Proiect Inteligență artificială

Joc de Șah cu AI

Student:

Galiș oliver-Valentin

Îndrumator:

Moldovan Cristian

1. Introducere:

Scopul proiectului este de a realiza o aplicație de tip joc de șah, care să permită interacțiunea dintre un jucător și o entitate artificială (AI) capabilă să calculeze mutările optime. Proiectul este implementat în limbajul Python, utilizând biblioteca Pygame pentru realizarea interfeței grafice.

Aceasta aplicație urmărește sa simuleze jocul de șah, oferind totodată o demonstrație practica a modului in care principiile de inteligentă artificială pot fi aplicate in jocurile de strategie.

2. Obiectivele proiectului:

- Implementarea completă a regulilor jocului de șah(mutări valide, promovare, en-passant, rocadă, etc.);
- Crearea unei interfețe grafice intuitive și dinamice;
- Dezvoltarea unei componente AI bazate pe evaluarea poziționalăși pe căutarea în arborele de mutări;
- Realizarea unui sistem modular, compus din trei fisiere Python independente, interconectate.

3. Descrierea modulelor

3.1 Modulul ChessEngin

Modulul reprezintă componenta principala a aplicației, funcționează ca un motor logic pentru aplicație, fiind responsabil pentru reprezentarea stării jocului de șah și pentru aplicarea regulului:

- Mutările valide ale fiecărei piese;
- Verificarea situațiilor de șah și șah mat;
- Implementarea mutărilor speciale(rocada, en-passant, promovare);
- Gestionarea istoricului mutărilor.

3.1.1 Reprezentarea tablei de joc

Tabla de joc este reprezentata printr-o matrice bidimensionala(lista de liste) 8x8, unde fiecare element conține o piesa sau un spațiu liber.

```
self.board = [
        ["bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"],
        ["bP", "bP", "bP", "bP", "bP", "bP", "bP", "bP"],
        ["--", "--", "--", "--", "--", "--", "--", "--"],
        ["--", "--", "--", "--", "--", "--", "--", "--"],
        ["--", "--", "--", "--", "--", "--", "--", "--"],
        ["--", "--", "--", "--", "--", "--", "--", "--"],
        ["wP", "wP", "wP", "wP", "wP", "wP", "wP", "wP"],
        ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"]
]
```

Fiecare cod de piesa este format din doua caractere:

- Primul caracter reprezintă culoarea piesei, ‘w’ pentru alb sau ‘b’ pentru negru;
- Al doilea caracter reprezintă tipul piesei, ‘P’-pion, ‘R’-tură, ‘N’-cal, ‘B’-nebun, ‘K’-rege, ‘Q’-regină.

3.1.2 Execuția unei mutări

Mutarea unei piese este rezultată de metoda MakeMove() care are următoarele caracteristicile:

- Golește pătratul de start;
- Plasează piesa in poziția de final;
- Salvează mutarea in istoricul mutărilor;
- Schimba jucătorului activ.

Funcția are tratamente suplimentare pentru mutările speciale cum ar fi :

- Promovarea pionilor
- Mutările de tip en-passant;
- Rocada.

```
def makeMove(self, move):
        self.board[move.startRow][move.startCol] = "--"
        self.board[move.endRow][move.endCol] = move.pieceMoved
        self.moveLog.append(move) # log the move so we can undo it later
        self.whiteToMove = not self.whiteToMove # swap players

        # update the king's location if moved
        if move.pieceMoved == 'wK':
            self.whiteKingLocation = (move.endRow, move.endCol)
        elif move.pieceMoved == 'bK':
            self.blackKingLocation = (move.endRow, move.endCol)

        # pawn promotion
        if move.isPawnPromotion:
            promotedPiece = 'Q'
            self.board[move.endRow][move.endCol] =
move.pieceMoved[0] + promotedPiece

        # en passant
        if move.isEnpassantMove:
            self.board[move.startRow][move.endCol] = "--"                    #capturing the pawn
        # update enPassantPossible variable
```

```
        if move.pieceMoved[1] == 'P' and abs(move.startRow -   move.endRow) == 2: # only on 2
square pawn advances
            self.enPassantPossible = ((move.startRow +
move.endRow)//2, move.startCol)
        else:
            self.enPassantPossible = () # reset enPassantPossible if not a 2 square pawn advance

        # castle move
        if move.isCastleMove:
            if move.endCol - move.startCol == 2: # KingSide castle
                self.board[move.endRow][move.endCol - 1] =
self.board[move.endRow][move.endCol + 1]
                self.board[move.endRow][move.endCol + 1] = "--"
        else: # QueenSide castle
            self.board[move.endRow][move.endCol + 1] =
self.board[move.endRow][move.endCol - 2]
            self.board[move.endRow][move.endCol - 2] = "--"

        self.enPassantPossibleLog.append(self.enPassantPossible)
```

3.1.3 Anularea unei mutari

Funcția undoMove() permite inversarea efectului unei mutări cea ce este esențial pentru algoritmul AI, deoarece acesta testează virtual mutări și trebuie să poată reveni la starea inițiala după fiecare simulare.

```
def undoMove(self):
        if len(self.moveLog) != 0: # make sure that there is a move to undo
            move = self.moveLog.pop()
            self.board[move.startRow][move.startCol] = move.pieceMoved
            self.board[move.endRow][move.endCol] =
move.pieceCaptured
            self.whiteToMove = not self.whiteToMove # swap players back

        # update the king's location if moved
        if move.pieceMoved == 'wK':
            self.whiteKingLocation=(move.startRow,move.startCol)
        elif move.pieceMoved == 'bK':
            self.blackKingLocation=(move.startRow,move.startCol)
        # undo en passant
        if move.isEnpassantMove:
            self.board[move.endRow][move.endCol] = "--" # leave landing square blank
            self.board[move.startRow][move.endCol]= move.pieceCaptured
            self.enPassantPossibleLog.pop()
            self.enPassantPossible=self.enPassantPossibleLog[-1]
        # undo castle move
        if move.isCastleMove:
```

```
            if move.endCol - move.startCol == 2: # KingSide
                self.board[move.endRow][move.endCol+1] = self.board[move.endRow][move.endCol-
1]
                self.board[move.endRow][move.endCol-1] = "--"
            else: # QueenSide
        self.board[move.endRow][move.endCol - 2] =        self.board[move.endRow][move.endCol+1]
                self.board[move.endRow][move.endCol+1] = "--"

        # reset the checkmate and stalemate flags
        self.checkMate = False
        self.staleMate = False
```

3.1.4 Determinarea mutărilor valide

Funcția getValidMoves() returnează toate mutările permise în poziția curenta, respectând regulile jocului și asigurându-se ca regele propriu nu rămâne in șah daca acesta este deja in șah.

Pentru fiecare mutare se parcurge următoare logica:

- Se aplica mutarea;
- Se verifica daca regele propriu este in șah;
    - Daca regele este in șah mutarea este eliminata si se revine la poziția inițiala;
    - Daca regele nu sete in șah mutarea este validata si se poate schimba tura către celălalt jucător.

```
def getValidMoves(self):
        tempEnPassantPossible = self.enPassantPossible
        tempCastleRights = CastleRights(self.currentCastlingRights.wks,
        self.currentCastlingRights.bks,
self.currentCastlingRights.wqs, self.currentCastlingRights.bqs)
        # Generate all possible moves
        moves = self.getAllPossibleMoves()
        if self.whiteToMove:
            self.getCastleMoves(self.whiteKingLocation[0], self.whiteKingLocation[1], moves)
        else:
            self.getCastleMoves(self.blackKingLocation[0], self.blackKingLocation[1], moves)
        # For each move, make the move
        for i in range(len(moves)-1, -1, -1): # when removing from a list, go backwards
            self.makeMove(moves[i])
        # Generate all opponent's moves
        # For each of opponent's moves, see if they attack your king
            self.whiteToMove = not self.whiteToMove
            if self.inCheck():
                moves.remove(moves[i]) # If they do attack your king, not a valid move
                self.whiteToMove = not self.whiteToMove
```

```
                self.undoMove()

            if len(moves) == 0: # either checkmate or stalemate
                if self.inCheck():
                    self.checkMate = True
                else:
                    self.staleMate = True
            else:
                self.checkMate = False
                self.staleMate = False

            self.enPassantPossible = tempEnPassantPossible
            self.currentCastlingRights = tempCastleRights

        return moves
```

3.1.5 Generarea mutărilor pentru fiecare tip de piesa

Exemplele de generare a mutărilor posibile pentru piese, care ține cont de regulile jocului

- Aceasta funcție generează mutările posibile pentru pioni și tine cont de marginile tablei, regulile mutări de tip en-passant, promovarea și prezența pieselor adversare sau proprii din jur:

```
def getPawnMoves(self, row, col, moves):
        if self.whiteToMove: # white pawn moves
            if self.board[row-1][col] == "--": # 1 square move
                moves.append(Move((row, col), (row-1, col), self.board))
                if row == 6 and self.board[row-2][col] == "--": # 2 square move
                    moves.append(Move((row, col), (row-2, col), self.board))
            if col - 1 >= 0: # captures to the left
                if self.board[row-1][col-1][0] == 'b': # enemy piece to capture
                    moves.append(Move((row, col), (row-1, col-1), self.board))
                elif (row-1, col-1) == self.enPassantPossible:
                    moves.append(Move((row, col), (row-1, col-1), self.board, isEnpassantPossible =
True))
            if col + 1 <= 7: # captures to the right
                if self.board[row-1][col+1][0] == 'b': # enemy piece to capture
                    moves.append(Move((row, col), (row-1, col+1), self.board))
                elif (row-1, col+1) == self.enPassantPossible:
                    moves.append(Move((row, col), (row-1, col+1), self.board, isEnpassantPossible =
True))
        else: # black pawn moves
            if self.board[row+1][col] == "--": # 1 square move
                moves.append(Move((row, col), (row+1, col), self.board))
```

```
                if row == 1 and self.board[row+2][col] == "--": # 2 square move
                    moves.append(Move((row, col), (row+2, col), self.board))
            if col - 1 >= 0: # captures to the left
                if self.board[row+1][col-1][0] == 'w': # enemy piece to capture
                    moves.append(Move((row, col), (row+1, col-1), self.board))
                elif (row+1, col-1) == self.enPassantPossible:
                    moves.append(Move((row, col), (row+1, col-1), self.board, isEnpassantPossible =
True))
            if col + 1 <= 7: # captures to the right
                if self.board[row+1][col+1][0] == 'w': # enemy piece to capture
                    moves.append(Move((row, col), (row+1, col+1), self.board))
                elif (row+1, col+1) == self.enPassantPossible:
                    moves.append(Move((row, col), (row+1, col+1), self.board, isEnpassantPossible =
True))
```

- Funcție responsabila de calcularea mutărilor posibile pentru cal, verifica limitele tablei si prezenta pieselor proprii sau adversare:

```
def getKnightMoves(self, row, col, moves):
        knightMoves = ((-2, -1), (-1, -2), (1, -2), (2, -1), (2, 1), (1, 2), (-1, 2), (-2, 1))
        allyColor = "w" if self.whiteToMove else "b" # friendly piece
        for m in knightMoves:
            endRow = row + m[0]
            endCol = col + m[1]
            if 0 <= endRow < 8 and 0 <= endCol < 8: # on board
                endPiece = self.board[endRow][endCol]
                if endPiece[0] != allyColor: # not a friendly piece (empty or enemy piece)
                    moves.append(Move((row, col), (endRow, endCol), self.board))
```

- Funcție care se ocupa cu calcularea mutărilor posibile pentru nebun, ținând cont de marginile tablei si piesele din jur:

```
def getBishopMoves(self, row, col, moves):
        directions = ((-1, -1), (-1, 1), (1, -1), (1, 1)) # 4 diagonals
        enemyColor = "b" if self.whiteToMove else "w"
        for d in directions:
            for i in range(1, 8): # bishop can move max 7 squares
                endRow = row + d[0] * i
                endCol = col + d[1] * i
                if 0 <= endRow < 8 and 0 <= endCol < 8: # on board
                    endPiece = self.board[endRow][endCol]
                    if endPiece == "--": # empty space valid
                        moves.append(Move((row, col), (endRow, endCol), self.board))
                    elif endPiece[0] == enemyColor: # enemy piece valid
                        moves.append(Move((row, col), (endRow, endCol), self.board))
                        break
                    else: # friendly piece invalid
                        break
                else: # off board
```

```
            break
```

- Funcție care calculează mutările posibile pentru tură, ținând cont de marginile tablei si piesele din jur:

```
def getRookMoves(self, row, col, moves):
        directions = ((-1, 0), (0, -1), (1, 0), (0, 1)) # up, left, down, right
        enemyColor = "b" if self.whiteToMove else "w"
        for dir in directions:
            for i in range(1, 8):
                endRow = row + dir[0] * i
                endCol = col + dir[1] * i
                if 0 <= endRow < 8 and 0 <= endCol < 8: # on board
                    endPiece = self.board[endRow][endCol]
                    if endPiece == "--": # empty space valid
                        moves.append(Move((row, col), (endRow, endCol), self.board))
                    elif endPiece[0] == enemyColor: # enemy piece valid
                        moves.append(Move((row, col), (endRow, endCol), self.board))
                        break
                    else: # friendly piece invalid
                        break
                else: # off board
                    break
```

- Funcție care calculează mutările posibile pentru regina, de fapt folosește o combinație dintre mutările pentru nebun si tura:

```
def getQueenMoves(self, row, col, moves):
        self.getRookMoves(row, col, moves)
        self.getBishopMoves(row, col, moves)
```

- Funcție care calculează mutările posibile pentru rege, tine cont de pătratele care sunt sub atac

```
def getKingMoves(self, row, col, moves):
        kingMoves = ((-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1))
        allyColor = "w" if self.whiteToMove else "b" # friendly piece
        for i in range(8):
            endRow = row + kingMoves[i][0]
            endCol = col + kingMoves[i][1]
            if 0 <= endRow < 8 and 0 <= endCol < 8: # on board
                endPiece = self.board[endRow][endCol]
                if endPiece[0] != allyColor: # not a friendly piece (empty or enemy piece)
                    moves.append(Move((row, col), (endRow, endCol), self.board))
```

3.1.6 Implementarea mutărilor speciale

- En-passant, captura speciala de pion este posibilă doar imediate după ce adversarul a mutat un pion cu două pătrate. Acest tip de mutare este introdus in logica de mutări pe care o pot face pioni;

- Rocada, se poate face fie pe partea regelui(partea stânga a tablei) sau in partea reginei(partea dreapta a tablei), ea se poate face daca regele sau tura nu au fost mutate, dacă pătratele dintre rege si tura nu sunt sub atac și dacă pătratele dintre tura si rege sunt libere.

Rocada in partea regelui se face astfel:

```
def getKingsideCastleMoves(self, row, col, moves):
        # ensure squares exist on board to avoid IndexError
        if col + 2 > 7:
            return
        # squares between king and rook must be empty
        if self.board[row][col+1] == "--" and self.board[row][col+2] == "--":
        # rook expected at col+3 for kingside; ensure it's on board and is a rook of the right color
            rookCol = col + 3
            if rookCol <= 7:
                rookPiece = self.board[row][rookCol]
                expectedColor = 'w' if self.whiteToMove else 'b'
                if rookPiece[1] == 'R' and rookPiece[0] == expectedColor:
                    # ensure squares king moves through/into are not under attack
                    if not self.squareUnderAttack(row, col+1) and not self.squareUnderAttack(row,
col+2):
                        moves.append(Move((row, col), (row, col+2), self.board, isCastleMove = True))
```

Rocada in parte regelui se face astfel:

```
def getQueenSideCastleMoves(self, row, col, moves):
        # ensure squares exist on board to avoid IndexError
        if col - 3 < 0:
            return
        # squares between king and rook must be empty
        if self.board[row][col-1] == "--" and self.board[row][col-2] == "--" and self.board[row][col-3]
== "--":
            # rook expected at col-4 for queen side; ensure it's on board and is a rook of the right
color
            rookCol = col - 4
            if rookCol >= 0:
                rookPiece = self.board[row][rookCol]
                expectedColor = 'w' if self.whiteToMove else 'b'
                if rookPiece[1] == 'R' and rookPiece[0] == expectedColor:
                    # ensure squares king moves through/into are not under attack
                    if not self.squareUnderAttack(row, col-1) and not self.squareUnderAttack(row, col-
2):
                        moves.append(Move((row, col), (row, col-2), self.board, isCastleMove = True))
```

3.2 Modulul ChessAI

Componenta de inteligenta artificiala care folosește un algoritm NegaMax cu Alpha-Beta Pruning, o varianta optimizată a algoritmului MinMax.

Scopul acestui algoritm este să evalueze toate mutările posibile și să aleagă varianta care maximizează șansele de câștig pentru AI. Evaluarea unei poziții se face bazata atât pe valoarea materială a pieselor, cat si pe poziția acestora pe tabla.

3.2.1 Principiul general

La baza AI-ului folosește algoritmul de MinMax, care are ca scop maximizarea scorul poziției proprii si de a minimiza avantajul oponentului.

Aceasta presupune construirea unui arbore de joc unde:

- Fiecare nod reprezintă o configurație a tablei(o stare de joc);
- Fiecare muchie reprezintă o mutare posibila;
- Nivelul arborelui alternează intre mutările jucătorului si ale adversarului.

În practica adâncimea arborelui este limitata iar pozițiile finale sunt evaluate cu o funcție euristica.

3.2.2 Optimizarea prin algoritmul NegaMax

În locul implementării algoritmului MinMax, AI-ul folosește o versiune mai compacta numite NegaMax.

Aceasta se bazează pe simetria dintre jucători: avantajul unuia este dezavantajul celuilalt.

Astfel în loc de scrierea funcției separatoare pentru maximizare si minimizare se folosește o singura funcție

NegaMax = max(-NegaMax(rezultat(s, a), d - 1))

Aceasta abordare reduce codul și creste eficiența

3.2.3 Optimizarea prin Alpha-Beta Pruning

Căutarea completa prin arborele de joc este extrem de costisitoare. Pentru reducerea complexității algoritmul utilizează Alpha-Beta Pruning care elimină

ramurile irelevante din arbore – adică reducerea pozițiilor care nu pot influenta decizia finala.

α(alpha) - cea mai buna valoare găsita pana acum pentru jucătorul MAX(AI-ul)

B(beta) - cea mai buna valoare găsita pentru jucătorul MIN(oponent)

Dacă la un moment dat $\alpha \geq \beta$ se poate întrerupe explorarea acelei ramuri, întrucât nu poate produce un rezultat mai bun decât ceea ce deja s-a găsit

Astfel complexitatea scade semnificativ, iar performata creste permițând cutarea mai adâncă în acel timp.

```
def findMoveNegaMaxAlphaBeta(gs, validMoves, depth, alpha, beta, turnMultiplier, rootDepth):
        # prefer faster mates: if this position is terminal, return mate score adjusted by distance
        if gs.checkMate:
            # if side to move is checkmated, it's a loss for the side to move
            mate_distance = rootDepth - depth
            if gs.whiteToMove:
                return - (CHECKMATE - mate_distance)
            else:
                return CHECKMATE - mate_distance
        if gs.staleMate:
            return STALEMATE
        if depth == 0:
            # use quiescence search at leaf; pass rootDepth for mate-distance accounting
            return turnMultiplier * quiescence(alpha, beta, gs, turnMultiplier, rootDepth)
        # order moves to improve pruning
        orderedMoves = orderMoves(validMoves, gs)
        maxScore = -CHECKMATE
        for move in orderedMoves:
            gs.makeMove(move)
            nextMoves = gs.getValidMoves()
            score = -findMoveNegaMaxAlphaBeta(gs, nextMoves, depth - 1, -beta, -alpha, -
turnMultiplier, rootDepth)
            if score > maxScore:
                maxScore = score
                gs.undoMove()
            if maxScore > alpha: # pruning
                alpha = maxScore
            if alpha >= beta:
                break
        return maxScore
```

3.2.4 Ordinea mutarilor

Pentru a maximiza eficiența tăierii Alpha-Beta, mutările sunt evaluate in ordinea descrescătoare a “probabilități de succes”.

Acesta este realizata cu funcția orderMoves(), care acorda prioritate pe baza următoarelor criterii:

- Capturile sunt favorizate(valoare piesei capturate – valoarea piesei care capturează);
- Promovarea de pioni cu bonus semnificativ;
- Mutările către centrul tablei primesc o mica bonificație.

```
def orderMoves(moves, gs):
        # prefer captures (victim value - attacker value) and promotions
        def score(move):
          score_val = 0
        # capture: higher victim value -> higher priority, lower attacker value -> higher priority
          if move.pieceCaptured != "--":
          victim = piecesScore.get(move.pieceCaptured[1], 0)
          attacker = piecesScore.get(move.pieceMoved[1], 0)
          score_val += 1000 + (victim * 10 - attacker) # MVV-LVA style
          # pawn promotions
          if move.isPawnPromotion:
            score_val += 900
          # small tie-breaker: prefer center moves (optional)
            center_bonus = 3 - (abs(3.5 - move.endRow) + abs(3.5 - move.endCol))
            score_val += int(center_bonus)
          return -score_val # negative because we'll sort ascending
        return sorted(moves, key=score)
```

Aceasta strategie numita Move Orering creste eficiența Alpha-Beta deoarece cele mai promițătoare mutări sunt verificate primele determinând tăieri mai rapide in arbore.

3.2.5 Căutarea Quiescence (liniștita)

O problemă frecventa în algoritmi de tip MinMax este efectul de instabilitate la frunzele arborelui, pozițiile analizate pot fi imediat afectate de capturi iminente, ceea ce duce la evaluări eronate.

Pentru a preveni acest lucru, se poate aplica o căutare suplimentara numita quwscene search care continuă explorarea doar pentru mutările de tip captură.

Astfel Ai-ul nu taie analiza într-o poziție tactica instabila.

```
def quiescence(alpha, beta, gs, turnMultiplier, rootDepth):
        # terminal check first so mates discovered in quiescence are distance-weighted
        if gs.checkMate:
          mate_distance = rootDepth # quiescence is called at depth == 0 so use rootDepth
          if gs.whiteToMove:
            return - (CHECKMATE - mate_distance)
```

```
            else:
                return CHECKMATE - mate_distance
        if gs.staleMate:
            return STALEMATE

        stand_pat = turnMultiplier * scoreBoard(gs)
        if stand_pat >= beta:
            return beta
        if alpha < stand_pat:
            alpha = stand_pat
        # generate captures only
        capture_moves = [m for m in gs.getAllPossibleMoves() if m.pieceCaptured != "--" or
m.isEnpassantMove]
        capture_moves = orderMoves(capture_moves, gs)
        for move in capture_moves:
            gs.makeMove(move)
            score = -quiescence(-beta, -alpha, gs, -turnMultiplier, rootDepth)
            gs.undoMove()
            if score >= beta:
                return score
            if score > alpha:
                alpha = score
                return alpha
```

3.2.6 Evaluarea tablei în funcție de material

Aceasta componenta cuantifica importanta relativa a fiecărei tip de piesa.

Definirea valorilor materială a pieselor se face astfel:

```
piecesScore = {'K': 0, 'Q': 9, 'R': 5, 'B': 3, 'N': 3, 'P': 1}
```

Pe baza acestei valori funcția scoreMaterial() parcurge întreaga tablă și aduna/scade punctajul aferent fiecărei piese astfel:

```
def scoreMaterial(board):
        material = 0
        for r in range(len(board)):
            for c in range(len(board[r])):
                piece = board[r][c]
                if piece != "--":
                    color = piece[0]
                    ptype = piece[1]
                    val = piecesScore.get(ptype, 0)
                    if color == 'w':
                        material += val
                    else:
                        material -= val
        return material
```

Rezultatul este un scor pozitiv dacă albul are avantaj material sau negativ dacă negru are avantaj

3.2.7 Evaluarea tablei in funcție de poziție

Pe lângă valoarea piesei în șah contează și unde se afla piesa pe tabla. Pentru a lua în considerare acest aspect aplicația folosește un tabel de poziționare pentru fiecare tip de piesa în parte care oferă o bonificație numerica fiecărei poziții a piesei pe tabla.

- Pentru pioni

```
pawnScores = [
        [8, 8, 8, 8, 8, 8, 8, 8],
        [8, 8, 8, 8, 8, 8, 8, 8],
        [5, 6, 6, 7, 7, 6, 6, 5],
        [2, 3, 3, 5, 5, 3, 3, 2],
        [1, 2, 3, 4, 4, 3, 2, 1],
        [1, 1, 2, 3, 3, 2, 1, 1],
        [1, 1, 1, 0, 0, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0]
        ]
```

- Pentru cai

```
knightScores = [
        [1, 1, 1, 1, 1, 1, 1, 1],
        [1, 2, 2, 2, 2, 2, 2, 1],
        [1, 2, 3, 3, 3, 3, 2, 1],
        [1, 2, 3, 4, 4, 3, 2, 1],
        [1, 2, 3, 4, 4, 3, 2, 1],
        [1, 2, 3, 3, 3, 3, 2, 1],
        [1, 2, 2, 2, 2, 2, 2, 1],
        [1, 1, 1, 1, 1, 1, 1, 1]
        ]
```

- Pentru nebuni

```
bishopScores = [
        [4, 3, 2, 1, 1, 2, 3, 4],
        [3, 4, 3, 2, 2, 3, 4, 3],
        [2, 3, 4, 3, 3, 4, 3, 2],
        [1, 2, 3, 4, 4, 3, 2, 1],
        [1, 2, 3, 4, 4, 3, 2, 1],
        [2, 3, 4, 3, 3, 4, 3, 2],
        [3, 4, 3, 2, 2, 3, 4, 3],
```

```
        [4, 3, 2, 1, 1, 2, 3, 4]
        ]
```

- Pentru ture

```
rookScores = [
        [4, 3, 4, 4, 4, 4, 3, 4],
        [4, 4, 4, 4, 4, 4, 4, 4],
        [1, 1, 2, 3, 3, 2, 1, 1],
        [1, 2, 3, 4, 4, 3, 2, 1],
        [1, 2, 3, 4, 4, 3, 2, 1],
        [1, 1, 2, 3, 3, 2, 1, 1],
        [4, 4, 4, 4, 4, 4, 4, 4],
        [4, 3, 4, 4, 4, 4, 3, 4]
        ]
```

- Pentru regină

```
queenScores = [
        [1, 1, 1, 3, 1, 1, 1, 1],
        [1, 2, 3, 3, 3, 1, 1, 1],
        [1, 4, 3, 3, 3, 4, 2, 1],
        [1, 2, 3, 3, 3, 2, 2, 1],
        [1, 2, 3, 3, 3, 2, 2, 1],
        [1, 2, 3, 3, 3, 4, 2, 1],
        [1, 1, 2, 3, 3, 1, 1, 1],
        [1, 1, 1, 3, 1, 1, 1, 1]
        ]
```

- Pentru rege

```
kingScores = [
        [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, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [2, 2, 1, 0, 0, 1, 2, 2],
        [4, 4, 3, 1, 1, 3, 4, 4]
        ]
```

3.2.8 Combinarea celor doua scoruri

Funcția scoreBoard unește valoarea materiala și cea pozițional într-un scor unic, adăugând valoarea materiala a fiecărei piesa cu bonusul pozițional corespunzător fiecărei locații și ajustează scorul în funcție de culoare(pozitiv pentru alb , negativ pentru negru).

```
def scoreBoard(gs):
```

```
        if gs.checkMate:
            if gs.whiteToMove:
                return -CHECKMATE # black wins
            else:
                return CHECKMATE # white wins
        elif gs.staleMate:
            return STALEMATE
        score = 0
        for row in range(len(gs.board)):
            for coll in range(len(gs.board[row])):
                square= gs.board[row][coll]
                if square != "--":
                    #score it positionally
                    if square[1] in piecePositionScores:
                        piecePositionScore= piecePositionScores[square[1]]
                        if square[0] == 'w':
                            score += piecePositionScore[row][coll]
                        elif square[0] == 'b':
                     score-=piecePositionScore[7-row][7- coll]
                    #score it by piece value
                    if square[0] == 'w':
                        score += piecesScore[square[1]]
                    elif square[0] == 'b':
                        score -= piecesScore[square[1]]
        return score
```

3.2.10 Implementarea finala a algoritmului

Funcția findBestMove() folosește căutarea iterativă pe adâncime. Astfel AI-ul analizează întâi toate mutările la o adâncime mica apoi creste treptat nivelul de analiză la DEPTH = 4

Aceasta abordare strategica asigura un răspuns rapid și relativ bun chiar și în poziții complexe.

```
def findBestMove(gs, validMoves):
        bestMove = None
        # iterative deepening
        maxDepth = DEPTH
        for depth in range(1, maxDepth + 1):
            alpha = -CHECKMATE
            beta = CHECKMATE
            # order root moves for better pruning
            orderedMoves = orderMoves(validMoves, gs)
            bestScore = -CHECKMATE
            rootTurn = 1 if gs.whiteToMove else -1
            for move in orderedMoves:
```

```
            gs.makeMove(move)
            nextMoves = gs.getValidMoves()
            # pass the current root search depth so mate distance can be computed
            score=-findMoveNegaMaxAlphaBeta(gs,nextMoves, depth - 1,-beta,-alpha,-
rootTurn,depth)
        gs.undoMove()
            if score > bestScore:
              bestScore = score
              bestMove = move
            if bestScore > alpha:
              alpha = bestScore
            if alpha >= beta:
              break
        return bestMove
```

3.2.11 Avantajele acestei abordări

Prin combinarea acestor componente, AI-ul reușește să simuleze un nivel de joc competitiv și logic, capabil sa analizeze consecințele propriilor decizi și să evite capturile oponentului.

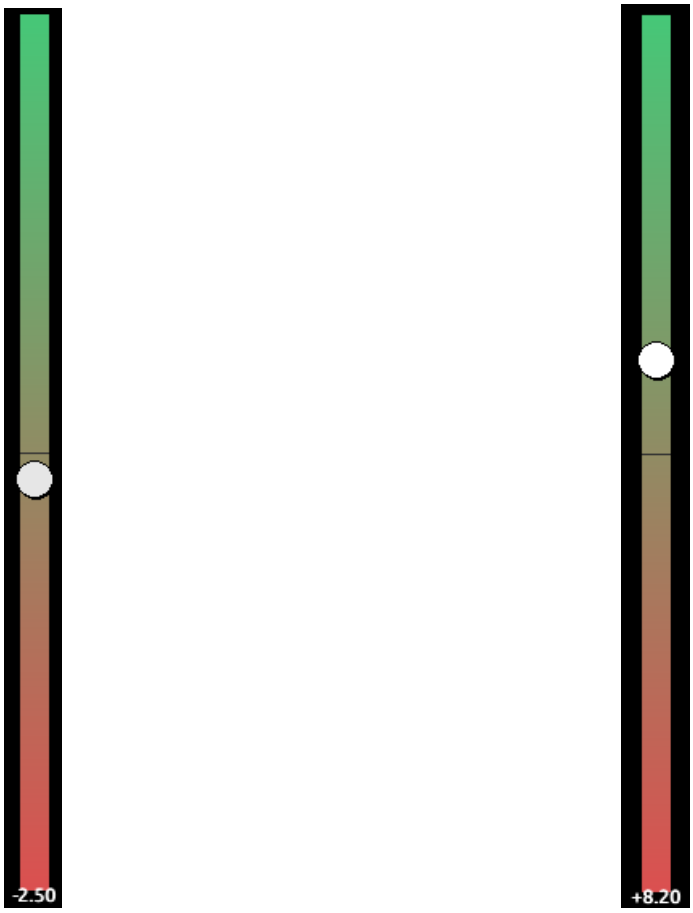
3.3 Modului ChessMain

Acest modul utilizează biblioteca Pygame pentru afișarea tablei de joc a pieselor și interacțiuni cu utilizatorul.

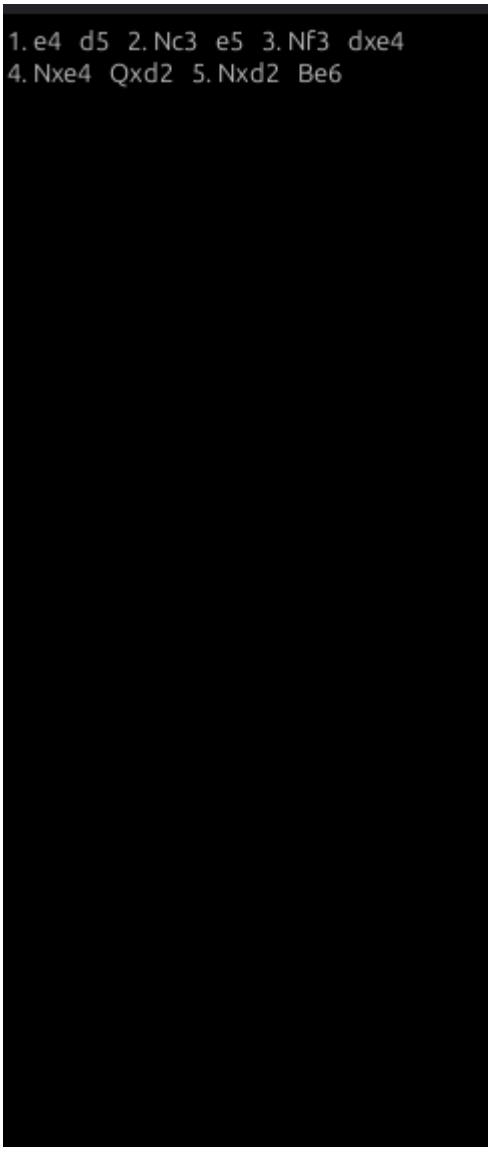
Ca funcționalități principale acest modul are animația pieselor când acestea sunt mutate atât manual de utilizator cat si in mod automat de către AI. Cât și anularea mutărilor prin apăsarea tastei Z sau resetarea jocului prin apăsarea tastei R.

Modulul permite afișarea mai multor elemente grafice cum ar fi:

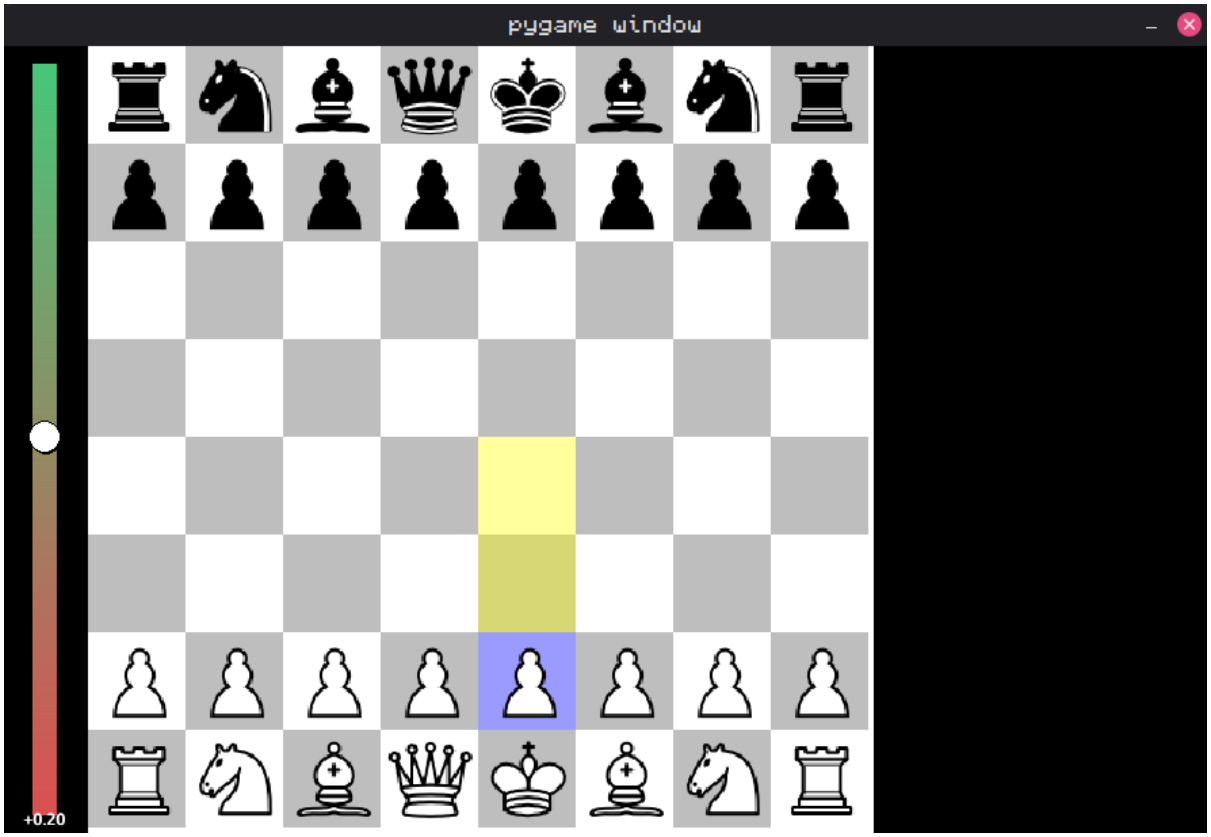
- o bara de evaluare care se face automat de către AI in partea stânga, evaluarea este pozitiva daca alb are avantaj iar negativa daca negru are avantaj;



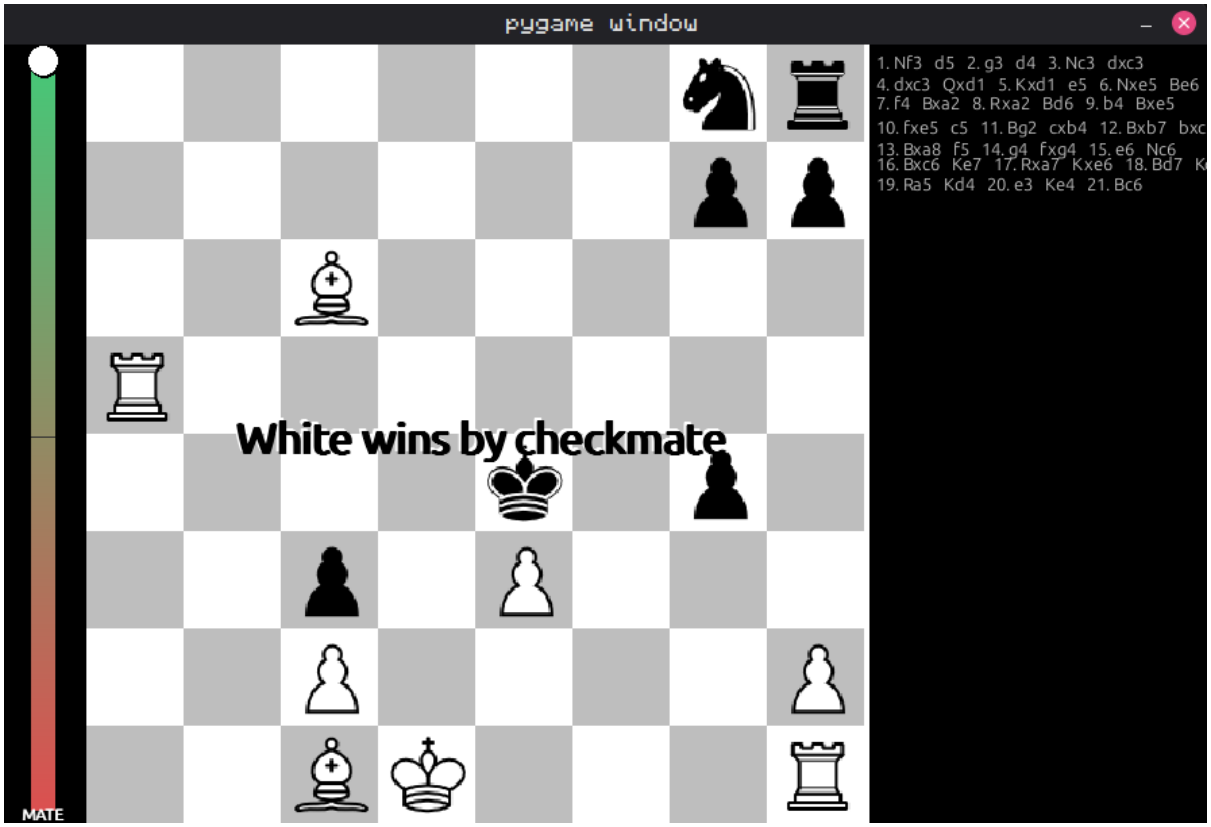
- istoricul mișcărilor in partea dreapta.



Modulul va evidenția mutările posibile pe care utilizatorul le poate face în momentul in care el apasă pe o piesa atâta timp cat este rândul lui sa facă o mutare.



Daca unul dintre jucători câștiga pe ecran va apărea care parte a câștigat(alb sau negru).



În caz contrar in care este remiza va apărea mesajul “Stale Mate”

4. Bibliografie

[1] Chess Programming Wiki, Main Page – Resources for Computer Chess Algorithms, disponibil la:
https://www.chessprogramming.org/Main_Page .

[2] Wikipedia, Computer Chess, disponibil la:
https://en.wikipedia.org/wiki/Computer_chess .

[3] Wikipedia, Minimax Algorithm, disponibil la:
https://en.wikipedia.org/wiki/Minimax .

[4] Wikipedia, Negamax Algorithm, disponibil la:
https://en.wikipedia.org/wiki/Negamax.

[5] Wikipedia, Alpha–Beta Pruning, disponibil la:
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning.

[6] Wikimedia Commons, Chess piece set image (Chess_bdt60.png), disponibil la:
https://commons.wikimedia.org/wiki/File:Chess_bdt60.png .

[7] Pygame Community, Pygame Documentation – Game development with Python, disponibil la:
https://www.pygame.org/docs/ .

[8] Python Software Foundation, Python 3.12 Documentation, disponibil la:
https://docs.python.org/3/ .

5. Fișiere

Pentru fișierele necesare rulări acestei aplicați se poate vizita:

https://github.com/HappyPlayer72/ProiectInteligentaArtificiala

Versiunea de Python recomandată este 3.12.11

Pentru instalarea dependentelor se poate folosi următoarea metoda:

În fișierul sursa se rulează următoarele comenzi:

```
python -m venv venv
venv/bin/activate                #pentru Linux
```

venv\Scripts\activate #pentru Windows

pip install -r requirements.txt

Pentru rularea aplicați se rulează fișierul ChessMain.py