Chess Engine Algorithms: A Comprehensive Study

Asray Gopa

June 13, 2024

Abstract

This paper presents an in-depth study of the algorithms employed in a chess engine developed in C. The core algorithms include the Minimax algorithm with Alpha-Beta pruning, an evaluation function, move generation, an opening book, parallel search, and game over checks. Each algorithm is discussed in detail, including the mathematical foundations, logical structure, and implementation nuances.

1 Introduction

A chess engine is a complex software that can play the game of chess autonomously. It involves a combination of artificial intelligence, heuristic evaluation, and search algorithms to make decisions. This paper delves into the core algorithms used in the chess engine, providing a thorough explanation of their mathematical underpinnings and logical implementation.

2 Minimax Algorithm with Alpha-Beta Pruning

The Minimax algorithm is a recursive decision-making algorithm used in two-player games like chess. It aims to minimize the possible loss for the worst-case scenario or maximize the potential gain.

2.1 Mathematical Foundation

The Minimax algorithm can be mathematically described as:

$$V(n) = \begin{cases} \max_{a \in A(n)} V(a) & \text{if } n \text{ is a maximizing node} \\ \min_{a \in A(n)} V(a) & \text{if } n \text{ is a minimizing node} \end{cases}$$

Where $V(n)$ is the value of node n and $A(n)$ is the set of available actions at node n.

2.2 Alpha-Beta Pruning

Alpha-Beta pruning is an optimization technique for the Minimax algorithm. It reduces the number of nodes evaluated by the search tree.

$$\alpha = \max(\alpha, V(n)) \quad \text{for maximizing player}$$

$$\beta = \min(\beta, V(n)) \quad \text{for minimizing player}$$

Pruning occurs when $\beta \leq \alpha$, meaning further exploration of this branch will not yield a better outcome.

2.3 Implementation

The pseudocode for Minimax with Alpha-Beta pruning is:

```
function minimax(node, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or node is a terminal node:
        return the heuristic value of node

    if maximizingPlayer:
        maxEval = -infinity
        for each child of node:
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return maxEval
    else:
        minEval = +infinity
        for each child of node:
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return minEval
```

3 Evaluation Function

The evaluation function is critical for assessing the board state. It returns a numerical value indicating the favorability of the position.

3.1 Components of the Evaluation Function

1. **Piece Values**: Assign values to different pieces (e.g., Pawn=100, Knight=320).
2. **Piece-Square Tables**: Provides additional value based on a piece's position on the board.
3. **Mobility**: The number of legal moves available to a player.
4. **King Safety**: Assesses the safety of the king considering factors like pawn shields and nearby attacks.
5. **Pawn Structure**: Evaluates the position and formation of pawns (e.g., isolated pawns, doubled pawns).

3.2 Mathematical Representation

The evaluation function E can be represented as:

$$E = \sum_{i=1}^{n} P_i + \sum_{i=1}^{n} PS_i + M + KS + PS$$

Where P_i is the piece value, PS_i is the piece-square table value, M is the mobility score, KS is the king safety score, and PS is the pawn structure score.

3.3 Implementation

The evaluation function in C might look like:

```c
int evaluate_board() {
    int score = 0;
    // Evaluate pieces
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j].piece != EMPTY) {
                score += evaluate_position(board[i][j], i, j);
            }
        }
    }
    // Add other evaluation factors like mobility, king safety, and pawn structure
    score += mobility * 10;
    score += king_safety(WHITE) - king_safety(BLACK);
    score += pawn_structure(WHITE) - pawn_structure(BLACK);
    return score;
}
```

4 Move Generation

Move generation is responsible for creating a list of all legal moves for a player.

4.1 Piece-Specific Moves

Each piece has specific movement rules which need to be encoded:

- **Pawns**: Move forward, capture diagonally, handle en passant, and promotion.
- **Knights**: Move in an L-shape.
- **Bishops**: Move diagonally.
- **Rooks**: Move horizontally or vertically.
- **Queens**: Combine the movements of rooks and bishops.
- **Kings**: Move one square in any direction, handle castling.

4.2 Move Validation

Moves must be validated to ensure they are within the bounds of the board and do not leave the player in check.

```
bool is_valid_move(int x, int y) {
    return x >= 0 && x < SIZE && y >= 0 && y < SIZE;
}
```

5 Opening Book

The opening book contains a set of pre-computed opening moves to guide the engine during the initial phase of the game.

5.1 Benefits

- **Speed**: Pre-determined moves save computational resources.
- **Quality**: Using well-known openings helps in achieving a stronger position.

5.2 Implementation

A simple hash map can be used to store and retrieve opening moves:

```
typedef struct {
    char position[100];
    Move move;
} OpeningEntry;

typedef struct {
    OpeningEntry *entries;
    size_t size;
    size_t capacity;
} OpeningBook;
```

6 Parallel Search

Parallel search utilizes multi-threading to improve the efficiency of the Minimax algorithm.

6.1 Implementation

By dividing the search tree among multiple threads, the engine can explore more nodes in the same amount of time.

```
typedef struct {
    int depth;
    int alpha;
    int beta;
    bool isMaximizing;
    int result;
} SearchArgs;

void* parallel_search(void* args) {
    SearchArgs* search_args = (SearchArgs*)args;
    search_args->result = minimax_alpha_beta(search_args->depth,
                                             search_args->isMaximizing,
                                             search_args->alpha,
                                             search_args->beta);
    return NULL;
}
```

7 Game Over Checks

The engine includes comprehensive checks for checkmate and stalemate.

7.1 Checkmate

Occurs when the player's king is in check and no legal moves can remove the threat.

7.2 Stalemate

Occurs when the player has no legal moves and the king is not in check.

8 Conclusion

This paper has provided an in-depth examination of the algorithms used in a chess engine, covering the Minimax algorithm with Alpha-Beta pruning, evaluation function, move generation, opening book, parallel search, and game over checks. These algorithms form the foundation of a functional chess engine capable of playing competitive games.