Report Homework2: Chess Game

I used the Python chess library for this exercise 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.

```
"""
    This is the main driver file. It will be responsible for displaying the object.

    NOTES IMPORTANT: When the program is running, hover the mouse over the board
    because it sometimes crashes!!!!!
"""

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

from minmax_heuristic.MinMaxBest import MyBestHeuristic
from minmax_heuristic.PredictiveMinMax import MyPredictiveHeuristic

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 of loading the images on 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 chessboard 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 minimax best and predictive minimax 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()
    best_minmax = MyBestHeuristic()
    predictive_minmax = MyPredictiveHeuristic()
```

```
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:
                predictive_minmax.H_L(board, 1)
                board.push(predictive_minmax.move)
            else:
                best_minmax.H_L(board, 2)
                board.push(best_minmax.move)
            gameOver = board.is_game_over()
        drawGameState(screen, make_chess_matrix(board))
```

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

Heuristic: MINMAX

Everything needed for the 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 minimax 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: MIMIMAX BEST

For step 1, I defined and implemented a different possible variation on the αβMinMaxL lookup which speeds up the computation.

For point a of step 1, I created the MyBestHeuristic where only the subtrees of the most promising nodes are explored (i.e. the best k nodes according to the H_L evaluation, where L can go from 0 to L, obviously the bigger the better involves , but the lower the acceleration). Experimenting with some small values on L to set it to a suitable value, I decided to put L=4.

All the libraries you need are imported.

```
import random
from typing import Any

import chess
from chess import Board

import minmax_heuristic
from minmax_heuristic import weight
```

We define the heuristic class and a variable called move. We define the __init__ and __repr__ methods and the method that defines the evaluation metric.

```
class MyBestHeuristic:
    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
```

The H_L method for the best minimax algorithm is defined. We then look for the best subnodes, simulating the move within the list, calling the minimax algorithm function, assigning the value to the move, and putting the subnode and its value into an array called possible_moves. Then sorting is done in the list.

Checks if the length of the possible_moves array is greater than 0 and if so, the checks defined in the code visit the best sub-nodes. Finally, the found values (subnode and subnode value) are assigned to the variable best_move.

The check is carried out if L = 4 (since the best value of L has been chosen to have the best behavior and the best acceleration) and if it is different from the indicated values (10000 and -10000). If control passes then the board and its value are written using the write function, defined later, in a .csv file called "ts".

This last procedure is also applied to the next_move as you can see in the code below.

Below is the code that matches the description.

```
# minimax best (best subnodes)
def H_L(self, state: Board, l: int):
    if l == 0:
        return self.H_0(state)
    possible_moves = []
    neighbors = list(state.legal_moves)
    random.shuffle(neighbors)
    for neighbor in neighbors:
        # move simulation
        state.push(neighbor)
        minmax = minmax_heuristic.MyHeuristic()
        # assigning value to the move
        neighbor_value = minmax.H_L(state, 1)
        possible_moves.append((neighbor, neighbor_value))
        state.pop()
    # sorting
    possible_moves = sorted(possible_moves, key=lambda x: (x[1]))
    filtered_possible_moves = []
```

```
if len(possible_moves) > 0:
    if state.turn:
        for i in range(0, 2):
            if len(possible_moves) > 0:
                filtered_possible_moves.append(possible_moves.pop())
    else:
        filtered_possible_moves = possible_moves[0:2]
    possible_moves = filtered_possible_moves
    best_move = possible_moves.pop()[0]
    state.push(best_move)
    best_value = self.H_L(state, l - 1)
    if l == 4 and best_value != 10000 and best_value != -10000:
        MyBestHeuristic.write(state, best_value)
    state.pop()
```

```
    while len(possible_moves) > 0:
        next_move = possible_moves.pop()[0]
        state.push(next_move)
        next_value = self.H_L(state, l - 1)
        if l == 4 and next_value != 10000 and next_value != -10000:
            MyBestHeuristic.write(state, next_value)
        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
else:
    if state.turn:
        return -10000
    else:
        return 10000
```

The method below is used to have an array of integer values inside the csv file. In fact, it is called from the list variable in the write method.

```
@staticmethod
def matrix_to_int(board):
    l = [None] * 64
    for sq in chess.scan_reversed(board.occupied_co[chess.WHITE]):
        l[sq] = board.piece_type_at(sq)
    for sq in chess.scan_reversed(board.occupied_co[chess.BLACK]):
        l[sq] = -board.piece_type_at(sq)
    # l.append(int(board.turn))
    return [0 if v is None else v for v in l]
```

The write method is defined to create a .csv file named ts and write values into it. This file will contain the representation of a board with its value.

```
@staticmethod
def write(board, value):
    f = open("./ts.csv", "a")
    list = MyBestHeuristic.matrix_to_int(board)
    f.write("[")
    for item in list:
        f.write(f"{item} ")
    f.write("],")
    f.write(f"{value}\n")
    f.close()
```

Heuristic: PREDICTIVE MIMIMAX

For step 2, I used the version of MinMax speed-up that was developed in step 1 to create a training set for learning H_L evaluation of states.

For the first point of step 2, I created, inside the PredictiveMinMax file, a Model class, and the MyPredictiveHeuristic heuristic.

First, you have the libraries that will be used for the development of the code imported.

```
import random

from sklearn.linear_model import LinearRegression
from typing import Any

import pandas as pandas
from chess import Board

from minmax_heuristic.MinMaxBest import MyBestHeuristic
```

Subsequently, the model class is defined with a variable called model and two methods inside:

- __init__, where inside the pandas library the ts.csv file is opened and read and two arrays are defined. The first (X) is for the board representation and its column is called "data". The second (y) is for its value and its column is called "target". Finally, a linear representation is given to the model;
- predict, where it has a variable x which calls the matrix_to_int() method to get the integer values of the matrix (of the board) and returns the integer value of the model which calls the predict method on the variable x.

```
class Model:
    model: Any

    def __init__(self):
        ds = pandas.read_csv("./ts.csv")
        X = []
        for single in ds["data"]:
            x = single.replace("[", "").replace("]", "").replace(" ", ",").split(",")
            x.pop()
            X.append([int(numeric_string) for numeric_string in x])
        y = []
        for single in ds["target"]:
            y.append(single)
        self.model = LinearRegression()
        self.model.fit(X, y)

    def predict(self, board):
        x = [MyBestHeuristic.matrix_to_int(board)]
        return int(self.model.predict(x))
```

Subsequently, the MyPredictiveHeuristic class was defined where two variables (move and model) are defined, the methods __init__, __repr__, and H_0 for the evaluation metric of the model which calls the predict method on the board.

Finally, the H_L method is defined which contains the minimax algorithm defined on pages 9-11 of this report, and the .csv file has been populated.

```
class MyPredictiveHeuristic:
    move: Any
    model = Model()

    def __init__(self):
        pass

    def __repr__(self):
        pass

    # metric of evaluation
    def H_0(self, state: Board):
        return self.model.predict(state)
```

```
# 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
```

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

For the second point of step 2, I evaluated the performance of the $_{predictiveL}$MinMax1 against the increase rate $MinMax_L$ and against the increase rate $MinMax_{L/2}$.

In 9 games $_{predictiveL}$MinMax1 was set with L = 1 and $MinMax_L$ with L = 4. In the other 9 games, $_{predictiveL}$MinMax1 was set with L = 1 and $MinMax_{L/2}$ with L = 2.

For the first table, I used $_{predictiveL}$MinMax1 with L=1 for the White while I used $MinMax_L$ with L=4 for the Black. Furthermore, with these values of L, it can be seen how slow the game time is.

The time taken for the game from when the program is executed, the board with the pieces created and loaded, the moves of the pieces defined, and the game finished is about 00:54.22 for the games played, so let's say a minute. While the time it takes to get one of the two kings into check or checkmate with the board already created and the pieces already loaded is about 00:32.42, so let's say thirty/thirty-five seconds. OBVIOUSLY, THE PLAYING TIME VARIES ACCORDING TO THE MOVES OF THE GAME THAT THE TWO ARTIFICIAL INTELLIGENCES ARE PLAYING. HOWEVER, YOU CAN NOTE HOW IN SOME CASES THE BATCH IS FASTER THAN OTHERS DEPENDING ON THE DEPTH (THEREFORE L).

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

For the second table, I used $_{predictiveL}$MinMax1 with L=1 for the White while I used $MinMax_{L/2}$ with L=2 for the Black. Furthermore, with these values of L, it can be seen that the game time is faster for $MinMax_{L/2}$ with L=2.

The time taken for the game from when the program is executed, the board with the pieces created and loaded, the moves of the pieces defined, and the game finished is about 00:17.72 for the games played, so let's say twenty/twenty-five seconds. While the time it takes to get one of the two kings into check or checkmate with the board already created and the pieces already loaded is about 00:07.63, so let's say five/ten seconds. OBVIOUSLY, THE PLAYING TIME VARIES ACCORDING TO THE MOVES OF THE GAME THAT THE TWO ARTIFICIAL INTELLIGENCES ARE PLAYING. HOWEVER, YOU CAN NOTE HOW IN SOME CASES THE BATCH IS FASTER THAN OTHERS DEPENDING ON THE DEPTH (THEREFORE L).

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

For the third point of step 2, I evaluated the performance of the predictive $_{predictiveL}$MinMaxL (using the prediction as a static evaluation, so L=4) against the $_{speed\text{-}up}$MinMax$_{L}$ and against the $_{speed\text{-}up}$MinMax$_{L/2}$.

In 9 games $_{predictiveL}$MinMaxL was set with L = 2 and $_{speed\text{-}up}$MinMax$_{L}$ with L = 2. In the other 9 games, $_{predictiveL}$MinMaxL was set with L = 2 and $_{speed\text{-}up}$MinMax$_{L/2}$ with L = 1.

For the first table, I used $_{predictiveL}$MinMaxL with L=2 for the White while I used MinMax$_{L}$ with L=2 for the Black. The predictive in this case win a few times.

The time taken for the game from when the program is executed, the board with the pieces created and loaded, the moves of the pieces defined and the game finished is about 00:40.70 for the games played, so let's say forty/forty-five seconds. While the time it takes to get one of the two kings into check or checkmate with the board already created and the pieces already loaded is about 00:16.59, so let's say fifteen/twenty seconds. OBVIOUSLY, THE PLAYING TIME VARIES ACCORDING TO THE MOVES OF THE GAME THAT THE TWO ARTIFICIAL INTELLIGENCES ARE PLAYING. HOWEVER, YOU CAN NOTE HOW IN SOME CASES THE BATCH IS FASTER THAN OTHERS DEPENDING ON THE DEPTH (THEREFORE L).

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

For the second table, I used $_{predictiveL}$MinMaxL with L=2 for the White while I used MinMax$_{L/2}$ with L=1 for the Black. Also, in this case, the predictive wins several times against the minimax best.

The time taken for the game from when the program is executed, the board with the pieces created and loaded, the moves of the pieces defined, and the game finished is about 00:16.89 for the games played, so let's say fifteen/twenty seconds. While the time it takes to get one of the two kings into check or checkmate with the board already created and the pieces already loaded is about 00:08.43, so let's say five/ten seconds. OBVIOUSLY, THE PLAYING TIME VARIES ACCORDING TO THE MOVES OF THE GAME THAT THE TWO ARTIFICIAL INTELLIGENCES ARE PLAYING. HOWEVER, YOU CAN NOTE HOW IN SOME CASES THE BATCH IS FASTER THAN OTHERS DEPENDING ON THE DEPTH (THEREFORE L).

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

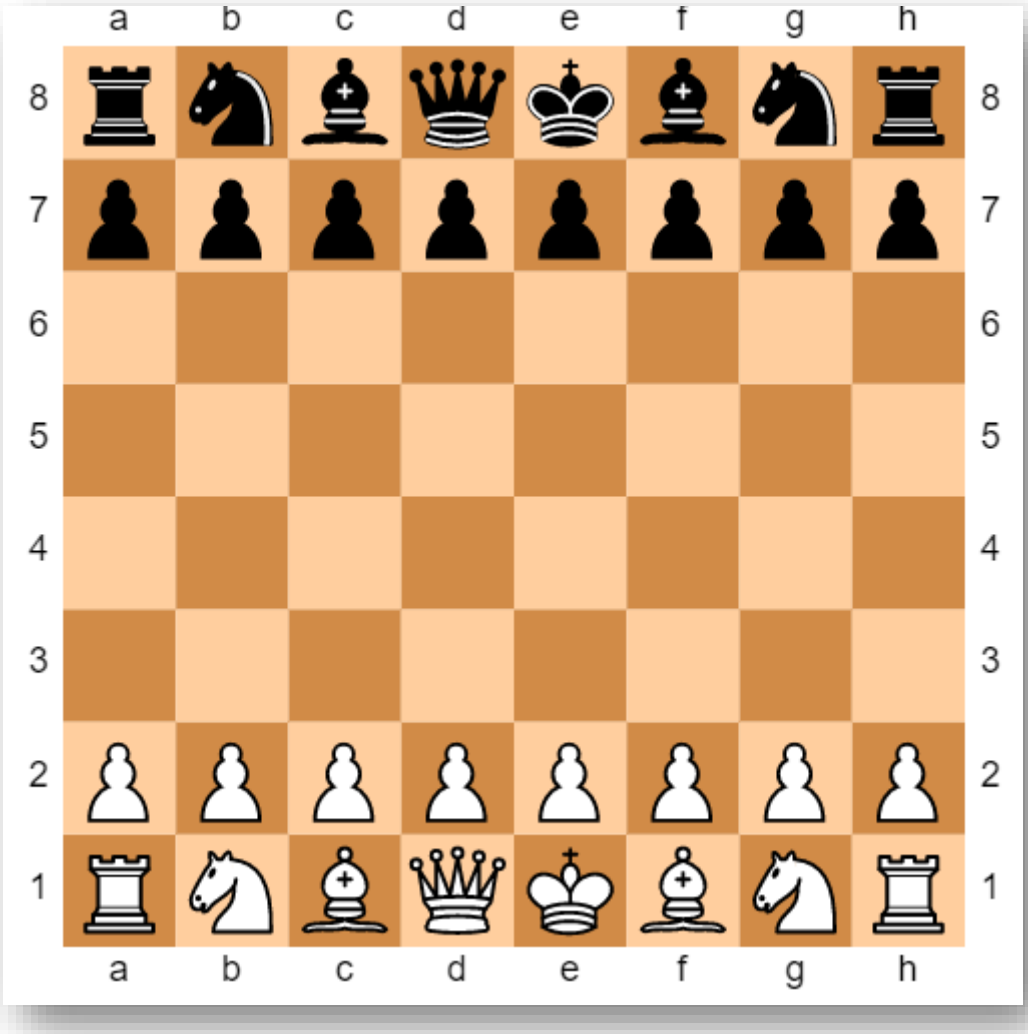
FILE TS.CSV

This is the file .csv with a few lines of data. Inside this file are a match of the board and its value.

```
data,target
[4 2 3 5 6 3 2 4 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 -1 -1 -
[4 0 3 5 6 3 2 4 1 1 1 1 1 1 1 1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 -1 -1 -
[4 2 3 5 0 3 2 4 1 1 1 1 1 6 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -2 0 0 -1 -1 -1
[4 0 3 5 6 3 2 4 1 1 1 1 1 0 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -2 0 0 -1 -1 -1
[4 2 3 5 6 3 2 4 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 -2 0 0 -1 -1 0
[4 2 3 5 0 3 2 4 1 1 1 1 1 0 1 1 0 0 0 0 0 6 0 0 0 0 0 0 0 1 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 -2 0 0 -1 -1 0
[4 2 3 5 6 3 2 4 1 1 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 -1 0 0 0 0 0 -2 0 0 0 0 -2 0 0 -1 -1 0
[4 2 3 5 0 3 2 4 1 1 1 1 1 6 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 -1 0 0 0 0 0 -2 0 0 0 0 -2 0 0 -1 -1 0
[4 2 3 5 6 3 2 4 1 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 -1 0 0 0 0 0 -2 0 0 0 -1 -2 0 0 -1 -1
[4 0 3 5 6 3 2 4 1 1 0 1 1 0 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 -1 0 0 0 0 0 -2 0 0 0 -1 -2 0 0 -1 -1
```

RESULT

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



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