**Title**

**Chess Engine Project**

**Abstract**

This project involves the development of a chess engine capable of playing chess autonomously. The engine uses advanced algorithms such as alpha-beta pruning, iterative deepening, and Zobrist hashing to evaluate and make optimal moves. The project includes components for board representation, move generation, search algorithms, and evaluation functions.

**1. Introduction**

* **Objective**: To develop a chess engine that can play chess at a competitive level.
* **Scope**: The project covers board representation, move generation, search algorithms, evaluation functions, and user interface for playing against the engine.
* **Motivation**: Creating a chess engine involves solving complex problems in artificial intelligence, search algorithms, and game theory, making it an excellent project for learning and applying these concepts.

**2. Project Structure**

* **Board Representation**: The board is represented using a 1D array of 64 elements, each representing a square on the chessboard.
* **Move Generation**: Generates all possible legal moves for a given board position.
* **Search Algorithms**: Implements alpha-beta pruning, iterative deepening, and quiescence search to find the best move.
* **Evaluation Function**: Evaluates board positions to determine the best move.
* **Transposition Table**: Uses Zobrist hashing to store and retrieve previously evaluated positions.

**3. Key Components**

**3.1 Board Representation**

* **Files**: board.cpp, board.hpp
* **Description**: Represents the chessboard and provides functions to manipulate and query the board state.
* **Key Functions**:
  + make\_move(Move& move): Updates the board state with a given move.
  + unmake\_move(Move& move): Reverts the board state to before the given move.
  + load\_fen(const std::string& fen): Loads a board position from a FEN string.
  + to\_fen(): Converts the current board state to a FEN string.
* **Implementation Details**:
  + The board is represented as a 1D array of 64 elements.
  + Each element represents a square on the chessboard and can hold values representing different pieces or an empty square.
  + The make\_move and unmake\_move functions handle the updating and reverting of the board state, ensuring that all aspects of the game state (e.g., castling rights, en passant squares) are correctly managed.

**3.2 Move Generation**

* **Files**: movegen.cpp, movegen.hpp
* **Description**: Generates all possible legal moves for a given board position.
* **Key Functions**:
  + generate\_moves(): Generates all legal moves for the current board position.
  + is\_legal\_move(Move& move): Checks if a given move is legal.
* **Implementation Details**:
  + The move generation function iterates through all pieces on the board and generates possible moves based on the piece type and position.
  + Special moves such as castling, en passant, and pawn promotion are handled separately to ensure all rules are correctly applied.
  + The is\_legal\_move function checks for move legality by ensuring that the move does not leave the king in check and adheres to all chess rules.

**3.3 Search Algorithms**

* **Files**: search.cpp, search.hpp
* **Description**: Implements search algorithms to find the best move.
* **Key Functions**:
  + search(): Main search function that uses iterative deepening.
  + alphaBeta(int alpha, int beta, int depth): Implements alpha-beta pruning.
  + quiescence(int alpha, int beta): Implements quiescence search to evaluate quiet positions.
* **Implementation Details**:
  + The search function uses iterative deepening to progressively deepen the search until a time limit is reached.
  + The alphaBeta function implements alpha-beta pruning to eliminate branches in the search tree that do not need to be explored, improving search efficiency.
  + The quiescence function extends the search at leaf nodes to evaluate quiet positions and avoid the horizon effect.

**3.4 Evaluation Function**

* **Files**: eval.cpp, eval.hpp
* **Description**: Evaluates board positions to determine the best move.
* **Key Functions**:
  + evaluate(): Evaluates the current board position and returns a score.
* **Implementation Details**:
  + The evaluation function assigns scores to board positions based on material balance, piece activity, king safety, and other factors.
  + The function uses a combination of heuristics and precomputed tables to efficiently evaluate positions.
  + Special considerations are made for endgame scenarios, where different evaluation criteria may apply.

**3.5 Transposition Table**

* **Files**: tt.cpp, tt.hpp
* **Description**: Uses Zobrist hashing to store and retrieve previously evaluated positions.
* **Key Functions**:
  + TT\_probe(TT\_t TT, u\_int64\_t hash, int depth, int alpha, int beta): Probes the transposition table for a given position.
  + TT\_store(TT\_t TT, u\_int64\_t hash, int depth, int score, EvalType eval\_type): Stores a position in the transposition table.
* **Implementation Details**:
  + The transposition table is a hash table that caches search results to avoid re-searching identical positions reached through different move orders.
  + Zobrist hashing is used to generate unique hashes for board positions, ensuring efficient indexing and retrieval.
  + The TT\_probe function checks if a position is already in the table and retrieves the stored score if available.
  + The TT\_store function stores the evaluated score of a position in the table, ensuring that deeper search results are preferred.

**4. Implementation Details**

**4.1 Zobrist Hashing**

* **Description**: Zobrist hashing is used to generate unique hashes for board positions.
* **Key Functions**:
  + zobrist\_init(): Initializes the Zobrist hashing tables with random values.
  + zobrist\_hash(): Computes the Zobrist hash for the current board position.
* **Implementation Details**:
  + Zobrist hashing uses precomputed random values for each piece on each square to generate a unique hash for any board position.
  + The zobrist\_init function initializes these random values at the start of the program.
  + The zobrist\_hash function computes the hash by XORing the random values for all pieces on the board, the current turn, castling rights, and en passant square.

**4.2 Iterative Deepening**

* **Description**: Iterative deepening is used to progressively deepen the search until a time limit is reached.
* **Key Functions**:
  + iterative\_search(): Performs iterative deepening search to find the best move.
* **Implementation Details**:
  + Iterative deepening starts with a shallow search and progressively increases the search depth.
  + This approach allows the engine to find a reasonably good move quickly and then refine it as more time is available.
  + The function keeps track of the best move found at each depth and uses it as a fallback if the search is interrupted.

**4.3 Alpha-Beta Pruning**

* **Description**: Alpha-beta pruning is used to eliminate branches in the search tree that do not need to be explored.
* **Key Functions**:
  + alphaBeta(int alpha, int beta, int depth): Implements alpha-beta pruning.
* **Implementation Details**:
  + Alpha-beta pruning maintains two values, 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.
  + As the search progresses, branches that cannot improve the current best score are pruned, reducing the number of nodes that need to be evaluated.
  + This technique significantly improves the efficiency of the search algorithm.

**4.4 Quiescence Search**

* **Description**: Quiescence search is used to evaluate quiet positions and avoid the horizon effect.
* **Key Functions**:
  + quiescence(int alpha, int beta): Implements quiescence search.
* **Implementation Details**:
  + Quiescence search extends the search at leaf nodes to include only "quiet" moves, such as captures and checks, that can significantly change the evaluation.
  + This helps to avoid the horizon effect, where the engine makes a poor move because it cannot see beyond a certain depth.
  + The function evaluates the position after each quiet move and continues until a stable position is reached.

**5. Results**

* **Performance**: The chess engine is capable of evaluating and making optimal moves within a reasonable time frame.
* **Strength**: The engine can play at a competitive level, making it suitable for playing against human opponents or other engines.
* **Example Games**: Provide examples of games played by the engine, highlighting its ability to make strategic decisions and handle complex positions.

**6. Conclusion**

* **Summary**: The chess engine project successfully implements key components required for autonomous chess play, including board representation, move generation, search algorithms, evaluation functions, and transposition table management.
* **Future Work**: Future improvements could include optimizing the evaluation function, adding more advanced search techniques, and improving the user interface.
* **Potential Enhancements**:
  + **Parallel Search**: Implementing parallel search techniques to take advantage of multi-core processors.
  + **Machine Learning**: Incorporating machine learning techniques to improve the evaluation function.
  + **Opening Book**: Adding an opening book to improve the engine's performance in the opening phase of the game.
  + **Endgame Tablebases**: Integrating endgame tablebases to ensure perfect play in the endgame.

**7. Resources Used**

**7.1 Tools**

* **Integrated Development Environment (IDE)**: Visual Studio Code
  + Used for writing, debugging, and testing the code.
* **Compiler**: GCC (GNU Compiler Collection)
  + Used for compiling the C++ code.
* **Version Control**: Git
  + Used for version control and collaboration.

**7.2 Libraries**

* **Standard Template Library (STL)**: C++ Standard Library
  + Provides data structures (e.g., vectors, maps) and algorithms used throughout the project.
* **Random**: <random>
  + Used for generating random numbers, particularly for Zobrist hashing and move selection.
* **Chrono**: <chrono>
  + Used for time management and measuring the duration of search operations.
* **Atomic**: <atomic>
  + Used for managing the searching flag in a thread-safe manner.
* **GTK**: <gtk/gtk.h>
  + Used for making GUI

**7.3 External Resources**

* **Chess Programming Wiki**: [Chess Programming Wiki](vscode-file://vscode-app/c:/Users/ijaza/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)
  + A valuable resource for understanding chess algorithms and techniques.