Documentation for Chess engine (Codename: Mindstorm)

Marcos Ashton

November 3, 2024

Abstract

This paper describes the design and implementation of a chess engine using bitboards, a high-efficiency data structure for chess representation. The engine uses advanced techniques such as minimax with alpha-beta pruning for decision-making and bitwise operations for move generation. We also discuss move legality checks, castling rights, en passant moves, and the evaluation function.

1 Introduction

Chess engines are computer programs designed to play chess at various skill levels. By leveraging efficient data structures and search algorithms, these engines can simulate human-like intelligence and strategy. This project implements a chess engine with a focus on bitboard representation for fast and efficient move generation and evaluation. This document describes the methodology, implementation details, and logic behind the chess engine.

2 Bitboard Representation

Bitboards are a compact and efficient way to represent chessboards in memory. Each square on an 8x8 chessboard is represented by a bit in a 64-bit integer, allowing each piece type to be tracked independently. For example, the bitboard for white pawns would have bits set in positions where pawns are located.

We define constants representing files and ranks to facilitate bitwise operations for move generation and attack detection:

```
const uint64_t FILE_A = 0x0101010101010101ULL;
const uint64_t FILE_H = 0x8080808080808080ULL;
const uint64_t RANK_1 = 0x00000000000000FFULL;
const uint64_t RANK_8 = 0xFF00000000000000ULL;
```

Each piece type is represented by a separate bitboard, allowing quick identification of piece locations and straightforward bitwise manipulation.

3 Move Generation

Move generation is a critical component of a chess engine. This engine implements move generation for each piece type, including pawns, knights, bishops, rooks, queens, and kings. Each piece type has unique movement rules, which are implemented using bitwise shifts and masks.

3.1 Pawn Moves

Pawn moves are generated with single and double advances and capture moves. Pawns also support en passant, a special capture move that is enabled under specific conditions.

```
uint64_t singleStep = (pawns << 8) & ~allPieces;
uint64_t doubleStep = ((pawns & RANK_2) << 16) & ~allPieces & ~(allPieces << 8);
```

3.2 Knight Moves

Knight moves involve "L" shaped jumps. The engine generates all potential moves and masks out illegal destinations using bitwise operations.

```
uint64_t knightAttacks = ((knight << 17) & ~FILE_A) | ((knight << 15) & ~FILE_H);
```

3.3 Sliding Pieces (Rooks, Bishops, and Queens)

Sliding pieces like rooks, bishops, and queens require continuous movement in straight or diagonal directions until encountering a piece. The function `slideMove()` is used to generate all potential moves for sliding pieces.

```
uint64_t slideMove(uint64_t piece, int direction, uint64_t blockers);
```

4 Move Legality Checks

The function `isMoveLegal()` checks whether a move leaves the king in check. Temporary board updates allow for testing moves while preserving the original state.

```
bool isMoveLegal(uint64_t fromSquare, uint64_t toSquare, bool isWhite);
```

To ensure valid moves, each move is validated to guarantee it does not expose the king to an attack.

5 Special Moves

The engine supports special moves, such as castling and en passant:

5.1 Castling

Castling is a special king move that requires specific conditions, including no intermediate pieces between the king and rook and no attacks on the squares crossed by the king.

```
bool canCastleKingside(bool isWhite);
bool canCastleQueenside(bool isWhite);
```

5.2 En Passant

En passant is handled by checking the en passant target square, set during a double pawn move. Valid en passant captures are generated accordingly.

```
uint64_t enPassantLeft = (pawns << 7) & ~FILE_H & enPassantTarget;
```

6 Minimax and Alpha-Beta Pruning

The core of the engine's decision-making process is the minimax algorithm with alpha-beta pruning. This approach enables efficient exploration of potential moves by cutting off branches that do not affect the outcome.

```
int minimax(int depth, bool isMaximizingPlayer, int alpha, int beta, bool isWhiteTurn);
```

The `minimax` function recursively explores moves up to a specified depth, evaluating positions with an evaluation function that considers material balance.

7 Evaluation Function

The evaluation function assesses the board position to determine the advantage. It assigns values to pieces and computes a score based on the material on each side. Positive scores favor White, while negative scores favor Black.

```
int evaluatePosition() {
    const int PAWN_VALUE = 100;
    const int KNIGHT_VALUE = 320;
    ...
    return whiteScore - blackScore;
}
```

8 Move History and Undo

To allow efficient exploration of the game tree, each move's effect on the board is recorded, enabling reversion with `undoMove()`.

```
void saveBoardState();
void undoMove();
```

This functionality enables backtracking during the minimax search, preserving game state integrity.

9 User Interface and Gameplay Loop

The engine features two modes: human vs. human and human vs. computer. The main gameplay loop handles user input and displays the board.

```
int main() {
    int choice;
    if (choice == 1) {
        gameLoop(); // Human vs Human
    } else {
        computerGameLoop(humanPlaysWhite);
    }
}
```

10 Conclusion

This chess engine demonstrates the effectiveness of bitboard-based move generation and minimax with alpha-beta pruning for efficient chess simulation. The engine supports all fundamental chess rules, including castling, en passant, and pawn promotion. This implementation can be expanded with additional features like opening books, endgame tablebases, and more sophisticated evaluation techniques.

References

[1] Chess Programming Wiki, `https://www.chessprogramming.org/Main_Page`