Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Fundamentals of Artificial Intelligence

Laboratory work #3

AI Chess Player – the Minimax Algorithm

Author:
Popa Eugeniu
std. gr. FAF-202

Supervisor:
Diana Marusic

Chișinău 2023

1 Task 1-2

Implement the MiniMax algorithm. Implement Alpha-Beta Prunning.

```python
def minimax(board, depth, alpha, beta, max_player, save_move, data):
    # Base case: If we've reached the desired depth or the game is over, evaluate the current state.
    if depth == 0 or board.is_terminal():
        data[1] = evaluate_board(board)  # Store the evaluation in data[1].
        return data
    if max_player:
        max_eval = -math.inf
        for piece, move in generate_possible_moves(board, max_player):
            board.make_move(piece, move[0], move[1], keep_history=True)  # Make a move on the board.
            evaluation = minimax(board, depth - 1, alpha, beta, False, False, data)[1]
            if save_move:
                if evaluation >= max_eval:
                    if evaluation > data[1]:
                        data.clear()  # Clear previous best moves since we found a better one.
                        data[1] = evaluation  # Update the best evaluation.
                        data[0] = [(piece, move, evaluation)]  # Store the current best move.
                    elif evaluation == data[1]:
                        data[0].append((piece, move, evaluation))  # Add to the list of best moves if tied.
            board.unmake_move(piece)  # Undo the move on the board.
            max_eval = max(max_eval, evaluation)  # Update the maximum evaluation.
            alpha = max(alpha, evaluation)  # Update the alpha value for pruning.
            if beta <= alpha:
                break  # Beta pruning: No need to explore further if beta is less than or equal to alpha.
        return data
    else:
        min_eval = math.inf
        for piece, move in generate_possible_moves(board, max_player):
            board.make_move(piece, move[0], move[1], keep_history=True)  # Make a move on the board.
            evaluation = minimax(board, depth - 1, alpha, beta, True, False, data)[1]
            board.unmake_move(piece)  # Undo the move on the board.
            min_eval = min(min_eval, evaluation)  # Update the minimum evaluation.
            beta = min(beta, evaluation)  # Update the beta value for pruning.
            if beta <= alpha:
                break  # Alpha pruning: No need to explore further if alpha is greater than or equal to beta.
        return data


def generate_possible_moves(board, max_player):
    moves = []
    for i in range(8):
        for j in range(8):
            piece = board[i][j]

            # Check if the current position contains a chess piece of the correct color.
            if isinstance(piece, ChessPiece) and (max_player == (piece.color != board.get_player_color())):
```

```
                valid_moves = piece.filter_moves(piece.get_moves(board), board)  # Generate valid moves for the piece.
                for move in valid_moves:
                    moves.append((piece, move))  # Add the piece-move pair to the list of possible moves.

    return moves
```

This is an implementation of a chess-playing engine using the minimax algorithm with alpha-beta pruning. It also includes functions for generating possible moves, evaluating the game state, and making moves for an AI player.

1. Minimax Algorithm with Alpha-Beta Pruning: - The "minimax" function is the heart of the chess engine. It is a recursive function that performs a depth-limited search of the game tree to find the best move for the AI player. - It takes several parameters, including the current board state ('board'), the search depth ('depth'), alpha and beta values for pruning, whether it's the maximizing player's turn ('max_player'), a flag to save the move ('save_move'), and a data structure to store the best move ('data').

2. Base House: - The function starts with a base case. If the depth is zero or the game is in a terminal state (e.g., checkmate or stalemate), it evaluates the current board state using the 'evaluate_board' function and stores the result in 'data[1]'.

3. Maximizing Player (AI): - If it's the AI player's turn (maximizing player), the function searches for the move that maximizes the AI's chances of winning. - It iterates through the possible moves generated by the 'generate_possible_moves' function for the AI player. - For each move, it temporarily makes the move on the board, recursively calls the 'minimax' function with the depth reduced by 1, and updates alpha and beta values accordingly. - The function keeps track of the best move(s) based on the evaluation scores. If a move with a higher score is found, it clears the previous best moves and updates the data structure with the new best move.

4. Minimizing Player (Opponent): - If it's the opponent's turn, the function searches for the move that minimizes the AI's chances of winning. The process is similar to the maximizing player, but it minimizes the evaluation scores.

5. Alpha-Beta Pruning: - Alpha-beta pruning is used to reduce the number of nodes evaluated in the minimax tree. It helps to improve the efficiency of the search. - If the beta value becomes less than or equal to the alpha value during the search, pruning occurs, and the function breaks out of the loop, as there is no need to explore further.

6. Generate Possible Moves: - The 'generate_possible_moves' function generates all possible moves for a given player on the current board. - It iterates through the board and identifies pieces belonging to the player whose turn it is. - For each piece, it calls the 'filter_moves' function to determine valid moves (moves that don't put the player's king in check) and appends the piece-move pair to the list of possible moves.

7. AI Move Selection: - The 'get_ai_move' function is responsible for determining the best move for the AI player. - It iterates through different depths to search for the best move. At each depth, it calls the 'minimax' function to find the best move(s) with the highest evaluation score. - It selects a random move from the best moves to introduce an element of randomness in the AI's decision-making.

8. Random Move for Opponent: - There's also a 'get_random_move' function that makes a random move for the opponent player. This function is useful for simulating a complete game when the AI is playing against a human or another AI.

This code can be used as the foundation for building a chess-playing engine. It demonstrates the core concepts of the minimax algorithm, alpha-beta pruning, move generation, and move evaluation, which are essential for creating a competitive chess-playing program.

2 Task 3

Implement an improved scoring (evaluation) method for MiniMax. For example, you could add values like KingSafetyValue, MobilityValue (nr. of legal moves to each side), PawnStructureValue (can include penalties for isolated pawns, doubled pawns, and bonuses for passed pawns or a strong pawn chain), etc.

```
def evaluate_board(board):
    # Scoring factors
    material_weight = 1.0
    positional_weight = 1.0
    king_safety_weight = 1.0
    mobility_weight = 0.1
    pawn_structure_weight = 0.1

    # Player colors
    player_color = board.get_player_color()
    opponent_color = 'black' if player_color == 'white' else 'white'

    # Evaluate material value
    material_score = evaluate_material_value(board)

    # Evaluate positional value
    player_positional_score = evaluate_positional_value(board, player_color)
    opponent_positional_score = evaluate_positional_value(board, opponent_color)
    positional_score = round(player_positional_score - opponent_positional_score, 3)

    # Evaluate king safety
    king_safety_score = evaluate_king_safety(board)

    # Evaluate mobility
    player_mobility_score = evaluate_mobility(board, player_color)
    opponent_mobility_score = evaluate_mobility(board, opponent_color)

    # Evaluate pawn structure
    pawn_structure_score = evaluate_pawn_structure(board)

    # Combine the scores with the specified weights
    total_score = (
            material_weight * material_score +
            positional_weight * positional_score +
            king_safety_weight * king_safety_score +
            mobility_weight * player_mobility_score +
            mobility_weight * opponent_mobility_score +
            pawn_structure_weight * pawn_structure_score
    )

    total_score = round(total_score, 3)
    print("\n")
    print(f"Material score: {material_score}")
    print(f"Positional score: {positional_score}")
    print(f"King safety score: {king_safety_score}")
    print(f"Player mobility score: {player_mobility_score}")
    print(f"Enemy mobility score: {opponent_mobility_score}")
    print(f"Pawn structure score: {pawn_structure_score}")
    print(f"Total score: {total_score}")

    return total_score


```

```
def evaluate_material_value(board):
    piece_values = {
        Pawn: 1,
        Knight: 3,
        Bishop: 3,
        Rook: 5,
        Queen: 9,
        King: 0,
    }

    material_score = 0
    for i in range(8):
        for j in range(8):
            piece = board[i][j]
            if isinstance(piece, ChessPiece):
                if piece.color == board.get_player_color():
                    material_score += piece_values[type(piece)]
                else:
                    material_score -= piece_values[type(piece)]
    return material_score


def evaluate_positional_value(board, player_color):
    # Positional values for each square on the chessboard
    positional_values = [
        [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2],
        [0.2, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.2],
        [0.2, 0.5, 0.7, 0.7, 0.7, 0.7, 0.5, 0.2],
        [0.2, 0.5, 0.7, 1.0, 1.0, 0.7, 0.5, 0.2],
        [0.2, 0.5, 0.7, 1.0, 1.0, 0.7, 0.5, 0.2],
        [0.2, 0.5, 0.7, 0.7, 0.7, 0.7, 0.5, 0.2],
        [0.2, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.2],
        [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
    ]
    position_score = 0
    for i in range(8):
        for j in range(8):
            piece = board[i][j]
            if isinstance(piece, ChessPiece) and piece.color == player_color:
                position_score += positional_values[i][j]
    return position_score


def evaluate_king_safety(board):
    king_safety_score = 0
    player_color = board.get_player_color()
    opponent_color = 'black' if player_color == 'white' else 'white'
    player_king = None
    for i in range(8):
        for j in range(8):
            piece = board[i][j]
            if isinstance(piece, King):
                if piece.color == player_color:
                    player_king = (i, j)
    if player_king:
        player_king_row, player_king_col = player_king
        # Factor: Presence of enemy queen or other enemy pieces near player's king
        # Iterate two squares in all directions
        for i in range(-2, 3):
            for j in range(-2, 3):
```

```
                # Calculate the target cell position relative to the player's king
                target_row = player_king_row + i
                target_col = player_king_col + j
                # Check if the target cell position is within the chessboard bounds
                if 0 <= target_row < 8 and 0 <= target_col < 8:
                    piece = board[target_row][target_col]
                    # Check if the piece in the target cell is an enemy queen
                    if isinstance(piece, Queen) and piece.color == opponent_color:
                        king_safety_score -= 0.5
                    # Check if the piece in the target cell is another enemy piece
                    elif isinstance(piece, ChessPiece) and piece.color == opponent_color:
                        king_safety_score -= 0.3
        # Factor: Presence of player pieces near player's king
        # Iterate two squares in all directions
        for i in range(-1, 2):
            for j in range(-1, 2):
                # Calculate the target cell position relative to the player's king
                target_row = player_king_row + i
                target_col = player_king_col + j
                # Check if the target cell position is within the chessboard bounds
                if 0 <= target_row < 8 and 0 <= target_col < 8:
                    piece = board[target_row][target_col]
                    # Check if the piece in the target cell is a player piece
                    if isinstance(piece, ChessPiece) and not (isinstance(piece, King)) and piece.color == player_color:
                        king_safety_score += 0.1
        # Factor: Pawn Shield (Checking for pawns in front of the player's king)
        front_piece = board[player_king_row + 1][player_king_col]
        left_front_piece = board[player_king_row + 1][player_king_col - 1]
        right_front_piece = board[player_king_row + 1][player_king_col + 1]
        if player_king_row >= 0 and isinstance(front_piece, Pawn):
            king_safety_score += 1
        if player_king_row >= 0 and isinstance(left_front_piece, Pawn):
            king_safety_score += 1
        if player_king_row >= 0 and isinstance(right_front_piece, Pawn):
            king_safety_score += 1
    king_safety_score = round(king_safety_score, 3)
    return king_safety_score


def evaluate_mobility(board, player_color):
    if player_color == board.get_player_color():
        # Calculate the mobility for the player
        player_moves = 0
        for i in range(8):
            for j in range(8):
                piece = board[i][j]
                if isinstance(piece, ChessPiece) and piece.color == player_color:
                    # Count the legal moves for each of the player's pieces
                    player_moves += len(piece.get_moves(board))
        return player_moves
    else:
        # Calculate the mobility for the opponent
        opponent_moves = 0
        for i in range(8):
            for j in range(8):
                piece = board[i][j]
                if isinstance(piece, ChessPiece) and piece.color == player_color:
                    # Count the legal moves for each of the opponent's pieces
```

```
                    opponent_moves += len(piece.get_moves(board))
        return opponent_moves


def evaluate_pawn_structure(board):
    isolated_pawns = 0  # Counter for isolated pawns
    strong_chain_pawns = 0  # Counter for pawns in strong pawn chains
    # Iterate through the chessboard
    for i in range(8):
        for j in range(8):
            piece = board[i][j]
            if isinstance(piece, Pawn) and piece.color == board.get_player_color():
                is_isolated = True  # Flag to detect isolated pawns
                is_supported = False  # Flag to detect pawns in strong chains
                # Check for friendly pawns on adjacent files (left and right)
                if j > 0:
                    left_piece = board[i][j - 1]
                    if isinstance(left_piece, Pawn) and left_piece.color == board.
    get_player_color():
                        is_isolated = False  # The pawn is not isolated
                        is_supported = True  # It's part of a strong pawn chain
                if j < 7:
                    right_piece = board[i][j + 1]
                    if isinstance(right_piece, Pawn) and right_piece.color == board.
    get_player_color():
                        is_isolated = False  # The pawn is not isolated
                        is_supported = True  # It's part of a strong pawn chain
                # Update counters based on the isolated and supported pawns
                if is_isolated:
                    isolated_pawns -= 1  # Isolated pawns reduce the score
                if is_supported:
                    strong_chain_pawns += 1  # Pawns in strong chains increase the
    score
    # Calculate the overall pawn structure score by summing isolated and strong chain
     pawns
    pawn_structure_score = isolated_pawns + strong_chain_pawns
    return pawn_structure_score
```

These evaluations are used to assess the current state of the game and help an AI player make informed moves. Here's an explanation of the functions:

1. evaluate_board(board): - This function is the main entry point for evaluating the chessboard. - It uses various scoring factors to assess the board's quality, such as material balance, king safety, mobility, and pawn structure. - The function combines these scores by applying specified weights to each factor and returns a total score. - It also prints individual component scores for debugging and analysis.

2. evaluate_material_value(board): - This function calculates the material value score on the chessboard. - It assigns values to each chess piece (Pawn, Knight, Bishop, Rook, Queen, King) and computes the material score by counting the number of each piece type for both players. - The score is adjusted based on which player's turn it is, and the result is returned.

2. evaluate_positional_value(board): - This function calculates the positional value score on the chessboard.

3. evaluate_king_safety(board): - This function assesses the safety of the player's king. - It checks for factors that could affect the king's safety, such as the presence of enemy pieces near the king. - The function considers the proximity of enemy queen and other pieces as well as the presence of friendly pieces near the king. - It assigns scores based on these factors, and the result is returned.

4. evaluate_mobility(board, player_color): - This function evaluates the mobility of the player's

pieces. - It takes the player's color and counts the legal moves available for all the player's pieces on the board. - The score is returned based on the player's mobility.

5. evaluate_pawn_structure(board): - This function assesses the pawn structure on the board. - It looks for isolated pawns and pawns in strong chains for the player. - Isolated pawns are pawns without friendly pawns on adjacent files, which reduces the score. - Pawns in strong chains, with friendly pawns on adjacent files, increase the score. - The function calculates the overall pawn structure score based on these criteria and returns it.

These evaluation functions are essential for building a chess-playing AI. They provide a way to assess the game state and make informed decisions about the best move to make. The individual components, such as material balance, king safety, and pawn structure, are combined to provide a comprehensive assessment of the board's position.

3 Task 4

Add at least one improvement to the MiniMax algorithm from the following list: Progressive Deepening, Transposition Tables, Opening Books, Move Ordering, Aspiration, Window, etc.

```
def get_ai_move(board):
    # Initialize a variable to store the best move
    best_move = None
    # Iterate through different depths
    for depth in range(1, board.depth + 1):
        print(f"Searching at depth {depth}...")
        # Use the minimax function to search for the best move at the current depth
        moves = minimax(board, depth, -math.inf, math.inf, True, True, [[], 0])
        # Check if valid moves were found at the current depth
        if moves and len(moves[0]) > 0:
            # Find the move with the best score
            best_score = max(moves[0], key=lambda x: x[2])[2]
            # Print all moves and their scores
            # print("All moves and scores:")
            # for move in moves[0]:
            #     print(f"Move: {move[0]} to {move[1]} with score {move[2]}")
            # Select a random move from the best moves with the highest score
            piece_and_move = random.choice([move for move in moves[0] if move[2] ==
    best_score])
            # Check if the selected piece and move are not None
            if piece_and_move[0] is not None and len(piece_and_move[1]) > 0 and
    isinstance(piece_and_move[1], tuple):
                # Update the best move if a better move was found
                best_move = piece_and_move
            # Check if the best move is not None and is a valid tuple
            if best_move[0] is not None and len(best_move[1]) > 0 and isinstance(
    best_move[1], tuple):
                piece = best_move[0]
                move = best_move[1]
                # Apply the best move to the board
                board.make_move(piece, move[0], move[1])
                # Print the best move and its score
                print(f"Best move: {piece} to {move} with score {best_score}")
            else:
                print("No valid move found.")
        else:
            print("No valid move found.")
    return True
```

The ‘get_ai_move‘ function is a component of a chess-playing program that uses the Minimax algorithm with Progressive Deepening to make informed and strategic moves. This function is responsible for selecting the best move for the AI player by considering different depths of search in the game tree.

Progressive Deepening:

1. Iterative Depth Search: The function begins by iteratively searching the game tree at increasing depths, starting from depth 1 up to a specified maximum depth. This approach is known as Progressive Deepening. It allows the AI to explore the game tree more deeply in a controlled and systematic manner.

2. Depth Parameter (‘depth’): At each iteration, the ‘depth’ variable is used to control the depth of search, starting from a depth of 1 and increasing with each iteration.

Minimax Algorithm:

3. Minimax Search: For each depth level, the Minimax algorithm is applied to find the best move for the AI player. The Minimax algorithm is a decision-making technique commonly used in games like chess. It explores the game tree to evaluate and select moves that maximize the AI player's chances of winning and minimize the opponent's chances.

4. Alpha-Beta Pruning: Within the Minimax search, the function employs alpha-beta pruning to improve efficiency. Alpha-beta pruning helps eliminate certain branches of the game tree that do not need to be explored further, reducing the number of nodes evaluated.

Selecting the Best Move:

5. Storing the Best Move: As the search progresses, the function keeps track of the best move found so far. The best move is stored in the ‘best_move‘ variable.

6. Selecting the Best Move: Once the search at a specific depth is complete, the function selects the best move based on the highest evaluation score (often referred to as the utility or score of the move). If multiple moves have the same highest score, one is randomly selected from them.

7. Applying the Best Move: The chosen best move is then applied to the chessboard by calling the ‘board.make_move‘ function, updating the game state.

Logging and Output:

8. Logging and Output: The function also provides detailed logging information at each depth, including the moves considered, their associated scores, and the selected best move. This information can be useful for debugging and analyzing the AI's decision-making process.

9. Completion and Return: The function completes its search at all specified depths and returns ‘True’ to indicate that a move has been selected and applied to the board.

Overall, the ‘get_ai_move‘ function combines Progressive Deepening with the Minimax algorithm to gradually explore deeper into the game tree, selecting the best move for the AI player based on the evaluations of different game states.

4 Conclusion

In this laboratory work, we have explored and implemented essential components for building a chess-playing engine. We started by implementing the Minimax algorithm with Alpha-Beta Pruning, which forms the core of the AI's decision-making process. This algorithm allows the AI to evaluate different game states and select the best move while efficiently pruning unnecessary branches of the game tree.

We then extended the capabilities of our AI by introducing an improved scoring (evaluation) method. This method considers factors such as material balance, positional advantages, king safety, mobility, and pawn structure to assess the quality of the current game state. These evaluations provide a holistic view of the board and help the AI make strategic and informed decisions.

Additionally, we implemented Progressive Deepening, which allows our AI to iteratively search the game tree at increasing depths. This approach ensures that the AI explores the most promising

moves first while maintaining control over the computational resources.

Throughout the implementation, we added logging and debugging features to better understand the AI's thought process, including the evaluation scores of various components and the selected best moves at each depth.

In conclusion, this laboratory work has provided a solid foundation for building a competitive chess-playing engine. It combines key elements such as the Minimax algorithm, evaluation functions, and Progressive Deepening to create an AI capable of making strategic decisions and playing chess effectively. This work can serve as a starting point for further enhancements and optimizations in the world of AI-powered chess engines.