(<u>The heuristics are discussed on page 6-8 for MinMax and on page 9-11 for Alpha Beta</u>)

Report: Chess Game

For this exercise, I used the Python chess library with move generation, move validation, and support for common formats. I also used Pygame, which is a Python language module dedicated to graphics. The library contains several graphic functions, useful for creating games and animations. The heuristics to be used within the main are also imported. The variables necessary for the images and the board to be built are defined.

```
import chess
from minmax_heuristic import MyHeuristic
from alphabeta_heuristic import MyHeuristic2
import pygame as p

WIDTH = HEIGHT = 512
# dimensions of a chess board are 8x8
DIMENSION = 8
SQ_SIZE = HEIGHT // DIMENSION
# for animations later on
MAX_FPS = 15
IMAGES = {}
```

The functions for uploading the images to the board and the board itself are then defined. The drawBoard function defines the size of rows and columns and the color they should have. With the drawPieces function the pieces inside the board are drawn.

```
# Initialize a global dictionary of images. This will be called exactly once in the main
def loadImages():
    pieces = ["wP", "wR", "wN", "wB", "wQ", "wK", "bP", "bR", "bN", "bB", "bQ", "bK"]
    for piece in pieces:
        IMAGES[piece] = p.transform.scale(p.image.load("img/" + piece + ".png"), (SQ_SIZE, SQ_SIZE))
    # Note: we can access an image by saying 'IMG['wp']'
```

```python
# Responsible for all the graphics within a current game state
def drawGameState(screen, board):
    # draw squares on the board
    drawBoard(screen)
    # add in piece highlighting or move suggestions (later)
    # draw pieces on top of those squares
    drawPieces(screen, board)
```

```python
"""
    Draw squares on the board. The top left square is always light.
"""



def drawBoard(screen):
    colors = [p.Color("white"), p.Color("grey")]
    for r in range(DIMENSION):
        for c in range(DIMENSION):
            color = colors[((r + c) % 2)]
            p.draw.rect(screen, color, p.Rect(c * SQ_SIZE, r * SQ_SIZE, SQ_SIZE, SQ_SIZE))
```

```
"""
    Draw the pieces on the board.
"""


def drawPieces(screen, board):
    for r in range(DIMENSION):
        for c in range(DIMENSION):
            piece = board[r][c]
            if piece != ".":
                if piece.isupper():
                    piece = "w" + piece
                else:
                    piece = "b" + piece.upper()
                screen.blit(IMAGES[piece], p.Rect(c * SQ_SIZE, r * SQ_SIZE, SQ_SIZE, SQ_SIZE))
```

A function is defined to create a chess matrix useful in visualization and in the various operations to be carried out once the program is run.

```
def make_chess_matrix(board):
    pgn = board.epd()
    foo = []
    pieces = pgn.split(" ", 1)[0]
    rows = pieces.split("/")
    for row in rows:
        foo2 = []
        for thing in row:
            if thing.isdigit():
                for i in range(0, int(thing)):
                    foo2.append('.')
            else:
                foo2.append(thing)
        foo.append(foo2)
    return foo
```

Then the main is defined where the screen is set, the images loaded and the heuristics of minmax and alpha beta pruning are recalled. So for the program to be running and if you do not have the Game Over, you play with the two heuristics creating the screen and the matrix of the board (therefore the game space).

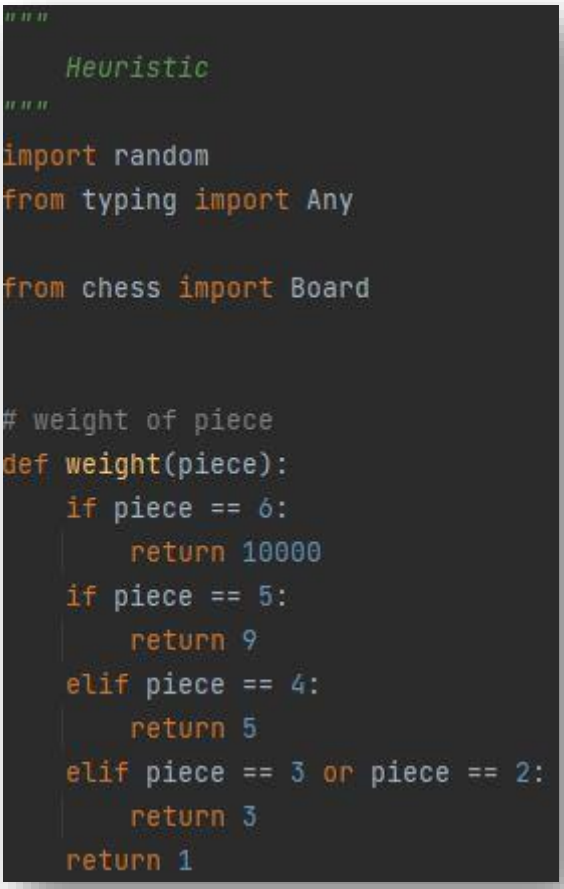
```
# The main driver of the code. This will update the graphics.
def main():
    p.init()
    screen = p.display.set_mode((WIDTH, HEIGHT))
    clock = p.time.Clock()
    screen.fill(p.Color("white"))
    # only do this once, before the while loop
    loadImages()
    running = True
    gameOver = False
    board = chess.Board()
    minmax = MyHeuristic()
    alphabeta = MyHeuristic2()
```

```
while running:
    for e in p.event.get():
        if e.type == p.QUIT:
            running = False
        clock.tick(MAX_FPS)
        p.display.flip()
        if not gameOver:
            if board.turn:
                minmax.H_L(board, 1)
                board.push(minmax.move)
            else:
                alphabeta.H_L(board, 2)
                board.push(alphabeta.move)
            gameOver = board.is_game_over()
        drawGameState(screen, make_chess_matrix(board))
```

```python
if __name__ == '__main__':
    main()
```

Heuristic: MINMAX

Everything needed for heuristic is imported. The weight of the pieces of the board is then defined.



```
"""
    Heuristic
"""
import random
from typing import Any

from chess import Board


# weight of piece
def weight(piece):
    if piece == 6:
        return 10000
    if piece == 5:
        return 9
    elif piece == 4:
        return 5
    elif piece == 3 or piece == 2:
        return 3
    return 1
```

The MyHeuristic class is defined (later recalled in the main) and two functions are defined inside it:

- ***H_0: which defines the evaluation metric (it is an evaluation function of the configuration situation in the game. It defines how good a given setup is for the player);***
- ***H_L: which defines the minmax algorithm. This heuristic search alternates the choice of which state to choose next, based on the minimum or maximum of the heuristic evaluation of the board position, depending on the player's turn. So, while Player A is trying to maximize the position rating, it is believed that the opponent is minimizing the same rating.***

```
class MyHeuristic:
    move: Any

    def __init__(self):
        pass

    def __repr__(self):
        pass

    # metric of evaluation
    def H_0(self, state: Board):
        pieces = state.piece_map()
        ret = 0
        for index in pieces:
            if pieces[index].color:
                ret += weight(pieces[index].piece_type)
            else:
                ret -= weight(pieces[index].piece_type)
        return ret
```

```python
# min max
def H_L(self, state: Board, l: int):
    if l == 0:
        return self.H_0(state)
    possible_moves = list(state.legal_moves)
    random.shuffle(possible_moves)
    if len(possible_moves) > 0:

        best_move = possible_moves.pop()
        state.push(best_move)
        best_value = self.H_L(state, l - 1)
        state.pop()
        while len(possible_moves) > 0:
            next_move = possible_moves.pop()
            state.push(next_move)
            next_value = self.H_L(state, l - 1)
            state.pop()
            if next_value > best_value and state.turn:
                best_value = next_value
                best_move = next_move
            if next_value < best_value and not state.turn:
                best_value = next_value
                best_move = next_move
        self.move = best_move
        return best_value
```

```python
else:
    if state.turn:
        return -10000
    else:
        return 10000
```

Heuristic: ALPHA BETA PRUNING

Everything needed for heuristic is imported. The weight of the pieces of the board is then defined.

```
"""
    Heuristic 2
"""
import random
from typing import Any

from chess import Board


# weight of piece
def weight(piece):
    if piece == 6:
        return 9999
    if piece == 5:
        return 9
    elif piece == 4:
        return 5
    elif piece == 3 or piece == 2:
        return 3
    return 1
```

The MyHeuristic2 class is defined (later recalled in the main) and two functions are defined inside it:

- ***H_0: which defines the evaluation metric (it is an evaluation function of the configuration situation in the game. It defines how good a given setup is for the player);***
- ***H_L: which defines the alpha beta pruning algorithm. Alpha-beta pruning is a search algorithm that seeks to reduce the number of nodes evaluated by the minimax algorithm in its search tree. It stops evaluating a move when at least one possibility has been found showing that the move is worse than a previously examined move.***

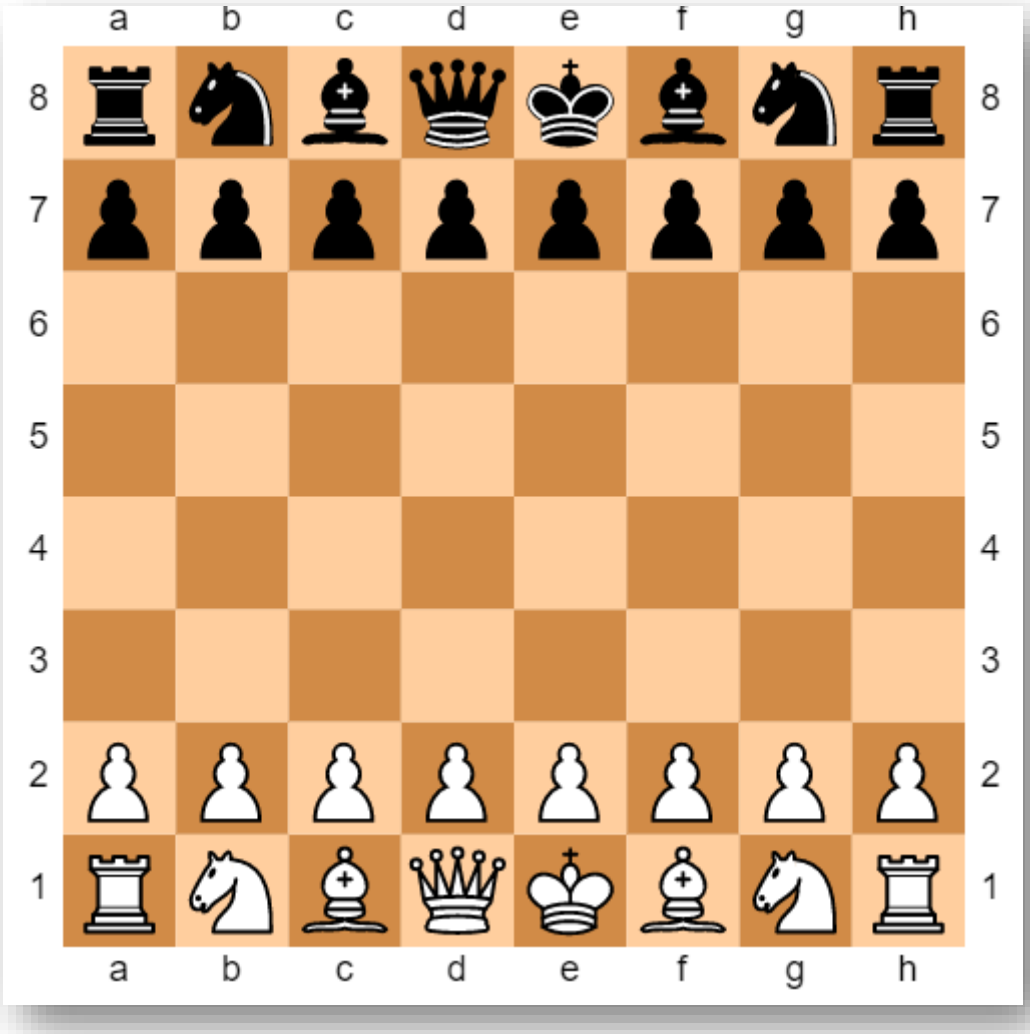
```
class MyHeuristic2:
    move: Any

    def __init__(self):
        pass

    def __repr__(self):
        pass

    # metric of evaluation
    def H_0(self, state: Board):
        pieces = state.piece_map()
        ret = 0
        for index in pieces:
            if pieces[index].color:
                ret += weight(pieces[index].piece_type)
            else:
                ret -= weight(pieces[index].piece_type)
        return ret
```

```python
# alpha beta pruning
def H_L(self, state: Board, l: int, alpha = None):
    if l == 0:
        return self.H_0(state)
    possible_moves = list(state.legal_moves)
    random.shuffle(possible_moves)
    if len(possible_moves) > 0:
        best_move = possible_moves.pop()
        state.push(best_move)
        best_value = self.H_L(state, l - 1, alpha)
        state.pop()
        while len(possible_moves) > 0:
            next_move = possible_moves.pop()
            state.push(next_move)
            next_value = self.H_L(state, l - 1, alpha)
            state.pop()
            if next_value > best_value and state.turn:
                best_value = next_value
                best_move = next_move
            if next_value < best_value and not state.turn:
                best_value = next_value
                best_move = next_move
```

```python
        if not alpha is None:
            if best_value < alpha and state.turn:
                break
            if best_value > alpha and not state.turn:
                break
    self.move = best_move
    return best_value
else:
    if state.turn:
        return -999999
    else:
        return 999999
```

RESULT

The defined board is represented in this way with all the chess pieces.



The two artificial intelligences start making their own moves and eventually you can find yourself in a stalemate or checkmate. In this case, black has put the white king in checkmate.

Statistics:

The time taken for the game to end since the program is run is 00:10:89.

On 9 games played, we can see how black through the minmax algorithm always wins by putting the white king using the alpha beta pruning algorithm in checkmate:

Color	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Octave	Ninth
W	Beaten	Beaten	Beaten	Beaten	Beaten	Beaten	Beaten	Beaten	Beaten
B	Winner	Winner	Winner	Winner	Winner	Winner	Winner	Winner	Winner

In fact, if the two algorithms are exchanged, we can see how white is able to prevail over black using the minmax algorithm.