Yuen Hay Russell Jordan

DGD 1

Lecturer: Mr Abdallah Ousman Peerally

Heuristics Algorithms

These are used to evaluate how "good" a board position is during AI decision-making. They're combined in the evaluateBoard() function:

```
int evaluateBoard() {
    return materialHeuristic() + positionalHeuristic();
}
```

materialHeuristic()

What it does:

This evaluates the total material on the board. It adds or subtracts points based on how many and what kind of pieces each player has.

{PAWN, 1}, {KNIGHT, 3}, {BISHOP, 3}, {ROOK, 5}, {QUEEN, 9}, {KING, 10000}

How it helps:

Encourages the AI to capture valuable pieces (e.g. queens, rooks).

Helps it avoid sacrificing major pieces unless there's good reason.

positionalHeuristic()

What it does:

This rewards pieces that are in strategic positions, specifically the 4 center squares (d4, d5, e4, e5):

```
for (int i = 3; i <= 4; ++i)
    for (int j = 3; j <= 4; ++j)
        if (board[i][j].color == WHITE) score++;
        else if (board[i][j].color == BLACK) score--;
```

How it helps:

Encourages board control by occupying the center, a key principle in chess.

Adds more nuance than just chasing material.

Rule Based Algorithm

These control the movement rules for each piece. They're used in isMoveLegal() and generateAllLegalMoves().

What it does:

Checks if a move follows the legal rules of chess for:

Pawns (including 2-square move, diagonal capture)

Knights (L-shape)

Bishops (diagonals, path clear)

Rooks (straight lines, path clear)

Queens (bishop + rook logic)

Kings (1 square in any direction)

How it helps:

Enforces real chess rules

Prevents illegal moves from being generated or executed

Pruning-Based Algorithm (Alpha-Beta Pruning)

This is an optimization of the minimax algorithm, implemented in your code as alphabeta():

int alphabeta(int depth, Color player, int alpha, int beta, bool maximizingPlayer)

What it does:

Searches ahead through possible moves (up to depth = 2)

}

Prunes branches that can’t possibly affect the final decision (beta ≤ alpha)

Why it's important:

Avoids checking every single move all the way to the bottom

Saves massive time, making deeper AI search possible

Helps the AI play faster and stronger without brute force

Code for Chess AI:

```
#include <SFML/Graphics.hpp>
#include <iostream>
#include <vector>
#include <unordered_map>
#include <limits>
#include <cmath>
#include <random>

enum Piece { EMPTY, PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING };
enum Color { NONE, WHITE, BLACK };

struct ChessPiece {
   Piece piece;
   Color color;
};

struct Move {
    int fromRow, fromCol, toRow, toCol;
};

const int BOARD_SIZE = 8;
ChessPiece board[BOARD_SIZE][BOARD_SIZE];

// Check if position inside board bounds
bool isInsideBoard(int r, int c) {
    return r >= 0 && r < BOARD_SIZE && c >= 0 && c < BOARD_SIZE;
}

// Check if move is legal for given color
bool isMoveLegal(const Move& move, Color color) {
    if (!isInsideBoard(move.fromRow, move.fromCol) || !isInsideBoard(move.toRow, move.toCol)) return false;
    ChessPiece from = board[move.fromRow][move.fromCol];
    ChessPiece to = board[move.toRow][move.toCol];
    if (from.color != color || to.color == color) return false;

    int dr = move.toRow - move.fromRow;
    int dc = move.toCol - move.fromCol;

    switch (from.piece) {
        case PAWN: {
            int dir = (color == WHITE ? -1 : 1);
            if (dc == 0 && dr == dir && to.piece == EMPTY) return true;
            if (dc == 0 && dr == 2 * dir && move.fromRow == (color == WHITE ? 6 : 1) &&
                board[move.fromRow + dir][move.fromCol].piece == EMPTY && to.piece == EMPTY) return true;
            if (std::abs(dc) == 1 && dr == dir && to.piece != EMPTY) return true;
            break;
        }
        case KNIGHT:
            return (std::abs(dr) == 2 && std::abs(dc) == 1) || (std::abs(dr) == 1 && std::abs(dc) == 2);
        case BISHOP:
            if (std::abs(dr) != std::abs(dc)) return false;
            for (int i = 1; i < std::abs(dr); ++i)
                if (board[move.fromRow + i * (dr > 0 ? 1 : -1)][move.fromCol + i * (dc > 0 ? 1 : -1)].piece != EMPTY)
                    return false;
            return true;
        case ROOK:
            if (dr != 0 && dc != 0) return false;
            for (int i = 1; i < std::max(std::abs(dr), std::abs(dc)); ++i) {
                int r = move.fromRow + i * (dr != 0 ? (dr > 0 ? 1 : -1) : 0);
                int c = move.fromCol + i * (dc != 0 ? (dc > 0 ? 1 : -1) : 0);
                if (board[r][c].piece != EMPTY) return false;
            }
            return true;
        case QUEEN: {
            board[move.fromRow][move.fromCol].piece = BISHOP;
            bool bishopLegal = isMoveLegal(move, color);
            board[move.fromRow][move.fromCol].piece = ROOK;
            bool rookLegal = isMoveLegal(move, color);
            board[move.fromRow][move.fromCol].piece = QUEEN;
            return bishopLegal || rookLegal;
        }
        case KING:
            return std::abs(dr) <= 1 && std::abs(dc) <= 1;
        default:
            return false;
    }
    return false;
```

```
}

// Heuristics

int materialHeuristic() {
    std::unordered_map<Piece, int> values = {
        {PAWN, 1}, {KNIGHT, 3}, {BISHOP, 3}, {ROOK, 5}, {QUEEN, 9}, {KING, 10000}
    };
    int score = 0;
    for (int i = 0; i < BOARD_SIZE; ++i)
        for (int j = 0; j < BOARD_SIZE; ++j) {
            ChessPiece cp = board[i][j];
            if (cp.piece != EMPTY) {
                int value = values[cp.piece];
                score += (cp.color == WHITE ? value : -value);
            }
        }
    return score;
}

int positionalHeuristic() {
    int score = 0;
    // Central squares d4,d5,e4,e5 = 3,3,4,4
    for (int i = 3; i <= 4; ++i)
        for (int j = 3; j <= 4; ++j) {
            if (board[i][j].color == WHITE) score++;
            else if (board[i][j].color == BLACK) score--;
        }
    return score;
}
```

```
int evaluateBoard() {
    return materialHeuristic() + positionalHeuristic();
}

// Check king presence for game end
bool kingExists(Color color) {
    for (int i = 0; i < BOARD_SIZE; ++i)
        for (int j = 0; j < BOARD_SIZE; ++j)
            if (board[i][j].piece == KING && board[i][j].color == color)
                return true;
    return false;
}

std::vector<Move> generateAllLegalMoves(Color color) {
    std::vector<Move> moves;
    for (int i = 0; i < BOARD_SIZE; ++i)
        for (int j = 0; j < BOARD_SIZE; ++j) {
            if (board[i][j].color == color) {
                for (int x = 0; x < BOARD_SIZE; ++x) {
                    for (int y = 0; y < BOARD_SIZE; ++y) {
                        Move move = { i, j, x, y };
                        if (isMoveLegal(move, color)) {
                            moves.push_back(move);
                        }
                    }
                }
            }
        }
    return moves;
}
```

```
void makeMove(const Move& move) {
    board[move.toRow][move.toCol] = board[move.fromRow][move.fromCol];
    board[move.fromRow][move.fromCol] = { EMPTY, NONE };
}

void undoMove(const Move& move, ChessPiece captured) {
    board[move.fromRow][move.fromCol] = board[move.toRow][move.toCol];
    board[move.toRow][move.toCol] = captured;
}

// Alpha-beta pruning minimax
int alphabeta(int depth, Color player, int alpha, int beta, bool maximizingPlayer) {
    if (depth == 0 || !kingExists(WHITE) || !kingExists(BLACK)) {
        return evaluateBoard();
    }

    Color opponent = (player == WHITE ? BLACK : WHITE);
    std::vector<Move> moves = generateAllLegalMoves(player);

    if (moves.empty()) {
        // No moves => checkmate or stalemate
        return evaluateBoard();
    }

    if (maximizingPlayer) {
        int maxEval = std::numeric_limits<int>::min();
        for (auto& move : moves) {
            ChessPiece captured = board[move.toRow][move.toCol];
            makeMove(move);
            int eval = alphabeta(depth - 1, opponent, alpha, beta, false);
            undoMove(move, captured);
            maxEval = std::max(maxEval, eval);
            alpha = std::max(alpha, eval);
            if (beta <= alpha)
                break;
        }
        return maxEval;
    } else {
        int minEval = std::numeric_limits<int>::max();
        for (auto& move : moves) {
            ChessPiece captured = board[move.toRow][move.toCol];
            makeMove(move);
            int eval = alphabeta(depth - 1, opponent, alpha, beta, true);
            undoMove(move, captured);
            minEval = std::min(minEval, eval);
            beta = std::min(beta, eval);
            if (beta <= alpha)
                break;
        }
        return minEval;
    }
}

Move selectBestMove(Color color) {
    std::vector<Move> bestMoves;
    int bestScore = std::numeric_limits<int>::min();

    std::vector<Move> moves = generateAllLegalMoves(color);
    if (moves.empty()) {
        return {0, 0, 0, 0};
    }
    Color opponent = (color == WHITE ? BLACK : WHITE);

    for (auto& move : moves) {
        ChessPiece captured = board[move.toRow][move.toCol];
        makeMove(move);
        int score = alphabeta(2, opponent, std::numeric_limits<int>::min(),
std::numeric_limits<int>::max(), false);
        undoMove(move, captured);

        if (score > bestScore) {
            bestScore = score;
            bestMoves.clear();
            bestMoves.push_back(move);
        } else if (score == bestScore) {
            bestMoves.push_back(move);
        }
    }

    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, bestMoves.size() - 1);
    return bestMoves[dis(gen)];
}

// Initialize board with pieces
void initializeBoard() {
    for (int i = 0; i < BOARD_SIZE; ++i)
        for (int j = 0; j < BOARD_SIZE; ++j)
            board[i][j] = { EMPTY, NONE };

    for (int j = 0; j < BOARD_SIZE; ++j) {
        board[1][j] = { PAWN, BLACK };
        board[6][j] = { PAWN, WHITE };
    }
    board[0][0] = board[0][7] = { ROOK, BLACK };
    board[0][1] = board[0][6] = { KNIGHT, BLACK };
    board[0][2] = board[0][5] = { BISHOP, BLACK };
    board[0][3] = { QUEEN, BLACK };
    board[0][4] = { KING, BLACK };

    board[7][0] = board[7][7] = { ROOK, WHITE };
    board[7][1] = board[7][6] = { KNIGHT, WHITE };
    board[7][2] = board[7][5] = { BISHOP, WHITE };
    board[7][3] = { QUEEN, WHITE };
    board[7][4] = { KING, WHITE };
}

// Draw the chessboard squares and pieces
void drawBoard(sf::RenderWindow& window, const std::vector<sf::RectangleShape>& squares) {
    for (const auto& square : squares)
        window.draw(square);

    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            ChessPiece cp = board[i][j];
            if (cp.piece != EMPTY) {
                switch (cp.piece) {
                    case PAWN: {
                        sf::CircleShape circle(30);
                        circle.setPosition(j * 80 + 10, i * 80 + 10);
                        circle.setFillColor(cp.color == WHITE ? sf::Color::White : sf::Color::Black);
                        window.draw(circle);
                        break;
                    }
                    case KNIGHT: {
                        sf::RectangleShape rect(sf::Vector2f(60, 30));
                        rect.setPosition(j * 80 + 10, i * 80 + 25);
                        rect.setFillColor(cp.color == WHITE ? sf::Color::White : sf::Color::Black);
                        window.draw(rect);
                        break;
                    }
                    case BISHOP: {
                        sf::CircleShape smallCircle(20);
                        smallCircle.setPosition(j * 80 + 20, i * 80 + 20);
                        smallCircle.setFillColor(cp.color == WHITE ? sf::Color::White : sf::Color::Black);
                        window.draw(smallCircle);
                        break;
                    }
                    case ROOK: {
                        sf::RectangleShape tallRect(sf::Vector2f(30, 60));
                        tallRect.setPosition(j * 80 + 25, i * 80 + 10);
                        tallRect.setFillColor(cp.color == WHITE ? sf::Color::White : sf::Color::Black);
                        window.draw(tallRect);
                        break;
                    }
                    case QUEEN: {
                        sf::CircleShape hexagon(30, 6);
                        hexagon.setPosition(j * 80 + 10, i * 80 + 10);
                        hexagon.setFillColor(cp.color == WHITE ? sf::Color::White : sf::Color::Black);
                        window.draw(hexagon);
                        break;
                    }
                    case KING: {
```

```
                        sf::CircleShape diamond(30, 4);
                        diamond.setPosition(j * 80 + 10, i * 80 + 10);
                        diamond.setFillColor(cp.color == WHITE ? sf::Color::White : sf::Color::Black);
                        window.draw(diamond);
                        break;
                    }
                    default:
                        break;
                }
            }
        }
    }
}

int main() {
    initializeBoard();

    sf::RenderWindow window(sf::VideoMode(640, 640), "Chess AI - Player vs AI");
    window.setFramerateLimit(60);

    // Pre-create squares for board background
    std::vector<sf::RectangleShape> squares;
    for (int i = 0; i < BOARD_SIZE; ++i)
        for (int j = 0; j < BOARD_SIZE; ++j) {
            sf::RectangleShape square(sf::Vector2f(80, 80));
            square.setPosition(j * 80, i * 80);
            square.setFillColor((i + j) % 2 == 0 ? sf::Color(238, 238, 210) : sf::Color(118, 150, 86));
            squares.push_back(square);
        }

    Color currentPlayer = WHITE;
    bool pieceSelected = false;
    int selectedRow = -1, selectedCol = -1;

    sf::Clock aiClock;
    const float aiMoveDelay = 1.0f; // 1 second delay for AI moves

    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();

            // Handle player input when it's white's turn
            if (currentPlayer == WHITE && event.type == sf::Event::MouseButtonPressed && event.mouseButton.button == sf::Mouse::Left) {
                int col = event.mouseButton.x / 80;
                int row = event.mouseButton.y / 80;

                if (!pieceSelected) {
                    if (isInsideBoard(row, col) && board[row][col].color == WHITE) {
                        pieceSelected = true;
                        selectedRow = row;
                        selectedCol = col;
                    }
                } else {
                    Move playerMove = { selectedRow, selectedCol, row, col };
                    if (isMoveLegal(playerMove, WHITE)) {
                        makeMove(playerMove);

                        if (!kingExists(BLACK)) {
                            std::cout << "You captured the Black King! You Win!\n";
                            window.close();
                            break;
                        }

                        currentPlayer = BLACK;
                        pieceSelected = false;
                        aiClock.restart();
                    } else {
                        // Change selected piece if clicking another white piece
                        if (isInsideBoard(row, col) && board[row][col].color == WHITE) {
                            selectedRow = row;
                            selectedCol = col;
                        } else {
                            pieceSelected = false;
                        }
                    }
                }
            }
        }

        // AI move if it's black's turn and delay passed
        if (currentPlayer == BLACK && aiClock.getElapsedTime().asSeconds() > aiMoveDelay) {
            Move bestMove = selectBestMove(BLACK);
            if (bestMove.fromRow == bestMove.toRow && bestMove.fromCol == bestMove.toCol) {
                std::cout << "Black has no legal moves. Game Over.\n";
                window.close();
                break;
            }

            makeMove(bestMove);
```

```
            if (!kingExists(WHITE)) {
                std::cout << "Black captured your King! You Lose!\n";
                window.close();
                break;
            }

            currentPlayer = WHITE;
            aiClock.restart();
        }

        // Draw everything
        window.clear();
        drawBoard(window, squares);

        // Highlight selected piece
        if (pieceSelected) {
            sf::RectangleShape highlight(sf::Vector2f(80, 80));
            highlight.setPosition(selectedCol * 80, selectedRow * 80);
            highlight.setFillColor(sf::Color(255, 255, 0, 100)); // translucent yellow
            window.draw(highlight);
        }

        window.display();
    }

    return 0;
}
```