In [1]:

```python
import chess
import random
from IPython.display import clear_output
import time
```

In [2]:

```python
#
# Agent abstract class
# Given a game state, the agent chooses a move between a set of legal moves
#

class Agent():
    """Agent class"""

    def __init__(self, program=None):
        self.program = program

    def find_best(self, game):
        """Given a game state, return the best move"""
        raise NotImplementedError
```

In [3]:

```python
#
# Game abstract class
# This class abstracts the concept of a game
# Implement the methods to define a real game
#
class Game():
    """Game class"""

    def __init__(self, initialState=None):
        self.state = initialState

    def neighbors(self, state):
        """Return the set of reachable states"""
        raise NotImplementedError

    def result(self, state, move):
        """Return the state produced by a move"""
        raise NotImplementedError

    def getState(self):
        """Return the current state"""
        return self.state

    def is_terminal(self, state):
        """Return true if this is a final state"""
        raise NotImplementedError

    def make_move(self, move):
        """Make a move changing the state"""
        raise NotImplementedError
```

In [4]:

```python
#
# ChessGame class
# This class overrides Game and allow to play chess
# The internal state is provided by the python-chess library
#
class ChessGame(Game):
    """Game implementation for chess"""

    def __init__(self):
        self.state = chess.Board() # initial board

    def neighbors(self, state):
        reachableStates = []
        for s in state.legal_moves:
            reachableState = self.result(state, s)
            reachableStates.append(reachableState)
        return reachableStates

    def result(self, state, move):
        # apply the given move producing a new board
        resultState = state.copy()
        resultState.push(move)
        return resultState

    def utility(self, state):
        # true if game ended, false otherwise
        outcome = state.outcome();
        if (outcome != None):
            return True;
        else:
            return False

    def is_terminal(self, state):
        return self.utility(state)

    def make_move(self, move):
        self.state.push(move)
```

In [5]:

```
#
# RandomAgent class
#
class RandomAgent(Agent):
    """An Agent that chooses a move randomly"""

    def find_best(self, game):

        possible_moves = list(game.getState().legal_moves)

        move = random.choice(possible_moves)

        print("Agent {} choosed {} as best move".format(game.getState().turn, move))
        return move
```

In [6]:

```
# PIECES VALUE
PAWN = 10
KNIGHT = 30
BISHOP = 30
ROOK = 50
QUEEN = 90
KING = 900
```

In [7]:

```
# POSITION VALUES
pawnEvalBlack = [
        0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
        5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,
        1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0,
        0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5,
        0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0,
        0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5,
        0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5,
        0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0
    ];

pawnEvalWhite = pawnEvalBlack
pawnEvalWhite.reverse()

knightEval = [
        -5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0,
        -4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0,
        -3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0,
        -3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0,
        -3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0,
        -3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0,
        -4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0,
        -5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0
    ];

bishopEvalBlack = [
     -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0,
     -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0,
     -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0,
     -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0,
     -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0,
     -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0,
     -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0,
     -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0
];

bishopEvalWhite = bishopEvalBlack.copy()
bishopEvalWhite.reverse()

rookEvalBlack = [
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5,
     -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5,
     -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5,
     -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5,
     -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5,
     -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5,
      0.0,   0.0, 0.0,  0.5,  0.5,  0.0,  0.0,  0.0
];

rookEvalWhite = rookEvalBlack.copy()
rookEvalWhite.reverse()

queenEval = [
     -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0,
     -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0,
     -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0,
     -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5,
      0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5,
     -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0,
     -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0,
     -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0
];

kingEvalBlack = [

     -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0,
     -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0,
     -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0,
     -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0,
     -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0,
     -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0,
      2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ,
      2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0
];

kingEvalWhite = kingEvalBlack.copy()
kingEvalWhite.reverse()
```

In [8]:

```
#
# GreedyAgent class
#
class GreedyAgent(Agent):
    """An Agent that chooses the best state in the neighborhood"""

    def find_best(self, game):
        states = game.neighbors(game.getState());

        best_value = -9999
        best_state = states[0]

        for s in states:
            current = self.evaluation(s)

            if current > best_value:
                best_value = current
                best_state = s

        print("Best move value:", best_value)
        return best_state.peek()



    def evaluation(self, state):
        value_black = 0;
        value_white = 0;

        pieces = state.piece_map()
        for p in pieces:
            if (state.piece_at(p).piece_type  == 1 and state.piece_at(p).color):
                value_white += PAWN + ( pawnEvalWhite[p] )
            elif (state.piece_at(p).piece_type == 2 and state.piece_at(p).color):
                value_white += KNIGHT + ( knightEval[p] )
            elif (state.piece_at(p).piece_type == 3 and state.piece_at(p).color):
                value_white += BISHOP + ( bishopEvalWhite[p] )
            elif (state.piece_at(p).piece_type == 4 and state.piece_at(p).color):
                value_white += ROOK + ( rookEvalWhite[p] )
            elif (state.piece_at(p).piece_type == 5 and state.piece_at(p).color):
                value_white += QUEEN + ( queenEval[p] )
            elif (state.piece_at(p).piece_type == 6 and state.piece_at(p).color):
                value_white += KING + ( kingEvalWhite[p] )
            elif (state.piece_at(p).piece_type == 1 and (not state.piece_at(p).color)):
                value_black += PAWN + ( pawnEvalBlack[p] )
            elif (state.piece_at(p).piece_type == 2 and (not state.piece_at(p).color)):
                value_black += KNIGHT + ( knightEval[p] )
            elif (state.piece_at(p).piece_type == 3 and (not state.piece_at(p).color)):
                value_black += BISHOP + ( bishopEvalBlack[p] )
            elif (state.piece_at(p).piece_type == 4 and (not state.piece_at(p).color)):
                value_black += ROOK + ( rookEvalBlack[p] )
            elif (state.piece_at(p).piece_type == 5 and (not state.piece_at(p).color)):
                value_black += QUEEN + ( queenEval[p] )
            elif (state.piece_at(p).piece_type == 6 and (not state.piece_at(p).color)):
                value_black += KING + ( kingEvalBlack[p] )

        if (not state.turn):
            return value_white - value_black
        else:
            return value_black - value_white
```

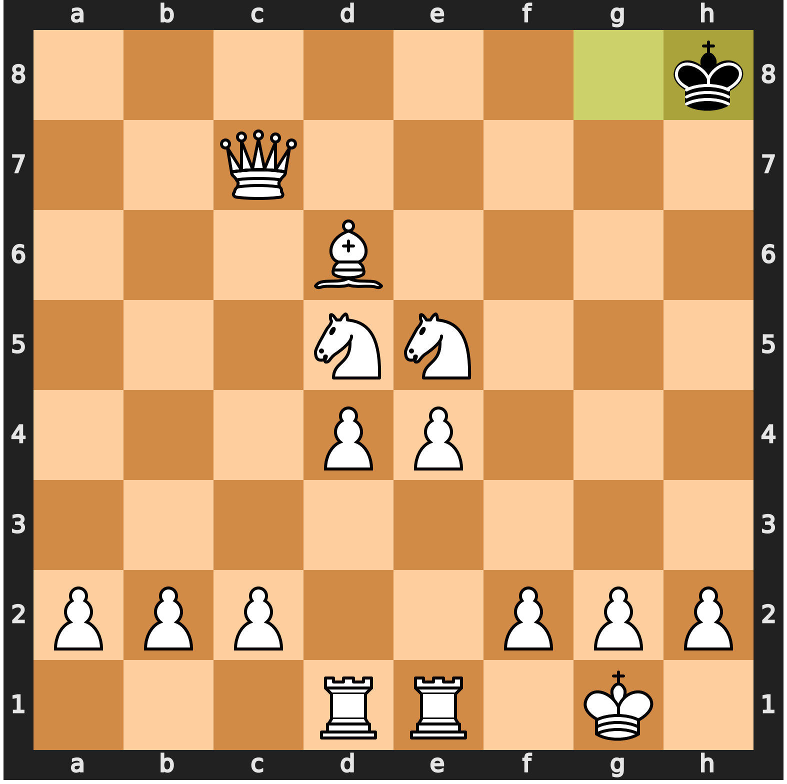
In [17]:

```
#
# MinMaxAgent class
#
class MinMaxAgent(Agent):
    """An Agent that uses minmax to evaluate the best move"""

    def __init__(self, level):
        self.level = level

    def find_best(self, game):
        states = game.neighbors(game.getState());
        mx = -9999
        ix = 0

        for s in states:
            h = self.H1(game, game.getState(), self.level)

            if h > mx:
                print("Best option found: ", s.peek())
                mx = h
                ix = s
        print("Agent {} choosed {} as best move with a score of {}".format(game.getState()

        return ix.peek()


    def H0(self, state):
        value_black = 0;
        value_white = 0;

        pieces = state.piece_map()
        for p in pieces:
            if (state.piece_at(p).piece_type  == 1 and state.piece_at(p).color):
                value_white += PAWN + ( pawnEvalWhite[p] )
            elif (state.piece_at(p).piece_type == 2 and state.piece_at(p).color):
                value_white += KNIGHT + ( knightEval[p] )
            elif (state.piece_at(p).piece_type == 3 and state.piece_at(p).color):
                value_white += BISHOP + ( bishopEvalWhite[p] )
            elif (state.piece_at(p).piece_type == 4 and state.piece_at(p).color):
                value_white += ROOK + ( rookEvalWhite[p] )
            elif (state.piece_at(p).piece_type == 5 and state.piece_at(p).color):
                value_white += QUEEN + ( queenEval[p] )
            elif (state.piece_at(p).piece_type == 6 and state.piece_at(p).color):
                value_white += KING + ( kingEvalWhite[p] )
            elif (state.piece_at(p).piece_type == 1 and (not state.piece_at(p).color)):
                value_black += PAWN + ( pawnEvalBlack[p] )
            elif (state.piece_at(p).piece_type == 2 and (not state.piece_at(p).color)):
                value_black += KNIGHT + ( knightEval[p] )
            elif (state.piece_at(p).piece_type == 3 and (not state.piece_at(p).color)):
                value_black += BISHOP + ( bishopEvalBlack[p] )
            elif (state.piece_at(p).piece_type == 4 and (not state.piece_at(p).color)):
                value_black += ROOK + ( rookEvalBlack[p] )
            elif (state.piece_at(p).piece_type == 5 and (not state.piece_at(p).color)):
                value_black += QUEEN + ( queenEval[p] )
            elif (state.piece_at(p).piece_type == 6 and (not state.piece_at(p).color)):
                value_black += KING + ( kingEvalBlack[p] )

        if (not state.turn):
            return value_white - value_black
        else:
            return value_black - value_white


    def Hl(self, game, state, l):
        if (l == 0):
            return self.H0(state)
        if (state.turn):
            return max([self.Hl(game, x, l-1) for x in game.neighbors(state)])
        else:
            return min([self.Hl(game, x, l-1) for x in game.neighbors(state)])
```

In [18]:

```
#
# Main
# The following lines of code generate an instance of a ChessGame
# You can choose the provided agents to see different results
#

game = ChessGame()
agent1 = GreedyAgent()
agent2 = GreedyAgent()
agent3 = MinMaxAgent(0)

display(game.getState())

while True:
    clear_output(wait=True)


    if (game.getState().turn):
        move = agent1.find_best(game)
    else:
        move = agent3.find_best(game)

    game.make_move(move)

    display(game.getState())

    time.sleep(0.5)

    if (game.is_terminal(game.getState())):
        display(game.getState().outcome())
        break
```

```
Best option found:  g8h8
Agent False choosed g8h8 as best move with a score of 375.5
```



```
Outcome(termination=<Termination.FIVEFOLD_REPETITION: 5>, winner=None)
```

In [ ]: