | · | · | · | · | · | · | · |
6 | · | · | · | · | · | · | · | · |
5 | · | · | · |   | ○ | · | · | · |
4 | · | · | · | ○ | ● | · | · | · |
3 | · | · | · | · | · | · | · | · |
2 | ● | · | · | · | · | · | · | · |
1 | ● | · | · | · | · | · | · | ○ |
    a   b   c   d   e   f   g   h
score: 1/0
```

moves: 55
h1:b1 h1:c1 h1:d1 h1:e1 h1:f1 h1:g1 h1:g2 h1:h2 h1:f3 h1:h3 h1:h4 h1:h5
h1:h6 h1:h7 d4:d1 d4:g1 d4:b2 d4:d2 d4:f2 d4:c3 d4:d3 d4:e3 d4:a4 d4:b4
d4:c4 d4:c5 d4:b6 d4:a7 e5:h2 e5:g3 e5:f4 e5:f5 e5:g5 e5:h5 e5:d6 e5:e6
e5:f6 e5:c7 e5:e7 e5:g7 e5:b8 e5:e8 a8:a3 a8:a4 a8:a5 a8:a6 a8:c6 a8:a7
a8:b7 a8:b8 a8:c8 a8:d8 a8:e8 a8:f8 a8:g8

```
White player

   8 | ○ | x | x | x | x | x | x | ● |

   7 | x | x | x | · | x | · | x | x |

   6 | x | x | x | x | x | x | · | x |

   5 | x | · | x |   | ○ | x | x | x |

   4 | x | x | x | ○ | ● | x | · | x |

   3 | x | · | x | x | x | x | x | x |

   2 | ● | x | · | x | · | x | x | x |

   1 | ● | x | x | x | x | x | x | ○ |

       a   b   c   d   e   f   g   h
score: 1/0

h1:f1
```

Listing 27: Beginning of the execution of the program from Listing 26

2.4.2   Differential Testing and Property-Based Testing

Observing the previous random games allows us to quickly detect potential bugs. This approach is very convenient for seeing in detail how our program works. But we also want a way to automatically test our game. To do so, we test the game by checking properties that it must satisfy. We also generate possible moves in a simple (and inefficient) way in which we have high confidence, precisely because it is straight-forward to implement, and compare the moves generated by our game with those produced by the simple implementation. All these tests are performed on randomly generated games. This way, we can easily produce a very large number of tests. This testing approach is known as *property-based testing* [10] and *differential testing* [11]. Property-based testing consists of verifying that certain invariant properties are always respected during the execution of randomly generated scenarios. Differential testing compares the results of two different implementations of the same algorithm: an optimized implementation (our move generator with magic bitboards) and a simple but obviously correct reference implementation.

The code in Listing 28 illustrates this testing approach.

- On lines 8 to 12, the `TestResult` structure holds the result of a given test. The `passed` attribute is true when the test succeeds and false otherwise. The `message` attribute stores the diagnostic that is used in case of failure (see the `run_test` function on line 165).

60

- On line 14, the `pass` function returns a `TestResult` indicating that the test ran successfully.

- On line 15, the `fail(string msg)` function returns a `TestResult` indicating that the test failed with the error diagnostic `msg`.

- On lines 17 to 44, the `slow_moves_generation` function is defined, which returns the list of possible moves in the game configuration `yolah` passed as parameter (line 182). This function serves as the reference for the differential tests. As we mentioned, this way of generating moves is much simpler than the magic bitboard approach. We have high confidence in the correctness of `slow_moves_generation`. The loops on lines 19 and 20 iterate over all squares on the board. We construct the move's source square `from` on line 21, then check on line 22 that this square contains a piece belonging to the player for whom we are generating the move list. The functions `square_of` and `get` are defined precisely in Listings 39–42, but `Square square_of(int rank, int file)` creates a square from coordinates `(rank, file)` and `uint8_t Yolah::get(Square sq)` returns the type of element (one of the players, a hole, or a free square) at square `sq`. The loops on lines 23 and 24 iterate over all possible directions for the piece, taking care to filter out the null displacement on line 25. Finally, the loop from line 28 to 37 moves in the chosen direction `(di, dj)` as long as we do not leave the board and there are no obstacles. If we leave the board on line 29, or an obstacle is detected on line 33, we break out of the loop to try a new direction. We do not forget on line 42 to add `Move::none()` if no move was possible. This way of building all possible moves is very intuitive (but inefficient).

- On lines 46 to 53, the `check_move_count` function is our first differential test. The first parameter `fast` is the move list obtained using magic bitboards, and the second parameter `expected` is the move list obtained with the previous function. In this test, we verify that the number of moves produced by both approaches is identical.

- On lines 55 to 84, the `check_move_lists_equal` function is our second and last differential test. We verify that the move lists are identical, by first sorting them on lines 58 and 59, then checking that both move lists are now identical on lines 60 and 61. The rest of the function builds an error message by computing the differences between the two lists.

- On lines 86 to 92, the `check_undo` function verifies the following property. Let `yolah` be any game configuration and `m` a possible move in this position. Let `Yolah before = yolah;`, then perform the following actions.

```
yolah.play(m);
yolah.undo(m);
Yolah after = yolah;
```

We verify that undoing a move works correctly by testing whether `before == after` on line 87.

- The second property we test is given in the `check_game_over_moves` function on lines 94 to 101. We test that the only possible move in a terminal game position is `Move::none()`. Indeed, the players have no more moves available and must therefore pass their turn. On line 96, we verify that there is only one move and that it is indeed `Move::none()`.

- On lines 103 to 119, the `check_none_move_execution` function tests the property that `Move::none()` should only increment the move count in the game. To do this, the loop on line 106 checks on line 107 that the content of each square has not changed after executing the null move. Then, on line 115, we test whether the move count has been properly incremented.

- On lines 121 to 156, the `check_regular_move_execution` function tests the properties that a regular move must satisfy. On lines 126 to 132, we verify that the squares not involved in the move have not changed. On lines 133 to 137, we verify that the move count has been incremented, that a hole has appeared on the move's source square, and finally that the piece is now on the move's destination square.

- On lines 158 to 234, the `random_games` function runs all tests on `nb_games` random games on lines 178 to 219. Lines 221 to 233 display a test summary.

- Listing 29 shows the result of executing the `main`. Even though, as Dijkstra said, "Program testing can be used to show the presence of bugs, but never to show their absence," and one must always remain cautious, the large number of game configurations explored nonetheless gives us good confidence in the validity of our Yolah implementation. Now that we have tested our implementation, we study its performance.

```
1  namespace test {
2  constexpr string_view RED    = "\033[1;31m";
3  constexpr string_view GREEN  = "\033[1;32m";
4  constexpr string_view YELLOW = "\033[1;33m";
5  constexpr string_view RESET  = "\033[0m";
6  constexpr string_view BOLD   = "\033[1m";
7
8  struct TestResult {
9      bool passed;
10     string message;
11     operator bool() const { return passed; }
12 };
13
14 TestResult pass() { return {true, ""}; }
15 TestResult fail(string msg) { return {false, std::move(msg)}; }
16
17 vector<Move> slow_moves_generation(const Yolah& yolah) {
```

```
18      vector<Move> res;
19      for (int i = 0; i < 8; i++) {
20          for (int j = 0; j < 8; j++) {
21              Square from = square_of(i, j);
22              if (yolah.get(from) != yolah.current_player()) continue;
23              for (int di = -1; di <= 1; di++) {
24                  for (int dj = -1; dj <= 1; dj++) {
25                      if (di == 0 && dj == 0) continue;
26                      int ii = i + di;
27                      int jj = j + dj;
28                      for(;;) {
29                          if (ii < 0 || ii >= 8 || jj < 0 || jj >= 8) {
30                              break;
31                          }
32                          Square to = square_of(ii, jj);
33                          if (yolah.get(to) != FREE) break;
34                          res.emplace_back(from, to);
35                          ii += di;
36                          jj += dj;
37                      }
38                  }
39              }
40          }
41      }
42      if (res.empty()) res.push_back(Move::none());
43      return res;
44  };

45

46  TestResult check_move_count(const MoveList& fast,
47                              const vector<Move>& expected) {
48      if (fast.size() != expected.size()) {
49          return fail(format("# of moves: expected {} got {}",
50                             expected.size(), fast.size()));
51      }
52      return pass();
53  }

54

55  TestResult check_move_lists_equal(MoveList& fast,
56                                    vector<Move>& expected,
57                                    const Yolah& yolah) {
58      sort(begin(fast), end(fast));
59      sort(begin(expected), end(expected));
```

63

```
60      if (equal(begin(fast), end(fast),
61                begin(expected), end(expected))) {
62          return pass();
63      }
64      ostringstream oss;
65      oss << "move lists differ\n" << yolah << '\n';
66      vector<Move> only_in_fast, only_in_expected;
67      set_difference(begin(fast), end(fast),
68                     begin(expected), end(expected),
69                     back_inserter(only_in_fast));
70      set_difference(begin(expected), end(expected),
71                     begin(fast), end(fast),
72                     back_inserter(only_in_expected));
73      if (!only_in_expected.empty()) {
74          oss << "  Only in expected: ";
75          for (const auto& m : only_in_expected) oss << m << ' ';
76          oss << '\n';
77      }
78      if (!only_in_fast.empty()) {
79          oss << "  Only in fast: ";
80          for (const auto& m : only_in_fast) oss << m << ' ';
81          oss << '\n';
82      }
83      return fail(oss.str());
84  }
85
86  TestResult check_undo(const Yolah& before, const Yolah& after) {
87      if (before == after) return pass();
88      ostringstream oss;
89      oss << "undo failed\n  Previous state:\n" << before
90          << "\n  State after undo:\n" << after << '\n';
91      return fail(oss.str());
92  }
93
94  TestResult check_game_over_moves(const Yolah& yolah,
95                                   const MoveList& moves) {
96      if (moves.size() == 1 && moves[0] == Move::none()) return pass();
97      ostringstream oss;
98      oss << "only Move::none() should be available when game is over\n"
99          << YolahWithMoves(yolah, moves) << '\n';
100     return fail(oss.str());
101 }
```

```
102
103   TestResult check_none_move_execution(const Yolah& before,
104                                        const Yolah& after) {
105       bool ok = true;
106       for (Square sq = SQ_A1; sq <= SQ_H8; ++sq) {
107           if (before.get(sq) != after.get(sq)) {
108               ok = false;
109               break;
110           }
111       }
112       if (!ok) {
113           return fail("Move::none() must not change the board content");
114       }
115       if (before.nb_plies() + 1 != after.nb_plies()) {
116         return fail("Move::none() must increment the number of plies");
117       }
118       return pass();
119   }
120
121   TestResult check_regular_move_execution(const Yolah& before,
122                                           const Yolah& after, Move m) {
123       Square from = m.from_sq();
124       Square to = m.to_sq();
125       bool ok = true;
126       for (Square sq = SQ_A1; sq <= SQ_H8; ++sq) {
127           if (sq == from || sq == to) continue;
128           if (before.get(sq) != after.get(sq)) {
129               ok = false;
130               break;
131           }
132       }
133       if (ok && before.nb_plies() + 1 == after.nb_plies() &&
134           after.get(from) == HOLE &&
135           after.get(to) == before.current_player()) {
136               return pass();
137       }
138       ostringstream oss;
139       oss << "Move execution incorrect\n" << after
140           << "\n  Move: " << m << '\n';
141       if (!ok) {
142           oss << " Squares not concerned by the move must not change";
143       }
```

65

```
144      if (after.get(from) != HOLE) {
145          oss << "  From square should be hole\n";
146      }
147      if (after.get(to) != before.current_player()) {
148          oss << " To square should contain a piece from "
149              << (before.current_player() == BLACK ? "black" : "white")
150              << " player\n";
151      }
152      if (before.nb_plies() + 1 != after.nb_plies()) {
153          oss << " The number of plies must be incremented\n";
154      }
155      return fail(oss.str());
156  }
157
158  void random_games(size_t nb_games, optional<uint64_t> seed) {
159      MoveList fast_moves;
160      random_device rd;
161      mt19937 mt(seed.value_or(rd()));
162      size_t total_tests = 0;
163      size_t passed_tests = 0;
164
165      auto run_test = [&](TestResult result) -> bool {
166          total_tests++;
167          if (result) {
168              passed_tests++;
169              return true;
170          }
171          cout << format("{}FAIL:{} {}\n", RED, RESET, result.message);
172          return false;
173      };
174
175      cout << format("{}\n=== Running Random Games Tests ===\n{}",
176                     BOLD, RESET);
177
178      for (size_t i = 0; i < nb_games; i++) {
179        Yolah yolah;
180        while (!yolah.game_over()) {
181          yolah.moves(fast_moves);
182          vector<Move> expected_moves = slow_moves_generation(yolah);
183
184          if (!run_test(
185                  check_move_count(fast_moves, expected_moves))) break;
```

```
186        if (!run_test(
187              check_move_lists_equal(fast_moves,
188                                     expected_moves, yolah))) break;
189
190        uniform_int_distribution<int> d(0, fast_moves.size() - 1);
191        Move m = fast_moves[d(mt)];
192        Yolah before = yolah;
193        yolah.play(m);
194
195        if (m == Move::none()) {
196            if (!run_test(
197                  check_none_move_execution(before, yolah))) break;
198        } else {
199            if (!run_test(
200                  check_regular_move_execution(before,
201                                               yolah, m))) break;
202        }
203
204        yolah.undo(m);
205        if (!run_test(check_undo(before, yolah))) break;
206        yolah.play(m);
207     }
208
209     if (!yolah.game_over()) continue;
210
211     yolah.moves(fast_moves);
212     if (!run_test(
213           check_game_over_moves(yolah, fast_moves))) continue;
214
215     yolah.play(Move::none());
216     yolah.moves(fast_moves);
217     if (!run_test(
218           check_game_over_moves(yolah, fast_moves))) continue;
219  }
220
221  cout << format("\n{}=== Test Summary ==={}\n", BOLD, RESET);
222  cout << format("Total tests: {}\n", total_tests);
223  cout << format("Passed: {}{}{}\n", GREEN, passed_tests, RESET);
224  cout << format("Failed: {}{}{}\n",
225               (passed_tests == total_tests ? GREEN : RED),
226               total_tests - passed_tests, RESET);
227  if (passed_tests == total_tests) {
```

```
228          cout << format("{}All tests passed!{}\n", GREEN, RESET);
229      } else {
230          double pass_rate = 100.0 * passed_tests / total_tests;
231          cout << format("{}Pass rate: {:.2f}%{}\n", YELLOW, pass_rate,
232                          RESET);
233      }
234  }}
235
236  int main() {
237    init_all_magics();
238    test::random_games(10000, 42);
239  }
```

Listing 28: Property-based and differential tests used to validate our implementation.

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic chapter02.cpp\
 -o chapter02
$ ./chapter02
=== Running Random Games Tests ===

=== Test Summary ===
Total tests: 2223036
Passed: 2223036
Failed: 0
All tests passed!
```

Listing 29: Results of executing the program from Listing 28

2.4.3 Performance Testing

We analyze the performance of our Yolah implementation using the program in Listing 30, which is similar to the program in Listing 26. However, unlike the latter, the program in Listing 30 does not pause during game play and displays some statistics about the games.

- On line 6, `black_wins` counts the number of wins by the black player throughout the `nb_games` games. On lines 28 to 30, we display the number of black wins and the percentage this represents relative to all games played.

- On line 7, `white_wins` counts the number of wins by the white player. The associated statistics are displayed on lines 31 to 33.

- On line 8, we do the same for the number of draws using the `draws` variable.

- On lines 11 to 16, we play a random game, by obtaining the list of available moves on line 12, then selecting a random move on line 14, and playing this move on line 15.

- We play one million games as we can see in the `main` on line 40.

```
1  void play_random_games(size_t nb_games,
2                          optional<uint64_t> seed = nullopt) {
3      MoveList moves;
4      random_device rd;
5      mt19937 mt(seed.value_or(rd()));
6      size_t black_wins = 0;
7      size_t white_wins = 0;
8      size_t draws = 0;
9      for (size_t i = 0; i < nb_games; i++) {
10         Yolah yolah;
11         while (!yolah.game_over()) {
12             yolah.moves(moves);
13             uniform_int_distribution<int> d(0, moves.size() - 1);
14             Move m = moves[d(mt)];
15             yolah.play(m);
16         }
17         auto [black_score, white_score] = yolah.score();
18         if (black_score > white_score) {
19             black_wins++;
20         } else if (white_score > black_score) {
21             white_wins++;
22         } else {
23             draws++;
24         }
25     }
26     cout << format("\n=== Game Statistics ===\n");
27     cout << format("Total games: {}\n", nb_games);
28     cout << format("Black wins:  {} ({:.1f}%)\n",
29                    black_wins,
30                    100.0 * black_wins / nb_games);
31     cout << format("White wins:  {} ({:.1f}%)\n",
32                    white_wins,
33                    100.0 * white_wins / nb_games);
34     cout << format("Draws:       {} ({:.1f}%)\n",
35                    draws,
36                    100.0 * draws / nb_games);
37 }
38 int main() {
39     init_all_magics();
```

```
40      play_random_games(1000000, 42);
41  }
```

Listing 30: We play one million random games to evaluate the performance of our Yolah implementation

We first analyze the function call traces to determine the most time-consuming parts of the code. We use the `perf` [12] tool to collect performance data and `stackcollapse-perf.pl`, from the `FlameGraph` [13] project, to analyze the call traces. The command we use is the following:

```
$ g++ -std=c++20 -O3 -g -march=native -Wall -Wpedantic chapter02.cpp\
-o chapter02
$ sudo perf record -F 999 --call-graph dwarf -g ./chapter02
$ sudo perf script | stackcollapse-perf.pl > profile.txt
```

We asked Claude [1] to generate a Python script to visualize the call traces as a tree, which is shown in Listing 31.

- The `play_random_games` function accounts for 96.79% of the program's execution time.

- The sequence of function calls can be read by following the tree branches. For example, we can see below the different function calls that led to the execution of `Magic::index`.

```
play_random_games
├── Yolah::moves
        ├── moves_bb
           └── Magic::index               8.87%
```

we can also see that the execution of `Magic::index` through this call chain represents 8.87% of the program's execution time.

- Note that the order of functions does not represent the order in the listing; the tree leaves are sorted by execution time percentage.

- Note also that we can easily deduce the time spent in a function excluding its function calls. For example, for the `play_random_games` function, its execution time accounts for 96.79% of the program's execution time. If we subtract from this number the sum (82.25%+5.33%+3.89%+3.86%), which represents the execution time percentages of the functions `Yolah::moves`, `std::uniform_int_distribution`, `Yolah::game_over`, and `Yolah::play`, we obtain 1.46%, which gives the percentage of total execution time spent in the rest of the `play_random_games` function. The proportion of time spent in the rest of the function is therefore $1.46/96.79 = 1.5\%$.

- The tree makes it easy to see that if we wish to improve our program's execution time, we should focus on the `Yolah::moves` function called by `play_random_games`. Indeed, this function's execution time represents 82.25% of the program's total execution time.

```
play_random_games                       96.79%
├── Yolah::moves                        82.25%
│      ├── moves_bb                      33.07%
│      │   └── Magic::index              8.87%
│      ├── Move::Move                    13.86%
│      └── pop_lsb                       13.85%
│          └── lsb                       13.18%
├── std::uniform_int_distribution<...    5.33%
├── Yolah::game_over                     3.89%
│   └── shift<>                          2.14%
└── Yolah::play                          3.86%
```

Listing 31: Function call traces identifying the most expensive functions

Thanks to Listing 31, we know that we must focus our efforts on the `Yolah::moves` function. We again use the `perf` [12] tool, but this time to access the processor's hardware performance counters (PMU – Performance Monitoring Unit) [14]. These counters measure events such as branch mispredictions, cache misses, etc. These counters will help us identify the weaknesses of our implementation[19]. We use the following commands to analyze the behavior of our program.

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic chapter02.cpp\
-o chapter02
$ perf stat -r 10 -e cycles,instructions,branches,branch-misses,\
L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses,\
l1d_pend_miss.pending,l1d_pend_miss.pending_cycles ./chapter02
```

The results are presented in Listing 32.

```
 Performance counter stats for './chapter02' (10 runs):

   17 675 748 046      cycles                        ( +-  0.14% )
   24 357 812 457      instructions   # 1.38 IPC     ( +-  0.03% )
    2 429 931 966      branches                      ( +-  0.03% )
      274 551 866      branch-misses  # 11.30%       ( +-  0.03% )
    3 132 479 051      L1-dcache-loads               ( +-  0.03% )
      147 715 205      L1-dcache-load-misses # 4.71% ( +-  0.16% )
       53 089 459      LLC-loads                     ( +-  1.24% )
           24 898      LLC-load-misses # 0.05%       ( +- 31.02% )
    2 413 363 382      l1d_pend_miss.pending         ( +-  0.78% )
    2 202 499 896      l1d_pend_miss.pending_cycles  ( +-  0.70% )

         4.23 +- 0.01 seconds time elapsed
```

Listing 32: Performance counters for the program in Listing 30.

- The `-r 10` option of the `perf` command above runs the program ten times, to compute the average and the standard deviation[20] of the obtained results.

---

[19]To delve deeper into optimization on modern processors, you may consult [15, 16, 17, 18].
[20]More precisely, the standard deviation as a percentage of the mean.

- The `cycles` counter indicates the number of cycles performed by the processor during our program's execution[21].

- The `instructions` counter measures the number of instructions executed by our program. The number of instructions executed per cycle, or Instructions Per Cycle (IPC), is 1.38. Modern processors are superscalar: they have the ability to execute multiple instructions per cycle. The higher this number, the more the processor's execution units are utilized. We will compare this IPC with that of the optimized versions we develop later.

- The `branches` counter tallies the total number of executed branches (conditional, unconditional, indirect, function calls, and returns).

- The `branch-misses` counter indicates the number of times the branch predictor was wrong. Modern processors, to keep their execution units busy, execute instructions speculatively, that is, before knowing the result of conditional branches. When the prediction is correct, the processor has saved time. In case of error, the speculatively executed instructions are discarded, and the processor resumes execution at the correct location after the branch. These branch mispredictions are costly. In our case, the branch predictor is wrong in 11.30% of cases. This rate is high and is the first thing we will try to improve.

- The `L1-dcache-loads` counter measures the number of read accesses to the first-level cache, which is the fastest cache (but also the smallest).

- The `L1-dcache-load-misses` counter indicates the number of reads that did not find the data in the first-level cache. These reads had to be fetched from higher-level caches or from main memory. Here we have an L1 cache miss rate of 4.71%. Our program accesses the magic bitboard tables in a way that is unpredictable for the processor, and these tables cannot fit entirely in the L1 cache. Given these considerations, this miss rate seems reasonable.

- The `LLC-loads` counter measures accesses to the last-level cache, the one placed just before main memory.

- The `LLC-load-misses` counter indicates the number of reads that did not find the data in the last-level cache (Last Level Cache (LLC)). We observe that this number is very low: out of the 147,175,205 accesses that missed in the L1 cache, only 24,898 required an access to main memory. This is easily explained, as the magic bitboard tables can fit entirely in the last-level cache.

- Regarding the next two counters, we were curious to study the memory-level parallelism Memory Level Parallelism (MLP) present in modern processors and to examine this characteristic in our program.

- The `l1d_pend_miss.pending` counter records, at each cycle, the number of pending memory requests following L1 cache misses. It is a cumulative sum: if during 100 cycles there are 3 in-flight requests, this counter increments by 300.

---

[21]In user mode and kernel mode. The counters are enabled when a process is running and disabled when it loses the CPU.

- The `l1d_pend_miss.pending_cycles` counter counts the number of cycles during which at least one memory request is pending.

- The ratio of these two counters gives the MLP:

$$\text{MLP} = \frac{\texttt{l1d\_pend\_miss.pending}}{\texttt{l1d\_pend\_miss.pending\_cycles}} = \frac{2\,413\,363\,382}{2\,202\,499\,896} \approx 1{,}10$$

This value indicates that on average, only 1.10 memory requests are in flight simultaneously when the processor is waiting for data. An MLP of 1 means that memory accesses are strictly sequential: the processor waits for one request to complete before launching another. Modern processors can handle multiple simultaneous requests. Our MLP close to 1 suggests that our code does not take advantage of memory-level parallelism, probably due to dependencies in the `moves` function code that prevent the processor from launching multiple requests in parallel. However, this observation should be put into perspective: with an L1 miss rate of only 4.71%, the vast majority of memory accesses (95.29%) are satisfied by the L1 cache within a few cycles. The low MLP is therefore only problematic for the 4.71% of accesses that actually generate a wait. We will compare this MLP with the optimized versions we develop later.

```
1  void moves(uint8_t player, MoveList& moves) const {
2      Move* move_list = moves.move_list;
3      uint64_t occupied = black | white | holes;
4      uint64_t bb = player == BLACK ? black : white;
5      while (bb) {
6          Square from = pop_lsb(bb);
7          uint64_t b = moves_bb(from, occupied) & ~occupied;
8          while (b) {
9              *move_list++ = Move(from, pop_lsb(b));
10         }
11     }
12     if (move_list == moves.move_list) [[unlikely]] {
13         *move_list++ = Move::none();
14     }
15     moves.last = move_list;
16 }
```

Listing 33: Recap of the `moves` function code

Given the previous results on the various counters, the criterion that seems most important to improve is the high branch misprediction rate. Listing 33 recalls the code of the `moves` function. We can also use `perf` to analyze more precisely where the branch mispredictions occur in the source code. The command we use is the following:

```
$ g++ -std=c++20 -O3 -g -march=native -Wall -Wpedantic chapter02.cpp\
-o chapter02
$ sudo perf record -e branch-misses:pp ./chapter02
$ sudo perf annotate --stdio
```

A portion of the annotated code produced by `perf annotate` is shown in Listing 34.

- The code generated by the compiler is optimized, and it is not always easy to see the correspondence with Listing 33. The annotated code shows the assembly generated by the compiler and interleaves source lines to try to show the correspondence with the original code.

- The percentages indicate the proportion of branch mispredictions attributed to each instruction relative to the entire program.

- The branch mispredictions related to the **while** (bb) loop on line 5 of Listing 33 represent: $9.24\% + 3.52\% = 12.76\%$.

- The branch mispredictions related to the **while** (b) loop on line 8 of Listing 33 represent: $4.50\% + 6.69\% + 67.28\% = 78.47\%$.

- The **while** (b) loop is therefore the one generating the most branch mispredictions. This is explained by the fact that the branch history for this loop does not contain regular patterns that would allow predicting future branches. Indeed, the number of iterations of this loop depends on the number of set bits in b, which represents the possible destination squares for a piece. This number varies depending on the positions of the pieces and the configuration of obstacles on the board, making any prediction difficult for the processor.

- The **while** (bb) loop, even though it generates far fewer branch mispredictions, is simple to optimize. Indeed, we know for certain that there are always exactly four iterations, since each player has four pieces. To optimize this loop, we fully unroll it to eliminate all associated branches. This unrolling also offers opportunities for instruction-level parallelism (Instruction Level Parallelism (ILP)), as the processor can simultaneously execute the independent computations for each of the four pieces.

```
Yolah::moves(unsigned char, MoveList&) const:
        :   408:   while (b) {
  4.50  : 8f2e:   and %r10,%rbp
  6.69  : 8f31:   je 8f65
        :   266:   : data((to << 6) + from) {}
  0.00  : 8f33:   mov %r9,%r11
        :   407:   uint64_t b = moves_bb(from, occupied) & ~occupied;
  0.00  : 8f36:   mov %rbp,%rcx
  0.00  : 8f39:   nopl 0x0(%rax)
        :   410:   std::__countr_zero:
  0.00  : 8f40:   xor %esi,%esi
        :   409:   *move_list++ = Move(from, pop_lsb(b));
  0.00  : 8f42:   add $0x2,%r11
  0.00  : 8f46:   tzcnt %rcx,%rsi
  0.00  : 8f4b:   shl $0x6,%esi
  0.00  : 8f4e:   add %eax,%esi
        :   408:   while (b) {
  0.00  : 8f50:   blsr %rcx,%rcx
  0.00  : 8f55:   mov %si,-0x2(%r11)
 67.28  : 8f5a:   jne 8f40
  0.00  : 8f5c:   popcnt %rbp,%rbp
  0.00  : 8f61:   lea (%r9,%rbp,2),%r9
        :   405:   while (bb) {
  9.24  : 8f65:   test %rdi,%rdi
  3.52  : 8f68:   jne 8ed8
```

Listing 34: Results of `perf annotate` identifying the code sections causing the most branch mispredictions. The percentages indicate the proportion of branch mispredictions attributed to each instruction relative to the entire program.

The new version of the `moves` function from Listing 33 with the **while** (bb) loop (line 5) fully unrolled is given in Listing 35.

- On lines 6 to 9, we retrieve the squares of the player's four pieces.

- On lines 11 to 14, we obtain the bitboards of possible moves for the player's four pieces.

- On lines 16 to 18, we create the moves for the piece located at square `from0`.

- On lines 19 to 21, we create the moves for the piece located at square `from1`.

- On lines 22 to 24, we create the moves for the piece located at square `from2`.

- On lines 25 to 27, we create the moves for the piece located at square `from3`.

```
1  void moves(uint8_t player, MoveList& moves) const {
2      Move* move_list = moves.move_list;
3      uint64_t occupied = black | white | holes;
4      uint64_t bb = player == BLACK ? black : white;
```

```
5
6        Square from0 = pop_lsb(bb);
7        Square from1 = pop_lsb(bb);
8        Square from2 = pop_lsb(bb);
9        Square from3 = pop_lsb(bb);
10
11       uint64_t b0 = moves_bb(from0, occupied) & ~occupied;
12       uint64_t b1 = moves_bb(from1, occupied) & ~occupied;
13       uint64_t b2 = moves_bb(from2, occupied) & ~occupied;
14       uint64_t b3 = moves_bb(from3, occupied) & ~occupied;
15
16       while (b0) {
17           *move_list++ = Move(from0, pop_lsb(b0));
18       }
19       while (b1) {
20           *move_list++ = Move(from1, pop_lsb(b1));
21       }
22       while (b2) {
23           *move_list++ = Move(from2, pop_lsb(b2));
24       }
25       while (b3) {
26           *move_list++ = Move(from3, pop_lsb(b3));
27       }
28       if (move_list == moves.move_list) [[unlikely]] {
29           *move_list++ = Move::none();
30       }
31       moves.last = move_list;
32   }
```

Listing 35: New version of the `moves` function with the first **while** loop unrolled.

We run the following command again, and the results are given in Listing 36.

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic chapter02.cpp\
 -o chapter02
$ perf stat -r 10 -e cycles,instructions,branches,branch-misses,\
L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses,\
l1d_pend_miss.pending,l1d_pend_miss.pending_cycles ./chapter02
```

This new version allows us to reduce the program's execution time by $1 - \frac{3.06}{4.23} \approx$ 27.7%. We can observe in Listing 36 that we improved the IPC, which goes from 1.38 to 1.85, that we reduced the branch misprediction rate from 11.30% to 10.16%, and that the L1 cache miss rate went from 4.71% to 4.12%. The MLP, which measures the average number of memory requests in flight simultaneously, also improves from 1.10 to 1.35, indicating that the processor better exploits memory-level parallelism to

hide cache access latency. Unrolling the first **while** loop thus allowed the processor to access more independent instructions and execute more instructions per cycle overall.

```
Performance counter stats for './chapter02' (10 runs):

    12 678 510 624     cycles                        ( +-  0.07% )
    23 599 166 797     instructions   # 1.85 IPC     ( +-  0.03% )
     2 329 596 530     branches                      ( +-  0.03% )
       236 669 629     branch-misses  # 10.16%       ( +-  0.04% )
     3 510 465 683     L1-dcache-loads               ( +-  0.04% )
       144 745 234     L1-dcache-load-misses # 4.12% ( +-  0.09% )
        51 642 976     LLC-loads                     ( +-  0.59% )
            11 608     LLC-load-misses # 0.02%       ( +- 12.06% )
     2 372 461 194     l1d_pend_miss.pending         ( +-  0.80% )
     1 756 784 236     l1d_pend_miss.pending_cycles  ( +-  0.77% )


         3.06 +- 0.01 seconds time elapsed
```

Listing 36: Performance counters for the program in Listing 35.

We shall see later whether it is necessary to continue improving the **moves** function. Indeed, its execution time may be negligible compared to the rest of the program in the following chapters. However, we need to run random games for some artificial players. It is therefore worthwhile to improve its execution time. A very effective way to optimize code is to delete it! We do not need to generate the list of all moves and then choose one randomly only after this generation. We can choose a random move from the start. This move list generation is also responsible for nearly all the branch mispredictions in the program. The **Move Yolah::random_move(mt19937& mt)** function in Listing 37 returns a random move in the current game position. We coded this function to eliminate as many branches as possible. This programming style, called *branchless*, favors arithmetic and logical operations over conditional instructions, thus avoiding the penalties associated with branch mispredictions.

```
1  Move Yolah::random_move(mt19937& mt) const {
2      uint64_t occupied = black | white | holes;
3      uint64_t player_bb = current_player() == BLACK ? black : white;
4      Square from0 = pop_lsb(player_bb);
5      Square from1 = pop_lsb(player_bb);
6      Square from2 = pop_lsb(player_bb);
7      Square from3 = pop_lsb(player_bb);
8      uint64_t b0 = moves_bb(from0, occupied) & ~occupied;
9      uint64_t b1 = moves_bb(from1, occupied) & ~occupied;
10     uint64_t b2 = moves_bb(from2, occupied) & ~occupied;
11     uint64_t b3 = moves_bb(from3, occupied) & ~occupied;
12     int n0 = popcount(b0);
13     int n1 = popcount(b1);
14     int n2 = popcount(b2);
```

```cpp
15        int n3 = popcount(b3);
16        int n = n0 + n1 + n2 + n3;
17        if (n == 0) [[unlikely]] {
18            return Move::none();
19        }
20        uniform_int_distribution<int> d(0, n - 1);
21        int bit = d(mt);
22        int bb_index = (bit >= n0) + (bit >= n0+n1) + (bit >= n0+n1+n2);
23        bit -= (bb_index > 0)*n0 + (bb_index > 1)*n1 + (bb_index > 2)*n2;
24        Square from = array{ from0, from1, from2, from3 }[bb_index];
25        uint64_t bb = array{ b0, b1, b2, b3 }[bb_index];
26        Square to = Square(countr_zero(_pdep_u64(1ULL << bit, bb)));
27        return Move(from, to);
28    }
29    void play_random_games_fast(size_t nb_games,
30                                optional<uint64_t> seed = nullopt) {
31        random_device rd;
32        mt19937 mt(seed.value_or(rd()));
33        size_t black_wins = 0;
34        size_t white_wins = 0;
35        size_t draws = 0;
36        for (size_t i = 0; i < nb_games; i++) {
37            Yolah yolah;
38            while (!yolah.game_over()) {
39                Move m = yolah.random_move(mt);
40                yolah.play(m);
41            }
42            auto [black_score, white_score] = yolah.score();
43            if (black_score > white_score) {
44                black_wins++;
45            } else if (white_score > black_score) {
46                white_wins++;
47            } else {
48                draws++;
49            }
50        }
51        cout << format("\n=== Game Statistics ===\n");
52        cout << format("Total games: {}\n", nb_games);
53        cout << format("Black wins:  {} ({:.1f}%)\n",
54                       black_wins,
55                       100.0 * black_wins / nb_games);
56        cout << format("White wins:  {} ({:.1f}%)\n",
```

```
57                    white_wins,
58                    100.0 * white_wins / nb_games);
59     cout << format("Draws:        {} ({:.1f}%)\n",
60                    draws,
61                    100.0 * draws / nb_games);
62 }
63 int main() {
64     init_all_magics();
65     play_random_games_fast(1000000, 42);
66 }
```

Listing 37: Generating a random move without first generating the entire move list.

- On lines 4 to 7, we retrieve the squares of the current player's pieces.

- On lines 8 to 11, we obtain the bitboards of possible moves for each of the player's piece positions.

- On lines 12 to 15, we obtain, using the `popcount` function which counts the number of set bits in a bitboard, the number of moves for each of the bitboards `b0, b1, b2`, and `b3`.

- On lines 17 to 19, if there are no possible moves, we pass our turn.

- On lines 20 to 21, we randomly generate the move number in the `bit` variable, which corresponds to one of the bits of `b0, b1, b2`, or `b3`.

- On line 22, we retrieve the index of the bitboard b_i where move `bit` is located. For example, if `n0 == 5` and `n1 == 12` and `bit == 7`, we need to look for the move in `b1` and the move corresponds to the second set bit in `b1`. Note that we compute this index without any conditional branch. We replaced a cascade of **if**/**else if** with a sum of comparisons: each comparison (`bit >= ...`) returns `0` or `1`, and their sum directly gives the desired index.

- On line 23, we compute the index of the set bit within the selected bitboard. To do this, we subtract the number of set bits in the previous bitboards b_i. Here again we use the *branchless* style.

- On line 24, we select the source square corresponding to the random move.

- On line 25, we select the bitboard b_i corresponding to the random move.

- On line 26, now that we have the move's source square, the bitboard `bb`, and the index of the set bit in `bb`, we can construct the random move by building the destination square. To compute the move's destination square, we could use the following code.

```
Square to;
while (bb) {
```

79

```
    int tz = countr_zero(bb); // position of the rightmost set bit
    if (bit-- == 0) {
        to = Square(tz);
        break;
    }
    bb &= bb - 1; // clear rightmost set bit
}
```

However, the `_pdep_u64` function (Parallel Bits Deposit), which is based on a Bit Manipulation Instruction Set 2 (BMI2) instruction[22], allows this code to be performed more efficiently. Indeed, `_pdep_u64(1ULL << bit, bb)` returns a bitboard with a single set bit, located at the position of the `bit`-th set bit of `bb`.

- On line 39, in the `play_random_games_fast` function, we use the `random_move` function to directly generate a random move, instead of first creating the entire list of possible moves and then selecting one at random from it.

We test our program's characteristics again with `perf`; the results are given in Listing 38.

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic chapter02.cpp\
-o chapter02
$ perf stat -r 10 -e cycles,instructions,branches,branch-misses,\
L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses,\
l1d_pend_miss.pending,l1d_pend_miss.pending_cycles ./chapter02

Performance counter stats for './chapter02' (10 runs):

    6 293 794 538      cycles                              ( +-  0.22% )
   15 430 554 773      instructions   # 2.43 IPC           ( +-  0.05% )
      792 545 398      branches                            ( +-  0.05% )
        2 354 243      branch-misses  # 0.30%              ( +-  0.26% )
    3 499 135 163      L1-dcache-loads                     ( +-  0.07% )
      144 657 644      L1-dcache-load-misses # 4.14% ( +-  0.11% )
       52 835 597      LLC-loads                           ( +-  0.90% )
           11 787      LLC-load-misses # 0.02%             ( +-  8.82% )
    2 329 553 474      l1d_pend_miss.pending               ( +-  0.87% )
    1 708 715 065      l1d_pend_miss.pending_cycles ( +-  0.82% )

           1.53 +- 0.00 seconds time elapsed
```

Listing 38: Performance counters for the program in Listing 37.

This last optimization cuts the execution time in half, going from 3.06 to 1.53 seconds. The IPC improves from 1.85 to 2.43. The most notable result concerns branch mispredictions: the rate drops from 10.16% to 0.30%. This near-elimination of branch mispredictions is explained by the *branchless* programming style adopted, which was made possible because we no longer need to generate the entire move list in

---

[22]This is an x86-64 instruction (Intel Haswell+, AMD Zen+).

this version. By replacing **if**/**else** conditional structures with arithmetic expressions (sums of comparisons), we eliminated the unpredictable branches that the processor's branch predictor could not anticipate. The `_pdep_u64` instruction also contributes to this improvement by avoiding a loop with a variable number of iterations.

2.5   What's Next

We now have at our disposal an efficient implementation of Yolah, whose complete code is given in the following section. With this implementation, we will be able to develop increasingly capable artificial players for our game in the remainder of this book.

2.6   Complete Board Game Code

The complete board code is given in listings 39, 40, 41, and 42. The program execution result is given in listing 43. We can see that over one million random games, black appears to have the advantage. Note that this does not prove that the game is winning for black.

```cpp
#include <algorithm>
#include <array>
#include <bit>
#include <cstdint>
#include <format>
#include <iostream>
#include <optional>
#include <random>
#include <set>
#include <sstream>
#include <string>
#include <string_view>
#include <utility>
#include <vector>
#include <immintrin.h>
using namespace std;

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

```cpp
25  // 1   0    1     2     3     4     5     6     7
26  //     a    b     c     d     e     f     g     h
27  enum Square : int8_t {
28      SQ_A1, SQ_B1, SQ_C1, SQ_D1, SQ_E1, SQ_F1, SQ_G1, SQ_H1,
29      SQ_A2, SQ_B2, SQ_C2, SQ_D2, SQ_E2, SQ_F2, SQ_G2, SQ_H2,
30      SQ_A3, SQ_B3, SQ_C3, SQ_D3, SQ_E3, SQ_F3, SQ_G3, SQ_H3,
31      SQ_A4, SQ_B4, SQ_C4, SQ_D4, SQ_E4, SQ_F4, SQ_G4, SQ_H4,
32      SQ_A5, SQ_B5, SQ_C5, SQ_D5, SQ_E5, SQ_F5, SQ_G5, SQ_H5,
33      SQ_A6, SQ_B6, SQ_C6, SQ_D6, SQ_E6, SQ_F6, SQ_G6, SQ_H6,
34      SQ_A7, SQ_B7, SQ_C7, SQ_D7, SQ_E7, SQ_F7, SQ_G7, SQ_H7,
35      SQ_A8, SQ_B8, SQ_C8, SQ_D8, SQ_E8, SQ_F8, SQ_G8, SQ_H8,
36      SQ_NONE,
37      SQUARE_ZERO = 0,
38      SQUARE_NB   = 64
39  };
40
41  // 8  56      57      58      59      60      61      62      63
42  // 7  48      49      50      51      52      53      54      55
43  // 6  40      41      42      43      44      45      46      47
44  // 5  32      33      34      35      36      37      38      39
45  // 4  24      25      26      27      28      29      30      31
46  // 3  16      17      18      19      20      21      22      23
47  // 2  8       9       10      11      12      13      14      15
48  // 1  0       1       2       3       4       5       6       7
49  //  FILE_A FILE_B FILE_C FILE_D FILE_E FILE_F FILE_G FILE_H
50  enum File : uint8_t {
51      FILE_A, FILE_B, FILE_C, FILE_D, FILE_E,
52      FILE_F, FILE_G, FILE_H, FILE_NB
53  };
54
55  //  RANK_8 | 56  57  58  59  60  61  62  63
56  //  RANK_7 | 48  49  50  51  52  53  54  55
57  //  RANK_6 | 40  41  42  43  44  45  46  47
58  //  RANK_5 | 32  33  34  35  36  37  38  39
59  //  RANK_4 | 24  25  26  27  28  29  30  31
60  //  RANK_3 | 16  17  18  19  20  21  22  23
61  //  RANK_2 | 8   9   10  11  12  13  14  15
62  //  RANK_1 | 0   1   2   3   4   5   6   7
63  //           a   b   c   d   e   f   g   h
64  enum Rank : uint8_t {
65      RANK_1, RANK_2, RANK_3, RANK_4, RANK_5,
66      RANK_6, RANK_7, RANK_8, RANK_NB
```

```
67  };
68  // Directions from sq = 27 (d4):
69  //
70  //   +------+------+------+------+------+------+------+------+
71  // 8 |  56  |  57  |  58  |  59  |  60  |  61  |  62  |  63  |
72  //   +------+------+------+------+------+------+------+------+
73  // 7 |  48  |  49  |  50  |  51  |  52  |  53  |  54  |  55  |
74  //   +------+------+------+------+------+------+------+------+
75  // 6 |  40  |  41  |  42  |  43  |  44  |  45  |  46  |  47  |
76  //   +------+------+------+------+------+------+------+------+
77  // 5 |  32  |  33  |NW 34 |N  35 |NE 36 |  37  |  38  |  39  |
78  //   +------+------+------+------+------+------+------+------+
79  // 4 |  24  |  25  |W  26 |>>27<<|E  28 |  29  |  30  |  31  |
80  //   +------+------+------+------+------+------+------+------+
81  // 3 |  16  |  17  |SW 18 |S  19 |SE 20 |  21  |  22  |  23  |
82  //   +------+------+------+------+------+------+------+------+
83  // 2 |   8  |   9  |  10  |  11  |  12  |  13  |  14  |  15  |
84  //   +------+------+------+------+------+------+------+------+
85  // 1 |   0  |   1  |   2  |   3  |   4  |   5  |   6  |   7  |
86  //   +------+------+------+------+------+------+------+------+
87  //     a      b      c      d      e      f      g      h
88  enum Direction : int8_t {
89      NORTH = 8,
90      EAST  = 1,
91      SOUTH = -NORTH, // -8
92      WEST  = -EAST,  // -1
93      NORTH_EAST = NORTH + EAST, // 9
94      SOUTH_EAST = SOUTH + EAST, // -7
95      SOUTH_WEST = SOUTH + WEST, // -9
96      NORTH_WEST = NORTH + WEST  // 7
97  };
98  // ORTHOGONAL moves from d4:      DIAGONAL moves from d4:
99  //
100 // 8 .  .  .  *  .  .  .  .      .  .  .  .  .  .  .  *
101 // 7 .  .  .  *  .  .  .  .      *  .  .  .  .  .  *  .
102 // 6 .  .  .  *  .  .  .  .      .  *  .  .  .  *  .  .
103 // 5 .  .  .  *  .  .  .  .      .  .  *  .  *  .  .  .
104 // 4 *  *  * sq  *  *  *  *      .  .  . sq  .  .  .  .
105 // 3 .  .  .  *  .  .  .  .      .  .  *  .  *  .  .  .
106 // 2 .  .  .  *  .  .  .  .      .  *  .  .  .  *  .  .
107 // 1 .  .  .  *  .  .  .  .      *  .  .  .  .  .  *  .
108 //    a  b  c  d  e  f  g  h      a  b  c  d  e  f  g  h
```

```
109  enum MoveType {
110      ORTHOGONAL,
111      DIAGONAL
112  };
113
114  // Black player
115  constexpr uint8_t BLACK = 0;
116  // White Player
117  constexpr uint8_t WHITE = 1;
118  // Square is destroyed
119  constexpr uint8_t HOLE  = 2;
120  // Square is free
121  constexpr uint8_t FREE  = 3;
122
123  // FileABB: bitboard for file A, bits on file A are set to 1,
124  // all other bits are set to 0. Other file bitboards are derived
125  // by shifting FileABB to the left.
126  //
127  //    FileABB            FileBBB            FileCBB
128  // 8  1 . . . . . . .    . 1 . . . . . .    . . 1 . . . . .
129  // 7  1 . . . . . . .    . 1 . . . . . .    . . 1 . . . . .
130  // 6  1 . . . . . . .    . 1 . . . . . .    . . 1 . . . . .
131  // 5  1 . . . . . . .    . 1 . . . . . .    . . 1 . . . . .
132  // 4  1 . . . . . . .    . 1 . . . . . .    . . 1 . . . . .
133  // 3  1 . . . . . . .    . 1 . . . . . .    . . 1 . . . . .
134  // 2  1 . . . . . . .    . 1 . . . . . .    . . 1 . . . . .
135  // 1  1 . . . . . . .    . 1 . . . . . .    . . 1 . . . . .
136  //    a b c d e f g h    a b c d e f g h    a b c d e f g h
137  //
138  //    FileDBB            FileEBB            FileFBB
139  // 8  . . . 1 . . . .    . . . . 1 . . .    . . . . . 1 . .
140  // 7  . . . 1 . . . .    . . . . 1 . . .    . . . . . 1 . .
141  // 6  . . . 1 . . . .    . . . . 1 . . .    . . . . . 1 . .
142  // 5  . . . 1 . . . .    . . . . 1 . . .    . . . . . 1 . .
143  // 4  . . . 1 . . . .    . . . . 1 . . .    . . . . . 1 . .
144  // 3  . . . 1 . . . .    . . . . 1 . . .    . . . . . 1 . .
145  // 2  . . . 1 . . . .    . . . . 1 . . .    . . . . . 1 . .
146  // 1  . . . 1 . . . .    . . . . 1 . . .    . . . . . 1 . .
147  //    a b c d e f g h    a b c d e f g h    a b c d e f g h
148  //
149  //    FileGBB            FileHBB
150  // 8  . . . . . . 1 .    . . . . . . . 1
```

84

```
151  // 7 . . . . . . . 1 .   . . . . . . . 1
152  // 6 . . . . . . . 1 .   . . . . . . . 1
153  // 5 . . . . . . . 1 .   . . . . . . . 1
154  // 4 . . . . . . . 1 .   . . . . . . . 1
155  // 3 . . . . . . . 1 .   . . . . . . . 1
156  // 2 . . . . . . . 1 .   . . . . . . . 1
157  // 1 . . . . . . . 1 .   . . . . . . . 1
158  //   a b c d e f g h   a b c d e f g h
159  constexpr uint64_t FileABB = 0x0101010101010101;
160  constexpr uint64_t FileBBB = FileABB << 1;
161  constexpr uint64_t FileCBB = FileABB << 2;
162  constexpr uint64_t FileDBB = FileABB << 3;
163  constexpr uint64_t FileEBB = FileABB << 4;
164  constexpr uint64_t FileFBB = FileABB << 5;
165  constexpr uint64_t FileGBB = FileABB << 6;
166  constexpr uint64_t FileHBB = FileABB << 7;
167
168  // Rank1BB: bitboard for rank 1, bits on rank 1 are set to 1,
169  // all other bits are set to 0. Other rank bitboards are derived
170  // by shifting Rank1BB to the left by multiples of 8.
171  //
172  //   Rank1BB          Rank2BB          Rank3BB
173  // 8 . . . . . . . .   . . . . . . . .   . . . . . . . .
174  // 7 . . . . . . . .   . . . . . . . .   . . . . . . . .
175  // 6 . . . . . . . .   . . . . . . . .   . . . . . . . .
176  // 5 . . . . . . . .   . . . . . . . .   . . . . . . . .
177  // 4 . . . . . . . .   . . . . . . . .   . . . . . . . .
178  // 3 . . . . . . . .   . . . . . . . .   1 1 1 1 1 1 1 1
179  // 2 . . . . . . . .   1 1 1 1 1 1 1 1   . . . . . . . .
180  // 1 1 1 1 1 1 1 1 1   . . . . . . . .   . . . . . . . .
181  //   a b c d e f g h   a b c d e f g h   a b c d e f g h
182  //
183  //   Rank4BB          Rank5BB          Rank6BB
184  // 8 . . . . . . . .   . . . . . . . .   . . . . . . . .
185  // 7 . . . . . . . .   . . . . . . . .   . . . . . . . .
186  // 6 . . . . . . . .   . . . . . . . .   1 1 1 1 1 1 1 1
187  // 5 . . . . . . . .   1 1 1 1 1 1 1 1   . . . . . . . .
188  // 4 1 1 1 1 1 1 1 1   . . . . . . . .   . . . . . . . .
189  // 3 . . . . . . . .   . . . . . . . .   . . . . . . . .
190  // 2 . . . . . . . .   . . . . . . . .   . . . . . . . .
191  // 1 . . . . . . . .   . . . . . . . .   . . . . . . . .
192  //   a b c d e f g h   a b c d e f g h   a b c d e f g h
```

```
193  //
194  //     Rank7BB              Rank8BB
195  // 8 . . . . . . . .    1 1 1 1 1 1 1 1
196  // 7 1 1 1 1 1 1 1 1    . . . . . . . .
197  // 6 . . . . . . . .    . . . . . . . .
198  // 5 . . . . . . . .    . . . . . . . .
199  // 4 . . . . . . . .    . . . . . . . .
200  // 3 . . . . . . . .    . . . . . . . .
201  // 2 . . . . . . . .    . . . . . . . .
202  // 1 . . . . . . . .    . . . . . . . .
203  //   a b c d e f g h    a b c d e f g h
204  constexpr uint64_t Rank1BB = 0xFF;
205  constexpr uint64_t Rank2BB = Rank1BB << (8 * 1);
206  constexpr uint64_t Rank3BB = Rank1BB << (8 * 2);
207  constexpr uint64_t Rank4BB = Rank1BB << (8 * 3);
208  constexpr uint64_t Rank5BB = Rank1BB << (8 * 4);
209  constexpr uint64_t Rank6BB = Rank1BB << (8 * 5);
210  constexpr uint64_t Rank7BB = Rank1BB << (8 * 6);
211  constexpr uint64_t Rank8BB = Rank1BB << (8 * 7);
212
213  // Initial positions: Black starts at a1, e4, d5, h8.
214  // White starts at a8, d4, e5, h1.
215  //
216  // BLACK_INITIAL_POSITION       WHITE_INITIAL_POSITION
217  //
218  // 8 . . . . . . . B    W . . . . . . .
219  // 7 . . . . . . . .    . . . . . . . .
220  // 6 . . . . . . . .    . . . . . . . .
221  // 5 . . . B . . . .    . . . . W . . .
222  // 4 . . . . B . . .    . . . W . . . .
223  // 3 . . . . . . . .    . . . . . . . .
224  // 2 . . . . . . . .    . . . . . . . .
225  // 1 B . . . . . . .    . . . . . . . W
226  //   a b c d e f g h    a b c d e f g h
227  constexpr uint64_t BLACK_INITIAL_POSITION =
228  0b10000000'00000000'00000000'00001000'00010000'00000000'00000000'00000001;
229  constexpr uint64_t WHITE_INITIAL_POSITION =
230  0b00000001'00000000'00000000'00010000'00001000'00000000'00000000'10000000;
```

Listing 39: Complete game board code (part 1/4 – types, constants and bitboards)

```
234    // Returns the square at rank i, file j
235    constexpr Square square_of(int i, int j) {
236        return Square(i * 8 + j);
237    }
238
239    // Checks whether a square is within the valid range [A1, H8]
240    constexpr bool is_ok(Square s) {
241        return s >= SQ_A1 && s <= SQ_H8;
242    }
243
244    // Extracts the file (column) from a square: s % 8
245    constexpr File file_of(Square s) {
246        return File(s & 7);
247    }
248
249    // Extracts the rank (row) from a square: s / 8
250    constexpr Rank rank_of(Square s) {
251        return Rank(s >> 3);
252    }
253
254    // Returns a bitboard with all bits set on rank r
255    constexpr uint64_t rank_bb(Rank r) {
256        return Rank1BB << (8 * r);
257    }
258
259    // Returns a bitboard with all bits set on the rank of square s
260    constexpr uint64_t rank_bb(Square s) {
261        return rank_bb(rank_of(s));
262    }
263
264    // Returns a bitboard with all bits set on file f
265    constexpr uint64_t file_bb(File f) {
266        return FileABB << f;
267    }
268
269    // Returns a bitboard with all bits set on the file of square s
270    constexpr uint64_t file_bb(Square s) {
271        return file_bb(file_of(s));
272    }
273
274    // Returns a bitboard with only the bit corresponding to square s set
```

87

```cpp
275  constexpr uint64_t square_bb(Square s) {
276      return 1ULL << s;
277  }
278
279  // Moves a square in a given direction
280  constexpr Square operator+(Square s, Direction d) {
281      return Square(int(s) + int(d));
282  }
283
284  // Pre-increment operator to iterate over squares
285  Square& operator++(Square& d) {
286      return d = Square(int(d) + 1);
287  }
288
289  // Returns the least significant bit as a Square (index
290  // of first set bit)
291  constexpr Square lsb(uint64_t b) {
292      return Square(countr_zero(b));
293  }
294
295  // Extracts and clears the least significant bit, returns it as
296  // a Square
297  Square pop_lsb(uint64_t& b) {
298      const Square s = lsb(b);
299      b &= b - 1;
300      return s;
301  }
302
303  // Computes the Manhattan distance (|rank1 - rank2| + |file1 - file2|)
304  int manhattan_distance(Square sq1, Square sq2) {
305      int d_rank = abs(rank_of(sq1) - rank_of(sq2));
306      int d_file = abs(file_of(sq1) - file_of(sq2));
307      return d_rank + d_file;
308  }
309
310  // Shifts all bits of a bitboard in direction D. For horizontal and
311  // diagonal shifts, bits on the edge file are masked out to prevent
312  // wrapping around the board (e.g., a piece on file H shifted east
313  // would appear on file A)
314  template<Direction D>
315  constexpr uint64_t shift(uint64_t b) {
316      if constexpr (D == NORTH)
```

```
317         return b << NORTH;
318     else if constexpr (D == SOUTH)
319         return b >> -SOUTH;
320     else if constexpr (D == EAST)
321         return (b & ~FileHBB) << EAST;
322     else if constexpr (D == WEST)
323         return (b & ~FileABB) >> -WEST;
324     else if constexpr (D == NORTH_EAST)
325         return (b & ~FileHBB) << NORTH_EAST;
326     else if constexpr (D == NORTH_WEST)
327         return (b & ~FileABB) << NORTH_WEST;
328     else if constexpr (D == SOUTH_EAST)
329         return (b & ~FileHBB) >> -SOUTH_EAST;
330     else if constexpr (D == SOUTH_WEST)
331         return (b & ~FileABB) >> -SOUTH_WEST;
332     else return 0;
333 }
334
335 // Computes all squares reachable from sq by sliding in the given move
336 // type (orthogonal or diagonal), stopping at occupied squares or
337 // board edges. Used during magic bitboard initialization to build the
338 // move tables
339 uint64_t reachable_squares(MoveType mt, Square sq,
340                            uint64_t occupied) {
341     uint64_t  moves    = 0;
342     Direction o_dir[4]={NORTH, SOUTH, EAST, WEST};
343     Direction d_dir[4]={NORTH_EAST,SOUTH_EAST,SOUTH_WEST,NORTH_WEST};
344     for (Direction d : (mt == ORTHOGONAL ? o_dir : d_dir)) {
345         Square s = sq;
346         while (true) {
347             Square to = s + d;
348             if (!is_ok(to) || manhattan_distance(s, to) > 2) break;
349             uint64_t bb = square_bb(to);
350             if ((bb & occupied) != 0) break;
351             moves |= bb;
352             s = to;
353         }
354     }
355     return moves;
356 }
357
358 // Magic bitboard entry for one square. The mask isolates the relevant
```

89

```
359    // occupancy bits, the magic number maps them to a unique index, and
360    // the moves pointer references the precomputed move bitboards in the
361    // table
362    struct Magic {
363        uint64_t  mask;   // relevant occupancy mask (edges excluded)
364        uint64_t  magic;  // magic multiplier for this square
365        uint64_t* moves;  // pointer into the move lookup table
366        uint32_t  shift;  // right shift amount = 64 - popcount(mask)
367
368        // Computes the table index for a given occupancy configuration
369        uint32_t index(uint64_t occupied) const {
370            return uint32_t( ((occupied & mask) * magic) >> shift );
371        }
372    };
373
374    // Precomputed move lookup tables for all squares
375    uint64_t orthogonalTable[102400];
376    uint64_t diagonalTable[5248];
377
378    // Magic bitboard entries for each square
379    Magic orthogonalMagics[SQUARE_NB];
380    Magic diagonalMagics[SQUARE_NB];
381
382    // Combines orthogonal and diagonal magic table lookups to return a
383    // bitboard of all squares that a piece on sq can reach, given the
384    // board occupancy
385    uint64_t moves_bb(Square sq, uint64_t occupied) {
386        uint32_t idx_omoves = orthogonalMagics[sq].index(occupied);
387        uint32_t idx_dmoves = diagonalMagics[sq].index(occupied);
388        return orthogonalMagics[sq].moves[idx_omoves] |
389               diagonalMagics[sq].moves[idx_dmoves];
390    }
391
392    // Initializes magic bitboard tables for a given move type (orthogonal
393    // or diagonal). For each square, enumerates all possible occupancy
394    // patterns using the Carry-Rippler trick, computes the corresponding
395    // reachable squares, and stores them in the lookup table indexed by
396    // the magic hash
397    void init_magics(MoveType mt, uint64_t table[], Magic magics[]) {
398        static constexpr uint64_t O_MAGIC[64] = {
399            0x80011040002082,  0x40022002100040,  0x1880200081181000,
400            0x2080240800100080, 0x8080024400800800, 0x4100080400024100,
```

```
401         0xc080028001000a00, 0x80146043000080,   0x8120802080034004,
402         0x8401000200240,    0x202001282002044,   0x81010021000b1000,
403         0x808044000800,     0x300080800c000200, 0x8c000268411004,
404         0x810080058020c100, 0xc248608010400080, 0x30024040002000,
405         0x9001010042102000, 0x210009001002,      0xa0061d0018001100,
406         0x2410808004000600, 0x6400240008025001, 0xc10600010340a4,
407         0x628080044011,     0x4810014040002000, 0x380200080801000,
408         0x10018580080010,   0x101040080180180,   0x9208020080040080,
409         0x10400a21008,      0x6800104200010484, 0x21400280800020,
410         0x9400402008401001, 0x8430006800200400, 0x8104411202000820,
411         0x8010171000408,    0x1202000402001008, 0x881100904002208,
412         0x15a0800a49802100, 0x224001808004,      0x4420201002424000,
413         0xc04500020008080,  0x2503009004210008, 0x42801010010,
414         0x2000400090100,    0x8080011810040002, 0x44401c008046000d,
415         0x4000800521104100, 0x82000b080400080,   0x10821022420200,
416         0x9488a82104100100, 0x1004800041100,     0x81600a0034008080,
417         0xa00056210280400,  0x5124088200,        0x4210410010228202,
418         0x1802230840001081, 0x1002102000400901, 0x1100c46010000901,
419         0x281000408001003,  0xc001001c00028809, 0x10020008008c4102,
420         0x280005008c014222,
421     };
422     static constexpr uint64_t D_MAGIC[64] = {
423         0x811100100408200,  0x412100401044020,   0x404044c00408002,
424         0xa0c070200010102,  0x104042001400008,   0x8802013008080000,
425         0x1001008860080080, 0x20220044202800,    0x2002610802080160,
426         0x4080800808610,    0x91c2800a10a0132,   0x400242401822000,
427         0x8530040420040001, 0x142010c210048,     0x8841820801241004,
428         0x804212084108801,  0x2032402094100484, 0x402202110010210a2,
429         0x8010000800202020, 0x800240421a800,     0x62200401a00444,
430         0x224082200820845,  0x106021492012000,   0x8481020082849000,
431         0x40a110c59602800,  0x10020108020400,    0x208c020844080010,
432         0x2000480004012020, 0x8001004004044000, 0xa044104128080200,
433         0x1108008015cc1400, 0x8284004801844400, 0x8180a020c2004,
434         0x9101004080100,    0x8840264108800c0,   0xc004200900200900,
435         0x8040008020020020, 0x20010802e1920200, 0x80204000480a0,
436         0xc0a80a100008400,  0x4018808114000,     0x90092200b9000,
437         0x80020c0048000400, 0x6018005500,        0x80a0204110a00,
438         0x4018808407201,    0x6050040806500280, 0x108208400c40180,
439         0x803081210840480,  0x201210402200200,   0x200010400920042,
440         0x902000a884110010, 0x851002021004,      0x43c08020120,
441         0x6140500501010044, 0x200a04440400c028, 0x14a002084046000,
442         0x10002409041040,   0x100022020500880b, 0x1000000000460802,
```

```
443            0x21084104410,        0x8000001053300104, 0x4000182008c20048,
444            0x112088105020200,
445        };
446        int size = 0;
447        vector<uint64_t> occupancies;
448        vector<uint64_t> possible_moves;
449        for (Square sq = SQ_A1; sq <= SQ_H8; ++sq) {
450            occupancies.clear();
451            possible_moves.clear();
452            Magic& m = magics[sq];
453            uint64_t edges = ((Rank1BB | Rank8BB) & ~rank_bb(sq)) |
454                                ((FileABB | FileHBB) & ~file_bb(sq));
455            uint64_t moves_bb = reachable_squares(mt, sq, 0) & ~edges;
456            m.mask = moves_bb;
457            m.shift = 64 - popcount(m.mask);
458            m.magic = (mt == ORTHOGONAL ? O_MAGIC : D_MAGIC)[sq];
459            m.moves = table + size;
460            uint64_t b = 0;
461            do {
462                occupancies.push_back(b);
463                possible_moves.push_back(reachable_squares(mt, sq, b));
464                b = (b - moves_bb) & moves_bb;
465                size++;
466            } while (b);
467            for (size_t j = 0; j < occupancies.size(); j++) {
468                int32_t index = m.index(occupancies[j]);
469                m.moves[index] = possible_moves[j];
470            }
471        }
472    }
473
474    // Initializes both orthogonal and diagonal magic bitboard tables
475    void init_all_magics() {
476        init_magics(ORTHOGONAL, orthogonalTable, orthogonalMagics);
477        init_magics(DIAGONAL, diagonalTable, diagonalMagics);
478    }
```

Listing 40: Complete game board code (part 2/4 – utility functions and magic bitboards)

```
472    // Compact move representation: packs source and destination squares
```

```cpp
// into a 16-bit integer (6 bits each). Move::none() encodes a
// pass (data = 0, i.e., from = a1, to = a1), used when a player has
// no legal move
class Move {
    uint16_t data;
public:
    constexpr Move() = default;
    constexpr explicit Move(uint16_t d) : data(d) {}
    constexpr explicit Move(Square from, Square to)
        : data((to << 6) + from) {}
    constexpr Square from_sq() const {
        return Square(data & 0x3F);
    }
    constexpr Square to_sq() const {
        return Square((data >> 6) & 0x3F);
    }
    constexpr uint16_t raw() const { return data; }
    static constexpr Move none() { return Move(0); }
    constexpr bool operator==(const Move& m) const {
        return data == m.data;
    }
    constexpr bool operator!=(const Move& m) const {
        return data != m.data;
    }
    constexpr bool operator<(const Move& m) const {
        return make_pair(from_sq(), to_sq()) <
                make_pair(m.from_sq(), m.to_sq());
    }
    constexpr explicit operator bool() const {
        return data != 0;
    }
};

// Prints a move in "from:to" notation (e.g., "d4:d7")
ostream& operator<<(ostream& os, const Move& m) {
    static constexpr string_view square2string[SQUARE_NB] = {
        "a1", "b1", "c1", "d1", "e1", "f1", "g1", "h1",
        "a2", "b2", "c2", "d2", "e2", "f2", "g2", "h2",
        "a3", "b3", "c3", "d3", "e3", "f3", "g3", "h3",
        "a4", "b4", "c4", "d4", "e4", "f4", "g4", "h4",
        "a5", "b5", "c5", "d5", "e5", "f5", "g5", "h5",
        "a6", "b6", "c6", "d6", "e6", "f6", "g6", "h6",
```

```
515          "a7", "b7", "c7", "d7", "e7", "f7", "g7", "h7",
516          "a8", "b8", "c8", "d8", "e8", "f8", "g8", "h8",
517      };
518      os << square2string[m.from_sq()] << ':'
519          << square2string[m.to_sq()];
520      return os;
521  }
522
523  // Stack-allocated move list with a fixed capacity. Avoids heap
524  // allocation during move generation.
525  // Over millions of random games, the observed maximum number of
526  // moves never exceeded 73; 75 is a safe upper bound
527  static constexpr uint16_t MAX_NB_MOVES = 75;
528  class MoveList {
529      Move move_list[MAX_NB_MOVES], *last;
530  public:
531      constexpr MoveList() : last(move_list) {}
532      constexpr const Move* begin() const { return move_list; }
533      constexpr const Move* end() const { return last; }
534      constexpr Move* begin() { return move_list; }
535      constexpr Move* end() { return last; }
536      constexpr size_t size() const {
537          return last - move_list;
538      }
539      constexpr const Move& operator[](size_t i) const {
540          return move_list[i];
541      }
542      constexpr Move& operator[](size_t i) { return move_list[i]; }
543      constexpr Move* data() { return move_list; }
544      friend class Yolah;
545  };
546
547  // Forward declaration of test function
548  namespace test {
549      void random_games(size_t nb_games,
550                        optional<uint64_t> seed = nullopt);
551  }
552  struct YolahWithMoves;
553
554  // Board game. Uses three bitboards (black, white, holes) to
555  // represent the board. Each player starts with 4 pieces. On each
556  // turn, a piece slides orthogonally or diagonally to a free square;
```

94

```
557    // the departure square becomes a hole. The game ends when no piece
558    // can move. The player with the most moves wins
559    class Yolah {
560        uint64_t black = BLACK_INITIAL_POSITION;
561        uint64_t white = WHITE_INITIAL_POSITION;
562        uint64_t holes = 0;          // destroyed squares
563        uint8_t black_score = 0;   // number of moves played by black
564        uint8_t white_score = 0;   // number of moves played by white
565        uint8_t ply = 0;             // current ply (0 = black's first turn)
566    public:
567        // The game is over when no piece of either player has an adjacent
568        // free square
569        constexpr bool game_over() const {
570            uint64_t possible = ~holes & ~black & ~white;
571            uint64_t players  = black | white;
572            uint64_t around_players = shift<NORTH>(players) |
573                shift<SOUTH>(players) | shift<EAST>(players) |
574                shift<WEST>(players) | shift<NORTH_EAST>(players) |
575                shift<NORTH_WEST>(players) | shift<SOUTH_EAST>(players) |
576                shift<SOUTH_WEST>(players);
577            return (around_players & possible) == 0;
578        }
579
580        // Returns BLACK (0) or WHITE (1) based on parity of ply
581        constexpr uint8_t current_player() const {
582            return ply & 1;
583        }
584
585        constexpr uint8_t nb_plies() const {
586            return ply;
587        }
588
589        constexpr pair<uint8_t, uint8_t> score() const {
590            return {black_score, white_score};
591        }
592
593        // Returns the content of a square: BLACK, WHITE, HOLE, or FREE
594        constexpr uint8_t get(Square sq) const {
595            uint64_t bb = square_bb(sq);
596            if (holes & bb) return HOLE;
597            if (black & bb) return BLACK;
598            if (white & bb) return WHITE;
```

95

```
599            return FREE;
600        }
601
602        constexpr uint8_t get(int i, int j) const {
603            return get(square_of(i, j));
604        }
605
606        // Generates all legal moves for the given player into the
607        // provided MoveList. Since each player always has exactly
608        // 4 pieces, the first loop is fully unrolled (version 2)
609        // to avoid branch mispredictions. If no move is available,
610        // a single Move::none() is added
611        void moves(uint8_t player, MoveList& moves) const {
612            Move* move_list = moves.move_list;
613            uint64_t occupied = black | white | holes;
614            uint64_t bb = player == BLACK ? black : white;
615
616            // Version 1: generic loop
617            // while (bb) {
618            //     Square from = pop_lsb(bb);
619            //     uint64_t b = moves_bb(from, occupied) & ~occupied;
620            //     while (b) {
621            //         *move_list++ = Move(from, pop_lsb(b));
622            //     }
623            // }
624
625            // Version 2: unrolled loop for the 4 pieces
626            Square from0 = pop_lsb(bb);
627            Square from1 = pop_lsb(bb);
628            Square from2 = pop_lsb(bb);
629            Square from3 = pop_lsb(bb);
630
631            uint64_t b0 = moves_bb(from0, occupied) & ~occupied;
632            uint64_t b1 = moves_bb(from1, occupied) & ~occupied;
633            uint64_t b2 = moves_bb(from2, occupied) & ~occupied;
634            uint64_t b3 = moves_bb(from3, occupied) & ~occupied;
635
636            while (b0) {
637                *move_list++ = Move(from0, pop_lsb(b0));
638            }
639            while (b1) {
640                *move_list++ = Move(from1, pop_lsb(b1));
```

96

```cpp
641            }
642            while (b2) {
643                *move_list++ = Move(from2, pop_lsb(b2));
644            }
645            while (b3) {
646                *move_list++ = Move(from3, pop_lsb(b3));
647            }
648            if (move_list == moves.move_list) [[unlikely]] {
649                *move_list++ = Move::none();
650            }
651
652            moves.last = move_list;
653        }
654
655        // Generates moves for the current player
656        void moves(MoveList& moves) const {
657            this->moves(current_player(), moves);
658        }
659
660        // Selects a uniformly random legal move without allocating a
661        // move list while minimizing branches
662        Move random_move(mt19937& mt) const {
663            uint64_t occupied = black | white | holes;
664            uint64_t player_bb =
665                current_player() == BLACK ? black : white;
666            Square from0 = pop_lsb(player_bb);
667            Square from1 = pop_lsb(player_bb);
668            Square from2 = pop_lsb(player_bb);
669            Square from3 = pop_lsb(player_bb);
670            uint64_t b0 = moves_bb(from0, occupied) & ~occupied;
671            uint64_t b1 = moves_bb(from1, occupied) & ~occupied;
672            uint64_t b2 = moves_bb(from2, occupied) & ~occupied;
673            uint64_t b3 = moves_bb(from3, occupied) & ~occupied;
674            int n0 = popcount(b0);
675            int n1 = popcount(b1);
676            int n2 = popcount(b2);
677            int n3 = popcount(b3);
678            int n = n0 + n1 + n2 + n3;
679            if (n == 0) [[unlikely]] {
680                return Move::none();
681            }
682            uniform_int_distribution<int> d(0, n - 1);
```

97

```
683              int bit = d(mt);
684              int bb_index = (bit >= n0) + (bit >= n0 + n1) +
685                             (bit >= n0 + n1 + n2);
686              bit -= (bb_index > 0) * n0 + (bb_index > 1) * n1 +
687                     (bb_index > 2) * n2;
688              Square from = array{from0, from1, from2, from3}[bb_index];
689              uint64_t bb = array{ b0, b1, b2, b3 }[bb_index];
690              // _pdep_u64 extracts the bit-th destination from bb without
691              // a loop. Example: bb = 0b01010100 (3 set bits),
692              // bit = 2 (3rd move):
693              //    1ULL << 2           = 0b00000100
694              //    _pdep_u64(0b100,bb) = 0b01000000  (the 3rd set bit of bb)
695              //    countr_zero(...)    = 6            (square index)
696              //
697              // Without _pdep_u64, the equivalent code would be:
698              // for (int i = 0; i < bit; i++) bb &= bb - 1;
699                                             // bb &= bb - 1
700                                             // clears the least
701                                             // significant bit of bb
702              // Square to = countr_zero(bb);
703              Square to = Square(countr_zero(_pdep_u64(1ULL << bit, bb)));
704              return Move(from, to);
705         }
706
707     // Applies a move: moves the piece from source to destination,
708     // turns the source square into a hole, and increments the
709     // player's score.
710     // For Move::none() (pass), only the ply counter is incremented
711     void play(Move m) {
712         if (m != Move::none()) [[likely]] {
713             uint64_t pos1 = square_bb(m.from_sq());
714             uint64_t pos2 = square_bb(m.to_sq());
715             if (ply & 1) {
716                 white ^= pos1 | pos2;
717                 white_score++;
718             } else {
719                 black ^= pos1 | pos2;
720                 black_score++;
721             }
722             holes |= pos1;
723         }
724         ply++;
```

```
725        }
726
727        // Reverses a move: restores the piece, removes the hole,
728        // decrements score
729        void undo(Move m) {
730            ply--;
731            if (m != Move::none()) [[likely]] {
732                uint64_t pos1 = square_bb(m.from_sq());
733                uint64_t pos2 = square_bb(m.to_sq());
734                if (ply & 1) {
735                    white ^= pos1 | pos2;
736                    white_score--;
737                } else {
738                    black ^= pos1 | pos2;
739                    black_score--;
740                }
741                holes ^= pos1;
742            }
743        }
744
745        constexpr bool operator==(const Yolah& other) const {
746            return black == other.black
747                && white == other.white
748                && holes == other.holes
749                && black_score == other.black_score
750                && white_score == other.white_score
751                && ply == other.ply;
752        }
753
754        constexpr bool operator!=(const Yolah& other) const {
755            return !(*this == other);
756        }
757    };
```

Listing 41: Complete game board code (part 3/4 – Move, MoveList and Yolah classes)

```
742    // Pairs a Yolah board with its legal moves for display purposes
743    struct YolahWithMoves {
744        const Yolah& yolah;
745        const MoveList& moves;
```

```cpp
746    YolahWithMoves(const Yolah& y,
747                   const MoveList& m) : yolah(y), moves(m) {}
748  };
749
750  // Displays the board with Unicode box-drawing, marking
751  // reachable squares
752  ostream& operator<<(ostream& os, const YolahWithMoves& ym) {
753      char grid[8][8];
754      const Yolah& yolah = ym.yolah;
755      const MoveList& moves = ym.moves;
756      static constexpr string_view players[] = {
757          "Black player",
758          "White player"
759      };
760      static constexpr char content[] = {'B', 'W', 'H', '.'};
761      uint8_t player = yolah.current_player();
762      os << players[player] << '\n';
763      for (int i = 0; i < 8; i++) {
764          for (int j = 0; j < 8; j++) {
765              Square dst = square_of(i, j);
766              if (any_of(begin(moves), end(moves), [&](const Move& m) {
767                  return m.to_sq() == dst;
768              })) {
769                  grid[i][j] = player == BLACK ? 'X' : 'x';
770              }
771              else grid[i][j] = content[yolah.get(dst)];
772          }
773      }
774      os << "\n    ┌─┬─┬─┬─┬─┬─┬─┬─┐\n";
775      for (int i = 7; i >= 0; i--) {
776          os << i + 1 << " │";
777          for (int j = 0; j < 8; j++) {
778              char c = grid[i][j];
779              if (c == 'B') os << " ○ ";
780              else if (c == 'W') os << " ● ";
781              else if (c == 'H') os << "   ";
782              else if (c == '.') os << " · ";
783              else os << ' ' << c << ' ';
784              os << "│";
785          }
786          os << '\n';
787          if (i > 0) os << "  ├─┼─┼─┼─┼─┼─┼─┼─┤\n";
```

```
788        }
789        os << "  └──────┴──────┴──────┴──────┴──────┴──────┴──────┘\n";
790        os << "     a    b    c    d    e    f    g    h\n";
791        const auto [black_score, white_score] = yolah.score();
792        os << "score: " << int(black_score) << '/'
793           << int(white_score) << '\n';
794        return os;
795    }
796
797    // Displays the board without move annotations
798    ostream& operator<<(ostream& os, const Yolah& yolah) {
799        return os << YolahWithMoves(yolah, MoveList());
800    }
801
802    // Plays random games using standard move generation. When
803    // STEP_BY_STEP is true, prints board and moves at each turn
804    // and waits for user input
805    template<bool STEP_BY_STEP = true>
806    void play_random_games(size_t nb_games,
807                           optional<uint64_t> seed = nullopt) {
808        MoveList moves;
809        random_device rd;
810        mt19937 mt(seed.value_or(rd()));
811        size_t black_wins = 0;
812        size_t white_wins = 0;
813        size_t draws = 0;
814
815        for (size_t i = 0; i < nb_games; i++) {
816            Yolah yolah;
817            while (!yolah.game_over()) {
818                if constexpr (STEP_BY_STEP) cout << yolah << '\n';
819                yolah.moves(moves);
820                if constexpr (STEP_BY_STEP) {
821                    sort(begin(moves), end(moves));
822                    cout << format("# moves: {}\n", moves.size());
823                    for (const auto& m : moves) {
824                        cout << m << ' ';
825                    }
826                    cout << "\n\n";
827                    cout << YolahWithMoves(yolah, moves) << '\n';
828                }
829                uniform_int_distribution<int> d(0, moves.size() - 1);
```

```
830              Move m = moves[d(mt)];
831              if constexpr (STEP_BY_STEP) {
832                  cout << m << '\n';
833                  string _;
834                  getline(std::cin, _);
835              }
836              yolah.play(m);
837          }
838          if constexpr (STEP_BY_STEP) cout << yolah << '\n';
839
840          auto [black_score, white_score] = yolah.score();
841          if (black_score > white_score) {
842              black_wins++;
843          } else if (white_score > black_score) {
844              white_wins++;
845          } else {
846              draws++;
847          }
848      }
849
850      if constexpr (!STEP_BY_STEP) {
851          cout << format("\n=== Game Statistics ===\n");
852          cout << format("Total games: {}\n", nb_games);
853          cout << format("Black wins:  {} ({:.1f}%)\n", black_wins,
854                          100.0 * black_wins / nb_games);
855          cout << format("White wins:  {} ({:.1f}%)\n", white_wins,
856                          100.0 * white_wins / nb_games);
857          cout << format("Draws:       {} ({:.1f}%)\n", draws,
858                          100.0 * draws / nb_games);
859      }
860  }
861
862  // Plays random games using random_move (no move list allocation)
863  void play_random_games_fast(size_t nb_games,
864                              optional<uint64_t> seed = nullopt) {
865      random_device rd;
866      mt19937 mt(seed.value_or(rd()));
867      size_t black_wins = 0;
868      size_t white_wins = 0;
869      size_t draws = 0;
870      for (size_t i = 0; i < nb_games; i++) {
871          Yolah yolah;
```

```
872        while (!yolah.game_over()) {
873            Move m = yolah.random_move(mt);
874            yolah.play(m);
875        }
876        auto [black_score, white_score] = yolah.score();
877        if (black_score > white_score) {
878            black_wins++;
879        } else if (white_score > black_score) {
880            white_wins++;
881        } else {
882            draws++;
883        }
884    }
885    cout << format("\n=== Game Statistics ===\n");
886    cout << format("Total games: {}\n", nb_games);
887    cout << format("Black wins:  {} ({:.1f}%)\n", black_wins,
888                    100.0 * black_wins / nb_games);
889    cout << format("White wins:  {} ({:.1f}%)\n", white_wins,
890                    100.0 * white_wins / nb_games);
891    cout << format("Draws:       {} ({:.1f}%)\n", draws,
892                    100.0 * draws / nb_games);
893 }
894
895 // Differential and property-based tests: differential testing
896 // validates move generation by comparing against a naive reference
897 // implementation; property-based testing checks invariants like undo
898 // reversibility and game-over detection
899 namespace test {
900    namespace {
901        using format;
902
903        constexpr string_view RED    = "\033[1;31m";
904        constexpr string_view GREEN  = "\033[1;32m";
905        constexpr string_view YELLOW = "\033[1;33m";
906        constexpr string_view RESET  = "\033[0m";
907        constexpr string_view BOLD   = "\033[1m";
908
909        struct TestResult {
910            bool passed;
911            string message;
912            operator bool() const { return passed; }
913        };
```

```
914
915        TestResult pass() { return {true, ""}; }
916        TestResult fail(string msg) {
917            return {false, move(msg)};
918        }
919
920        // Reference move generator: brute-force loop over all
921        // squares/directions
922        vector<Move> slow_moves_generation(const Yolah& yolah) {
923            vector<Move> res;
924            for (int i = 0; i < 8; i++) {
925                for (int j = 0; j < 8; j++) {
926                    Square from = square_of(i, j);
927                    if (yolah.get(from) != yolah.current_player()) {
928                        continue;
929                    }
930                    for (int di = -1; di <= 1; di++) {
931                        for (int dj = -1; dj <= 1; dj++) {
932                            if (di == 0 && dj == 0) continue;
933                            int ii = i + di;
934                            int jj = j + dj;
935                            for(;;) {
936                                if (ii < 0 || ii >= 8 ||
937                                    jj < 0 || jj >= 8) break;
938                                Square to = square_of(ii,jj);
939                                if (yolah.get(to) != FREE) break;
940                                res.emplace_back(from, to);
941                                ii += di;
942                                jj += dj;
943                            }
944                        }
945                    }
946                }
947            }
948            if (res.empty()) res.push_back(Move::none());
949            return res;
950        }
951
952        // Verifies that magic and naive generators produce the same
953        // move count
954        TestResult check_move_count(const MoveList& fast,
955                                    const vector<Move>& expected) {
```

```
956            if (fast.size() != expected.size()) {
957                return fail(format("# of moves: expected {} got {}",
958                                    expected.size(), fast.size()));
959            }
960            return pass();
961        }
962
963        // Sorts magic and naive lists and checks element-wise
964        // equality, reports differences
965        TestResult check_move_lists_equal(MoveList& fast,
966                                          vector<Move>& expected,
967                                          const Yolah& yolah) {
968            sort(begin(fast), end(fast));
969            sort(begin(expected), end(expected));
970            if (equal(begin(fast), end(fast),
971                      begin(expected), end(expected))) {
972                return pass();
973            }
974            ostringstream oss;
975            oss << "move lists differ\n" << yolah << '\n';
976            vector<Move> only_in_fast, only_in_expected;
977            set_difference(begin(fast), end(fast),
978                           begin(expected), end(expected),
979                           back_inserter(only_in_fast));
980            set_difference(begin(expected), end(expected),
981                           begin(fast), end(fast),
982                           back_inserter(only_in_expected));
983            if (!only_in_expected.empty()) {
984                oss << "  Only in expected: ";
985                for (const auto& m : only_in_expected) {
986                    oss << m << ' ';
987                }
988                oss << '\n';
989            }
990            if (!only_in_fast.empty()) {
991                oss << "  Only in fast: ";
992                for (const auto& m : only_in_fast) oss << m << ' ';
993                oss << '\n';
994            }
995            return fail(oss.str());
996        }
997
```

105

```
998         // Verifies that undo restores the board to its previous state
999         TestResult check_undo(const Yolah& before,
1000                               const Yolah& after) {
1001             if (before == after) return pass();
1002             ostringstream oss;
1003             oss << "undo failed\n  Previous state:\n" << before
1004                 << "\n  State after undo:\n" << after << '\n';
1005             return fail(oss.str());
1006         }

1007
1008         // When the game is over, the only legal move must be
1009         // Move::none()
1010         TestResult check_game_over_moves(const Yolah& yolah,
1011                                          const MoveList& moves) {
1012             if (moves.size() == 1 && moves[0] == Move::none()) {
1013                 return pass();
1014             }
1015             ostringstream oss;
1016             oss << "only Move::none() should be available when"
1017                 << " game is over\n"
1018                 << YolahWithMoves(yolah, moves) << '\n';
1019             return fail(oss.str());
1020         }

1021
1022         // Move::none() must not alter the board, only increment the
1023         // ply counter
1024         TestResult check_none_move_execution(const Yolah& before,
1025                                              const Yolah& after) {
1026             bool ok = true;
1027             for (Square sq = SQ_A1; sq <= SQ_H8; ++sq) {
1028                 if (before.get(sq) != after.get(sq)) {
1029                     ok = false;
1030                     break;
1031                 }
1032             }
1033             if (!ok) {
1034                 return fail("Move::none() must not change the board
    ↪  content");
1035             }
1036             if (before.nb_plies() + 1 != after.nb_plies()) {
1037                 return fail("Move::none() must increment the number of
    ↪  plies");
```

106

```
1038                    }
1039                    return pass();
1040               }
1041
1042          // A regular move must: place a piece at 'to', create a hole
1043          // at 'from', increment ply, and leave all other squares
1044          // unchanged
1045          TestResult check_regular_move_execution(const Yolah& before,
1046                                                   const Yolah& after,
1047                                                   Move m) {
1048               Square from = m.from_sq();
1049               Square to = m.to_sq();
1050               bool ok = true;
1051               for (Square sq = SQ_A1; sq <= SQ_H8; ++sq) {
1052                    if (sq == from || sq == to) continue;
1053                    if (before.get(sq) != after.get(sq)) {
1054                         ok = false;
1055                         break;
1056                    }
1057               }
1058               if (ok && before.nb_plies() + 1 == after.nb_plies() &&
1059                    after.get(from) == HOLE &&
1060                    after.get(to) == before.current_player()) {
1061                         return pass();
1062               }
1063               ostringstream oss;
1064               oss << "Move execution incorrect\n" << after
1065                    << "\n  Move: " << m << '\n';
1066               if (!ok) {
1067                    oss << " Squares not concerned by the move must not
    change";
1068               }
1069               if (after.get(from) != HOLE) {
1070                    oss << "  From square should be hole\n";
1071               }
1072               if (after.get(to) != before.current_player()) {
1073                    oss << " To square should contain a piece from "
1074                         << (before.current_player() == BLACK ? "black" :
1075                                                                "white")
1076                         << " player\n";
1077               }
1078               if (before.nb_plies() + 1 != after.nb_plies()) {
```

```
1079                    oss << " The number of plies must be incremented\n";
1080                }
1081                return fail(oss.str());
1082            }
1083        }

1084

1085        // Runs nb_games random games, testing move generation, play/undo,
1086        // and game_over at each step
1087        void random_games(size_t nb_games, optional<uint64_t> seed) {
1088            MoveList fast_moves;
1089            random_device rd;
1090            mt19937 mt(seed.value_or(rd()));
1091            size_t total_tests = 0;
1092            size_t passed_tests = 0;

1093

1094            auto run_test = [&](TestResult result) -> bool {
1095                total_tests++;
1096                if (result) {
1097                    passed_tests++;
1098                    return true;
1099                }
1100                cout << format("{}FAIL:{} {}\n", RED, RESET,
1101                                    result.message);
1102                return false;
1103            };

1104

1105            cout << format("{}\n=== Running Random Games Tests ===\n{}",
1106                                BOLD, RESET);

1107

1108            for (size_t i = 0; i < nb_games; i++) {
1109                Yolah yolah;
1110                while (!yolah.game_over()) {
1111                    yolah.moves(fast_moves);
1112                    vector<Move> expected_moves =
1113                        slow_moves_generation(yolah);
1114                    if ( !run_test(
1115                            check_move_count(fast_moves,
1116                                                expected_moves)) ) break;
1117                    if ( !run_test(
1118                            check_move_lists_equal(fast_moves,
1119                                                expected_moves,
1120                                                yolah)) ) break;
```

108

```
1121                    uniform_int_distribution<int> d(0,
1122                                        fast_moves.size()-1);
1123                Move m = fast_moves[d(mt)];
1124                Yolah before = yolah;
1125                yolah.play(m);
1126                if (m == Move::none()) {
1127                    if (!run_test(
1128                            check_none_move_execution(before,
1129                                                yolah))) break;
1130                } else {
1131                    if (!run_test(
1132                            check_regular_move_execution(before,
1133                                                yolah,
1134                                                m))) break;
1135                }
1136                yolah.undo(m);
1137                if (!run_test(check_undo(before, yolah))) break;
1138                yolah.play(m);
1139            }
1140            if (!yolah.game_over()) continue;
1141            yolah.moves(fast_moves);
1142            if (!run_test(
1143                    check_game_over_moves(yolah,
1144                                    fast_moves))) continue;
1145            yolah.play(Move::none());
1146            yolah.moves(fast_moves);
1147            if (!run_test(
1148                    check_game_over_moves(yolah,
1149                                    fast_moves))) continue;
1150        }
1151
1152        cout << format("\n{}=== Test Summary ==={}\n", BOLD, RESET);
1153        cout << format("Total tests: {}\n", total_tests);
1154        cout << format("Passed: {}{}{}\n", GREEN, passed_tests,
1155                        RESET);
1156        cout << format("Failed: {}{}{}\n",
1157                        (passed_tests == total_tests ? GREEN : RED),
1158                        total_tests - passed_tests, RESET);
1159        if (passed_tests == total_tests) {
1160            cout << format("{}All tests passed!{}\n", GREEN, RESET);
1161        } else {
1162            double pass_rate = 100.0 * passed_tests / total_tests;
```

```
1163                cout << format("{}Pass rate: {:.2f}%{}\n", YELLOW,
1164                                pass_rate, RESET);
1165            }
1166        }
1167    }
1168
1169    int main() {
1170        init_all_magics();
1171        //play_random_games<true>(1000000, 42);
1172        play_random_games<false>(1000000, 42);
1173        //test::random_games(10000, 42);
1174        //play_random_games_fast(1000000, 42);
1175    }
```

Listing 42: Complete game board code (part 4/4 – display, random games, tests and main)

```
$ g++ -std=c++20 -O3 -march=native -Wall -Wpedantic chapter02.cpp\
-o chapter02
$ ./chapter02
=== Game Statistics ===
Total games: 1000000
Black wins:  499124 (49.9%)
White wins:  395468 (39.5%)
Draws:       105408 (10.5%)

real    0m3,075s
user    0m3,067s
sys     0m0,008s
```

Listing 43: Results of executing the board code program (listings 39–42).

Chapter 3

Artificial 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

BMI2 Bit Manipulation Instruction Set 2. 80

ILP Instruction Level Parallelism. 74

IPC Instructions Per Cycle. 72, 76, 80

LLC Last Level Cache. 72

MLP Memory Level Parallelism. 72, 76

SMT Satisfiability Modulo Theories. 23, 24

Z3 Z3 Theorem Prover. 24, 25, 33

Bibliography

[1] Anthropic. *Claude Opus 4.6*. https://www.anthropic.com/claude. Large Language Model. 2026.

[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.

[10] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. ACM. 2000, pp. 268–279.

[11] William M McKeeman. "Differential testing for software". In: *Digital Technical Journal*. Vol. 10. 1. Digital Equipment Corporation, 1998, pp. 100–107.

[12] Linux Kernel Developers. *perf: Linux profiling with performance counters*. https://perf.wiki.kernel.org/. Accessed: 2025-01-21. 2025.

[13] Brendan Gregg. *FlameGraph: Stack trace visualizer*. https://github.com/brendangregg/FlameGraph. Includes stackcollapse-perf.pl. Accessed: 2025-01-21. 2011.

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Chapter 20: Performance Monitoring. 2024. Chap. 20.

[15]   Denis Bakhvalov. *Performance Analysis and Tuning on Modern CPUs*. 2nd. Accessed: 2025-01-28. 2024.

[16]   Fedor G. Pikus. *The Art of Writing Efficient Programs*. Packt Publishing, 2021. ISBN: 978-1800208117.

[17]   Sergey Kulikov. *Algorithms for Modern Hardware*. Accessed: 2025-01-28. 2022. URL: https://en.algorithmica.org/hpc/.

[18]   Ahmad Yasin. *A Top-Down Method for Performance Analysis and Counters Architecture*. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Intel Corporation. 2014.