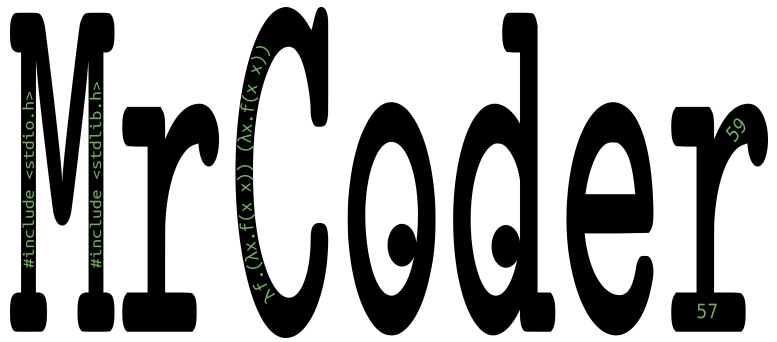
Yolah Board Game

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

Pascal Garcia

December 19, 2025


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

C’est en forgeant qu’on devient forgeron

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Yolah Game

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

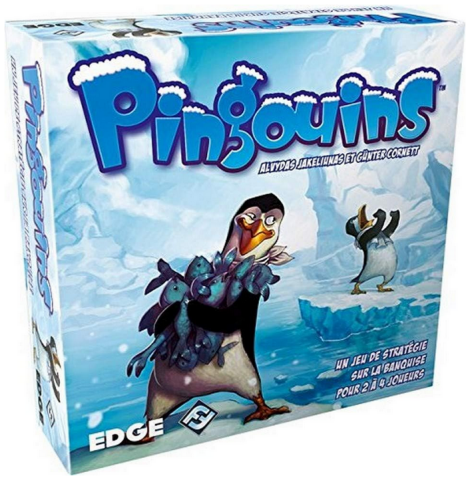


Figure 1.1: The Pingouins game box

Important

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

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

1.1.1 Game Rules

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

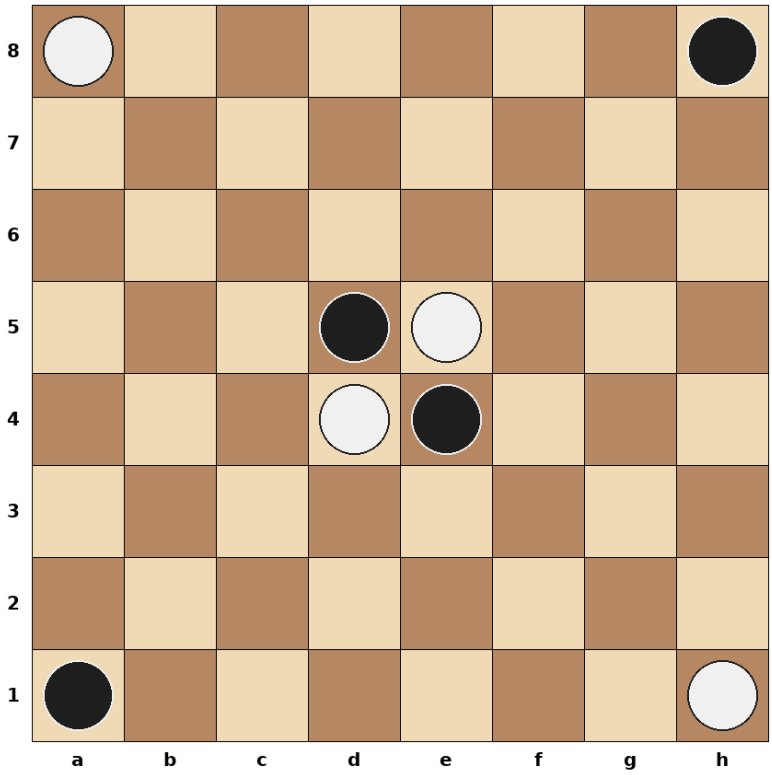


Figure 1.2: The initial configuration of the Yolah game

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

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

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

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

To summarize, the rules of Yolah are as follows:

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

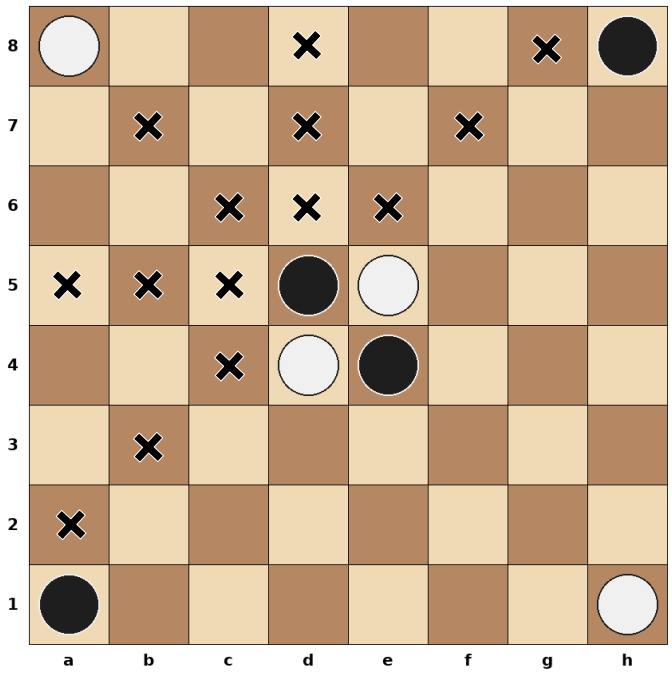


Figure 1.3: Possible moves (small black crosses) for the black piece located on square `d5`

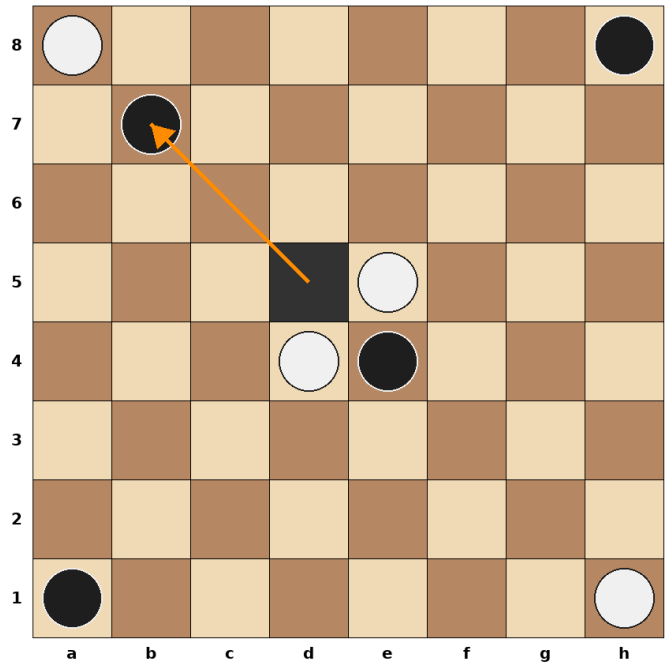


Figure 1.4: Black just moved from `d5` to `b7`. The starting square `d5` becomes inaccessible and impassable for the rest of the game

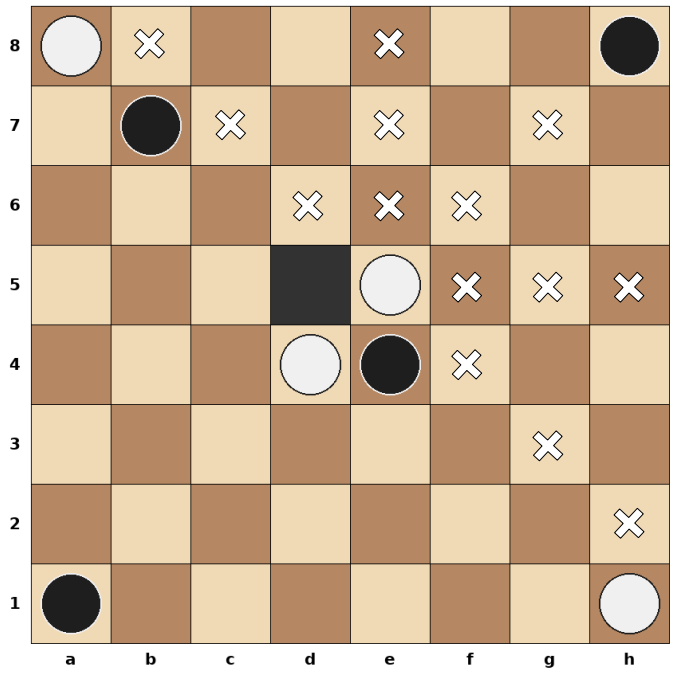


Figure 1.5: Possible moves (small white crosses) for the white piece located on square `e5`

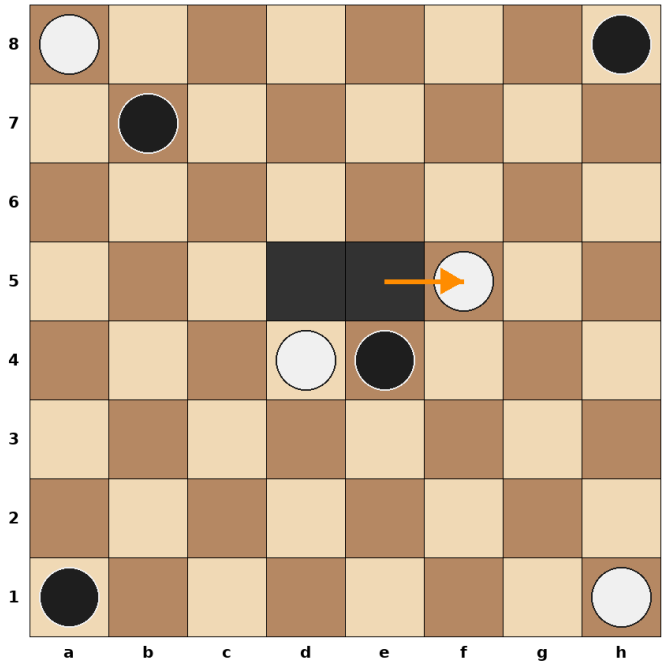


Figure 1.6: White just moved from `e5` to `f5`. The starting square `e5` becomes inaccessible and impassable for the rest of the game. The score is one point each (each player has moved once)

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

1.1.2 Interesting Characteristics of Yolah for Developing AIs

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

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

1.1.3 Game Example

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

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

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

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

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

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

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

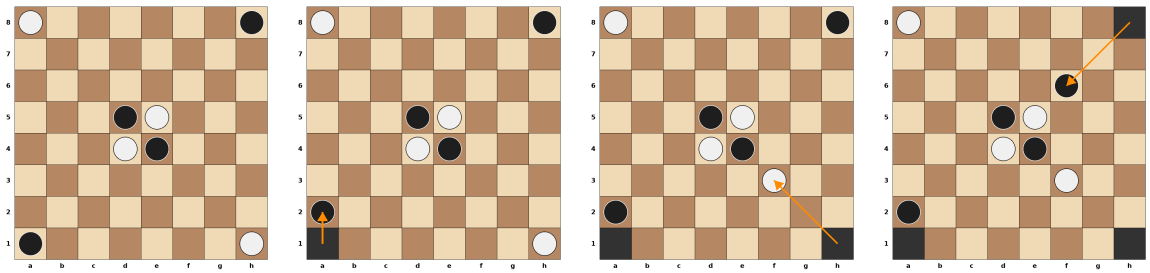
1.1.4 What's Next

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

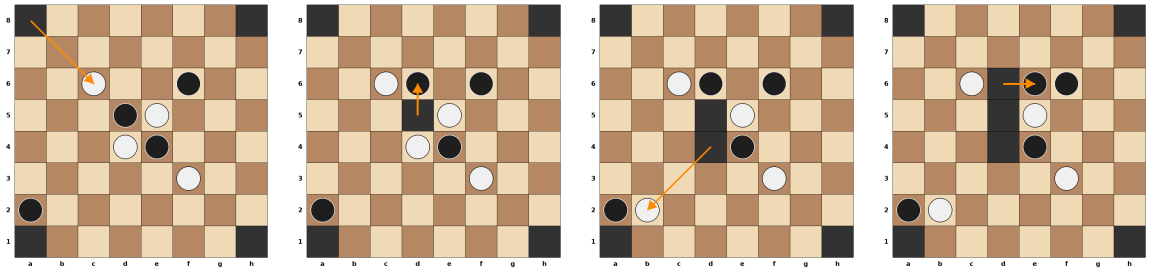
Chapter 3 describes the common interface for our different AIs. Chapter 4 presents a very simple AI based on Monte Carlo search. The next AI, described in Chapter **??**, is an evolution of the previous one and will allow, unlike the Monte Carlo AI, the development of a game tree. Note that these two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

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

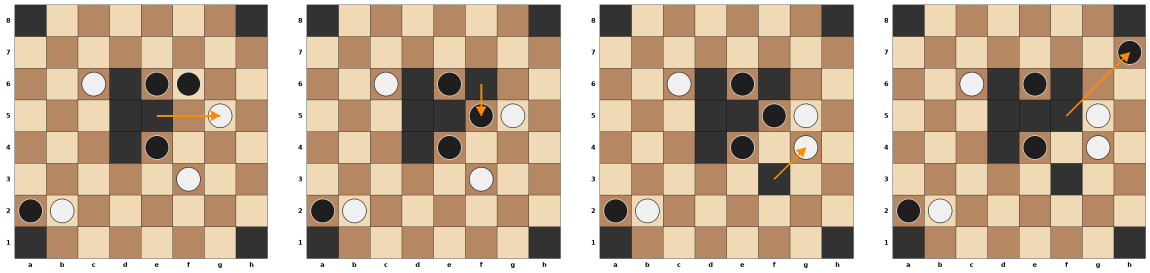
Happy reading!



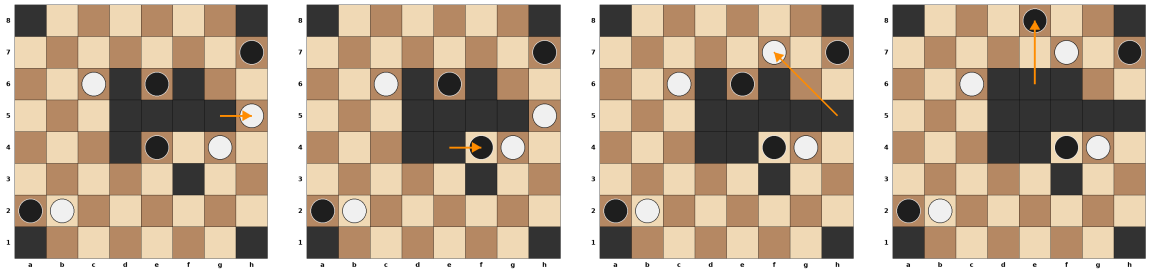
(a) Initial position (b) Move 1 (Black) (c) Move 2 (White) (d) Move 3 (Black)



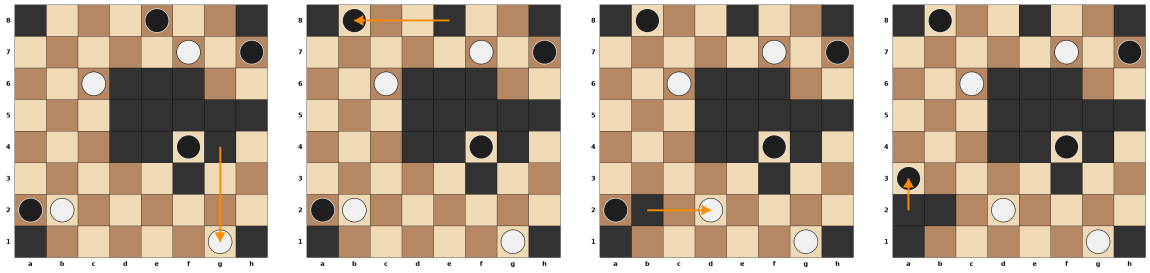
(e) Move 4 (White) (f) Move 5 (Black) (g) Move 6 (White) (h) Move 7 (Black)



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

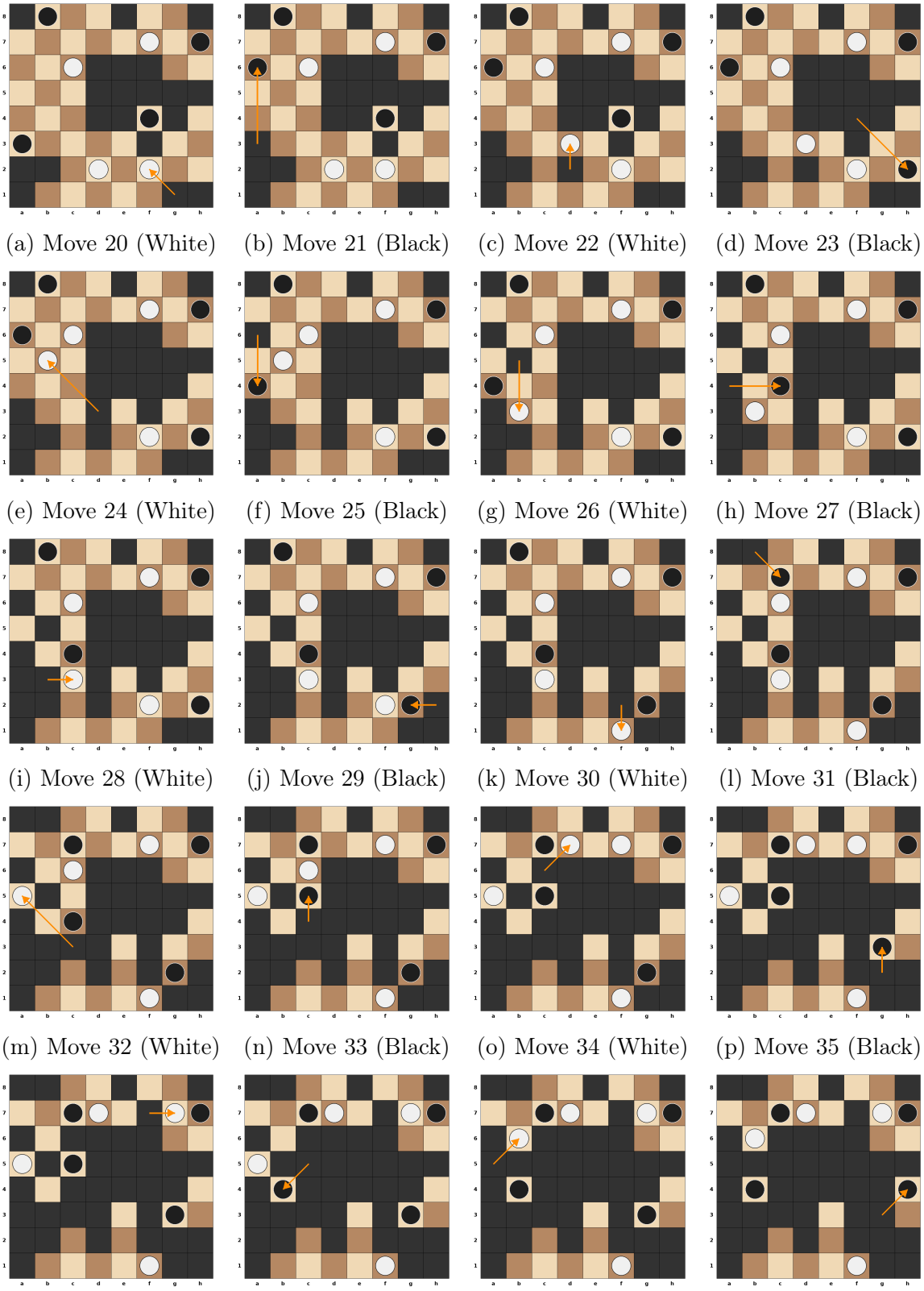


(m) Move 12 (White) (n) Move 13 (Black) (o) Move 14 (White) (p) Move 15 (Black)



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

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



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

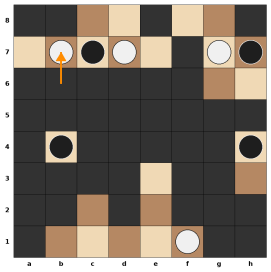
(e) Move 24 (White) (f) Move 25 (Black) (g) Move 26 (White) (h) Move 27 (Black)

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

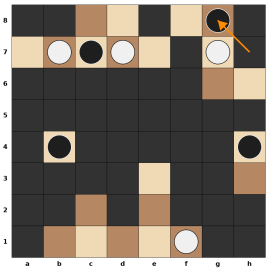
(m) Move 32 (White) (n) Move 33 (Black) (o) Move 34 (White) (p) Move 35 (Black)

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

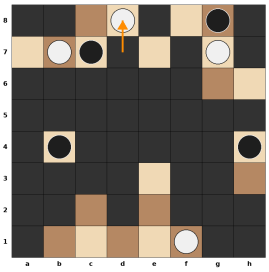
Figure 1.8: Game example between two AIs - moves 20 to 39



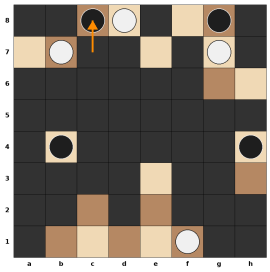
(a) Move 40 (White)



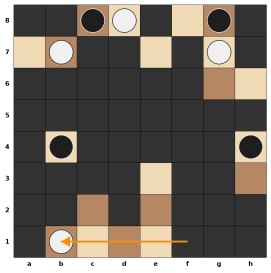
(b) Move 41 (Black)



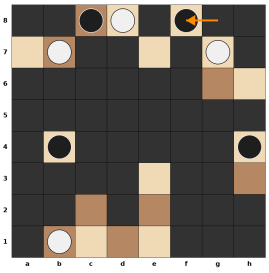
(c) Move 42 (White)



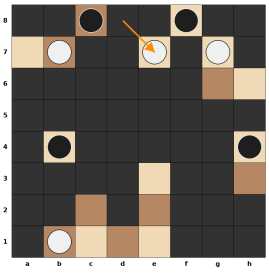
(d) Move 43 (Black)



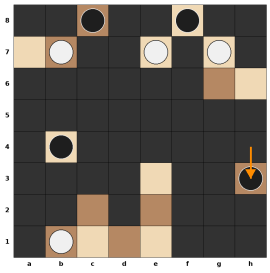
(e) Move 44 (White)



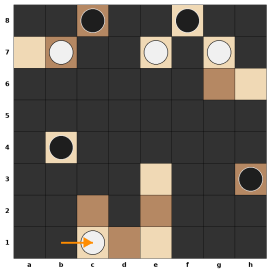
(f) Move 45 (Black)



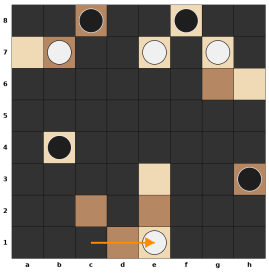
(g) Move 46 (White)



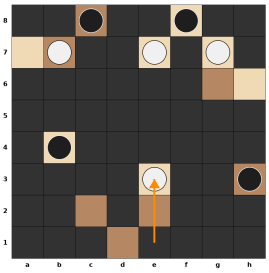
(h) Move 47 (Black)



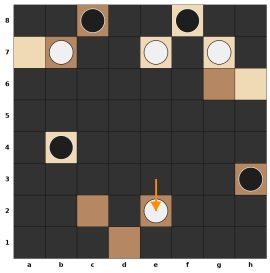
(i) Move 48 (White)



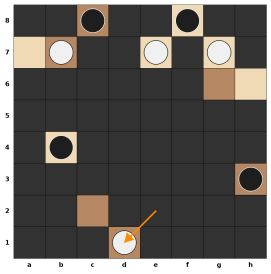
(j) Move 49 (White)



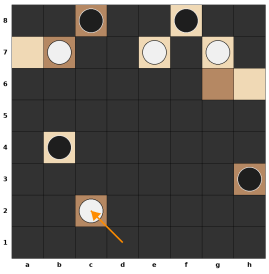
(k) Move 50 (White)



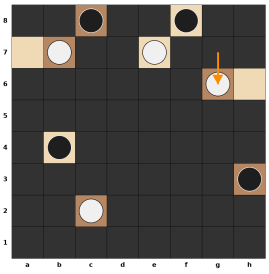
(l) Move 51 (White)



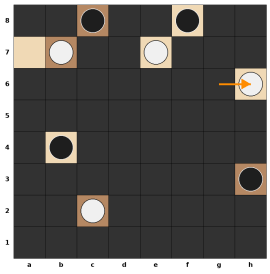
(m) Move 52 (White)



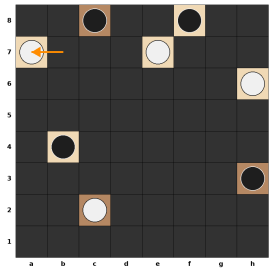
(n) Move 53 (White)



(o) Move 54 (White)



(p) Move 55 (White)



(q) Move 56 (White)

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

Chapter 2

Game Engine

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

2.1 Data Structures

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

Table 2.1: Position of each board square in the bitboard

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

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

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

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

Listing 1: Board squares

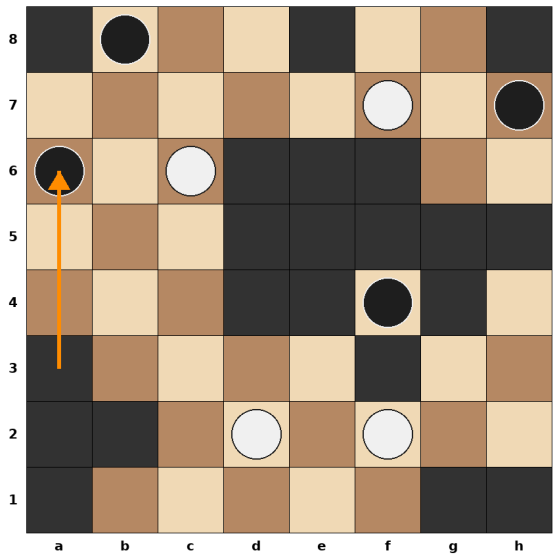


Figure 2.1: Board configuration corresponding to move 21 of the game given as an example in the previous chapter (Figure 1.8b)

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

- The positions of black pieces on the board (Table 2.2) are represented by the bitboard: `0b0000001010000000000000010000000000100000000000000000000000000000`.
- The positions of white pieces on the board (Table 2.3) are represented by the bitboard: `0b0000000000010000000000100000000000000000000000000001010000000000`.
- The positions of holes (Table 2.4) are represented by the bitboard: `0b1001000100000000001110001111100001011000001000010000001111000001`.

Table 2.2: Black piece positions

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

Table 2.3: White piece positions

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

Table 2.4: Destroyed square positions (holes)

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

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

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

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

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

The attributes of the `Yolah` class are:

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

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

```
1  class Yolah {
2      uint64_t black = BLACK_INITIAL_POSITION;
```

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

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

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

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

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

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

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

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

2.2 Game Over Test

To test for game over, we will need some enumerations and constants (Listings 3 and 5) and the `shift` function (Listing 4). The `NORTH` constant in the `Direction` enumeration (Listing 3) equals 8. Why this value? Let us revisit below, see Table 2.6, the table

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

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).

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

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

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

Listing 3: Directions

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

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

Listing 4: Shift by direction

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

`black` == 1000000000000000000000000000010000010000000000000000000000000001

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

1000000000000000000000000000010000010000000000000000000000000001

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

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

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

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

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

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

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

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

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

Table 2.9: Game board

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

Table 2.10: Positions around the pieces

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

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

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

Listing 5: Files and ranks

The function that tests for game over is called `game_over` and is described in Listing

6. Its implementation is straightforward:

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

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

Listing 6: Game over test

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

2.3 Generating Possible Piece Moves

Move generation will use a technique called *magic bitboards*. To understand this technique, I will first present magic perfect hashing through a simple example. I drew inspiration from [4] for this example.

2.3.1 Magic Perfect Hashing Function

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

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

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

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

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

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

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

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

[5] `tzcnt` on Intel.

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

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

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

Listing 7: Magic bitboard perfect hashing

In this function:

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

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

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

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

Listing 8: Random search for the `MAGIC` constant

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

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

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

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

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

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

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

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

```
1  from z3 import *
2
3  positions = {}
4  for i in range(64):
5      positions[1 << i] = i
6
7  solver = Solver()
8  MAGIC  = BitVec('magic', 64)
9  K = 6
10 bitboards = list(positions.keys())
11
12 def index(magic, k, bitboard):
13     return magic * bitboard >> (64 - k)
14
15 for i in range(64):
16     index1 = index(MAGIC, K, bitboards[i])
17     for j in range(i + 1, 64):
18         index2 = index(MAGIC, K, bitboards[j])
19         solver.add(index1 != index2)
20
21 if solver.check() == sat:
22     model = solver.model()
23     m = model[MAGIC].as_long()
24     print(f'found magic for K = {K}: {m:#x}')
25     size = 1 << K
26     table = [-1] * size
27     for bitboard, pos in positions.items():
28         table[index(m, K, bitboard) & size - 1] = pos
29     print(f'constexpr uint8_t positions[{size}] = {{')
30     for i in range(size):
31         print(f'{table[i]},', end='')
32     print('\n};')
```

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

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

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

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

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

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

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

In *C++* we would have:

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

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

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

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

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

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

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

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

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

Listing 11: Micro-benchmark execution results from Listing 10

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

2.3.2 Magic Bitboards for Piece Moves

Pour chaque configuration du plateau de jeu, nous voulons pouvoir, pour une case donnée, donner la liste des coups possibles pour cette configuration de jeu. Par exemple, pour le plateau de la figure 2.2 dont les coups possibles pour la pièce en `d3` sont représentées par des étoiles dans la table 2.11, on voudrait très efficacement pouvoir obtenir le bitboard de la table 2.12 indiquant les différents coups possibles.

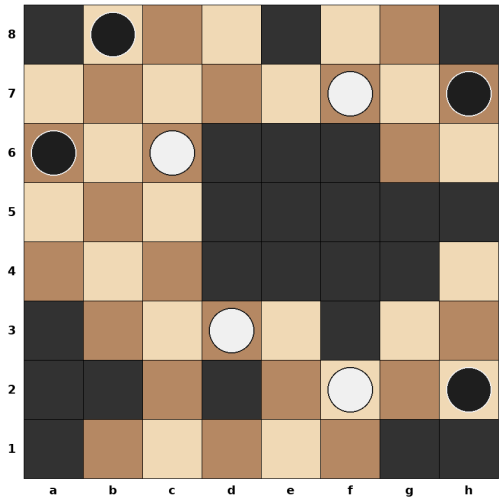


Figure 2.2: Board position example (after move 23 in figure 1.8)

Table 2.11: Possible moves for white piece at d3

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

Prenons comme exemple la case `d3`. Nous avons tout d'abord besoin d'un masque qui va nous permettre d'isoler les cases du plateau atteignables par la pièce en `d3` sans prendre en considération les obstacles. Ce masque est représenté dans la table 2.13. Soit `mask` le masque de la table 2.13 et `board` le bitboard de la table 2.14 représentant le plateau de jeu de la figure 2.2. On obtient le bitboard `occupancies`, donnant les obstacles sur les trajectoires possibles de la pièce en `d3` en faisant : `occupancies = mask & board`. Ce bitboard est donné dans la table 2.15. Il y a un 1 pour chaque case contenant une pièce ou un trou. Ce qu'il nous faut, c'est de pouvoir indexer une table, appelons la `uint64_t MOVES_D3[]`, grâce au bitboard `occupancies`, pour obtenir le bitboard des coups possibles à partir de la case `d3`. Ce bitboard est représenté dans la table 2.12. Il ne nous restera plus qu'à parcourir chacun des bits à 1 dans ce bitboard

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

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

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

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

Table 2.14: Board bitboard

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

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

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

pour connaitre les cases où l'on peut se rendre et créer le coup correspondant. Pour obtenir l'index dans `MOVES_D3`, on va utiliser la technique des magic bitboard que nous avons décrite dans la section précédente. Pour la case `d3`, il va nous falloir trouver une valeur `K` et une valeur `MAGIC` (voir listing 7) pour pouvoir obtenir le bitboard des coups possibles en faisant : `MOVES_D3[occupancies * MAGIC >> (64 - K)]`.

Pour trouver une constante `MAGIC` pour la case `d3`, nous allons avoir besoin d'énumérer tous les placements d'obstacles possibles sur les cases où peut se rendre la pièce en `d3`. Cela veut dire que l'on doit énumérer tous les sous-ensembles de 1 du bitboard de la table 2.13. Huit sous-ensembles de cette énumération sont présentés dans la figure 2.3. Il y a 25 cases sur la trajectoire de la pièce en `d3` (voir la table 2.13), cela veut dire qu'il y a $2^{25} = 33,554,432$ sous-ensembles à énumérer ! Nous voudrions réduire ce nombre de configurations des obstacles possibles sur la trajectoire de la pièce. En effet, comme nous le verrons lorsque nous détaillerons le code pour déterminer la valeur de la constante `MAGIC`, la place mémoire pour stocker les coups possibles pour une case donnée va dépendre de ce nombre de configurations.

Pour réduire ce nombre de configurations, nous n'allons pas considérer les cases sur les bords du plateau pour enlever les bits à 1 sur les lignes 1 et 8 et les colonnes a et h du masque de la table 2.13. On obtient le bitboard de la table 2.16. Il nous manquera l'information sur les bords, mais on supposera que s'il est possible de jouer en `d7` par exemple, il sera aussi possible de jouer en `d8`. Bien sûr il se peut qu'il y ait un obstacle en `d8`, mais on pourra facilement éliminer ce coup impossible comme nous le verrons dans la suite de ce chapitre. Avec cette réduction, il nous reste $2^{17} = 131072$ sous-ensembles à énumérer. C'est bien mieux, mais nous allons pouvoir encore réduire ce nombre.

Table 2.16: Masque pour les cases atteignables pour la pièce en `d3` en excluant les bords du plateau

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

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

(a) Empty (0 bits)

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

(b) 1 bit set

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

(c) 2 bits set

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

(d) 3 bits set

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

(e) 9 bits set

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

(f) 18 bits set (full)

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

En effet, on peut scinder les mouvements d'une pièce en mouvements diagonaux et orthogonaux. Nous obtiendrons deux tableaux : `MOVES_D3_DIAG` et `MOVES_D3_ORTHO`. Nous aurons à consulter deux tableaux au lieu d'un seul, pour obtenir les coups possibles en `d3` (ou n'importe quelle autre case), mais nous allons gagner beaucoup de place mémoire. Nous verrons comment combiner les valeurs de ces deux tableaux plus loin dans ce chapitre. Au lieu du masque de la table 2.16, nous aurons maintenant les deux masques de la figure 2.4. Pour le masque orthogonal de la figure 2.4a, on a $2^{10} = 1024$ sous-ensembles et pour le masque diagonal de la figure 2.4b, on a $2^7 = 128$ sous-ensembles, on passe donc de $2^{17} = 131072$ à $2^{10} + 2^7 = 1152$ configurations d'obstacles à considérer pour la pièce en `d3` !

	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

(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 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

(b) Diagonal mask

Figure 2.4: Diagonal and orthogonal masks for d3 without edge squares

Nous pouvons maintenant étudier comment trouver les constantes `MAGIC` et `K` pour les coups diagonaux et orthogonaux de chacune des cases du plateau. Le code est donné dans le listing 12[9]. Une exécution de ce programme[10] est donnée dans le 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
```

[9] J'ai essayé d'utiliser le solveur Z3 pour obtenir ces constantes mais sans succès.

[10] Au vu du générateur aléatoire et de son initialisation, le programme pourra trouver des constantes différentes pour des exécutions différentes.

[11] La sortie est reformatée pour être plus lisible.

```
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
10 constexpr Rank rank_of(Square s) { return Rank(s >> 3); }
11
12 constexpr uint64_t rank_bb(Rank r) { return Rank1BB << (8 * r); }
13
14 constexpr uint64_t rank_bb(Square s) { return rank_bb(rank_of(s)); }
15
16 constexpr uint64_t file_bb(File f) { return FileABB << f; }
17
18 constexpr uint64_t file_bb(Square s) { return file_bb(file_of(s)); }
19
20 constexpr uint64_t square_bb(Square s) {
21     return uint64_t(1) << s;
22 }
23
24 constexpr Square operator+(Square s, Direction d) {
25     return Square(int(s) + int(d));
26 }
27
28 int manhattan_distance(Square sq1, Square sq2) {
29     int d_rank = std::abs(rank_of(sq1) - rank_of(sq2));
30     int d_file = std::abs(file_of(sq1) - file_of(sq2));
31     return d_rank + d_file;
32 }
33
34 enum MoveType {
35     HORIZONTAL,
36     DIAGONAL
37 };
38
39 uint64_t reachable_squares(MoveType mt, Square sq, uint64_t occupied) {
40     uint64_t  moves    = 0;
41     Direction h_dir[4] = {NORTH, SOUTH, EAST, WEST};
42     Direction d_dir[4] = {NORTH_EAST,SOUTH_EAST,SOUTH_WEST,NORTH_WEST};
43     for (Direction d : (mt == HORIZONTAL ? h_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              s = to;
51              moves |= bb;
52          }
53      }
54      return moves;
55  }
56
57  std::pair<int, uint64_t> magic_for_square(MoveType mt, Square sq) {
58      using namespace std;
59      uint64_t edges = ((Rank1BB | Rank8BB) & ~rank_bb(sq)) |
60                          ((FileABB | FileHBB) & ~file_bb(sq));
61      uint64_t moves_bb = reachable_squares(mt, sq, 0) & ~edges;
62      vector<uint64_t> occupancies;
63      vector<uint64_t> possible_moves;
64      uint64_t b = 0;
65      int size = 0;
66      do {
67          occupancies.push_back(b);
68          possible_moves.push_back(sliding_moves(HORIZONTAL, 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<int> seen(1 << k);
78      vector<uint64_t> moves(1 << k);
79      for (int 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 : {HORIZONTAL, DIAGONAL}) {
103         stringstream ss_k, ss_magic;
104         ss_k << format("int {}_K[64] = {{",
105                        mt == HORIZONTAL ? "H" : "D");
106         ss_magic << format("uint64_t {}_MAGIC[64] = {{",
107                            mt == HORIZONTAL ? "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

Dans le listing 12,

- À la ligne 39, la fonction `uint64_t reachable_squares(MoveType mt, Square sq, uint64_t occupied)` va nous permettre de créer le masque des cases accessibles à partir de la case `sq` et pour le type de mouvement `mt`, qui sera soit horizontal (`HORIZONTAL` à la ligne 35), soit diagonal (`DIAGONAL` à la ligne 36). Le masque créé sera similaire aux masques de la figure 2.4.
  - TO DO

```
1  int H_K[64] = {
2      12, 11, 11, 11, 11, 11, 11, 12,
3      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 H_MAGIC[64] = {
    0xc80031080604000,  0xaa4020004000b004, 0x880100020040880,
    0x100081001012014,  0x8880030400800800, 0x2200300104120008,
    0x4100010024408200, 0xc80022480064100,
    0x112802180004000,  0x401000482000,     0x801000200080,
    0x105000810030220,  0x408800400810800,  0xa006004890044200,
    0xa52000200114804,  0x101001090d90002,
    0x1280208000c00084, 0x400404005601000,  0x808010002009,
    0x8041010020481000, 0x3010010040802,    0x2808004000200,
    0x3840040010080211, 0x30200a0000806401,
    0x880a280014004,    0x900600040401000,  0x4002001005102c0,
    0x1000100100230008, 0x4402080080040080, 0x2000c00800a0080,
    0x9000500440200,    0x8040200008041,
    0x208204000800080,  0x40004305002082,   0x108104301002000,
    0x124210019001000,  0x60580082800400,   0x8004000905001d00,
    0x160021004001108,  0x1040a2000104,
    0x40008020428010,   0x100420095000c000, 0x5e004421820010,
    0x202002070420009,  0x8005011088010024, 0x200d000400090022,
    0x4088420001008080, 0x40900400a0520007,
    0x403082280004300,  0x718200040008080,  0x2a0210408200,
    0x3401040a1000a100, 0x8000a80040080,    0x4202800600040180,
    0x8100100841028400, 0x802100005080,
    0x12014021001a82,   0x900250010400081,  0x1004020003249,
    0x8104610010000409, 0x4043001006880025, 0x2001804231006,
    0x4600100648130094, 0x2211170120844402,
};
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] = {
    0x8804040002000064, 0x10020208400004,   0x200000000413,
    0xa000021000000206, 0x830000204020160,  0x108000004000104,
    0x4020002a0402100,  0x28008000000010,
    0x2001040c08c,      0x110050000008,     0x80008001502030,
    0x4000200049440101, 0x4001c08051200092, 0x81000d240090020,
    0x101c000808240020, 0x4802080088010409,
    0x8c00040400,       0x1006800001840402, 0x20008004016000,
    0x4001000000060318, 0x2008020002000408, 0x140000100000030,
    0x2480029000004800, 0x640101060020910,
    0x2860841801a0000,  0x8282404004040ac,  0x4400280922005260,
    0x1020000000082006, 0x400200401,        0x9200140400080008,
    0xa000811090090100, 0x1011a10282302890,
    0x8000104020060019, 0x1000140809811101, 0x40c080020020803,
    0x4000040000848002, 0x130001080801000,  0xa90002420500310,
    0x2845420800040100, 0x8020908044920004,
    0x2180040000000000, 0x8880431022900104, 0x830101008c000918,
    0x20000000110,      0x8290401030841125, 0x21881002000400,
    0x3000000001181010, 0x1000000000210004,
    0x18280040c008080,  0x800109000181d240, 0x200a1c0040865020,
    0x20022000102000,   0x902000a00,        0x8020008100108800,
    0x30000142f0004804, 0x800600400000,
    0x100040180410040,  0xc08a011040004,    0x238400120000200,
    0x1004506042,       0x400080008080000,  0x1280000000065001,
    0x4211260006a0800,  0x1000020234000080,
};
```

Listing 13

2.3.3 Making and Unmaking Moves

2.4 Testing with Random Games

2.5 What's Next

2.6 Complete Commented Game Board Code

Chapter 3

AI Players

Chapter 4

Monte Carlo Player

Chapter 5

MCTS Player

Chapter 6

Minmax Player

Chapter 7

Minmax with Neural Network Player

Chapter 8

AI Tournament

Chapter 9

Conclusion

Acronymes

SMT Satisfiability Modulo Theories. 23, 24

Z3 Z3 Theorem Prover. 23, 24, 33

Bibliography

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

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

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

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

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

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

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

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