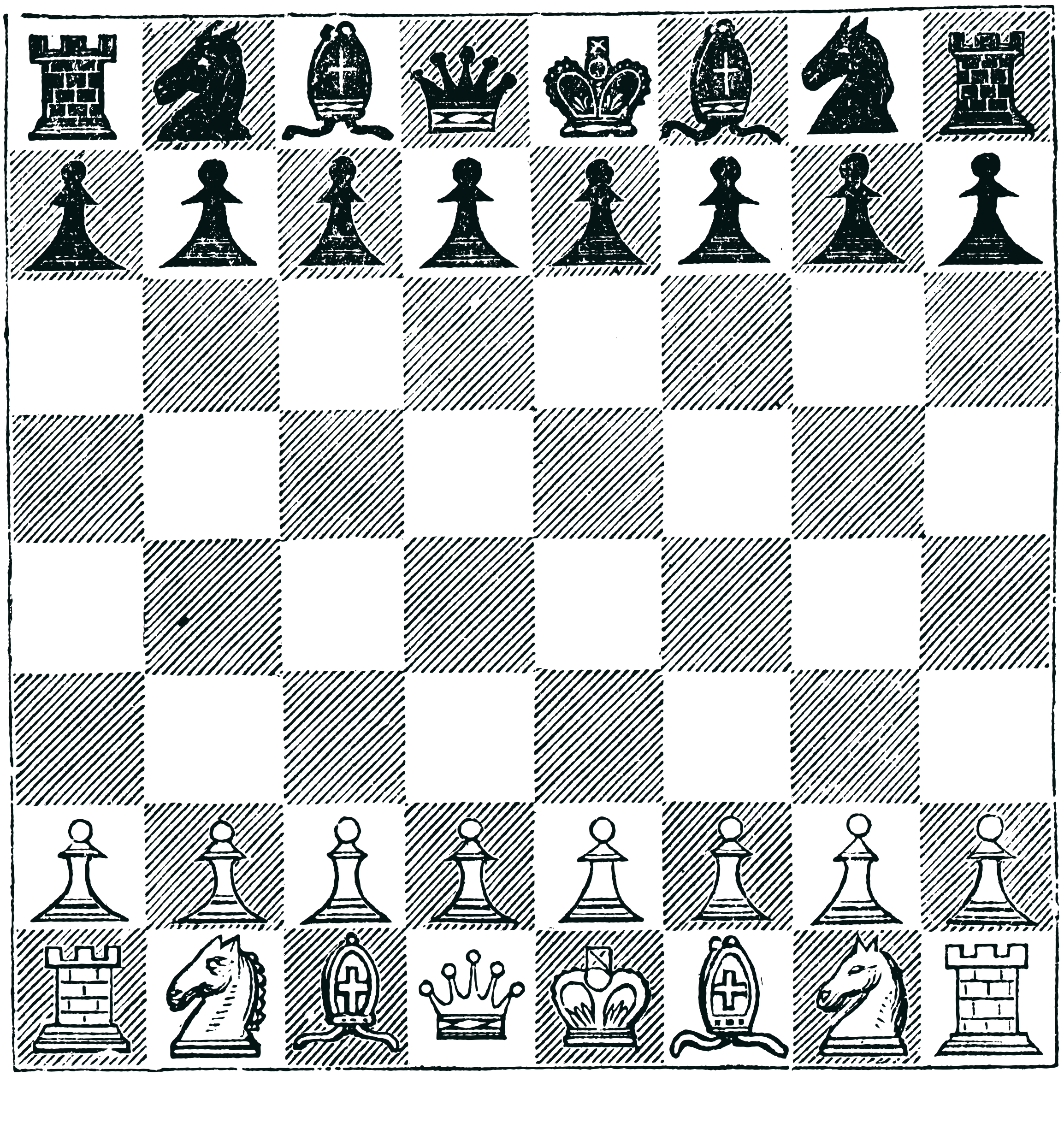
MATTEO ALLEGRINI, FRANCESCO GIACOPPO

THINKING IN MOVES



A PRACTICAL EXPLORATION OF MINIMAX AND ALPHA-BETA PRUNING IN CHESS

Contents

1 Introduction

Chess is one of the most complex and fascinating strategic games, where the ability to plan and anticipate an opponent's move plays a fundamental role.

The main objective of this project is to develop a chess engine capable of correctly representing the rules of the game on a playable board, on which the user will be able to play both against another player or against a cpu.

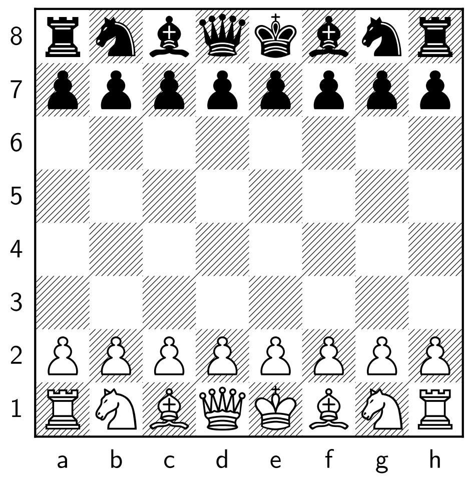
Therefore this document, which is aimed to explain as clearly as possible the structure of our project, will be divided in three big sections:

1. The first section, the biggest one, will be focused on every aspect of the **board implementation**, from the internal and external representation of the board to the legal move generator.
2. In the second section we will tackle the problem of the implementation of a **chess-bot** playing (pseudo) reasoned moves. To this aim we will mate use of the well known minmax algorithm and the alpha beta pruning.
3. The last section, considerably smaller then the other two, will just have the task to explain how we combined all of what we had done in the first two section to obtain a fully playable game. In other words, it will be focused on the **game engine** of our project.

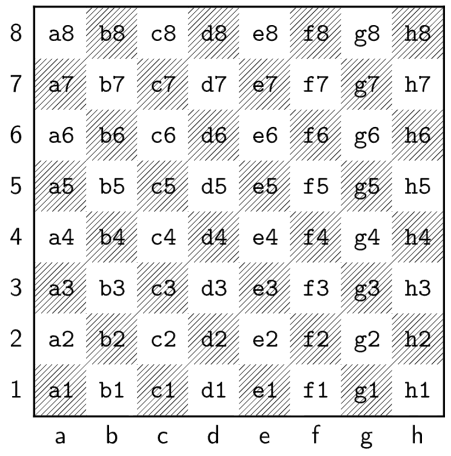
1.1 Rules of the game

Before starting with the explanation of our project, we are going to explain here briefly the main rules of the game of chess. Of course if one is already familiar with this game, he can skip this section right away.

Chess is a strategic game for two players, played in a square board consisting of 64 squares arranged in an 8×8 grid. On the chess board live 16 white pieces and 16 black pieces and each of the two players, referred to as "White" and "Black", controls either the white or the black ones. The pieces are arranged on the board in the following way.



The numbers on the chessboard from 1 to 8 denote the **ranks** of the board, while the letters from 'a' to 'h' denote the **files** of the board. These symbols are useful to give a name to the squares of the board, as shown in the following figure.


8 a8 b8 c8 d8 e8 f8 g8 h8
7 a7 b7 c7 d7 e7 f7 g7 h7
6 a6 b6 c6 d6 e6 f6 g6 h6
5 a5 b5 c5 d5 e5 f5 g5 h5
4 a4 b4 c4 d4 e4 f4 g4 h4
3 a3 b3 c3 d3 e3 f3 g3 h3
2 a2 b2 c2 d2 e2 f2 g2 h2
1 a1 b1 c1 d1 e1 f1 g1 h1
a b c d e f g h

Each type of piece has a different pattern of movement, for instance:

- White pawns (♙) can only be moved towards Black's direction by one square and by two squares if they have not been moved yet. Moreover, they capture pieces only in diagonal, always moving ahead.
- White rooks (♖) can move any number of squares horizontally or vertically, but cannot jump over other pieces. They capture an enemy piece by landing on its square.
- White knights (♘) move in an "L" shape: two squares in one direction and then one square perpendicular to it. They can jump over other pieces and capture an enemy piece by landing on its square.
- White bishops (♗) can move any number of squares diagonally, but cannot jump over other pieces. They capture an enemy piece by landing on its square.
- White queens (♕) can move any number of squares in any direction: horizontally, vertically, or diagonally, but cannot jump over other pieces. They capture an enemy piece by landing on its square.
- White kings (♔) can move exactly one square in any direction: horizontally, vertically, or diagonally. They capture an enemy piece by landing on its square and cannot move into check.

Black pieces move in the exact same way.

If after the opponent move our king is under attack it is said to be **in check**. The only legal moves that a player can make when his king is in check are the ones that blocks that attack to their king (either by capturing the attacking piece, by blocking the attack with another piece or by just moving the king away from the attack). The goal of the game is to deliver a ”checkmate” to the rival king, that is, a check from which the opposite player cannot escape.

In addition to the moves already presented there are other ”special” of moves that involves specific pieces in specific situations. For instance we have:

- Catling: always involve a king and a rook that have never been moved before.
- Promotion: when a white pawn reaches the 7-th rank (or when a black one reaches the 1-th rank) it can be promoted to another type o piece (except for the king).
- En-passant: a special kind of pawn capture that can be made only by white pawns on the 6-th rank (or by black pawns on the 4-th rank) which can capture another pawn if it has been moved by two steps the move before.

Moreover, each type piece is usually associated with a specific value, corresponding to its general strength. The more common ones are the following:

Pawns: 1   Bishops: 3   Knights: 3   Rooks: 5   Queens: 9 .

Those values are to be considered as very rough approximations of the real strength of a piece in the game, which of course depend heavily also on the position of that piece in relation with the positions of all the other pieces on the board. Notice that we didn’t mention the value of king pieces, this is because it is not so important since in a chess game kings are never actually captured. Thus in theory kings have infinite value, but in practice it is rarely specified.

Now that we introduced the basic rules of chess, we can dive into the first section explaining how we managed the board implementation of our Chess engine.

2 Board Implementation

The board implementation of our Chess engine is basically divided in three main parts: the internal board representation, the move generator and the external board representation.

2.1 Internal board representation

Every chess program needs an internal board representation, that is, a way to represent the chess-board (as well as all its components) using the computer language.

For our chess engine we chose to represent the board using a **class-based representation**, which increase the readability of the code at the cost of slightly worse performances.

We noticed that every aspects of a chess game can be described using the following main objects: squares, pieces, moves. Thus we created a specific class for each of those types of object and, for each of them, we implemented useful methods needed for the engine to work.

Furthermore we created a fourth class, the Board class, which makes use of the preceding three to obtain a coherent full internal representation of the chess-board. In particular, as we will see better later, we represented the board in our code as a **double array**: a list of 8 lists, each of them containing 8 Square objects.

Let's now analyze in detail the four classes used in our internal representation.

2.1.1 Square class

First we created the Square class, whose every instance represents a **single square** on the board.

Since every square is characterized by a row and a column, and of course any square may or may not contain a piece, we defined the constructor of the Square class as follows:

```
class Square:
    def __init__(self, row, col, piece = None):
        self.row = row
        self.col = col
        self.piece = piece # Piece object
        ...
```

Thus, every square is initialized with its row and column and optionally with a piece on it.

In this class we also implemented some methods to check whether on a particular Square object there is a "team piece", a "rival piece" or if it is empty. These methods will be particularly useful in the move generator algorithms.

2.1.2 Move class

In order to encode the flow of a chess game inside the chess engine it is essential to represent every chess move in a clear, useful and coherent way. For this reason we created a specific class for every **single chess move**: the Move class.

This class record everything about a possible chess move: the initial and the final square, the moved piece, the (possible) captured piece, and many other useful information. The constructor is defined as follows.

```python
class Move:
    def __init__(self, initial, final, piece_moved = None, piece_captured =
        None, has_piece_moved = False, castling = False, en_passant = False,
         rating = 0):

        # initial and final are squares
        self.initial = initial
        self.final = final

        # remember the piece moved and the piece captured
        self.piece_moved = piece_moved
        self.piece_captured = piece_captured

        # remember if the moved piece had already been moved or not
        self.has_piece_moved = has_piece_moved

        # remember if the move is a castling move
        self.castling = castling

        # remember if the move is an en-passant move
        self.en_passant = en_passant

        # remember if the move gives a check
        self.check = False

        # remember if the move is a promotion
        self.promotion = False

        # Add a rating attribute to the Move object
        self.rating = rating
```

As we can see from the above code, the move constructor has a lot of optional attributes. This is because moves in chess can be very different from each other. For instance, we can have simple moves, capturing moves and special moves (promotion, castle or en passant). Furthermore, every of those move can also be a check[1] or a double check move. For this reason we need all those optional attributes. On the other hand, every move must always have an initial square and a final square, which indeed are the only two mandatory attributes.

2.1.3 Piece class

Here, we define a class to represent every **single piece** on the chessboard: the Piece class.

[1]A check is a move that puts the opposite king under attack.

This class has a slightly different structure compared to the previous two, because it is composed of one ”parent” class (the Piece class) and of 5 ”child” subclasses (one for each type of piece).

The constructor of the parent class and two of the child classes are defined as follows.

```
class Piece:
    def __init__(self, name, color, value):
        self.name = name
        self.color = color
        self.moved = False
        value_sign = 1 if color == 'white' else -1
        self.value = value * value_sign
    ...

class Pawn(Piece):
    def __init__(self, color):
      # Set the direction of movement of the pawn as an attribute
      if user_color == 'white':
        self.dir = -1 if color == 'white'  else 1
      else:
        self.dir = 1 if color == 'white'  else -1
      super().__init__('pawn', color, 1.0)

class Knight(Piece):
    def __init__(self, color):
        super().__init__('knight', color, 3.0)
```

This kind structure enable us to initialize any piece calling the corresponding subclass and specifying just the color of the piece. For example, to initialize a white knight we just have to write: `Knight('white')`, and we will never have to call the parent class.

Every instance of the Piece class stores the name, the color and the value of the piece corresponding to the called subclass. In order to facilitate the evaluation procedure of a board state, as we will see later, we store white pieces values as positive and black ones as negative.

Moreover, as we can see from the above code, if we call the Pawn subclass we also store in the `dir` attribute the direction of the Pawn, which of course change based on the chosen perspective of the board.

2.1.4 Board class

To conclude the internal representation of our chess engine we created the Board class, whose constructor is defined as follows.

```
class Board:
  def __init__(self):
    self.squares = []
    self.moveLog = []
    self.piece_positions = []
    self.checkmate = False
    self.stalemate = False
    self.white_turn = True
    self.w_castled = False
    self.b_castled = False

    self.create_board()
    self.add_pieces('white')
    self.add_pieces('black')
```

As we can see from the above code, we can define a Board object by just writing `board = Board()`, without passing any attributes. Once a Board object is assigned to a variable, a lot of internal attributes are initialized such as board squares, move log, piece positions, other than a lot of game state flags used to record if the board is in checkmate or stalemate, if it is White's turn or if White or Black have already castled.

Then, the board constructor calls the methods `self.create_board()` and `self.add_pieces(color)` for white and black.

The first one, as deducible from the name, is responsible to create the double array of Square objects which we use to represent the chessboard. This double array is stored in the internal attributed `self.squares`; in this way we'll be able to access anytime we want any square of the chessboard identified bu a `row` and by a `col` by just selecting `self.squares[row][col]`.

```
def create_board(self):
    self.squares = [[0,0,0,0,0,0,0,0] for col in range(COLS)]
    for row in range(ROWS):
        for col in range(COLS):
            self.squares[row][col] = Square(row, col)
```

The second method, called both for white and for black, places the pieces on the board by simply adding (where needed) to the Square objects attribute `piece` the piece of the color we want to add. Again, we just show the procedure for pawns and for knights since for the other pieces it is the same.

```
def add_pieces(self, color):
    ...
    # pawns
    for col in range(COLS):
        self.squares[row_pawn][col].piece = Pawn(color)

    # knights
    self.squares[row_other][files_to_cols['b']].piece = Knight(color)
    self.squares[row_other][files_to_cols['g']].piece = Knight(color)
    ...
```

In the `self.add_pieces(color)` method we also store, inside the attribute `self.piece_positions`, the initial piece positions of the added pieces. Those position will be of course updated every time a piece is moved.

Once a Board object is initialized, a lot of other methods are defined other than the two we just mentioned. Most of those methods (we will skip the easiest ones) will be explained in the following two subsections which covers the move generator and the external board representation.

2.2 Move generator

We call "Move generator" the entire logic of our chess engine that determines how pieces interact with each others on the chessboard. It provides a reliable foundation for both human gameplay and AI chess decision-making.

2.2.1 Legal move generation

Here we discuss how we managed to implement the legal move generator of our chess engine.

First we have to introduce some vocabulary. In chess, a **pseudo move** is a move that can be made on the board in a specific position by a specific piece without considering checks.

In chess programming a **pseudo-move generator** is a function that returns all possible pseudo-moves of a player, while a **legal-move generator** is of course a function that returns all possible legal moves of a player.

In our case, the most important method of the Board class, the method:

```
get_moves(row, col, look_for_checks = True, look_for_castling = True)
```

is both a pseudo and a legal move generator, based on the parameters that we pass when we call it. Let's now see how it works in details.

As we can see, the first two parameters (`row` and `col`) are obligatory, while the other two are optional and are True by default.

Indeed, this method is built to return a list of moves (pseudo or legal) of whatever piece lies in the position `row`,`col` of our board (accessed by `self.squares[row][col].piece`), and therefore a `row` and a `col` are always required. Of course, if the row and column provided as input are related to a square that doesn't contain any piece, it just return an empty list.

The third parameter, on the other hand, determine if the `get_moves()` method is a legal or a pseudo-move generator. Indeed, when `look_for_checks = True`, the method returns a list of legal moves, while when `look_for_checks = False` the returned moves are just pseudo moves.

Similarly, when the fourth parameter is passed as True, if the piece present in the position `row`, `col` is a king, the list of moves will contain also the castling moves (if possible), while if it is passed as False the move generator will ignore the presence of castling moves.

The structure of the `get_moves()` method is roughly the following:

- First we define four piece-specific move functions denoted `knight_moves()`, `pawn_moves()`, `straightline_moves(increments)` (for Bishops, Rooks, and Queens) and `king_moves()`.
- Then we look at what piece lies in the position `row`, `col` of our board and we call the corresponding function inside the method.

It is clear that the most difficult part is the first one, but the structure is always the same: find the pseudo moves and then, if required, filter them to obtain only the legal moves.

To obtain the pseudo-moves we obviously used different methods for different pieces:

- For the knights we check the 8 possible L-shaped moves a knight can make and then we add the move to the list of valid moves if the destination is empty or contains an opponent's piece.
- For pawns we had to distinguish between normal forward moves, diagonal captures and en passant captures. For normal forward moves we just add the move if the square in front of the pawn is empty (considering also the fact that in his first move a pawn can move by two steps). For diagonal captures we just add the move if the final square contains an opponent's piece and for en passant captures we also need to check the last move played, to make sure that the en passant is actually possible.
- For bishops, rooks and queens we use the same function `straightline_moves(increments)` but we pass a different parameter for each piece, based on the movements allowed for that specific piece. In this function we keep moving in a direction (given by the passed parameter `increments`) until we hit the edge of the board, a piece of the same color or an opponent piece which will be captured eventually, and at each step we add the move to the list of valid moves.
- For kings, since they can be moved by 1 square in any direction, we add the move for every direction if the final square is empty of contains a rival piece. For the castling moves, on the other hand, we use the function `is_castling_possible(row)` to determine if the castle move is actually possible, and if so we add it to the list of valid moves.

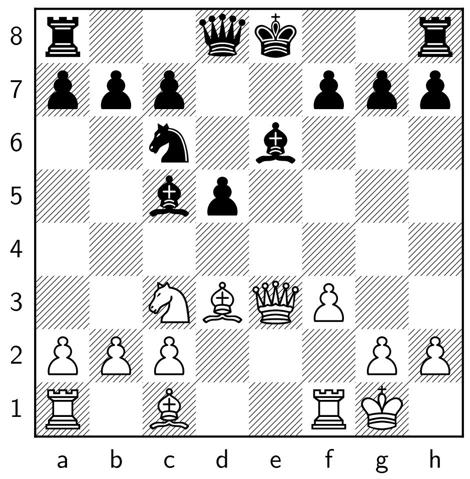
The method `is_castling_possible(row)` of the Board class is simply our way to check if the castling (both king-side and queen-side) of the king on the input row is possible or not. It works by checking the easiest necessary conditions, if there is actually a king in the selected square, if it has never been moved, if the rook have never been moved, and so on. If all these conditions are satisfied, then the methods loops through all the opponent pieces to check if any of those pieces attacks a squares involved in the castling, if at least one of them does, then the castling is not possible and the method returns False, otherwise it is possible and we return True.

Then, if we want only legal moves, for each pseudo-move we call the method `in_check(move)` which returns True if the move leaves the team king under attack (in such case we don't add the move to the list of valid moves) and returns False if it doesn't and thus we can append the move to the list of valid moves to return. The structure of the method `in_check(move)` is the following.

```python
def in_check(self, move):
    #make the move on the board
    self.make_move(move, testing = True)
    # check if the king is under attack
    for row in range(ROWS):
      for col in range(COLS):
          if self.squares[row][col].has_rival_piece(move.piece_moved.color):
            moves = self.get_moves(row, col, look_for_checks = False,
                look_for_castling = False)
            for m in moves:
              if isinstance(m.final.piece, King):
                self.undo_move(testing = True)
                return True
    self.undo_move()
    return False
```

Basically, every time a pseudo move is generated and we are asked for legal moves, the code runs virtually the move on the board using the `make_move(move)` method (which we will see later) and then it generates all the possible moves of the opponent player. If one of those moves ends up on the team king, it returns True, otherwise it returns False. When the process if ended, we undo the move given in input with the method `undo_move()`.

Notice that in the `in_check` method, when we generate all the possible moves of the opponent player, we pass the parameters `look_for_checks = False, look_for_castling = False`. This is very important because a piece can threat an attack even if it is pinned to the king. To understand this point better let's consider the following chess position.



In this case, after Black's move Bf5, the Black is under check, even if the white Queen is pinned to the King and therefore theoretically it cannot be moved.

For this reason it is of crucial importance to look only for pseudo moves inside the `in_check` method.

2.2.2 Making moves on the board

Now let's see how we manage the actually make and unmake a move on the board. To do those operations we use two methods of the Board class called `make_move` and `undo_move`, already cited above, which are very similar to each other.

The first one, defined as: `make_move(self, move, testing = False)` takes an obligatory parameter, the move we want to make, and an optional boolean parameter which is False by default. This method has basically the following structure:

1. **Making the move.** It removes the piece involved in the move from its starting square and it place it on its destination square. Then, it marks the piece as moved.
2. **Rules for Pawn.** If the moved piece is a Pawn then the method checks two special cases: the **en passant** and the **promotion**.
   - En passant: the captured Pawn (on the adjacent file) is removed.
   - Promotion: if a pawn reaches the back rank, it is promoted to a Queen.
3. **Castling.** If the move is a castling move then an appropriate flag is set (`w_castled` or `b_castled`). Then it determines if it's a **king-side castling** or **queen-side castling** and moves the rook to its new position marking it as moved.
4. **Updating the game history**: The move is stored in `moveLog`, so the game history can be printed later or used for undo a move.
5. If the optional parameter is not passed as True we:
   - Flag the move as a checking move if it leads to a check, using the method `is_move_check(move)`.
   - Detect, with the method `is_mate()`, if the move made leads to a checkmate or a stalemate and, if that is the case, update the state of the board.
   - Update the position of the pieces involved in the move.
6. **Ending and switching the turn**: the method calls the `next_turn()` method to pass the turn.

Let's see briefly the two methods of the Board class that we just cited:

- The `is_move_check(move)` method returns True if the move give as input delivers a check to the opposite king, and False otherwise. To implement this method we first looked at the "direct checks" (the possible checks made by the moved piece), and then we looked at the "indirect checks", that is, the checks made by other pieces once the move has been done. The second type of checks are also called **discovered checks**, since they occur when a move discovers the attack sliding piece (rook, bishop or queen) towards the opposite king. Of course, at the first check found the methods return True.
- The `is_mate()` method returns True if the opposite Player has no more legal moves and, in that case, if the previous move was a check then it is a **checkmate**, otherwise it is a **stalemate** (a draw).

The structure of the `undo_move(testing = False)` method is basically the reverse twin of `make_move`: its purpose is to revert the last move made on the board restoring the state of the game exactly as it was before the move. Notice however that this methods doesn't require a move as a parameter, this is because the last move is taken from the move log using the command `move = self.moveLog.pop()`, which assigns the last move made on the board to the variable `move` and deletes it from the move log. Furthermore, also in this case, if we call the `undo_move` method passing the parameter `testing = True`, then the positions of the pieces will not be updated. This is useful especially in the move generator, where combo of `make_move - undo_move` are very frequent and therefore it doesn't make sense to update and reset the piece positions every time.

Now that we have a clear understanding of the internal representation and of the move generator of our engine, we can pass to the explanation ot the external board representation.

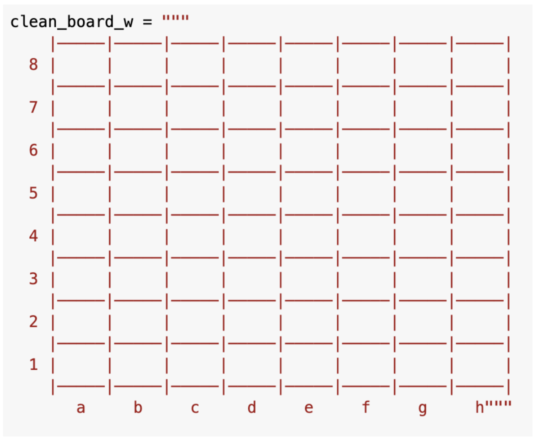
2.3 External board representation

In this section we explore the user interface that we implemented to represent the board as an output to the user, and we analyze the tools that we used to translate the user inputs into actual moves for our engine.

2.3.1 User interface

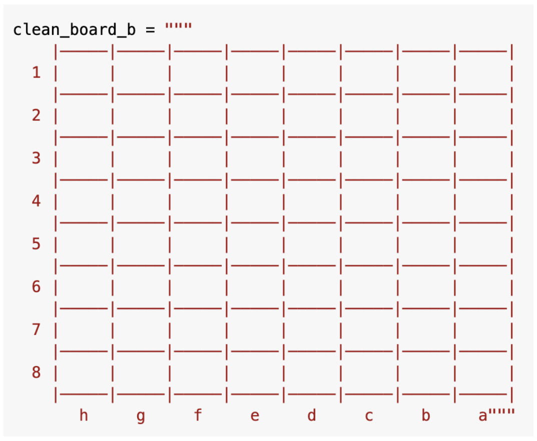
Since we had to work on Google Colab, which has a lot of limitations with the interaction of a virtual screen, we managed to create a method that prints the board in plain text every time we need it.

Moreover, since we wanted to give the user the choice to play with Black or White, we implemented this method in such a way that, given the user choice of the color, the printed board result rotated in Black's perspective or in White's perspective. To do that, we used as "empty boards" the two boards presented in Figure 1.



```
clean_board_w = """
    |─────|─────|─────|─────|─────|─────|─────|─────|
  8 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  7 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  6 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  5 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  4 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  3 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  2 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  1 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
       a     b     c     d     e     f     g     h"""
```

(a) White's perspective



```
clean_board_b = """
    |─────|─────|─────|─────|─────|─────|─────|─────|
  1 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  2 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  3 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  4 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  5 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  6 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  7 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
  8 |     |     |     |     |     |     |     |     |
    |─────|─────|─────|─────|─────|─────|─────|─────|
       h     g     f     e     d     c     b     a"""
```

(b) Black's perspective

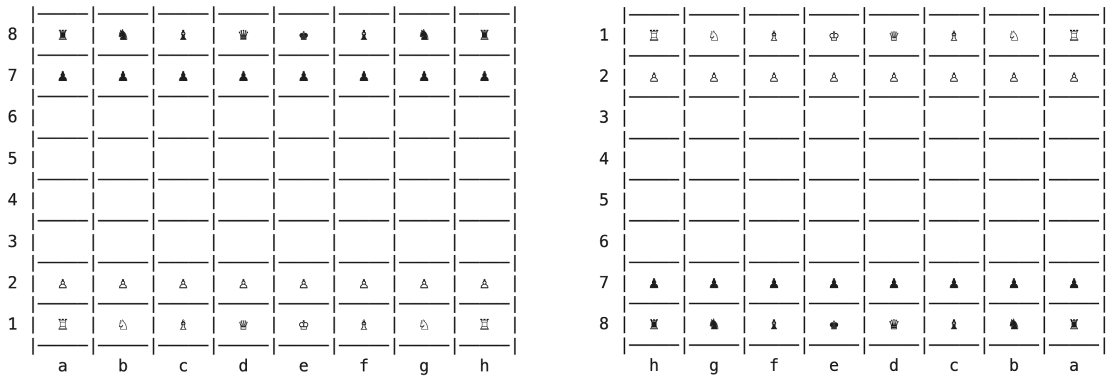
Figure 1: ASCII representation of the empty chessboards.

Now, to print the board as text we use a method of the Board class called `print_board()`, which is structured as follows:

- The empty board of the right perspective is stored (as a string) inside a variable `board_to_print`.
- Then we loop through all the squares of the board and when we find a square with a piece, we insert in the string board the ASCII character corresponding to the piece we found.

- Then we print the string with the piece on it.

The following figure shows how the initial boards looks like from White and Black's perspective.



```
  |-----|-----|-----|-----|-----|-----|-----|-----|
8 |  ♜  |  ♞  |  ♝  |  ♛  |  ♚  |  ♝  |  ♞  |  ♜  |
  |-----|-----|-----|-----|-----|-----|-----|-----|
7 |  ♟  |  ♟  |  ♟  |  ♟  |  ♟  |  ♟  |  ♟  |  ♟  |
  |-----|-----|-----|-----|-----|-----|-----|-----|
6 |     |     |     |     |     |     |     |     |
  |-----|-----|-----|-----|-----|-----|-----|-----|
5 |     |     |     |     |     |     |     |     |
  |-----|-----|-----|-----|-----|-----|-----|-----|
4 |     |     |     |     |     |     |     |     |
  |-----|-----|-----|-----|-----|-----|-----|-----|
3 |     |     |     |     |     |     |     |     |
  |-----|-----|-----|-----|-----|-----|-----|-----|
2 |  ♙  |  ♙  |  ♙  |  ♙  |  ♙  |  ♙  |  ♙  |  ♙  |
  |-----|-----|-----|-----|-----|-----|-----|-----|
1 |  ♖  |  ♘  |  ♗  |  ♕  |  ♔  |  ♗  |  ♘  |  ♖  |
  |-----|-----|-----|-----|-----|-----|-----|-----|
     a     b     c     d     e     f     g     h
```

(a) White's perspective

```
  |-----|-----|-----|-----|-----|-----|-----|-----|
1 |  ♖  |  ♘  |  ♗  |  ♔  |  ♕  |  ♗  |  ♘  |  ♖  |
  |-----|-----|-----|-----|-----|-----|-----|-----|
2 |  ♙  |  ♙  |  ♙  |  ♙  |  ♙  |  ♙  |  ♙  |  ♙  |
  |-----|-----|-----|-----|-----|-----|-----|-----|
3 |     |     |     |     |     |     |     |     |
  |-----|-----|-----|-----|-----|-----|-----|-----|
4 |     |     |     |     |     |     |     |     |
  |-----|-----|-----|-----|-----|-----|-----|-----|
5 |     |     |     |     |     |     |     |     |
  |-----|-----|-----|-----|-----|-----|-----|-----|
6 |     |     |     |     |     |     |     |     |
  |-----|-----|-----|-----|-----|-----|-----|-----|
7 |  ♟  |  ♟  |  ♟  |  ♟  |  ♟  |  ♟  |  ♟  |  ♟  |
  |-----|-----|-----|-----|-----|-----|-----|-----|
8 |  ♜  |  ♞  |  ♝  |  ♚  |  ♛  |  ♝  |  ♞  |  ♜  |
  |-----|-----|-----|-----|-----|-----|-----|-----|
     h     g     f     e     d     c     b     a
```

(b) Black's perspective

Figure 2: Initial text boards positions.

In this way, printing the board every time a move is made, we have been able to represent the flow of a chess game without a proper interactive user interface.

2.3.2 Input moves

Now, since on Colab we are only able to provide input with text, we first had to chose a standard notation accepted by our engine, and then we had to implement a method to translate an input move (in the right notation) into an actual move for our chess engine.

The notation that we chose follows the structure: `initial square - final square`, where each square is represented by a file and a rank.
So for example, considering an initial chess position as the one in Figure 2, to move the white d-Pawn the user must enter "d2d4", while to move the g-Knight to the f3 square the user must enter as input the line "g1f3".

Then, to encode the user move into an actual move, we use the method `get_right_notation(input_move)`.
This method first check the notation of the input move is correct and then, if it is so, it creates a Move object which correspond to the move entered by the user in the initial-final square notation. Once this move is created, this method also checks if it is a legal moves, calling the Board method `get_moves(initial_row, initial_col)` and checking if the new move is actually between the legal moves of the selected initial square. If we find that the user move is legal, then the method returns that move, otherwise it return a None value.

Inside the `get_right_notation` method, to go from ranks to rows and from files to columns, we make a large use of the **mapping dictionaries**. The mapping dictionaries are used to **translate** between initial-final square notation and the internal numerical representation of the board. They are initialized as empty dictionaries, and once the user chose the color to play with they are update in order to translate properly files with columns and ranks with rows and vice versa. In our code, function to update those dictionaries based on the chosen color is defined as follows.

```python
def get_swap_dict(color):
  if color == 'white':
    for row in rows:
      ranks_to_rows[row] = 8 - int(row)
    i = 0
    for col in cols:
      files_to_cols[col] = i
      i = i+1
  else:
    for row in rows:
      ranks_to_rows[row] = int(row) - 1
    i = 7
    for col in cols:
      files_to_cols[col] = i
      i = i - 1

  global rows_to_ranks, cols_to_files
  rows_to_ranks = {v : k for k,v in ranks_to_rows.items()}
  cols_to_files = {v : k for k,v in files_to_cols.items()}
```

These mappings make it easy to switch back and forth between the human-readable notation used in chess and the matrix coordinates used in the program.

3 Computer Moves

In this section we will talk about the methods that we used in order to obtain chess engine that is able to select ”reasoned” moves from a specific position with a discrete level of accuracy.

First we will talk about the well known Minmax algorithm, the **search function** we used in our engine, providing a simple example to help the reader understand the **basic functioning** of the algorithm. Then we will explain how we have been able to speed up considerably the performance of our engine with the help of the **alpha-beta pruning**. At last, we will see the **full implementation** of our main search algorithm and, after performing a complexity analysis on it, we will be able to discuss the point of strength of our search function, as well as its limitations.

3.1 Minmax algorithm

The **minimax rule** is a decision strategy used to **minimize** the worst-case loss or **maximize** the minimum gain. First developed for multi-player zero-sum games, it now applies more broadly to complex games and decision-making under uncertainty.

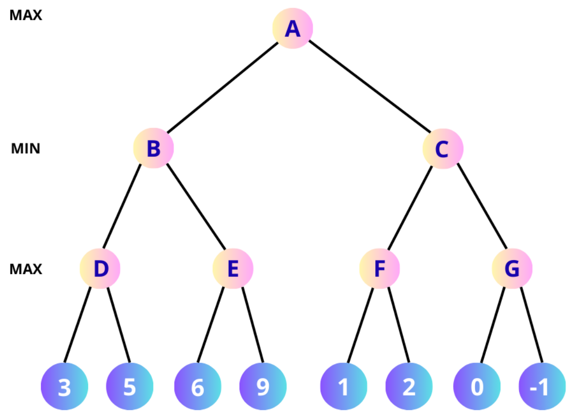


Figure 3: Fundamental graph of Minimax algorithm

Explanation of the algorithm. In a **one-ply search**, where only a single move is considered, the **max player** can simply evaluate all possible moves and choose the one with the highest score. However, in a **two-ply search**, the opponent also has a turn, and thus the situation becomes more complex. The opponent will select the move that **minimizes** the max player’s advantage, effectively choosing the move that leads to the **worst** possible outcome for the max player. Therefore, in this case, the value of each possible move for the max player is determined by the minimum score that the opponent can force. As the number of plies increases, the minimax algorithm **recursively** evaluates all possible sequences of moves, alternating between maximizing and minimizing at each level, until it reaches a predetermined search depth or a terminal game state. This approach ensures that the chosen move is the best possible under the assumption of **perfect play** by both sides.

Of course this reasoning can be made also for 3 ply, for 4 ply and so on. We call this concept **depth**: the number of plies that the algorithm search in order to obtain the best possible move. It follows that an higher depth increases the accuracy of the algorithm but of course it requires more computations.

In general, the **maximum value** is the highest value that the player can be sure to get without knowing the actions of the other players; equivalently, it is the lowest value the other players can force the player to receive when they know the player's action.

3.1.1 The score function

As we just saw, the Minmax algorithm works by **comparing the scores** of different chess positions. Thus, before starting with the implementation of our main algorithm, we need to build a method to evaluate a chess position anytime we need.

What is an evaluation? An evaluation of a chess position is basically an a heuristic function to determine the relative goodness of the position. Since in Chess there are two players and since a good position of one's reflects to a bad position of the other's, in order to have a unique evaluation function for both players it is common practice to assign positive scores as a good evaluation for White, and negative scores as a good evaluation for Black.

Of course, any possible evaluation function of a chess position will always be an approximation of the "real score" of the position. Indeed, if we could see to the end of the game in every line, the evaluation function would only have three possible outcomes: `white_won`, `draw`, `black_won`. In practice, however, we are not able to check every possible line of a chess game and therefore we have to make some sort of approximations of the real score using heuristic ideas.

In the recent literature, there are two main ways to build an evaluation function:

- The traditional **hand-crafted evaluation (HCE)**.
- Using a **multi-layer neural networks**.

In our chess engine we use a simple hand-crafted evaluation function, and therefore we are going to focus on that one.

In general, an hand-crafted evaluation function has the form:

$$S = f(k_1, k_2, \dots, k_n)$$

where $k_1, k_2, \dots, k_n$ are some sort of "features" of the chess position we want to evaluate (such as piece values, king safety, pawn structure, ...) and the value S returned by the function $f(\cdot)$ is the so-called **static evaluation** of the considered position.

However in practice the function f is reduced to be a linear polinomial function of the n observed features of the chess position we are evaluating, that is:

$$S = c_1k_1 + c_2k_2 + \cdots + c_nk_n \ .$$

In our case, the evaluation function is defined to be just the weighted sum of the values of the pieces on the board with a multiplicative factor different for each piece and which depends on the position of that particular piece on the board.

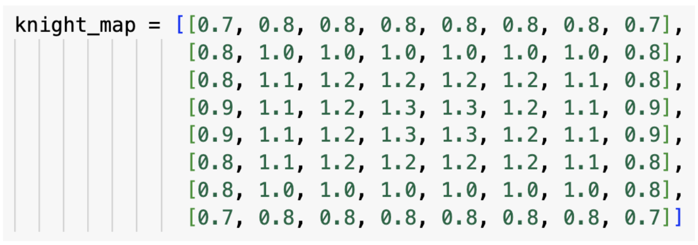
We implemented this evaluation function inside a method of the Board class called `score()`, which is defined as follows.

```python
def score(self):
    if self.checkmate:
        if self.white_turn:
            return -CHECKMATE # black wins
        else: return CHECKMATE # white wins
    elif self.stalemate: return STALEMATE
    score = 0
    # add white scores
    for piece_type in self.piece_positions[0]:
        for pos in self.piece_positions[0][piece_type]:
            piece = self.squares[pos[0]][pos[1]].piece
            score += piece.value * piece_position_scores['w' + piece_type][
                pos[0]][pos[1]]
    # add black scores
    for piece_type in self.piece_positions[1]:
        for pos in self.piece_positions[1][piece_type]:
            piece = self.squares[pos[0]][pos[1]].piece
            score += piece.value * piece_position_scores['b' + piece_type][
                pos[0]][pos[1]]
    return score
```

As we can see, our score function starts with an if-condition to check if the game has ended and, if so, it returns a large positive or negative number depending on which side won. If the game ended in stalemate, instead, the function returns the special constant `STALEMATE = 0` to indicate a draw.

After those conditions, we initialize the score value at zero and then we loop through the pieces on the chessboard and, for each piece, we compute the contribution of that piece and we sum it to the total score of the board.

As already explained, each contribution is obtained by multiplying the piece's value with a factor depending on the piece's position. Those factors are stored in a dictionary called `piece_position_scores`, which contains the **heat maps** for every white and black piece. The following figure shows the heat map of the knights.



```python
knight_map = [[0.7, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.7],
              [0.8, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.8],
              [0.8, 1.1, 1.2, 1.2, 1.2, 1.2, 1.1, 0.8],
              [0.9, 1.1, 1.2, 1.3, 1.3, 1.2, 1.1, 0.9],
              [0.9, 1.1, 1.2, 1.3, 1.3, 1.2, 1.1, 0.9],
              [0.8, 1.1, 1.2, 1.2, 1.2, 1.2, 1.1, 0.8],
              [0.8, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.8],
              [0.7, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.7]]
```

In this way, we can modulate the value of a piece relying on the fact that certain pieces are better in certain positions that in others. For example pawns gets better as they advance up the board since they may promote to a queen, knights are more valuable in the center than on the edge while kings are safer when they are in one of the team corners of the board.

When the loop ends, the score function returns a numeric score which, as already said, represent the **static evaluation** of the board. This means that this evaluation is based only on what is directly deducible from the board, without searching any line in advance (the minmax algorithm will have such a task).

3.1.2 Toy Example (Plain Minimax)

Let's now analyze a simple example to understand the functioning of the minmax algorithm applied to the game of Chess. We first provide the pseudocode of the algorithm in order to have a better view on its functioning.

```
function minimax(node, depth, maximizingPlayer):
    if depth = 0 or node is a terminal node then
        return the score of the node
    if maximizingPlayer then
        max_value = -infinity
        for each child of node do:
            score = minimax(child, depth - 1, FALSE)
            max_value = max(max_value, score)
        return max_value
    else  # minimizing player
        min_value = +infinity
        for each child of node do:
            score = minimax(child, depth - 1, TRUE)
            min_value = min(min_value, score)
        return min_value
```

Let's consider the situation in which we play as White against a computer playing as Black. For semplicity, let's consider a 3×3 board with just two pieces on it, as shown in Figure 4. We want to see what move will chose the Black after analyzing the position with a depth of 2 using as logic the minmax algorithm.

Let's now analyze in detail the **behavior** of the Minimax algorithm starting from the initial position of Figure 4 called with a depth of two:

- The **Black moves first** and its interest is to minimize the score. This means that the root node is a **minimizer**. The **root** is the position where the algorithm starts thinking, and then it branch out into **possible moves**. This node will choose the minimum between the two minimax scores below.
- Entering the **left branch** we find a **maximizer**. At this point, the algorithm considers the movements that the White can make in the **first** scenario.
    1. The algorithm check the board on the left and gives an evaluation of 0.
    2. Then, it checks the board on the right and gives an evaluation of +3.
    3. The parent node gets a minimax score of +3 since it is a **maximizer**.

- Entering the **right branch** we find a **maximizer**. At this point, the algorithm considers the movements that the White can make in the **second** scenario.
  1. The algorithm check the board on the left and gives an evaluation of 0.
  2. Then, it checks the board on the right and gives an evaluation of 0.
  3. The parent node gets a minimax score of 0, the **maximum** value that it can get from the nodes below.
- The algorithm chooses the **right branch**, since the root node is a **minimizer** that has to choose between +3 and 0. The fact that the root value will be 0 means that if Black moves first and plays optimally, the best White can achieve is 0.

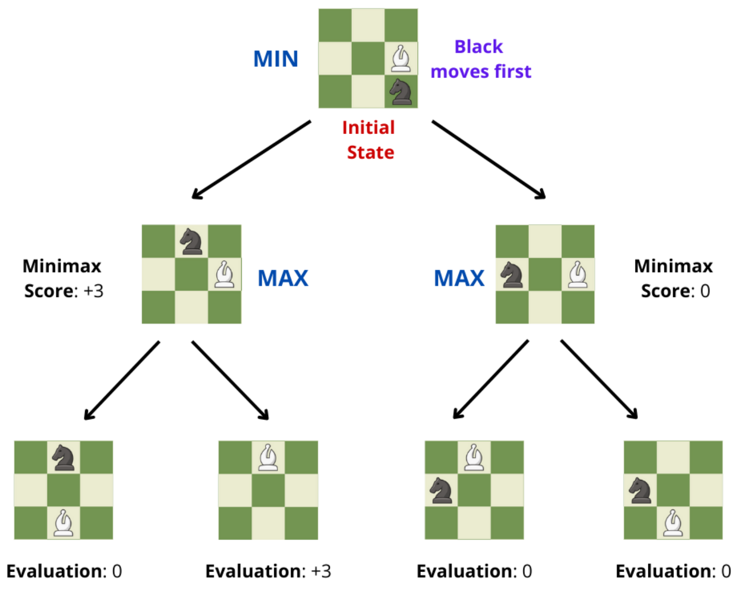


Figure 4: Example of Minimax algorithm.

Static evaluations At the very bottom, we have the evaluation values: on the left branch leaves we find 0 and +3, while on the right branch leaves we find 0 and 0. These are the static evaluations of the positions (how good they are for White, in this case).

3.2 Alpha-beta pruning

In this section we are going to talk about a method commonly used to speed up considerably the performances of the Minmax algorithm: the alpha-beta pruning.

What is it? The alpha–beta pruning is a technique designed to reduce the number of nodes explored in the minimax algorithm's game tree. The method works by discarding branches as soon as it becomes clear that a move cannot yield a better outcome than one already considered. A s a result, unnecessary evaluations are avoided.

It is important to underline that, when applied to a minimax tree, alpha–beta pruning produces the **same final decision** as the Minimax, but usually with a lot fewer computations.

We say "usually" because the the efficiency improvement of the alpha–beta pruning technique depends heavily on the order with which the moves are analyzed by the Minmax algorithm.

In particular, the alpha–beta pruning performs better when the moves to analyze are sorted from the best to the worst. In this case indeed we are able to cut a larger portion of the branches of the game tree and therefore to save more computations. On the other hand, in the unfortunate case in which the moves to analyze are sorted from the worst to the best, the alpha–beta pruning doesn't produce any savings of computations (in fact the performance in such case is even slightly worse than the one of the plain Minimax).

We will see later how we managed to create a function capable of ordering the moves following some basic heuristics and without having to look for lines ahead.

3.2.1 Toy Example (Minimax with Alpha-Beta pruning)

Let's now see, with a little example, how the alpha-Beta pruning technique works when applied to the Minmax algorithm.

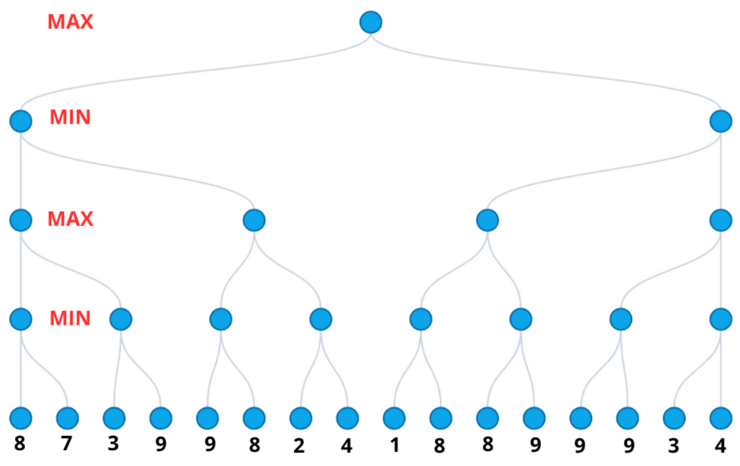


Figure 5: Example of minimax with alpha-beta pruning)

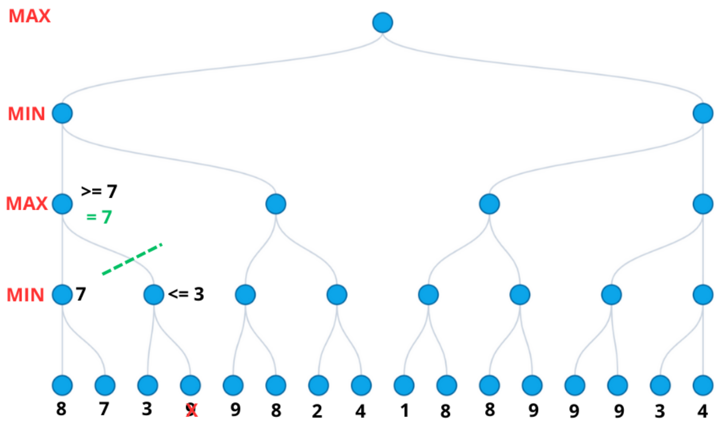
Let's consider the game tree shown in Figure 5 and assume the **root node** to be a **maximizer**, and not a minimizer like before. The depth also changed, here we have more layers in order to show in a better way how the pruning actually works.

This initial condition shows us an example of the static evaluation scores that the last nodes can get, now we have to climb the tree to how we can optimize the computations using the alpha-Beta pruning technique.

Left branch We start by analyzing the left branch of the root maximizer function. We start by the last level of the graph. We will update the picture step by step so that we can have a clear understanding of the functioning.

1. The **first minimizer starting from the left** looks at the 8 and (since it is a minimizer) it gets that it has to take a value ≤ 8. Then, it looks at the 7, and since $7 < 8$, the minimizer value updates to 7. The first minimizer now has the **value** of 7. Since its value is 7, we can also conclude that the maximizer above will have a value ≥ 7.
2. The **second minimizer starting from the left** looks at the 3 and gets that it has to take a value ≤ 3.

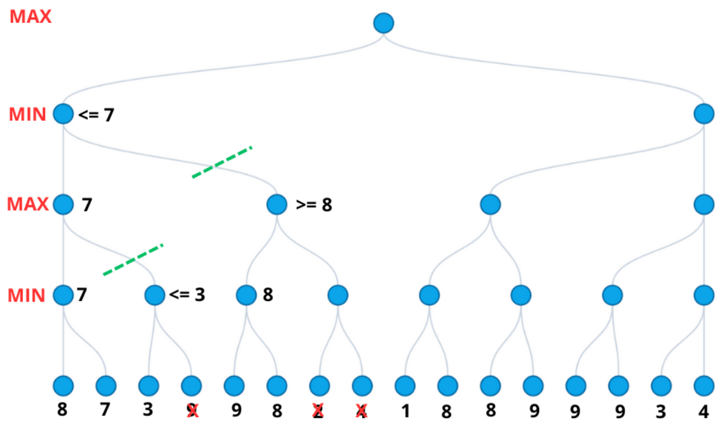
The current situation allows us to prune and **cut a branch**. Why? We said that the maximizer above (the first maximizer starting from the left on the third level) will take a value ≥ 7, while the second minimizer starting from the left can only take values ≤ 3. This allow us to cut the branch skipping the control on the 9. The **maximizer successfully updates the value** to 7 avoiding a control. Here's what's happening graphically.



The algorithm goes on and we now also know that the first **minimizer** starting from the left on the second level will have a value ≤ 7. It's time to evaluate the third and fourth **minimizers** starting from the left on the fourth level.

1. The third minimizer starting from the left looks at the 9 and gets that it has to have a value ≤ 9. Then it looks at the 8 and updates its value to 8. We stop here.
2. The **maximizer** above now knows that it has to have a value ≥ 8.

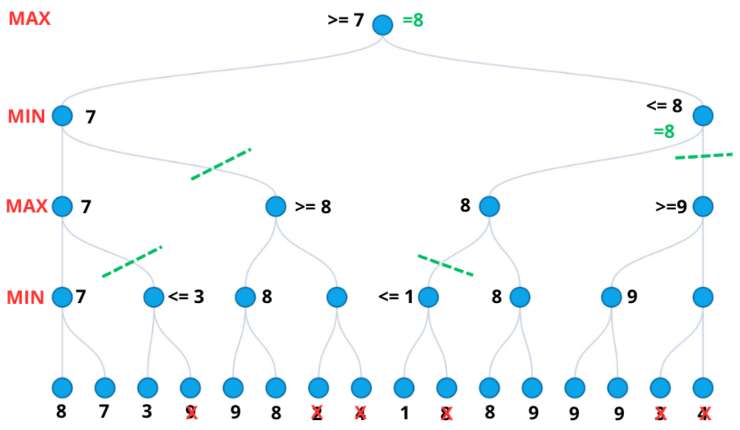
The left minimizer on the second level wants a value ≤ 7, and the right maximizer below will have a value ≥ 8. We prune the entire right branch of the second level minimizer. This allows us to avoid the control of the end nodes with scores 2 and 4, resulting in a big improvement because even the move generation of those nodes is skipped. Let's see graphically what's happening if the following picture.



Right branch We repeat the same reasoning for the main right branch that starts from the second level. Here we have to notice that:

1. The fifth **minimizer** on the fourth level starting from the left looks at the 1 first. It understand that it has to take a value ≤ 1.
2. Does the **maximizer** on top allow that? Not really. It is searching for a number ≥ 7. So we can avoid the control on the 8 and cut the branch off.
3. The maximizer above instantly takes the value of 8. Another move generation is avoided, and **computation is saved**.

Going on, we get the following graph.



Now, since the **root** node is a maximizing node searching for a value ≥ 7 and since the right node below takes exactly 8, we conclude that the root node takes a final score of 8.
This means that the maximizing player will chose the move which leads to the right branch and, if the following move played are optimal for both the players, the static evaluation of the game after 3 other moves will be 8.

This example showed briefly how the alpha–beta pruning technique is used along with the Minmax algorithm in order to save a lot of computations. Again, we'd like to point out that the same result as the one we got would have been obtained using the plain Minmax algorithm, but it would have been requires lots of extra operations.

3.3 Full implementation

In this section, we show our entire implementation in Python of the **minimax algorithm** improved with the **alpha-beta pruning**. The main function is defined as follows.

```python
def minmax_alphabeta(board, depth, alpha, beta, white_to_move):
  global best_move, position_counter, move_counter
  position_counter += 1

  if depth == 0 or board.checkmate or board.stalemate:
    return board.score()

  moves = board.get_all_moves()
  move_counter.append(len(moves))
```

```
# move ordering
moves = order(moves)

# white
if white_to_move:
  max_score = -CHECKMATE
  for move in moves:
    board.make_move(move, look_for_mates = True, update_positions = True)
    score = minmax_alphabeta(board, depth - 1, alpha, beta, False)
    if score > max_score:
      max_score = score
      if depth == DEPTH:
        best_move = move
        print('best move so far: ', cols_to_files[move.initial.col],
            rows_to_ranks[move.initial.row], '->', cols_to_files[move.
            final.col], rows_to_ranks[move.final.row])
    board.undo_move(update_positions = True)
    # pruning
    alpha = max(max_score, alpha)
    if alpha >= beta:
      break
  return max_score

# black
else:
  min_score = CHECKMATE
  for move in moves:
    board.make_move(move, look_for_mates = True, update_positions = True)
    score = minmax_alphabeta(board, depth - 1, alpha, beta, True)
    if score < min_score:
      min_score = score
      if depth == DEPTH:
        best_move = move
        print('best move so far: ', cols_to_files[move.initial.col],
            rows_to_ranks[move.initial.row], '->', cols_to_files[move.
            final.col], rows_to_ranks[move.final.row])
    board.undo_move(update_positions = True)
    # pruning
    beta = min(score, beta)
    if beta <= alpha:
      break
  return min_score
```

We already explored most of the functions that are present here in the previous sections, now we just have to speak about the function `order(moves)`.

The `order(moves)` function sorts the list of moves passed as a parameter from the "strongest" to the "weakest", following some heuristic features of moves in chess such as checks, captures, promotions and so on. Of course we don't know in advance what are the strongest move (it is the whole point of the Minmax algorithm to find them), but it turns out that even a moderately accurate ordering yields a good improvement how many branch of the game tree are actually pruned, which in turn results in even higher computational efficiency.

The code of our ordering function is the following.

```
def order(moves):
  for move in moves:
      if move.piece_captured is not None:
        move.rating = 1
        if isinstance(move.piece_moved, Pawn) and not isinstance(move.
            piece_captured, Pawn):
            move.rating += 1
      if move.check:
        move.rating += 1
      if move.promotion:
        move.rating += 1
      elif move.castling:
        move.rating += 1
  ordered_moves = sorted(moves, key=lambda move: move.rating, reverse=True)
  return ordered_moves
```

The base rating for every piece that **captures** is 1, but with some bonuses in particular cases. If a Pawn captures a piece that is not a Pawn it gets a bonus in this ranking system (+1). If a Pawn captures a Pawn, the ranking is not improved. If the piece that **captures** is a **checker**, then we add a bonus (+1), same for **promotion** and **castling**. The function returns ordered moves ranked as written.

3.3.1 Complexity analysis

Let's now perform the complexity analysis of our main searching algorithm (the Minmax algorithm with alpha-beta pruning).

To do that, we will consider the time complexity of every function used in the algorithm and then we will find the general time complexity by recalling the fact that this function is **recursive**.

Starting from the top we find that:

- The check on the `depth` variable has an asymptotic complexity of $O(1)$.
- The method `get_all_moves` scans the whole 8×8 board (constant 64 squares) and, for each square containing a piece of the current player, calls `get_moves` to generate all its legal moves. On a fixed 8×8 chessboard this is effectively constant time, since there is only a finite number of piece on the board (max 32) and every piece has only a finite number of possible legal moves. Thus, asymptotically, the complexity of the `get_all_moves` method is still $O(1)$.
- The `order` function explored before is a complete ordering of the moves, it has complexity of $O(b \log b)$ where b corresponds to the number of moves required to order. We call b the

branching factor, since it denotes how many valid moves we have per state and therefore how many branch each node will have. Of course in chess this number is not fixed, but changes relatively to the position for which we generate the moves. However, it is known that empirically the average branching factor is ≈ 35 moves per position.

The order function first loops through the moves in order to update the ratings, and then it uses the built-in function `sorted` to order the moves.

The loop has a complexity of $O(b)$, while the `sorted` function is in fact a **Timsort** algorithm (a sorting method derived by merge sort and insertion sort), that has complexity $O(b \log b)$ in the average and worst case. However, since the branching factor is variable but surely bounded, we can still say that asymptotically the whole order function has constant $O(1)$ time complexity.

- The `make_move` method includes basic assignments like attributes updates and pieces movements that have a costant time of execution $O(1)$. Then, the special checks for en passant, castling, etc.. are executed on a fixed number of squares which means again complexity of $O(1)$. Inside this function we find two other functions that we have to analyze: the `is_move_check(move)` and `is_mate` methods of the Board class. We will explore them in a bit. At the end of the function we have the `next_turn` method which has a $O(1)$ complexity.
- In the `is_move_check(move)` method we scan through all moves of the moved piece and through all the sliding pieces standing on the diagonals, on the file or on the rank of the opposite king square. Since all these loops are finite, we can practically say that the time complexity of this method is still $O(1)$.
- In the `is_mate` method again we scan of the chessboard and we generate the moves of each opponent piece that we find until we get at least one legal move. Since the number of opponent pieces is finite, as well as the squares of the board, the complexity of this method is again $O(1)$.
- The `undo_move` method has only constant time operations since it doesn't have any loops. Thus its time complexity is again $O(1)$.
- Lastly, the pruning is made by simple logic operations, thus we still have an $O(1)$ complexity.

Since all those operations have asymptotic constant time, to evaluate the time complexity of the Minmax algorithm we just have to consider the number of times the function calls itself inside the recursion.

The Minmax algorithm calls itself at any node of the game tree, thus we deduce that the time complexity of the whole algorithm is $O(b^d)$ where, as we just said, b denotes the average branching factor and d denotes the depth with which the algorithm is called in the first place.

On the other hand, it is empirically known that the **alpha-beta pruning** operations can improve, in the best case, the time complexity of the Minmax algorithm by a square root factor, from $O(b^d)$ to $O(b^{(d/2)})$. In the worst case, as already said before, when poor moves are explored first, pruning is minimal and the complexity remains $O(b^d)$. In practice, however, with reasonably good move ordering the time complexity often falls somewhere in between, around $O(b^{3d/4})$.

To conclude we found that the Minmax algorithm with alpha-beta pruning has theoretical time complexity of:

1. $O(b^d)$ in the **worst case**, when the moves are not ordered at all.
2. $O(b^{(d/2)})$ in the **best case**, when the moves are perfectly ordered.
3. $O(b^{(3d/4)})$ in the **average case**, when the moves are partially ordered.

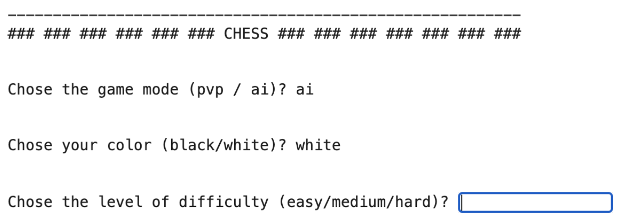
Space complexity. The Minmax algorithm has a total liner $O(d)$ space complexity where, again, d denotes the depth searched. This is because of the **recursive implementation** of the algorithm. Indeed, at each level of depth, the function call stores a small amount of local information, such as the current board state, the values of alpha and beta, and the moves being considered. Importantly, the algorithm does not store the entire tree in memory at once: it explores one branch at a time, and when it backtracks, the memory from the deeper calls is released. As a result, the memory usage is determined only by the longest chain of recursive calls, which is equal to the **maximum search depth**. This is why the space complexity of The Minmax algorithm is $O(d)$.

4 Gameplay Loop

In this section, significantly shorter than the previous two, we examine the structure of our Gameplay loop that we use to coordinate the interaction between the human player, the AI and the game state.

The first operations that we do once the Gamplay starts concern the request to the user for a series of information regarding the modalities of the game.
In particular we first ask the user to chose a game mode, between **PvP** and **AI**:

- If the PvP (plyer versus player) mode is chosen, then we just ask the user to chose the perspective of the printed board, and then the game can start.
- If the AI mode is chosen, we first ask the user to chose the color to play with (the AI will be assigned to the opposite color) and then we ask the user to chose a level of difficulty of the game.



```
---------------------------------------------------------
### ### ### ### ### ### CHESS ### ### ### ### ### ### ###


Chose the game mode (pvp / ai)? ai


Chose your color (black/white)? white


Chose the level of difficulty (easy/medium/hard)?
```

Figure 6: Example of modality request.

At coding level, when we ask to chose a level of difficulty, we are basically asking the user to set the **depth** parameter for the Minmax algorithm. In particular, the difficulties set the following level of search depth:

- **Easy** level has a depth of 2.
- **Medium** level has a depth of 3.
- **Hard** level has a depth of 4.

After this first part, we can initialize the swap dictionaries (based on the color chose by the use) and then we are ready to jump into the real Gameplay loop.

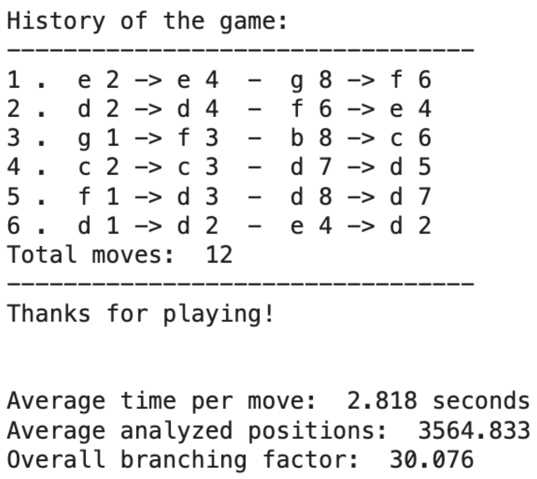
The structure of the gameplay loop is roughly the following:

- Print the board with all the changes done until now
- Look for gameover board status (checkmate or stalemate), and in case exit the loop.
- If it's the turn of a human player, ask for a move until a legal move is entered, then execute the move on the board.
- If it's the turn of the AI, generate all the possible moves from the current state of the board, look for the best move among them and execute it on the board.

- Loop again until a gameover occur or until the run gets blocked.

Once the game loop stops, the history of the game is printed out, as well as other useful information about the game, such as the **average time per move**, the **average analyzed positions** and **overall branching factor**.

Furthermore, since we wanted to print these information even if the script is interrupted before a gameover occurs, we handled the `KeybordInterrupt` issue with a `try/except` structure.
The following image shows an example of the endgame printed informations.



```
History of the game:
---------------------------------
1 .  e 2 -> e 4  -  g 8 -> f 6
2 .  d 2 -> d 4  -  f 6 -> e 4
3 .  g 1 -> f 3  -  b 8 -> c 6
4 .  c 2 -> c 3  -  d 7 -> d 5
5 .  f 1 -> d 3  -  d 8 -> d 7
6 .  d 1 -> d 2  -  e 4 -> d 2
Total moves:  12
---------------------------------
Thanks for playing!


Average time per move:  2.818 seconds
Average analyzed positions:  3564.833
Overall branching factor:  30.076
```

Figure 7: History of the game

5 Conclusions and further improvements

In conclusion, in this project we implemented a functional **chess engine** which handles both the **board implementation** and the **Minimax search algorithm with alpha-beta pruning**. These two things allowed the engine to **evaluate** positions and make **strategic** decisions while significantly reducing search space.

However, we didn't reached an amazing result regarding the depth that we have been able to search in reasonable amount of times, since at depth 5 we already got 20/30 second per move in average. Despite this fact, we are still satisfied of the work done since at depth 4 the AI plays like a farily good intermediate player, and even at depth 2 or 3 beginner players can experience some difficulties in defeating it.

On the other hand, we are aware of the great deals of improvements that may be done to speed up significantly the move generation algorithm and to increase the accuracy of the search algorithm. Thus, some further improvements may be:

- A bit-board internal representation of the board that, unlike the class-based representation, is less clear but significantly more efficient.
- The use of Transposition Tables that, subsequentally memorizing chess position, would have speed up a lot the move generator.
- The use of Quiescence Search in the Minmax algorithm, in order to be more accurate in the search of the best move by stopping the search only when a final leaf node doesn't correspond to a capture or a check move.
- The implementation of a brand new algorithm to detect checks in the legal move generator. Our algorithm based on the search of all the possible moves after a certain move is surely more intuitive but is extremely more expensive than other more sophisticated algorithms.