Eiaki Morooka

Chess AI Project

Period 7

Second year second semster (7th period)
School of ICT
Metropolia University of Applied Sciences
February 23, 2024

Abstract

This report outlines the development of a chess engine with artificial intelligence, undertaken as a second-year project for the Data Structure and Algorithm class at Metropolia University of Applied Sciences. Utilizing the Minimax algorithm, enhanced with Alpha-Beta pruning, and parallelized through multithreading, my project aimed to create a capable and efficient AI for playing chess. The implementation of Alpha-Beta pruning allowed my engine to ignore less promising moves, streamlining the decision-making process. By incorporating multithreading, I was able to extend the engine's search depth to 7 or 8 levels, improving its strategic foresight within a manageable computation time. This project demonstrates the practical application of complex algorithms and multithreading in developing a competitive yet straightforward chess AI. It reflects my journey through the challenges of algorithm optimization and parallel computing, offering insights into the foundational aspects of AI development in gaming.

Contents

1 Introduction

Chess has always been a classic testbed for artificial intelligence, challenging programmers to replicate or surpass human strategic thinking. My project dives into this challenge by developing a chess engine in C++. The heart of this engine is the Position object, which holds the chessboard's current state and handles all move operations, serving as the crucial element for game logic.

To make the AI, I implemented the Minimax algorithm, enhanced with Alpha-Beta pruning to sift through possible moves more efficiently. This approach helps in evaluating the game's future possibilities, allowing the engine to decide on the best moves by pruning unlikely paths, thus saving valuable computation time.

Given the complexity of chess and the depth of analysis required, I also incorporated multithreading in the main file. This parallelizes the computation, enabling the engine to explore game scenarios up to 7 or 8 moves ahead, striking a balance between depth of search and computational speed.

My goal was to create a chess engine that not only plays well but does so by efficiently navigating the vast sea of possible moves in chess. This project, crafted entirely in C++, was a hands-on journey into the intersection of AI and one of the world's oldest games, pushing the boundaries of what I could achieve with code, algorithms, and a bit of strategic thinking.

2 AI Algorithm

2.1 Minimax

The Minimax algorithm is a recursive strategy used for minimizing the possible loss for a worst-case scenario. When applied to chess, it is used to determine the best move by exploring all possible moves up to a certain depth. The function minimax(position, depth, maximizingPlayer) serves as the core of this strategy. It evaluates the game's positions to decide on the optimal move, considering both the player's and the opponent's perspectives.

At the base of the recursion, when the depth is zero, the algorithm evaluates the current board position using the evaluatePosition(position) function. This evaluation assigns a numerical value to the board state, indicating how favorable it is. Positive values are favorable for White, while negative values favor Black. For non-terminal nodes (depth < 0), the algorithm generates all legal moves from the current position and explores each move's outcomes recursively. When the algorithm operates in the maximizing mode, it looks for the move that will result in the highest evaluation. Conversely, in minimizing mode, it seeks the move that will lead to the lowest evaluation, simulating an opponent trying to minimize the score. Through this process of exploring and evaluating all possible moves up to the given depth, the algorithm identifies the best possible move from the current position, aiming to maximize the player's advantage while minimizing the opponent's.

The following is a speudo-code (for simplicity) of the minimax algorithm I used

```
func minimax(position, depth, maximizingPlayer)
    if depth == 0
        return evaluatePosition(position) # Evaluate position for terminal node

    moves = generateLegalMoves(position) # Generate all legal moves from position

    if maximizingPlayer
        maxEval = -infinity
        for each move in moves
            # Apply move to create new position
            childPosition = applyMove(position, move)
            # Recursive call for child position, toggle maximizingPlayer
            eval = minimax(childPosition, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval
    else
        minEval = +infinity
        for each move in moves
            # Apply move to create new position
            childPosition = applyMove(position, move)
            # Recursive call for child position, toggle maximizingPlayer
            eval = minimax(childPosition, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval

# Auxiliary function to evaluate the position
# Returns a numerical value representing the position's value
# Positive for favorable positions for WHITE, negative for BLACK

func evaluatePosition(position)
    stateValues = position.getStateValue() # Get state values for both players
    if position.turn == BLACK
        return stateValues[WHITE] - stateValues[BLACK]
    else
        return stateValues[BLACK] - stateValues[WHITE]
```

Listing 1: Pseudo-Code of the minimax used

Since it is impossible to do a full search, the evaluation function within the Minimax algorithm plays a crucial role in assessing the strength of a given board position in the game of chess. In this implementation, the evaluation is based on a weighted scoring system for the chess pieces, assigning specific values to each type of piece: Pawns are valued at 1 point, Knights at 3 points, and so on (see table below). This scoring

system reflects the general consensus on the relative value of chess pieces in terms of their mobility and utility in the game. The evaluation function calculates the total score for both White and Black by summing the values of all pieces each side has on the board. The final evaluation score is the difference between these totals, providing a numerical value that represents the board state's favorability. A positive score indicates a position favorable to White, while a negative score favors Black. This method offers a straightforward yet effective way to quantify the advantage or disadvantage at any point in the game, guiding the Minimax algorithm in its decision-making process to select the optimal move.

Chess Piece	Value
Pawn	1
Knight	3
Bishop	3
Rook	5
Queen	9
King	20

Table 1: Values assigned to each chess piece for evaluation.

This could only achieve a depth of 4 or 5 so I added also the alpha-beta pruning.

2.2 Alpha-Beta Pruning

Alpha-Beta Pruning optimizes the Minimax algorithm by introducing two parameters, 'alpha' and 'beta', which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively. By updating these parameters during recursion and employing conditional checks, the algorithm prunes (skips) branches that cannot possibly affect the final decision. This significantly reduces the number of nodes evaluated, leading to faster decision-making without compromising the outcome's accuracy.

```
func minimax_alphabeta(position, depth, maximizingPlayer, alpha, beta)
    # Base case remains the same

    if maximizingPlayer
        # Initialize maxEval as before
        for each move in moves
            # Apply move and recursive call with alpha and beta
            eval = minimax_alphabeta(childPosition, depth - 1, false, alpha, beta)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval) # Update alpha
            if beta <= alpha
                break # Pruning occurs here
        return maxEval
    else
        # Initialize minEval as before
        for each move in moves
            # Apply move and recursive call with alpha and beta
            eval = minimax_alphabeta(childPosition, depth - 1, true, alpha, beta)
            minEval = min(minEval, eval)
            beta = min(beta, eval) # Update beta
            if beta <= alpha
                break # Pruning occurs here
        return minEval
```

Listing 2: Pseudo-Code of Alpha-Beta Pruning Enhancements

Incorporating Alpha-Beta Pruning into the chess engine allowed for an exploration depth of 5 to 6 moves ahead on my personal computer. To further enhance the engine's capability and achieve greater depth in move exploration, I employed parallelization techniques using the C++ Standard Library's threading capabilities. This approach enabled the distribution of computational tasks across multiple threads, significantly reducing the time required to evaluate deeper game states and thereby improving the overall performance and strategic analysis of the chess engine.

2.3 Threading

By employing parallel computing through threading, the chess engine's ability to evaluate potential moves is significantly enhanced. Moves are first shuffled to ensure an even distribution of computational effort across threads. The work of evaluating moves is then divided among a determined number of threads, equal to the system's hardware concurrency, allowing for simultaneous evaluation. Each thread calculates the best move for its assigned portion of the move list, and through synchronization mechanisms (mutexes), updates the global best move and value if a better move is found. This method not only accelerates the move evaluation process by leveraging multicore processors but also maintains the integrity of the decision-making process by ensuring that the best move is selected based on a deeper and faster analysis.

```
func getBestMoveThread(position, depth, g)
    moves = generateLegalMoves(position)
    shuffle(moves, g) # Randomize moves to evenly distribute computational load

    bestValue = -infinity
    bestMove = null
    alpha = -infinity
    beta = infinity

    num_threads = hardwareConcurrency() # Number of available hardware threads
    threads = vector(num_threads)

    func processMoves(start, end)
        localBestValue = -infinity
        localBestMove = null

        for each move in moves[start:end]
            test_position = applyMove(position, move)
            moveValue = minimax_alphabeta(test_position, depth - 1, false, alpha, beta)

            if moveValue > localBestValue
                localBestValue = moveValue
                localBestMove = move

        # Critical section to update global best values
        lock(mutex)
        if localBestValue > bestValue
            bestValue = localBestValue
            bestMove = localBestMove
        unlock(mutex)

    # Divide work among threads and start them
    moves_per_thread = moves.size / num_threads
    for i = 0 to num_threads
        start = i * moves_per_thread
        end = (i + 1) * moves_per_thread
        if i == num_threads - 1
            end = moves.size # Last thread gets remainder

        threads[i] = startThread(processMoves, start, end)

    # Wait for all threads to complete
    for each thread in threads
        join(thread)

    return bestMove
```

Listing 3: Threading in Getting Best Move

Incorporating threading into the chess engine's "getBestMove" function, I was able to extend the search depth to 7 or 8 levels on my personal computer, a significant improvement over the depth achievable without parallel processing. This enhancement is a direct result of leveraging multiple threads to concurrently evaluate potential moves, thus allowing the engine to explore more possibilities within the same time frame. While this approach already provides a substantial boost in performance, it's important to note that further optimizations and advanced techniques could potentially unlock even deeper search capabilities. Such

enhancements could include more sophisticated move ordering, use of transposition tables, and adaptive depth adjustment based on the complexity of the position, among others. These additional optimizations promise to further enhance the engine's strategic depth and efficiency, pushing the boundaries of what can be achieved with parallel processing in chess analysis.

3 Future Development

To reach and surpass the search depths of over 20 moves achieved by modern chess engines, significant improvements are necessary. Key areas include refining search algorithms and adopting efficient board representations.

3.1 Improving Board Representation

Right now, the chessboard is represented using a 2D C++ vector, where each piece, like a white rook or knight, is marked by specific numbers (e.g., 0 for rook, 1 for knight). Switching to a bitboard system and using simple bit operations (AND, NOR, XOR) for managing moves could greatly speed up the game's decision-making process.

3.2 Move Ordering to Enhance Alpha-Beta Pruning

Currently, the move order is randomized. By implementing a good move ordering is a strategic enhancement that can significantly improve the efficiency of alpha-beta pruning. By evaluating promising moves earlier, the pruning process becomes more effective, allowing deeper searches in the same amount of time. This technique prioritizes moves that are likely to lead to a favorable outcome, such as captures or threats, thereby increasing the chances of cutting off less promising branches sooner. Integrating move ordering could be a pivotal step in refining the engine's performance and achieving faster, more accurate evaluations.

3.3 Using Transposition Tables

Adding transposition tables can makethe AI smarter and faster. These tables remember positions the engine has already looked at, along with their scores and how deeply they were analyzed. If the engine comes across a position it's seen before, it can just use the score from the table instead of figuring it out all over again. This saves time and lets the engine think deeper without extra work.

3.4 Use a Better Evaluation

Improving the evaluation function can significantly enhance move quality without necessarily deepening search depth. Currently, the evaluation relies on a simple weighted count of pieces. Enhancements could include considering positional factors such as piece mobility, pawn structure, king safety, control of the center, and specific piece placements.

3.5 Opening Books, Endgame Tablebases

Integrating Opening Books and Endgame Tablebases into the chess engine represents a strategic enhancement that significantly improves its opening and endgame performance.

Opening Books utilize a curated collection of effective opening moves from professional games, enabling the engine to make strong, informed moves right from the start. This not only conserves computational resources for later, more complex stages of the game but also positions the engine advantageously from the outset.

Endgame Tablebases contain pre-calculated, perfect play for endgame positions, ensuring the engine can navigate these critical phases with optimal moves. Notably, the last 6 pieces on the board are fully solved, meaning the engine can play these scenarios flawlessly. However, the utility of tablebases comes with a consideration for memory usage; for instance, the tablebase for all 6-piece endings occupies approximately

1.2 terabytes of memory. While substantial, the strategic advantage provided justifies the memory cost, particularly in competitive or analysis settings.

By adopting these enhancements, the engine not only gains a tactical edge in the opening and endgame but also leverages deep strategic insights without the need for real-time calculation, making it significantly more competitive and efficient.

4 Summary

I made a chess engine and enhanced the AI's decision-making and computational efficiency. Starting with the Minimax algorithm enhanced by alpha-beta pruning, I efficiently narrowed down the search for the best moves. The use of weighted material counts for evaluating positions laid the foundation for the AI's strategy.

By introducing parallel processing, I was able to dive deeper into possible moves, reaching new depths in analysis with the AI. I also discussed potential improvements like adopting bitboards for faster move calculations and considering advanced strategies such as opening books and endgame tablebases for better opening and endgame performance.

There is significant room for further development. Optimizing the board representation to bitboards could dramatically increase speed, and refining my move evaluation process could sharpen the AI's competitive edge.