**סיכום כללי פרויקט סיום באלגוריתמים מתקדמים**

**מגישים:   
עדי דגן 318792579  
עילאי גוזלן 318473295  
עפרי גרוס 32242820**

[**לינק להדגמת המצגת**](https://www.youtube.com/watch?v=IQtBj4tXJdI)[**לינק להדגמת המשחק**](https://www.youtube.com/watch?v=vj1CFtAJPt8)[**לינק לGITHUB**](https://github.com/OfriG/Checkers_Algorithm.git)

**מטרת הפרויקט:** לבנות משחק דמקה דיגיטלי הכולל ממשק גרפי ולוגיקה מלאה של חוקי דמקה, עם סוכן חכם שיודע לשחק בצורה אסטרטגית מול שחקן אנושי. המשחק מבוסס על אלגוריתם Minimax עם חיתוך אלפא-בטא, בתוספת חידושים ושינויים יצירתיים למשחק הקלאסי.

הסוכן החכם: פועל באמצעות Minimax עם Alpha-Beta Pruning   
בוחן את כל המהלכים החוקיים האפשריים ומדרג אותם לפי פונקציית הערכה.  
  
פונקציית evaluate אחראית לכימות איכות המצב הנוכחי של הלוח. כלומר, לקבוע עד כמה מצב המשחק נוח או טוב עבור השחקן הלבן- הבינה המלאכותית.

המטרה היא לתרגם את מצב הלוח לערך מספרי:  
ערך גבוה יותר = טוב ל -AIהשחקן הלבן.  
ערך נמוך יותר = טוב לשחקן היריב- השחקן האדום.

פונקציית ההערכה כוללת שקלול של:  
יתרון במספר חיילים (לבן מול אדום).  
יתרון בכמות המלכים – שקולים לניסיון במשחק.  
התחשבות בקרבה ל"חורים" בלוח – קרבה מסכנת מענישה את השחקן.  
שקלול זמינות Boost - יתרון קל למי שעדיין לא השתמש בו.  
השקלול של כל המרכיבים יחד נותן ל-AI תמונה אסטרטגית, גם של **ההווה** וגם של **הפוטנציאל העתידי** במצב הנתון.

**קושי במימוש האלגוריתם**

**החלטה על עומק החיפוש**  
בתחילה בחרנו בעומק גזירה של 3, כדי לאפשר לאלגוריתם לקבל החלטות "חכמות" יותר. אך עומק זה גרם לעיכובים משמעותיים בביצוע המהלכים, ופגע בזרימת המשחק.  
לאחר ניסוי וטעייה, בחרנו להפחית את העומק ל-2 – החלטה ששיפרה באופן ניכר את מהירות המשחק, תוך שמירה על רמת אתגר מספקת.

**התמודדות עם Boost**  
הכנסת אפשרות לביצוע Boost דרשה מהאלגוריתם להעריך מצבים לא רק לפי מצב הלוח, אלא גם לפי זמינות הבוסט, השפעתו העתידית, והאם כדאי "לבזבז" אותו במהלך הנוכחי.

**התחשבות במלכים ותור כפול**  
כששחקן מוכתר למלך, הוא מקבל תור נוסף – דבר שמוסיף מורכבות ללוגיקה של יצירת מהלכים ומעקב אחר סדר התורות.

**תכנון תנועה בלוח עם משבצות חסומות (חורים)**  
החורים בלוח הגבילו את תנועת החתיכות ויצרו מצבים בהם שחקנים "נחסמים", מה שדרש טיפול מיוחד בבדיקת מהלכים חוקיים, על מנת להתמודד עם האתגר החלטנו להגדיל את הלוח על מנת להצליח לשפר את יכולת התנועה של השחקנים.

קוד: Board

import pygame

from .constants import BLACK, ROWS, RED, SQUARE\_SIZE, COLS, WHITE, HOLES

from .piece import Piece

class Board:

    def \_\_init\_\_(self):

        self.board = []

        self.red\_left = self.white\_left = 12

        self.red\_kings = self.white\_kings = 0

        self.create\_board()

    def draw\_squares(self, win):

        win.fill(BLACK)

        for row in range(ROWS):

            for col in range(row % 2, COLS, 2):

                pygame.draw.rect(win, RED, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))

        for row, col in HOLES:

            pygame.draw.rect(win, (100, 100, 100), (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))

    def evaluate(self, boost\_available=None):

        score = self.white\_left - self.red\_left

        score += 0.5 \* (self.white\_kings - self.red\_kings)

        for row in range(ROWS):

            for col in range(COLS):

                piece = self.board[row][col]

                if isinstance(piece, Piece):

                    for hole\_r, hole\_c in HOLES:

                        dist = abs(row - hole\_r) + abs(col - hole\_c)

                        if dist == 1:

                            if piece.color == WHITE:

                                score -= 0.1

                            else:

                                score += 0.1

        if boost\_available:

            if boost\_available["WHITE"]:

                score += 0.2

            if boost\_available["RED"]:

                score -= 0.2

        return score

    def get\_all\_pieces(self, color):

        return [piece for row in self.board for piece in row if piece != 0 and piece is not None and piece.color == color]

    def move(self, piece, row, col):

        self.board[piece.row][piece.col], self.board[row][col] = self.board[row][col], self.board[piece.row][piece.col]

        piece.move(row, col)

        if row == 0 or row == ROWS - 1:

            piece.make\_king()

            if piece.color == WHITE:

                self.white\_kings += 1

            else:

                self.red\_kings += 1

    def get\_piece(self, row, col):

        if (row, col) in HOLES:

            return None

        return self.board[row][col]

    def create\_board(self):

        for row in range(ROWS):

            self.board.append([])

            for col in range(COLS):

                if (row, col) in HOLES:

                    self.board[row].append(None)

                elif col % 2 == ((row + 1) % 2):

                    if row < 3:

                        self.board[row].append(Piece(row, col, WHITE))

                    elif row > 6:

                        self.board[row].append(Piece(row, col, RED))

                    else:

                        self.board[row].append(0)

                else:

                    self.board[row].append(0)

    def draw(self, win):

        self.draw\_squares(win)

        for row in range(ROWS):

            for col in range(COLS):

                piece = self.board[row][col]

                if piece != 0 and piece is not None:

                    piece.draw(win)

    def remove(self, pieces):

        for piece in pieces:

            self.board[piece.row][piece.col] = 0

            if piece != 0:

                if piece.color == RED:

                    self.red\_left -= 1

                else:

                    self.white\_left -= 1

    def winner(self):

        if self.red\_left <= 0:

            return WHITE

        elif self.white\_left <= 0:

            return RED

        return None

    def get\_valid\_boost\_moves(self, piece):

        moves = {}

        if piece is None:

            return moves

        directions = []

        if piece.color == RED or piece.king:

            directions += [(-2, -2), (-2, 2)]

        if piece.color == WHITE or piece.king:

            directions += [(2, -2), (2, 2)]

        for dr, dc in directions:

            r, c = piece.row + dr, piece.col + dc

            if 0 <= r < ROWS and 0 <= c < COLS and (r, c) not in HOLES and self.board[r][c] == 0:

                moves[(r, c)] = []

        return moves

    def get\_valid\_moves(self, piece, allow\_boost=False, boost\_used=False):

        moves = {}

        if piece is None:

            return moves

        left = piece.col - 1

        right = piece.col + 1

        row = piece.row

        def is\_valid\_dest(r, c):

            return 0 <= r < ROWS and 0 <= c < COLS and (r, c) not in HOLES and self.board[r][c] == 0

        if piece.color == RED or piece.king:

            moves.update(self.\_traverse\_left(row - 1, max(row - 3, -1), -1, piece.color, left))

            moves.update(self.\_traverse\_right(row - 1, max(row - 3, -1), -1, piece.color, right))

        if piece.color == WHITE or piece.king:

            moves.update(self.\_traverse\_left(row + 1, min(row + 3, ROWS), 1, piece.color, left))

            moves.update(self.\_traverse\_right(row + 1, min(row + 3, ROWS), 1, piece.color, right))

        moves = {move: skipped for move, skipped in moves.items() if is\_valid\_dest(\*move)}

        if allow\_boost and not boost\_used:

            moves.update(self.get\_valid\_boost\_moves(piece))

        return moves

    def \_traverse\_left(self, start, stop, step, color, left, skipped=[]):

        moves = {}

        last = []

        for r in range(start, stop, step):

            if left < 0:

                break

            current = self.board[r][left]

            if current == 0 or current is None:

                if skipped and not last:

                    break

                elif skipped:

                    moves[(r, left)] = last + skipped

                else:

                    moves[(r, left)] = last

                if last:

                    prev\_r = r - step

                    prev\_c = left + 1 if step == -1 else left - 1

                    if 0 <= prev\_r < ROWS and 0 <= prev\_c < COLS and self.board[prev\_r][prev\_c] is None:

                        break

                    next\_row = max(r - 3, 0) if step == -1 else min(r + 3, ROWS)

                    moves.update(self.\_traverse\_left(r + step, next\_row, step, color, left - 1, skipped=last))

                    moves.update(self.\_traverse\_right(r + step, next\_row, step, color, left + 1, skipped=last))

                break

            elif isinstance(current, Piece) and current.color == color:

                break

            else:

                last = [current]

            left -= 1

        return moves

    def \_traverse\_right(self, start, stop, step, color, right, skipped=[]):

        moves = {}

        last = []

        for r in range(start, stop, step):

            if right >= COLS:

                break

            current = self.board[r][right]

            if current == 0 or current is None:

                if skipped and not last:

                    break

                elif skipped:

                    moves[(r, right)] = last + skipped

                else:

                    moves[(r, right)] = last

                if last:

                    prev\_r = r - step

                    prev\_c = right - 1 if step == -1 else right + 1

                    if 0 <= prev\_r < ROWS and 0 <= prev\_c < COLS and self.board[prev\_r][prev\_c] is None:

                        break

                    next\_row = max(r - 3, 0) if step == -1 else min(r + 3, ROWS)

                    moves.update(self.\_traverse\_left(r + step, next\_row, step, color, right - 1, skipped=last))

                    moves.update(self.\_traverse\_right(r + step, next\_row, step, color, right + 1, skipped=last))

                break

            elif isinstance(current, Piece) and current.color == color:

                break

            else:

                last = [current]

            right += 1

        return moves

קוד constants

import pygame

WIDTH, HEIGHT = 800, 800

ROWS, COLS = 10, 10

SQUARE\_SIZE = WIDTH // COLS

# Colors

RED = (255, 0, 0)

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

BLUE = (0, 0, 255)

GREEN = (0, 255, 0)

GREY = (128, 128, 128)

# Crown image

CROWN = pygame.transform.scale(pygame.image.load('assets/crown.png'), (44, 25))

# Holes (obstacles)

HOLES = [

    (4, 1),

    (4, 5),

    (5, 2),

    (5,6)

]

קוד game

import pygame

from .constants import RED, WHITE, BLUE, GREEN, SQUARE\_SIZE

from checkers.board import Board

class Game:

    def \_\_init\_\_(self, win):

        self.win = win

        self.\_init()

    def \_init(self):

        self.selected = None

        self.board = Board()

        self.turn = RED

        self.valid\_moves = {}

        self.boost\_used = {RED: False, WHITE: False}

        self.boost\_just\_used = {RED: False, WHITE: False}

        self.twice\_used = {RED: False, WHITE: False}

        self.twice\_pending = {RED: False, WHITE: False}

        self.just\_got\_king = False

        self.winner\_color = None

    def update(self):

        self.board.draw(self.win)

        self.draw\_valid\_moves(self.valid\_moves)

        pygame.display.update()

    def reset(self):

        self.\_init()

    def winner(self):

        w = self.board.winner()

        if w is not None:

            self.winner\_color = w

        return w

    def select(self, row, col, use\_boost=False, use\_twice=False):

        if self.selected:

            result = self.\_move(row, col, use\_boost, use\_twice)

            if not result:

                self.selected = None

                self.select(row, col, use\_boost, use\_twice)

        piece = self.board.get\_piece(row, col)

        if piece and piece != 0 and piece.color == self.turn:

            self.selected = piece

            allow\_boost = use\_boost and not self.boost\_used[self.turn]

            self.valid\_moves = self.board.get\_valid\_moves(

                piece,

                allow\_boost=allow\_boost,

                boost\_used=self.boost\_used[self.turn]

            )

            return True

        return False

    def \_move(self, row, col, use\_boost=False, use\_twice=False):

        piece = self.board.get\_piece(row, col)

        if self.selected and piece == 0 and (row, col) in self.valid\_moves:

            was\_king = self.selected.king

            is\_boost\_move = (

                abs(self.selected.row - row) == 2 and

                abs(self.selected.col - col) == 2 and

                use\_boost and not self.boost\_used[self.turn]

            )

            if use\_boost and self.boost\_used[self.turn]:

                print("Boost already used – move rejected")

                return False

            self.board.move(self.selected, row, col)

            skipped = self.valid\_moves[(row, col)]

            if skipped:

                self.board.remove(skipped)

            if is\_boost\_move:

                self.boost\_used[self.turn] = True

                self.boost\_just\_used[self.turn] = True

            else:

                self.boost\_just\_used[self.turn] = False

            if use\_twice and not self.twice\_used[self.turn] and skipped:

                self.twice\_used[self.turn] = True

                self.twice\_pending[self.turn] = True

                return True

            self.just\_got\_king = not was\_king and self.selected.king

            if self.just\_got\_king:

                self.twice\_pending[self.turn] = True

            if not self.twice\_pending[self.turn]:

                self.change\_turn()

        else:

            return False

        return True

    def change\_turn(self):

        self.twice\_pending[self.turn] = False

        self.valid\_moves = {}

        self.selected = None

        next\_turn = WHITE if self.turn == RED else RED

        if not self.twice\_pending[next\_turn]:

            self.turn = next\_turn

    def draw\_valid\_moves(self, moves):

        for move in moves:

            row, col = move

            color = GREEN if (

                self.selected and abs(self.selected.row - row) == 2 and abs(self.selected.col - col) == 2

            ) else BLUE

            pygame.draw.circle(

                self.win,

                color,

                (col \* SQUARE\_SIZE + SQUARE\_SIZE // 2, row \* SQUARE\_SIZE + SQUARE\_SIZE // 2),

                15

            )

    def get\_board(self):

        return self.board

    def ai\_move(self, board):

        if getattr(board, 'last\_boost\_used', False):

            self.boost\_used[WHITE] = True

            self.boost\_just\_used[WHITE] = True

            board.last\_boost\_used = False

        else:

            self.boost\_just\_used[WHITE] = False

        current\_kings = sum(1 for p in board.get\_all\_pieces(WHITE) if p.king)

        prev\_kings = sum(1 for p in self.board.get\_all\_pieces(WHITE) if p.king)

        self.board = board

        if current\_kings > prev\_kings:

            self.twice\_pending[WHITE] = True

        else:

            self.twice\_pending[WHITE] = False

            self.change\_turn()

    def detect\_ai\_boost(self, old\_board, new\_board):

        # Detect if WHITE used Boost by checking if move was a 2-step diagonal

        old\_pieces = {(p.row, p.col): p for p in old\_board.get\_all\_pieces(WHITE)}

        new\_pieces = {(p.row, p.col): p for p in new\_board.get\_all\_pieces(WHITE)}

        for (r\_old, c\_old), piece in old\_pieces.items():

            for (r\_new, c\_new) in new\_pieces:

                if abs(r\_new - r\_old) == 2 and abs(c\_new - c\_old) == 2:

                    return True

        return False

    def has\_valid\_moves(self, color):

        pieces = self.board.get\_all\_pieces(color)

        for piece in pieces:

            moves = self.board.get\_valid\_moves(piece, allow\_boost=not self.boost\_used[color], boost\_used=self.boost\_used[color])

            if moves:

                return True

        return False

קוד piece

from .constants import RED, WHITE, SQUARE\_SIZE, GREY, CROWN

import pygame

class Piece:

    PADDING = 15

    OUTLINE = 2

    def \_\_init\_\_(self, row, col, color):

        self.row = row

        self.col = col

        self.color = color

        self.king = False

        self.x = 0

        self.y = 0

        self.calc\_pos()

    def calc\_pos(self):

        self.x = SQUARE\_SIZE \* self.col + SQUARE\_SIZE // 2

        self.y = SQUARE\_SIZE \* self.row + SQUARE\_SIZE // 2

    def make\_king(self):

        self.king = True

    def draw(self, win):

        radius = SQUARE\_SIZE//2 - self.PADDING

        pygame.draw.circle(win, GREY, (self.x, self.y), radius + self.OUTLINE)

        pygame.draw.circle(win, self.color, (self.x, self.y), radius)

        if self.king:

            win.blit(CROWN, (self.x - CROWN.get\_width()//2, self.y - CROWN.get\_height()//2))

    def move(self, row, col):

        self.row = row

        self.col = col

        self.calc\_pos()

    def \_\_repr\_\_(self):

        return str(self.color)

קוד algorithm

from copy import deepcopy

import pygame

RED = (255, 0, 0)

WHITE = (255, 255, 255)

def minimax(position, depth, max\_player, game,

            boost\_available={"RED": True, "WHITE": True},

            alpha=float('-inf'), beta=float('inf')):

    # Base case

    if depth == 0 or position.winner() is not None:

        return position.evaluate(boost\_available), position

    if max\_player:

        maxEval = float('-inf')

        best\_move = None

        for move, used\_boost in get\_all\_moves(position, WHITE, game,

                                              boost\_available["WHITE"],

                                              boost\_used=not boost\_available["WHITE"]):

            new\_boost = boost\_available.copy()

            if used\_boost:

                new\_boost["WHITE"] = False

                game.boost\_just\_used[WHITE] = True  # 🔔 Show message AI used Boost

            else:

                game.boost\_just\_used[WHITE] = False

            evaluation = minimax(move, depth - 1, False, game, new\_boost, alpha, beta)[0]

            if evaluation > maxEval:

                maxEval = evaluation

                best\_move = move

            alpha = max(alpha, evaluation)

            if beta <= alpha:

                break

        return maxEval, best\_move

    else:

        minEval = float('inf')

        best\_move = None

        for move, used\_boost in get\_all\_moves(position, RED, game,

                                              boost\_available["RED"],

                                              boost\_used=not boost\_available["RED"]):

            new\_boost = boost\_available.copy()

            if used\_boost:

                new\_boost["RED"] = False

            evaluation = minimax(move, depth - 1, True, game, new\_boost, alpha, beta)[0]

            if evaluation < minEval:

                minEval = evaluation

                best\_move = move

            beta = min(beta, evaluation)

            if beta <= alpha:

                break

        return minEval, best\_move

def simulate\_move(piece, move, board, game, skip):

    board.move(piece, move[0], move[1])

    if skip:

        board.remove(skip)

    return board

def get\_all\_moves(board, color, game, boost\_available=True, twice\_available=False, boost\_used=False):

    moves = []

    for piece in board.get\_all\_pieces(color):

        # רגיל

        valid\_moves = board.get\_valid\_moves(piece, boost\_used=boost\_used)

        for move, skip in valid\_moves.items():

            temp\_board = deepcopy(board)

            temp\_piece = temp\_board.get\_piece(piece.row, piece.col)

            new\_board = simulate\_move(temp\_piece, move, temp\_board, game, skip)

            just\_got\_king = not temp\_piece.king and (move[0] == 0 or move[0] == board.board.\_\_len\_\_() - 1)

            if twice\_available or just\_got\_king:

                for piece2 in new\_board.get\_all\_pieces(color):

                    valid\_moves2 = new\_board.get\_valid\_moves(piece2, boost\_used=boost\_used)

                    for move2, skip2 in valid\_moves2.items():

                        temp\_board2 = deepcopy(new\_board)

                        temp\_piece2 = temp\_board2.get\_piece(piece2.row, piece2.col)

                        new\_board2 = simulate\_move(temp\_piece2, move2, temp\_board2, game, skip2)

                        moves.append((new\_board2, False))

            else:

                moves.append((new\_board, False))

        # Boost

        if boost\_available:

            boost\_moves = board.get\_valid\_moves(piece, allow\_boost=True, boost\_used=boost\_used)

            for move, skip in boost\_moves.items():

                if move not in valid\_moves:

                    temp\_board = deepcopy(board)

                    temp\_piece = temp\_board.get\_piece(piece.row, piece.col)

                    new\_board = simulate\_move(temp\_piece, move, temp\_board, game, skip)

                    moves.append((new\_board, True))

    return moves

def draw\_moves(game, board, piece):

    valid\_moves = board.get\_valid\_moves(piece)

    board.draw(game.win)

    pygame.draw.circle(game.win, (0, 255, 0), (piece.x, piece.y), 50, 5)

    game.draw\_valid\_moves(valid\_moves.keys())

    pygame.display.update()

קוד main

import pygame

from checkers.constants import WIDTH, HEIGHT, SQUARE\_SIZE, RED, WHITE

from checkers.game import Game

from minimax.algorithm import minimax

import tkinter as tk

from tkinter import messagebox

# initialize tkinter without opening a window

tk\_root = tk.Tk()

tk\_root.withdraw()

FPS = 60

pygame.init()

pygame.font.init()

WIN = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption('Checkers')

BOOST\_BTN = pygame.Rect(WIDTH - 140, HEIGHT - 50, 120, 40)

def get\_row\_col\_from\_mouse(pos):

    x, y = pos

    row = y // SQUARE\_SIZE

    col = x // SQUARE\_SIZE

    return row, col

def draw\_winner(win, color):

    font = pygame.font.SysFont(None, 72)

    text = f"{'RED' if color == RED else 'WHITE'} WINS!"

    text\_surface = font.render(text, True, (255, 255, 255))

    text\_rect = text\_surface.get\_rect(center=(WIDTH/2, HEIGHT/2))

    win.blit(text\_surface, text\_rect)

    pygame.display.update()

    pygame.time.wait(3000)

def draw\_message(win, message):

    font = pygame.font.SysFont(None, 36)

    text\_surface = font.render(message, True, (255, 255, 255))

    text\_rect = text\_surface.get\_rect(center=(WIDTH/2, HEIGHT - 70))

    pygame.draw.rect(win, (0, 0, 0), (0, HEIGHT - 90, WIDTH, 90))

    win.blit(text\_surface, text\_rect)

def draw\_boost\_button(win, game):

    if game.turn == RED and not game.boost\_used[RED]:

        pygame.draw.rect(win, (70, 130, 180), BOOST\_BTN)

        font = pygame.font.SysFont(None, 28)

        text = font.render("Use Boost", True, (255, 255, 255))

        text\_rect = text.get\_rect(center=BOOST\_BTN.center)

        win.blit(text, text\_rect)

def main():

    run = True

    clock = pygame.time.Clock()

    game = Game(WIN)

    boost\_mode = False

    boost\_button\_visible = True

    ai\_boost\_alert\_shown = False

    while run:

        clock.tick(FPS)

        if game.turn == WHITE:

            value, new\_board = minimax(game.get\_board(), 2, WHITE, game)

            if new\_board:

                new\_board.last\_boost\_used = game.detect\_ai\_boost(game.get\_board(), new\_board)

                game.ai\_move(new\_board)

            else:

                continue

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                run = False

            if event.type == pygame.MOUSEBUTTONDOWN:

                pos = pygame.mouse.get\_pos()

                if BOOST\_BTN.collidepoint(pos) and boost\_button\_visible and not game.boost\_used[RED] and game.turn == RED:

                    boost\_mode = True

                    boost\_button\_visible = False

                    draw\_message(WIN, "Boost mode activated!")

                else:

                    row, col = get\_row\_col\_from\_mouse(pos)

                    use\_twice = event.button == 2 and not game.twice\_used[RED]

                    if game.turn == RED:

                        game.select(row, col, boost\_mode, use\_twice)

                        boost\_mode = False

        game.update()

        if game.turn == RED and not game.boost\_used[RED] and boost\_button\_visible:

            draw\_boost\_button(WIN, game)

        # Game messages and popups

        if game.boost\_just\_used[RED]:

            draw\_message(WIN, "Boost used!")

        elif game.boost\_just\_used[WHITE] and not ai\_boost\_alert\_shown:

            messagebox.showinfo("AI Boost", "AI used Boost!")

            game.boost\_just\_used[WHITE] = False

            ai\_boost\_alert\_shown = True

        if game.twice\_pending[RED]:

            messagebox.showinfo("Twice Turn", "You get another turn (Twice)!")

            game.twice\_pending[RED] = False

        elif game.twice\_pending[WHITE]:

            messagebox.showinfo("Twice Turn", "AI gets another turn (Twice)!")

            game.twice\_pending[WHITE] = False

            # AI does the next move immediately

            value, new\_board = minimax(game.get\_board(), 2, WHITE, game)

            if new\_board:

                new\_board.last\_boost\_used = game.detect\_ai\_boost(game.get\_board(), new\_board)

                game.ai\_move(new\_board)

        if game.just\_got\_king:

            if game.turn == RED:

                messagebox.showinfo("King crowned!", "You got a King! You get another turn (Twice)!")

            else:

                messagebox.showinfo("King crowned!", "AI got a King! It gets another turn (Twice)!")

                game.twice\_pending[WHITE] = True

            game.just\_got\_king = False

        if game.turn == RED and not game.has\_valid\_moves(RED):

            draw\_message(WIN, "No valid moves left. WHITE wins!")

            pygame.display.update()

            pygame.time.wait(2000)

            messagebox.showinfo("Game Over", "No valid moves for RED. WHITE wins!")

            run = False

        elif game.turn == WHITE and not game.has\_valid\_moves(WHITE):

            draw\_message(WIN, "No valid moves left. RED wins!")

            pygame.display.update()

            pygame.time.wait(2000)

            messagebox.showinfo("Game Over", "No valid moves for WHITE. RED wins!")

            run = False

        winner = game.winner()

        if winner is not None:

            draw\_winner(WIN, winner)

            messagebox.showinfo("Game Over", f"{'RED' if winner == RED else 'WHITE'} wins!")

            run = False

        pygame.display.update()

    pygame.quit()

main()