import time  
import random  
from board import BOARD\_SIZE  
import pygame  
  
# ----------- Zobrist Hashing ----------  
ZOBRIST = {}  
  
def owner\_to\_id(owner):  
 if owner == 'Player1':  
 return 0  
 elif owner == 'Player2':  
 return 1  
 return 2  
  
def initialize\_zobrist():  
 global ZOBRIST  
 for row in range(BOARD\_SIZE):  
 for col in range(BOARD\_SIZE):  
 for owner\_id in range(3): # 0 (White), 1 (Black), 2 (None)  
 ZOBRIST[(row, col, owner\_id)] = random.getrandbits(64)  
# --------------------------------------  
  
def get\_all\_possible\_capture\_moves(game, player):  
 *""" Sadece capture (yeme) hamlelerini döndürür.  
 İnsan tarafı da fianco.py içerisinden buna erişebilir.  
 """* capture\_moves = []  
 for piece in game.pieces:  
 if piece.owner == player:  
 # Yeme için çift atlama bakıyoruz  
 if player == 'Player1':  
 poss = [  
 (piece.row + 2, piece.col + 2),  
 (piece.row + 2, piece.col - 2)  
 ]  
 else:  
 poss = [  
 (piece.row - 2, piece.col + 2),  
 (piece.row - 2, piece.col - 2)  
 ]  
 for (r, c) in poss:  
 is\_ok, \_, cap = game.is\_valid\_move(piece, r, c)  
 if is\_ok and cap is not None:  
 capture\_moves.append((piece, r, c, cap))  
 return capture\_moves  
  
def get\_all\_possible\_moves(game, player):  
 *""" Oyuncu player için tüm olası hamleleri döndürür.  
 Eğer herhangi bir capture hamlesi varsa, normal hamleler  
 listeden çıkar (mandatory capture).  
 """* all\_moves = []  
 capture\_moves = []  
  
 for piece in game.pieces:  
 if piece.owner == player:  
 if player == 'Player1':  
 poss = [  
 (piece.row + 1, piece.col),  
 (piece.row, piece.col + 1),  
 (piece.row, piece.col - 1),  
 (piece.row + 2, piece.col + 2),  
 (piece.row + 2, piece.col - 2)  
 ]  
 else:  
 poss = [  
 (piece.row - 1, piece.col),  
 (piece.row, piece.col + 1),  
 (piece.row, piece.col - 1),  
 (piece.row - 2, piece.col + 2),  
 (piece.row - 2, piece.col - 2)  
 ]  
 for (r, c) in poss:  
 is\_ok, \_, cap = game.is\_valid\_move(piece, r, c)  
 if is\_ok:  
 if cap is not None:  
 capture\_moves.append((piece, r, c, cap))  
 else:  
 all\_moves.append((piece, r, c, None))  
  
 # ZORUNLU YEME  
 if capture\_moves:  
 return capture\_moves  
 return all\_moves  
  
  
# ------------------ AI Search (Negamax) --------------------  
  
def make\_ai\_move(game):  
 *""" AI'nin hamle yapması. """* start\_ms = time.time() \* 1000  
 best\_move = None  
 depth\_reached = 1  
  
 # Olası hamle sayısına göre ufak adaptif derinlik limiti  
 num\_moves = len(get\_all\_possible\_moves(game, game.current\_player))  
 if num\_moves > 20:  
 local\_depth\_limit = 5  
 else:  
 local\_depth\_limit = 8  
  
 for depth in range(1, local\_depth\_limit + 1):  
 now = time.time() \* 1000  
 if now - start\_ms > game.iterative\_time\_limit:  
 break  
 game.prune\_count = 0  
 game.nodes\_searched = 0  
 move\_candidate = negamax\_root(game, depth, -999999, 999999)  
 if move\_candidate:  
 best\_move = move\_candidate  
 depth\_reached = depth  
 else:  
 break  
  
 if best\_move:  
 game.make\_move(best\_move)  
 print(f"[AI] depth={depth\_reached}, prune\_count={game.prune\_count}, nodes={game.nodes\_searched}")  
 # İstatistik güncelleme  
 game.total\_prunes += game.prune\_count  
 if game.prune\_count > game.max\_prune\_per\_move:  
 game.max\_prune\_per\_move = game.prune\_count  
 else:  
 print("[AI] No valid move found. pass")  
  
def negamax\_root(game, depth, alpha, beta):  
 moves = get\_all\_possible\_moves(game, game.current\_player)  
 moves = order\_moves(game, moves, depth)  
  
 best\_score = -999999  
 best\_move = None  
 original\_alpha = alpha  
  
 board\_hash = get\_board\_hash(game)  
 # TT key: (board\_hash, depth, alpha, beta)  
 # root'ta alpha/beta sabit gibi, ama yine de kaydedebiliriz  
  
 for move in moves:  
 prev = game.make\_move(move, store\_previous\_state=True)  
 game.current\_player = 'Player2' if game.current\_player == 'Player1' else 'Player1'  
  
 score = -negamax(game, depth - 1, -beta, -alpha, 1)  
  
 game.unmake\_move(prev)  
  
 if score > best\_score:  
 best\_score = score  
 best\_move = move  
 if best\_score > alpha:  
 alpha = best\_score  
 if alpha >= beta:  
 game.prune\_count += 1  
 break  
  
 # TT Bound ve best\_move saklama  
 if best\_score <= original\_alpha:  
 bound\_flag = 'UPPER'  
 elif best\_score >= beta:  
 bound\_flag = 'LOWER'  
 else:  
 bound\_flag = 'EXACT'  
  
 tt\_key = (board\_hash, depth, original\_alpha, beta)  
 game.ttable[tt\_key] = {  
 "score": best\_score,  
 "flag": bound\_flag,  
 "best\_move": best\_move  
 }  
  
 return best\_move  
  
def negamax(game, depth, alpha, beta, color):  
 if depth == 0 or game.game\_over:  
 game.nodes\_searched += 1  
 return evaluate\_position(game) \* color  
  
 # TT sorgusu  
 board\_hash = get\_board\_hash(game)  
 original\_alpha = alpha  
 tt\_key = (board\_hash, depth, alpha, beta)  
  
 game.tt\_accesses += 1  
 if tt\_key in game.ttable:  
 entry = game.ttable[tt\_key]  
 flag = entry.get("flag", "EXACT")  
 cached\_score = entry["score"]  
  
 # TT Bound kullanımı  
 if flag == 'EXACT':  
 return cached\_score  
 elif flag == 'LOWER': # score >= gerçek beta  
 alpha = max(alpha, cached\_score)  
 elif flag == 'UPPER': # score <= gerçek alpha  
 beta = min(beta, cached\_score)  
 if alpha >= beta:  
 return cached\_score  
  
 moves = get\_all\_possible\_moves(game, game.current\_player)  
 if not moves:  
 game.nodes\_searched += 1  
 return evaluate\_position(game) \* color  
  
 moves = order\_moves(game, moves, depth)  
 best\_score = -999999  
 best\_move = None  
  
 for move in moves:  
 prev = game.make\_move(move, store\_previous\_state=True)  
 game.current\_player = 'Player2' if game.current\_player == 'Player1' else 'Player1'  
 score = -negamax(game, depth - 1, -beta, -alpha, -color)  
 game.unmake\_move(prev)  
  
 if score > best\_score:  
 best\_score = score  
 best\_move = move  
 if best\_score > alpha:  
 alpha = best\_score  
 if alpha >= beta:  
 game.prune\_count += 1  
 # killer move  
 kset = game.killer\_moves.get(depth, set())  
 kset.add(move)  
 game.killer\_moves[depth] = kset  
 break  
  
 # TT'ye kaydet  
 if best\_score <= original\_alpha:  
 bound\_flag = 'UPPER'  
 elif best\_score >= beta:  
 bound\_flag = 'LOWER'  
 else:  
 bound\_flag = 'EXACT'  
  
 game.ttable[tt\_key] = {  
 "score": best\_score,  
 "flag": bound\_flag,  
 "best\_move": best\_move  
 }  
 return best\_score  
  
def evaluate\_position(game):  
 # Basit evaluate: Beyaz taş sayısı - Siyah taş sayısı  
 w = len([p for p in game.pieces if p.owner == 'Player1'])  
 b = len([p for p in game.pieces if p.owner == 'Player2'])  
 return w - b  
  
def get\_board\_hash(game):  
 h = 0  
 for p in game.pieces:  
 oid = owner\_to\_id(p.owner)  
 h ^= ZOBRIST[(p.row, p.col, oid)]  
 return h  
  
def order\_moves(game, moves, depth):  
 *"""  
 1) TT move  
 2) Capture  
 3) Killer  
 4) Normal  
 """* # TT move bul  
 board\_hash = get\_board\_hash(game)  
 best\_move\_from\_tt = None  
 # En basit yoldan: tabloyu gezip  
 for key, entry in game.ttable.items():  
 # key: (hash, d, a, b)  
 if key[0] == board\_hash and key[1] == depth:  
 bm = entry.get("best\_move", None)  
 if bm:  
 best\_move\_from\_tt = bm  
 break  
  
 captures = []  
 killers = []  
 normal = []  
 kset = game.killer\_moves.get(depth, set())  
  
 for m in moves:  
 piece, r, c, captured = m  
 if best\_move\_from\_tt is not None and m == best\_move\_from\_tt:  
 # TT move en başa  
 captures.insert(0, m) # liste başına ekleyelim  
 elif captured is not None:  
 captures.append(m)  
 elif m in kset:  
 killers.append(m)  
 else:  
 normal.append(m)  
  
 return captures + list(killers) + normal