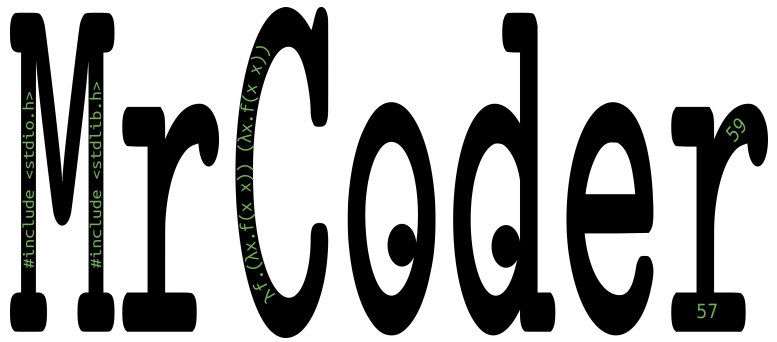
Yolah Board Game

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

Pascal Garcia

December 8, 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 Sarah, Hugo et Célya ❤

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 ☺)

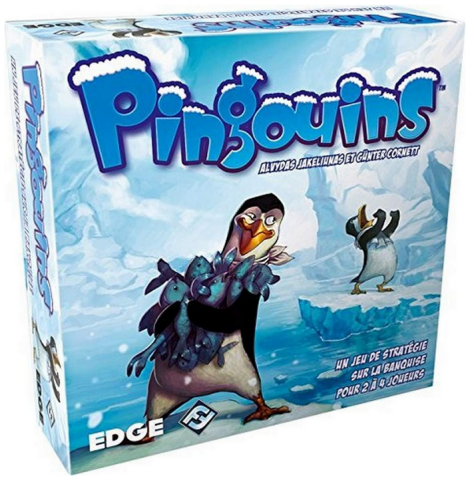


Figure 1.1: The Pingouins game box

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

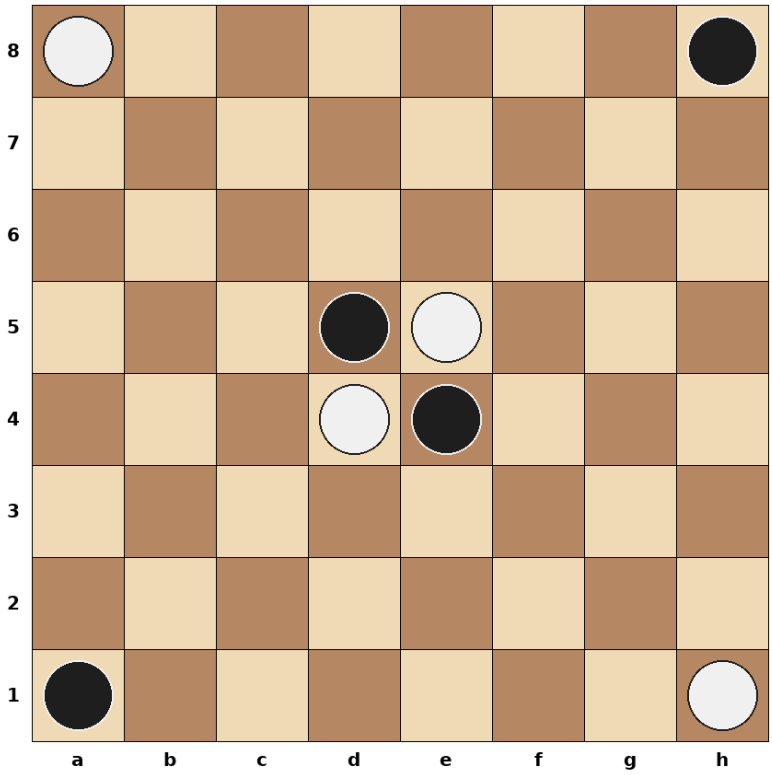


Figure 1.2: The initial configuration of the Yolah game

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.

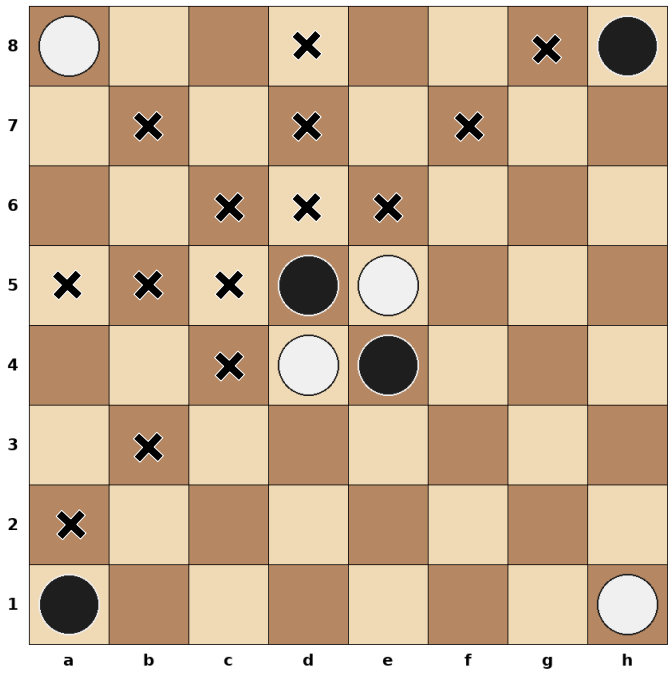


Figure 1.3: Possible moves (small black crosses) for the black piece located on square `d5`

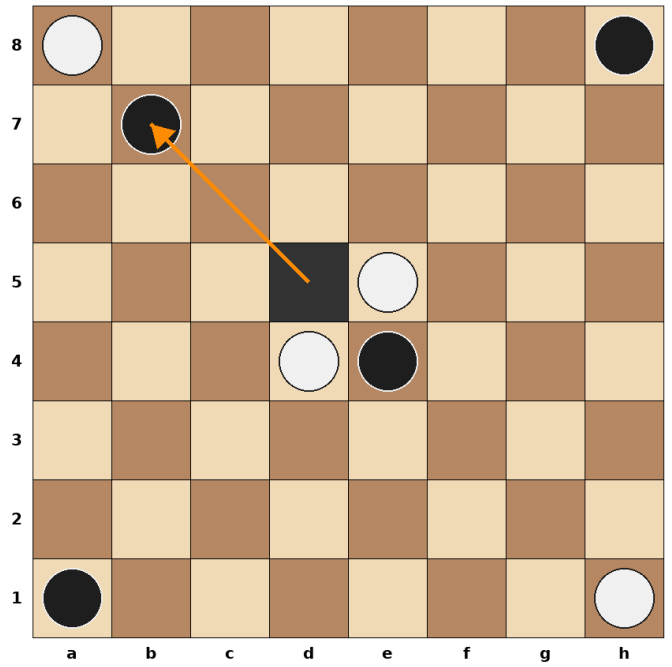


Figure 1.4: Black just moved from `d5` to `b7`. The starting square `d5` becomes inaccessible and impassable for the rest of the game

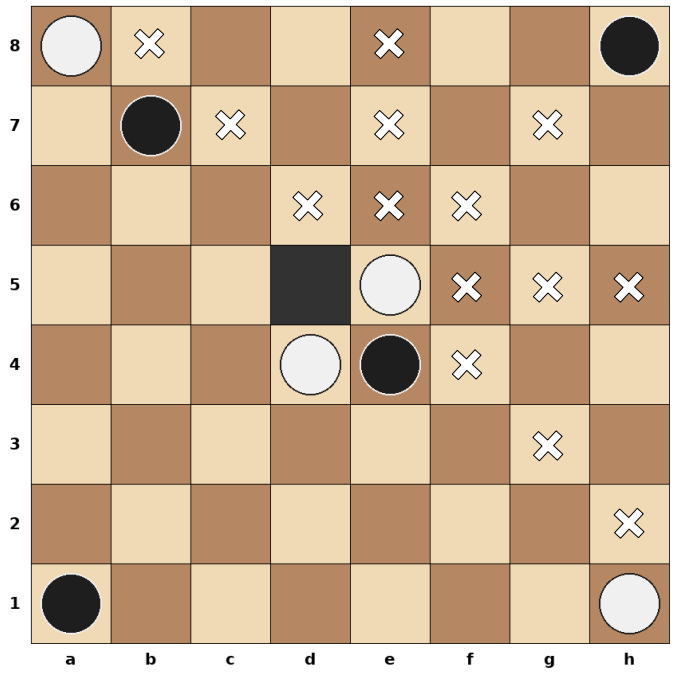


Figure 1.5: Possible moves (small white crosses) for the white piece located on square `e5`

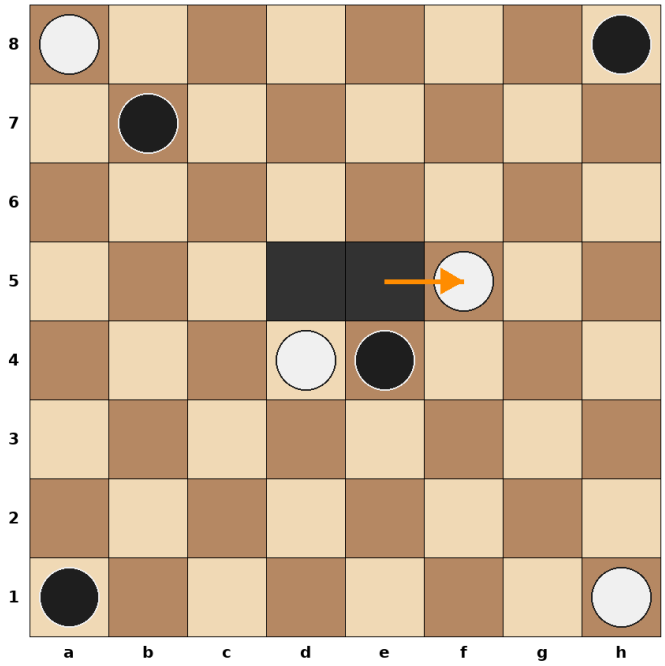


Figure 1.6: White just moved from `e5` to `f5`. The starting square `e5` becomes inaccessible and impassable for the rest of the game. The score is one point each (each player has moved once)

- 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.

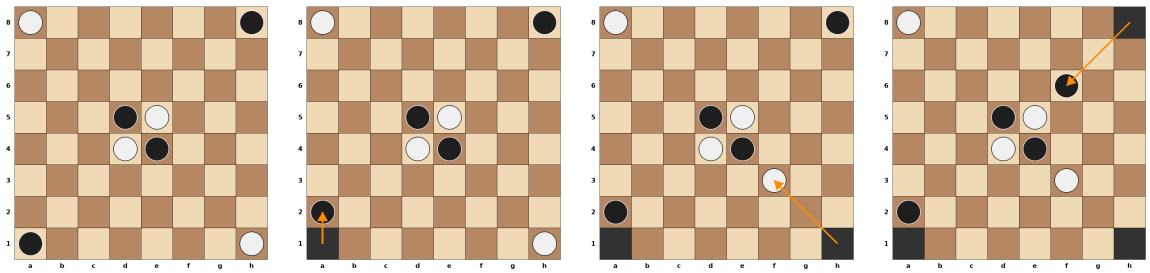
1.1.4 What's Next

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.

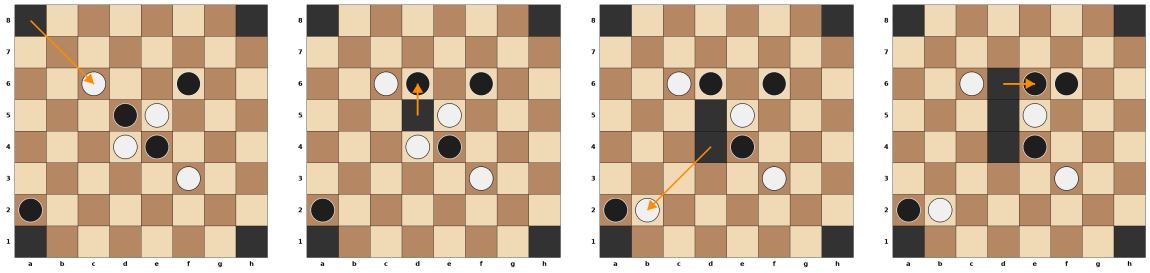
Chapter 3 describes the common interface for our different AIs. Chapter 4 presents a very simple AI based on Monte Carlo search. The next AI, described in Chapter **??**, is an evolution of the previous one and will allow, unlike the Monte Carlo AI, the development of a game tree. Note that these two 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.

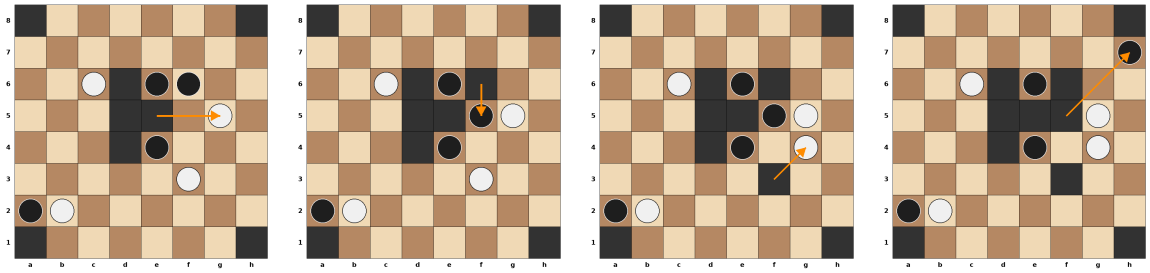
Happy reading!



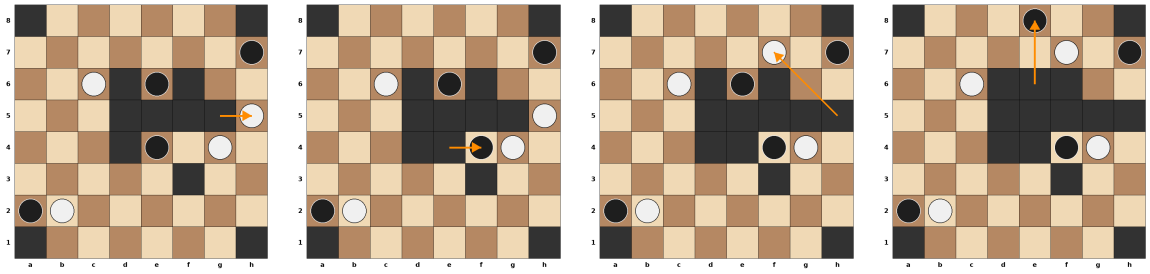
(a) Initial position (b) Move 1 (Black) (c) Move 2 (White) (d) Move 3 (Black)



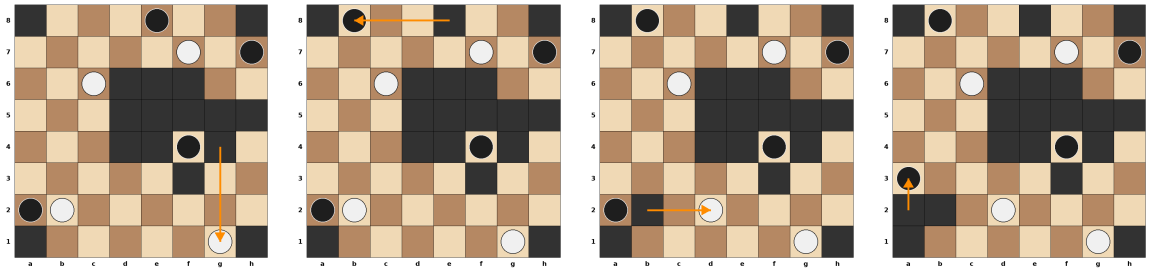
(e) Move 4 (White) (f) Move 5 (Black) (g) Move 6 (White) (h) Move 7 (Black)



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

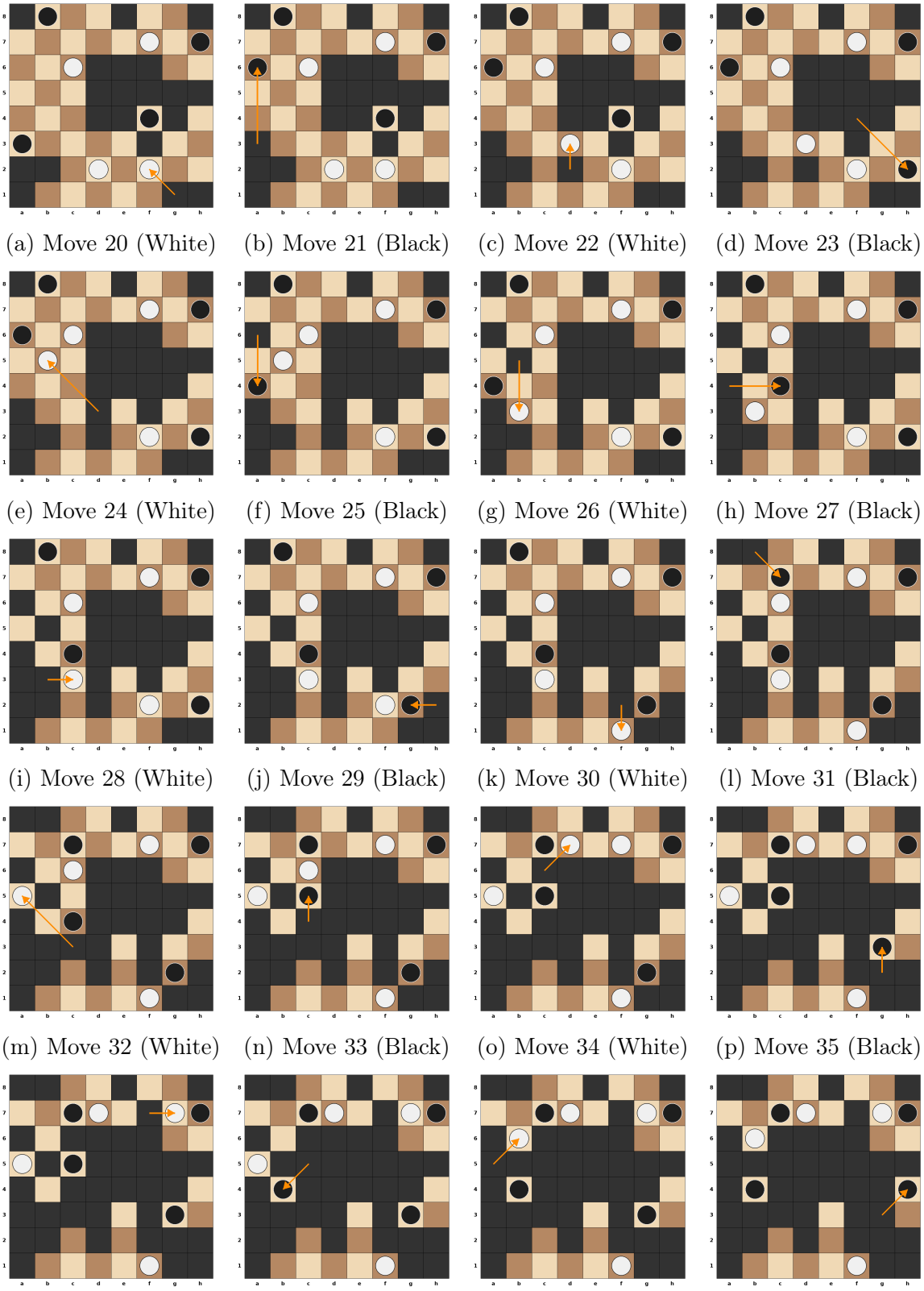


(m) Move 12 (White) (n) Move 13 (Black) (o) Move 14 (White) (p) Move 15 (Black)



(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)

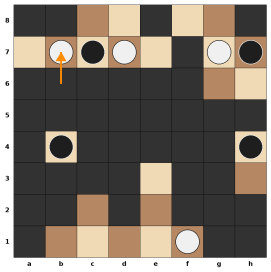
(e) Move 24 (White) (f) Move 25 (Black) (g) Move 26 (White) (h) Move 27 (Black)

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

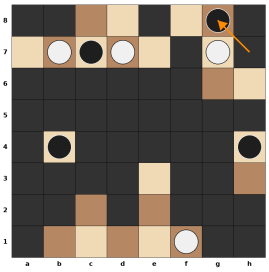
(m) Move 32 (White) (n) Move 33 (Black) (o) Move 34 (White) (p) Move 35 (Black)

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

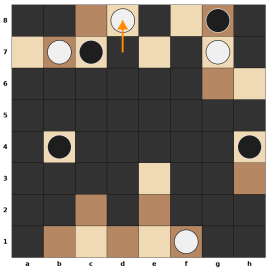
Figure 1.8: Game example between two AIs - moves 20 to 39



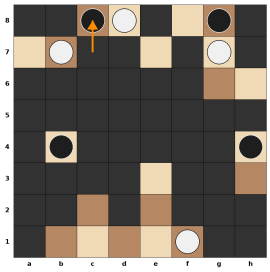
(a) Move 40 (White)



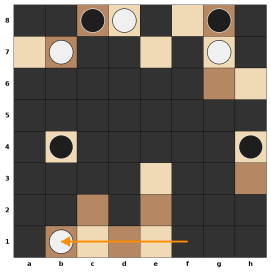
(b) Move 41 (Black)



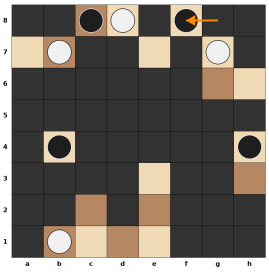
(c) Move 42 (White)



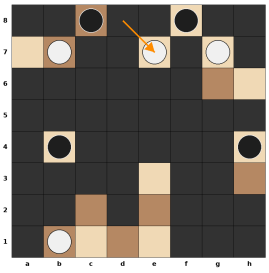
(d) Move 43 (Black)



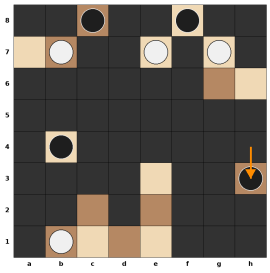
(e) Move 44 (White)



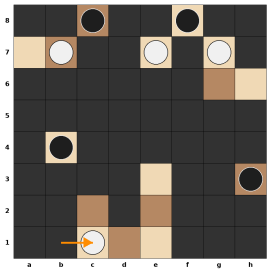
(f) Move 45 (Black)



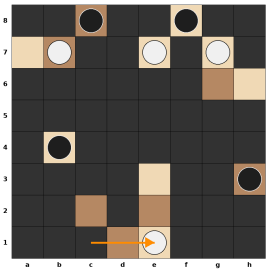
(g) Move 46 (White)



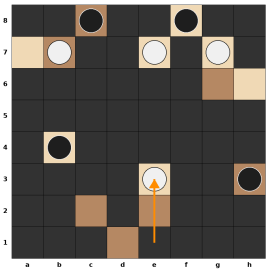
(h) Move 47 (Black)



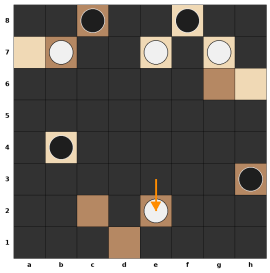
(i) Move 48 (White)



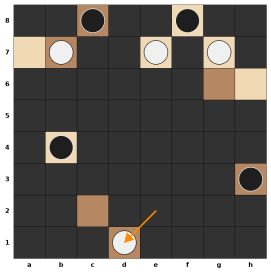
(j) Move 49 (White)



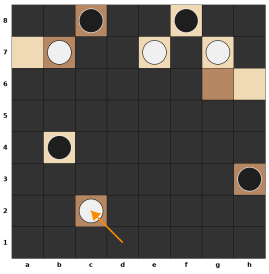
(k) Move 50 (White)



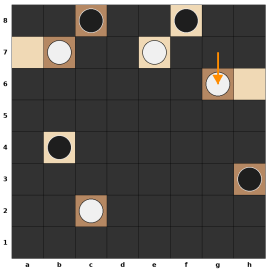
(l) Move 51 (White)



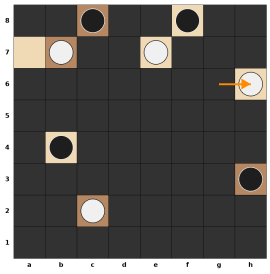
(m) Move 52 (White)



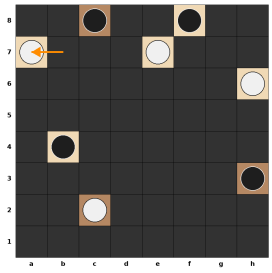
(n) Move 53 (White)



(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

Nous allons implémenter la gestion du jeu en C++ en essayant de créer une implémentation efficace pour pouvoir réaliser un maximum de parties à la seconde, ce qui sera important pour obtenir des IA de bons niveaux. Nous testerons notre implémentation en générant un maximum de parties aléatoires en un temps donné à la fin de ce chapitre. Nous allons nous inspirer de l'excellent moteur de jeu d'échecs Stockfish [3] pour les structures de données de Yolah.

2.1 Structure de données

Nous allons représenter la position des noirs, des blancs et des cases détruites par des des entiers non signés sur 64 bits : `uint64_t`. Nous appelerons bitboard ces entiers. Nous prenons des `uint64_t` car le plateau de Yolah contient 64 cases, nous avons donc un bit par case. La table 2.1 donne la position de chaque cases du plateau dans le bitboard. Cette information est représentée dans le code par l'énumération du listing 1 [1].

Table 2.1: Positions de chaque cases du plateau dans le 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] Par défaut, l'énumeration commence à la valeur 0, donc `SQ_A1` vaut 0, `SQ_A1` vaut 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: Les cases du plateau de jeu

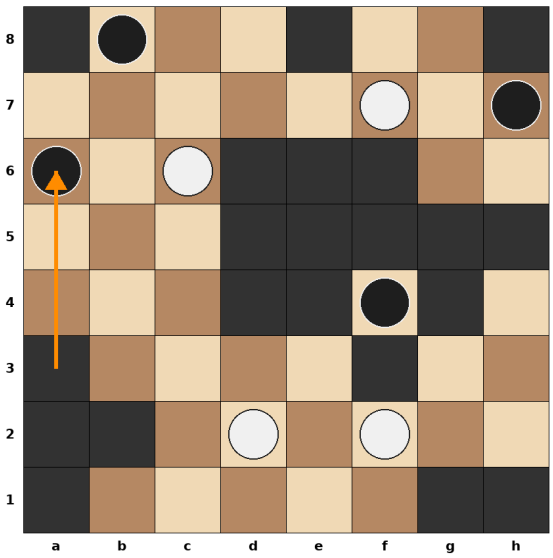


Figure 2.1: Configuration du plateau de jeu correspondant au coup 21 de la partie donnée en example dans le chapitre précédent (figure 1.8b)

Prenons comme exemple le plateau représenté dans la figure 2.1.

- Les positions des pièces noires sur le plateau (table 2.2) sont représentées par le bitboard : `0b0000001010000000000000010000000001000000000000000000000000000000`.
- Les positions des pièces blanches sur le plateau (table 2.3) sont représentées par le bitboard : `0b0000000000010000000000010000000000000000000000000000010100000000000`.
- Les positions des trous (table 2.4) sont représentées par le bitboard : `0b1001000100000000001110001111100001011000001000010000001111000001`.

Table 2.2: Positions des pièces noires

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

Table 2.3: Positions des pièces blanches

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

Table 2.4: Positions des cases détruites (les trous)

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

Les bitboards vont nous permettre une manipulation efficace du plateau, en utilisant des opérations bits à bits, et nécessiterons peu de place mémoire pour représenter celui-ci.

Nous représentons le plateau de jeu par la classe `Yolah` dont un extrait est donné dans le 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: Les attributs de la classe Yolah représentant le plateau de jeu

Les attributs de la classe `Yolah` sont :

- Ligne 6, `black` : le bitboard pour les pièces noires.
- Ligne 7, `white` : le bitboard pour les pièces blanches.
- Ligne 8, `holes` : le bitboard pour les trous (les cases détruites).
- Ligne 9, `black_score` : le score, ou le nombre de déplacements, du joueur noir.
- Ligne 10, `white_score` : le score, ou le nombre de déplacements, du joueur blanc.
- Ligne 11, `ply` : le nombre de coups joués par les deux joueurs depuis le début de la partie.

Un objet de type `Yolah` prendra en mémoire `syzeof(Yolah) == 32` octets. Notons qu'avec le padding [2], il n'aurait pas été judicieux d'écrire, par exemple,

```
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  };
```

car avec cette implémentation, nous aurions eu `syzeof(Yolah) == 48` octets.

La représentation en bitboard permet d'obtenir efficacement des informations sur le jeu. Par exemple pour voir les cases occupées par les pièces noires et les pièces blanches, il suffit d'effectuer : `black | white`. Pour les positions des tables 2.2 et 2.3, on obtiendrait en une seule opération très efficace [3] les positions des pièces noires et blanches de la table 2.5. En notation binaire, on obtient :

```
black | white
== 0000001010000000000000010000000000100000000000000000000000000000 |
   0000000000100000000001000000000000000000000000000010100000000000
== 0000001010100000000001010000000001000000000000001010000000000
```

Table 2.5: Positions des pièces noires et blanches que l'on obtient en faisant : `black | white`

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

Nous allons continuer d'utiliser des opérations bits-à-bits dans la suite de ce chapitre, pour tester notamment la fin d'une partie et la génération des coups possibles.

2.2 Test de fin de partie

Pour tester la fin de partie nous allons avoir besoin de quelques énumerations et constantes (listings 3 et 5) et de la function `shift` (listing 4). La constante `NORTH` de

[3] Vous pouvez trouver des informations sur la latence et le débit des instructions pour plusieurs processeurs via le lien suivant : https://agner.org/optimize/instruction_tables.pdf

l'énumeration `Direction` (listing 3) est égale à 8. Pourquoi avoir choisi cette valeur ? Reprenons ci-dessous, voir la table 2.6, la table vu au chapitre précédent représentant la position de chaque bit d'un bitboard dans le plateau. On peut voir que si l'on ajoute 8 au numéro de bit contenu dans une des cases, on obtient le numéro de bit de la case au nord ce cette dernière (sauf si l'on sort du plateau de jeu). Il en va de même pour les autres valeurs des constantes de l'énumération `Direction` (listing 3).

Table 2.6: Positions de chaque cases du plateau dans le bitboard. Remarquons que pour une case qui ne se trouve pas dans la rangée 8, bit_{i+8} est le bit correspondant à la case au nord de la case du 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)
```

```
8           return (b & ~FileHBB) << EAST;
9       else if constexpr (D == WEST)
10          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: Décalage selon une direction

La fonction `shift(uint64_t b)` donnée dans le listing 4 nous permet de décaler tous les bits à 1 du bitboard en paramètre dans la direction `D` [4]. Soit,

`black` == 1000000000000000000000000000010000010000000000000000000000000001

le bitboard représenté dans la table 2.7. Pour décaler les pièces noires d'une case vers le nord, on effectue : `shift<NORTH>(black)`. On obtient alors le bitboard

1000000000000000000000000000010000010000000000000000000000000001

représenté dans la table 2.8. Notons que la pièce en `h8` n'est plus sur le plateau.

Table 2.7: Plateau de jeu avant d'effectuer `shift<NORTH>`

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

Table 2.8: Plateau de jeu après avoir effectuer `shift<NORTH>`

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

On peut ainsi facilement obtenir toutes les cases directement au contact des pièces d'un plateau donné `board`, en effectuant l'opération suivante :

[4] Grâce aux `if constexpr`, lorsque l'on effectuera `shift<NORTH>(b)` par exemple, le compilateur transformera le code en : `return b << NORTH;`.

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

Pour le plateau de la table 2.9 on obtient les positions représentées par des étoiles dans la table 2.10.

Table 2.9: Plateau de jeu

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

Table 2.10: Positions autour des pièces

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

Dans le code de la fonction `shift` (listing 4) on peut voir l'utilisation des constantes `FileHBB` et `FileABB` (`BB` pour bitboard). Ces constantes vont nous permettre de masquer certains bits qui se retrouveraient après décalage dans une position incorrecte. Par exemple, si nous décalons dans la direction `EAST` le bitboard représenté dans la table 2.9, le pion blanc en `h1` se retrouverait en `a2`. Pour éviter cela, avant le décalage, on élimine les éléments de la colonne `h` pour qu'ils ne se retrouvent pas dans la colonne `a` (ligne 8 du listing 4). Les constantes `FileABB` et `FileHBB` ainsi que les énumerations pour les rangées et colonnes sont définies dans le listing 5.

```
enum File : uint8_t {
    FILE_A, FILE_B, FILE_C, FILE_D, FILE_E, FILE_F, FILE_G, FILE_H,
    FILE_NB
};
enum Rank : uint8_t {
    RANK_1, RANK_2, RANK_3, RANK_4, RANK_5, RANK_6, RANK_7, RANK_8,
    RANK_NB
};
constexpr uint64_t FileABB = 0x0101010101010101;
//   0x0101010101010101
//== 0b0000000100000001000000010000000100000001000000010000000100000001
//             a8      a7      a6      a5      a4      a3      a2      a1

constexpr uint64_t FileHBB = FileABB << 7;
//   0x1010101010101010
```

```
16 //== 0b1000000010000000100000001000000010000000100000001000000010000000
17 //     h8      h7      h6      h5      h4      h3      h2      h1
```

Listing 5: Colonnes et rangées

La fonction qui teste la fin de partie s'appelle `game_over` et elle est décrite dans le listing 6. Son implémentation est assez simple,

- À la ligne 2, on récupère dans `possible` le bitboard où il y a des 1 dans les positions libres.
- À la ligne 3, on crée le bitboard `players` qui contient des 1 dans les positions occupées par l'un des joueurs.
- Les lignes 4 à 8 permettent de créer le bitboard `around_players` qui contient des 1 aux positions autour de chacune des pièces des joueurs.
- Enfin, à la ligne 8, on teste s'il n'existe aucune position libre à côté d'un des joueurs. En effet, le et bit-à-bit ne gardera un 1 dans un bit du résultat, que si et seulement s'il y a un 1 à cette position dans le bitboard `possible` et dans le bitboard `around_players`. Donc que la case correspondante est libre et accessible par le joueur. Si le résultat de `around_players & possible` vaut 0, cela veut dire qu'aucune position n'est à la fois à côté d'un des joueurs et libre.

```
1  bool Yolah::game_over() const {
2      uint64_t possible = ~empty & ~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: Test de fin de partie

Maintenant que nous savons tester la fin d'une partie, nous allons étudier comment générer efficacement les coups possibles dans une position donnée.

2.3 Génération des mouvements possibles pour les pièces

La génération des coups possibles va utiliser une technique appelée les *magic bitboard*. Pour comprendre cette technique je vais tout d'abord la présenter sur un

exemple simple. Je me suis inspiré de l'article http://pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf pour cet exemple.

2.3.1 Hachage parfait

Supposons que nous devions trouver la position du seul bit à 1 dans un entier long non-signé (`uint64_t`). Par exemple, si nous appelons `position` la fonction permettant de calculer cette position, nous obtenons les résultats suivants :

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

Notons que les processeurs possèdent des instructions permettant de calculer efficacement cette fonction [5], et le compilateur `gcc` permet de calculer efficacement cette position avec la fonction `__builtin_ctzll(unsigned long long x)` [6], par exemple,

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

Mais nous allons coder cette fonction `position` en utilisant un hachage parfait afin d'illustrer cette technique, qui nous servira pour la génération des coups possibles par la suite.

En utilisant une table de hachage classique, nous pouvons tout d'abord remplir celle-ci avec les 64 valeurs et simplement consulter cette table par la suite, comme dans le code suivant :

```
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 }
```

La consultation de cette table de hachage est bien plus coûteuse que l'indexation d'un simple tableau [7]. Mais on ne peut pas directement utiliser un tableau, car les valeurs pour indexer celui-ci sont dans une plage de valeurs trop grande, par exemple la valeur `0x8000000000000000` (9, 223, 372, 036, 854, 775, 808 !). L'idée c'est de trouver une fonction de hachage efficace pour transformer chacune des 64 valeurs[8], que nous

[5] `tzcnt` sous 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] Nous mesurerons à la fin de cette section le gain apporté par le hachage parfait par rapport à l'utilisation d'une table de hachage classique.

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

allons appeler clés, dans la plage $[0, 2^6 - 1 = 63]$. De plus on veut qu'il n'y ait aucune collision, d'où le nom de hachage parfait. S'il y a des collisions des positions seraient erronées. Par exemple, si les clés `0b100` et `0b1000000` étaient transformées en l'indice 42, on devrait mettre dans `table[42]` la valeur 2 ou 6, mais on ne peut pas stocker plus d'une valeur au même emplacement.

La fonction de hachage parfait va avoir la forme suivante :

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

Dans cette fonction,

- À la ligne 2, la constante `K` donne le nombre de bits pour notre index. Ici avec `K == 6`, cela nous donne un maximum de $2^K = 2^6 = 64$ valeurs possibles dans la table.
- À la ligne 4, on peut voir la formule pour calculer l'index dans la table selon la clé `uint64_t key` donnée en paramètre. On multiplie la clé par la constante `MAGIC` et ensuite on décale vers la droite pour ne garder que les `K` bits de poids forts du résultat. Cette opération est très peu coûteuse, mais comment trouver cette constante magique ?

Une méthode simple pour essayer de trouver la constante `MAGIC` est de la générer au hasard jusqu'à trouver une valeur qui ne crée aucune collision ! Ce genre d'approche est utilisée dans Stockfish [3] pour générer les magic bitboard. Le listing 7 montre cette approche.

```
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 7: Recherche aléatoire de la constante `MAGIC`

- Aux lignes 6 à 8, nous mettons en place un générateur aléatoire pour générer aléatoirement des `uint64_t`.
- À la ligne 9, la table de correspondance `positions` va associer à un entier long ne contenant qu'un seul bit à 1, la position de ce dernier. Cette table est initialisée aux lignes 10 à 12.
- À la ligne 13, la constante `K` est le nombre de bits de notre clé obtenue par hachage parfait. Il faut que `K` permettent de représenter toutes les valeurs que l'on doit couvrir. Par exemple, ici nous avons 64 valeurs possibles, car nous

avons 64 positions possibles pour le bit à 1 dans un entier long sur 64 bits. Avec `K == 7`, on peut couvrir 2^7, soit 128 valeurs différentes. Notons que c'est plus que les 64 valeurs dont nous avons besoin et que `K == 6` suffirait (`K == 5` serait trop petit). Mais en essayant de trouver la constante `MAGIC` au hasard pour `K == 6`, ce programme ne trouvait pas en un temps raisonnable ☹ (nous verrons après une autre façon de faire pour réussir pour `K == 6` ☺).

- Aux lignes 14 à 16, la fonction `index` calcule l'index de clé `bitboard` (ce bitboard contient un seul bit à 1) en utilisant la formule pour notre hachage parfait.
- La boucle `while`, lignes 17 à 43, va boucler jusqu'à trouver la constante `MAGIC` qui permet de réaliser un hachage parfait (du coup cette boucle peut être infinie).
- À la ligne 18, on crée la valeur `MAGIC` au hasard.
- Le `set<uint64_t>` de la ligne 20 va nous permettre de mémoriser les index (obtenues grâce à la fonction `index`) que nous avons déjà utilisés pour vérifier que nous n'avons pas de collisions.
- Aux lignes 21 à 28, la boucle `for` va tester tous les bitboards, trouver l'index de chacun d'entre eux grâce au hachage (fonction `index`) et vérifier que nous n'avons pas de collision (ligne 23) avec les index obtenus pour les bitboard précédents.
- Enfin, aux lignes 29 à 42, si nous avons trouvé une constante `MAGIC` qui permet de créer le hachage parfait, nous affichons celle-ci puis nous affichons un tableau en *C++* qui contient la position du bit à 1 pour chacun des bitboard. La sortie de ce programme, pour une exécution donnée, est donnée ci-dessous.

```
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
};
```

Pour obtenir la position du bit à 1 pour le bitboard `0b1000` par exemple, il faut donc consulter la valeur suivante :

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

```
== positions[0x17]
== positions[23]
== 3
```

On remarque qu'il y a des places vacantes dans le tableau `positions`, celles contenant la valeur -1, donc on a perdu de la place. Nous n'aurions gaspillé aucune place si nous avions trouvé une constante `MAGIC` pour `K == 6`. Mais est-ce possible ?

Pour répondre à cette question nous allons procéder d'une autre manière. Nous ne voulons pas bien entendu balayer les 2^{64} $(9,223,372,036,854,775,808)$ valeurs possibles puis à chaque fois tester si la valeur permet le hachage parfait. Nous allons utiliser un solveur Satisfiability Modulo Theories (SMT) qui va nous permettre de fixer des contraintes avant de balayer tout l'espace de recherche. Le fait de fixer tout d'abord des contraintes va permettre d'élaguer l'espace de recherche. Notons que nous n'avons pas la garantie que la recherche soit rapide, mais pour ce problème, l'obtention d'une constante `MAGIC` pour `K == 6` sera très rapide. Si vous êtes intéressé par le fonctionnement d'un solveur SMT vous pouvez consulter [4] et [5].

```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 bitscan[{size}] = {{')
    for i in range(size):
        print(f'{table[i]},', end='')
    print('\n};')
```

Listing 8: Recherche avec un solveur SMT de la constante `MAGIC`

```
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
};
```

2.3.2 Magic bitboard pour les mouvements des pièces

2.4 Code complet commenté du plateau de jeu

```
#ifndef TYPES_H
#define TYPES_H
#include <cstdint>
#include <bit>
enum Square : int8_t {
    SQ_A1, SQ_B1, SQ_C1, SQ_D1, SQ_E1, SQ_F1, SQ_G1, SQ_H1,
    SQ_A2, SQ_B2, SQ_C2, SQ_D2, SQ_E2, SQ_F2, SQ_G2, SQ_H2,
    SQ_A3, SQ_B3, SQ_C3, SQ_D3, SQ_E3, SQ_F3, SQ_G3, SQ_H3,
    SQ_A4, SQ_B4, SQ_C4, SQ_D4, SQ_E4, SQ_F4, SQ_G4, SQ_H4,
    SQ_A5, SQ_B5, SQ_C5, SQ_D5, SQ_E5, SQ_F5, SQ_G5, SQ_H5,
    SQ_A6, SQ_B6, SQ_C6, SQ_D6, SQ_E6, SQ_F6, SQ_G6, SQ_H6,
    SQ_A7, SQ_B7, SQ_C7, SQ_D7, SQ_E7, SQ_F7, SQ_G7, SQ_H7,
    SQ_A8, SQ_B8, SQ_C8, SQ_D8, SQ_E8, SQ_F8, SQ_G8, SQ_H8,
    SQ_NONE,
    SQUARE_ZERO = 0,
    SQUARE_NB   = 64
};
```

```
18  enum Direction : int8_t {
19      NORTH = 8,
20      EAST  = 1,
21      SOUTH = -NORTH,
22      WEST  = -EAST,
23      NORTH_EAST = NORTH + EAST,
24      SOUTH_EAST = SOUTH + EAST,
25      SOUTH_WEST = SOUTH + WEST,
26      NORTH_WEST = NORTH + WEST
27  };
28  enum File : uint8_t {
29      FILE_A, FILE_B, FILE_C, FILE_D, FILE_E, FILE_F, FILE_G, FILE_H,
30      FILE_NB
31  };
32  enum Rank : uint8_t {
33      RANK_1, RANK_2, RANK_3, RANK_4, RANK_5, RANK_6, RANK_7, RANK_8,
34      RANK_NB
35  };
36  constexpr uint64_t FileABB = 0x0101010101010101;
37  constexpr uint64_t FileBBB = FileABB << 1;
38  constexpr uint64_t FileCBB = FileABB << 2;
39  constexpr uint64_t FileDBB = FileABB << 3;
40  constexpr uint64_t FileEBB = FileABB << 4;
41  constexpr uint64_t FileFBB = FileABB << 5;
42  constexpr uint64_t FileGBB = FileABB << 6;
43  constexpr uint64_t FileHBB = FileABB << 7;
44  constexpr uint64_t Rank1BB = 0xFF;
45  constexpr uint64_t Rank2BB = Rank1BB << (8 * 1);
46  constexpr uint64_t Rank3BB = Rank1BB << (8 * 2);
47  constexpr uint64_t Rank4BB = Rank1BB << (8 * 3);
48  constexpr uint64_t Rank5BB = Rank1BB << (8 * 4);
49  constexpr uint64_t Rank6BB = Rank1BB << (8 * 5);
50  constexpr uint64_t Rank7BB = Rank1BB << (8 * 6);
51  constexpr uint64_t Rank8BB = Rank1BB << (8 * 7);
52  constexpr uint64_t BLACK_INITIAL_POSITION =
53  0b10000000'00000000'00000000'00001000'00010000'00000000'00000000'00000001;
54  constexpr uint64_t WHITE_INITIAL_POSITION =
55  0b00000001'00000000'00000000'00010000'00001000'00000000'00000000'10000000;
56  constexpr uint64_t FULL = 0xFFFFFFFFFFFFFFFF;
57  constexpr Square make_square(File f, Rank r) {
58      return Square((r << 3) + f);
59  }
```

```
60 inline Square lsb(uint64_t b) {
61     return Square(std::countr_zero(b));
62 }
63 inline Square pop_lsb(uint64_t& b) {
64     const Square s = lsb(b);
65     b &= b - 1;
66     return s;
67 }
68 constexpr bool more_than_one(uint64_t b) {
69     return b & (b - 1);
70 }
71 template<Direction D>
72 constexpr uint64_t shift(uint64_t b) {
73     if constexpr (D == NORTH)           return b << 8;
74     else if constexpr (D == SOUTH)      return b >> 8;
75     else if constexpr (D == EAST)       return (b & ~FileHBB) << 1;
76     else if constexpr (D == WEST)       return (b & ~FileABB) >> 1;
77     else if constexpr (D == NORTH_EAST) return (b & ~FileHBB) << 9;
78     else if constexpr (D == NORTH_WEST) return (b & ~FileABB) << 7;
79     else if constexpr (D == SOUTH_EAST) return (b & ~FileHBB) >> 7;
80     else if constexpr (D == SOUTH_WEST) return (b & ~FileABB) >> 9;
81     else return 0;
82 }
83 constexpr uint64_t shift_all_directions(uint64_t b) {
84     uint64_t b1 = b & ~FileHBB;
85     uint64_t b2 = b & ~FileABB;
86     return b << 8 | b >> 8 | b1 << 1 | b1 << 9
87             | b1 >> 7 | b2 >> 1 | b2 << 7 | b2 >> 9;
88 }
89 #endif
```

Listing 9: Squares, files, ranks and directions

```
1 #ifndef MOVE_H
2 #define MOVE_H
3 #include "types.h"
4 #include <iostream>
5 class Move {
6     uint16_t data;
7 public:
```

```
8       Move() = default;
9       constexpr explicit
10      Move(uint16_t d) : data(d) {}
11      constexpr explicit
12      Move(Square from, Square to) : data((to << 6) + from) {}
13      constexpr Square to_sq() const {
14          return Square((data >> 6) & 0x3F);
15      }
16      constexpr Square from_sq() const {
17          return Square(data & 0x3F);
18      }
19      static constexpr Move none() {
20          return Move(0);
21      }
22      constexpr bool operator==(const Move& m) const {
23          return data == m.data;
24      }
25      constexpr bool operator!=(const Move& m) const {
26          return !(*this == m);
27      }
28      constexpr explicit operator bool() const {
29          return data != 0;
30      }
31      constexpr uint16_t raw() const {
32          return data;
33      }
34  };
35  std::ostream& operator<<(std::ostream& os, const Move& m);
36  std::istream& operator>>(std::istream& is, Move& m);
37  #endif
```

Listing 10: implémentation d'un coup

```
1   #ifndef GAME_H
2   #define GAME_H
3   #include <iostream>
4   #include <vector>
5   #include <utility>
6   #include "types.h"
7   #include "move.h"
```

```
8  #include "json.hpp"
9  #include "misc.h"
10 using json = nlohmann::json;
11 class Yolah {
12     uint64_t black = BLACK_INITIAL_POSITION;
13     uint64_t white = WHITE_INITIAL_POSITION;
14     uint64_t empty = 0;
15     uint8_t black_score = 0;
16     uint8_t white_score = 0;
17     uint8_t ply = 0;
18 public:
19     static constexpr uint8_t MAX_NB_MOVES = 75;
20     static constexpr uint8_t MAX_NB_PLIES = 120;
21     class MoveList {
22       Move moveList[MAX_NB_MOVES], *last;
23     public:
24       explicit MoveList() : last(moveList) {}
25       const Move* begin() const { return moveList; }
26       const Move* end() const { return last; }
27       Move* begin() { return moveList; }
28       Move* end() { return last; }
29       size_t size() const { return last - moveList; }
30       const Move& operator[](size_t i) const { return moveList[i]; }
31       Move& operator[](size_t i) { return moveList[i]; }
32       Move* data() { return moveList; }
33       friend class Yolah;
34     };
35     static constexpr uint8_t BLACK = 0;
36     static constexpr uint8_t WHITE = 1;
37     static constexpr uint8_t HOLE  = 2;
38     static constexpr uint8_t FREE  = 3;
39     constexpr std::pair<uint8_t, uint8_t> score() const;
40     constexpr int8_t score(uint8_t player) const;
41     constexpr uint8_t current_player() const;
42     static constexpr uint8_t other_player(uint8_t player);
43     uint8_t get(Square) const;
44     uint8_t get(File f, Rank r) const;
45     bool game_over() const;
46     void play(Move m);
47     void undo(Move m);
48     void moves(uint8_t player, MoveList& moves) const;
49     void moves(MoveList& moves) const;
```

```
50      bool valid(Move m) const;
51      constexpr uint64_t free_squares() const;
52      constexpr uint64_t occupied_squares() const;
53      constexpr uint64_t bitboard(uint8_t player) const;
54      constexpr uint16_t nb_plies() const;
55      constexpr uint64_t empty_bitboard() const;
56      std::string to_json() const;
57      static Yolah from_json(std::istream& is);
58      static Yolah from_json(const std::string&);
59      friend std::ostream& operator<<(std::ostream& os, const Yolah&
    ↪   yolah);
60  };
61  std::ostream& operator<<(std::ostream& os, const Yolah& yolah);
62  bool Yolah::game_over() const {
63      uint64_t possible = ~empty & ~black & ~white;
64      uint64_t players  = black | white;
65      uint64_t around_players = shift<NORTH>(players) |
    ↪   shift<SOUTH>(players) | shift<EAST>(players) |
66          shift<WEST>(players) | shift<NORTH_EAST>(players) |
    ↪   shift<NORTH_WEST>(players) |
67          shift<SOUTH_EAST>(players) | shift<SOUTH_WEST>(players);
68      return (around_players & possible) == 0;
69  }
70  #endif
```

Listing 11: implémentation du plateau

2.5 Test avec des parties aléatoires

2.6 La suite

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. 24, 25

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] 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.

[5] 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.