Developing a Chess Engine from the Ground Up

Chess is a game of perfect information. It provides players with complete knowledge of the current state of the game. There is no luck involved, no hidden elements, and no uncertainty. It is a game of pure skill and calculation. The simplicity of its rules paves the way for a world of immense complexity and strategy. This complexity makes chess an ideal, albeit challenging setting for the development of an intelligent agent.

Over the course of a semester, I challenged myself to build a chess engine from the ground up to obtain a deeper understanding of artificial intelligence algorithms, optimization techniques, and the software development life cycle. The primary constraint I set for this project was to avoid using any chess or AI libraries. While using libraries would have significantly sped up the development process, I find satisfaction and amusement in knowing that I am capable of writing a program from scratch that can beat me in a game of chess.

As this project comes to a close, I am reminded that I have only stepped foot in the world of chess programming. It has been the subject of extensive research, capturing the interest of mathematicians and researchers since the dawn of computer science. This documentation serves as a guide for my ongoing development and for anyone interested in creating their own chess engine.

Board Representation

The board representation is the internal data structure which describes the current state of the chess board and its pieces. The goal is to accurately and efficiently represent any position and minimize the overhead of generating and making moves. Various board representations come with both advantages and disadvantages, and often experience a tradeoff between space and time, which is a recurring concept in the development of chess engines.

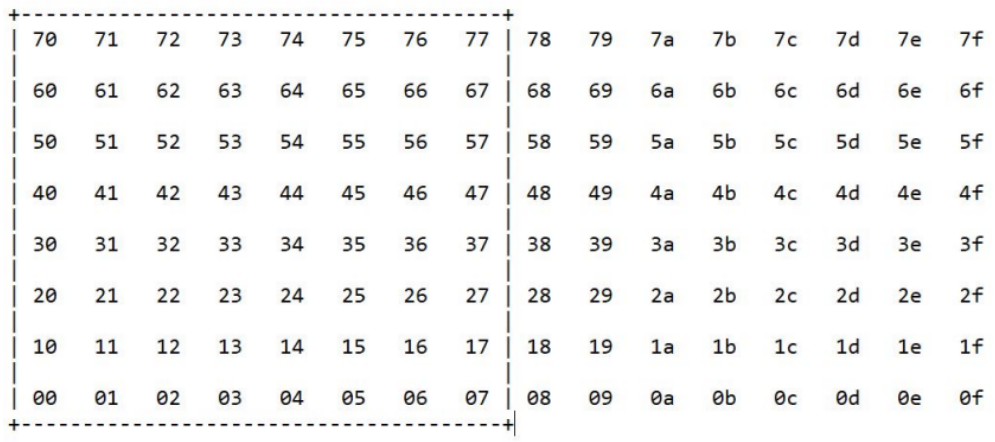
The simplest board representation is an 8x8 array, where each element describes a square holding a piece. This is the most intuitive approach, but runs into performance issues in move generation. Indexing the array requires the compiler to first calculate 8*rank+file. This problem can be mitigated by creating a single dimensional array of 64 elements, but leads to a new problem due to the potential wrap-around of pieces. To keep the pieces from wrapping around the board, one would need to implement additional edge cases leading to inefficiency.

Padded arrays solve both of these issues by being both one dimensional, and possessing a “padding” around the board. If a piece attempts to wrap around the board during move generation, it will first enter this padding, and a simple check will tell the move generator to stop moving in this direction. Although this method increases the memory footprint, the speed-up in move generation has been proved advantageous. One example of a padded array is the 0x88 board representation.

First imagine that a piece’s location is stored in a single byte with the first four bits indicating column and the second four bits representing row. A 16x16 board could be indexed with a single byte and would possess 8 layers of padding around the playable 8x8 board. If the playable board is placed in the lower left quadrant of this array, every invalid square would contain a 1 in the most significant bit of its row or column. Thus an invalid square can simply be determined through a bitwise “and” of the index and 0b10001000 (or 0x88).

```
// Checks if a square is valid
bool isSquareValid(unsigned char index) {
      return (index & 0x88) == 0;
}
```

Technically only the lower half of the array needs to be allocated since every square in the upper half is invalid. This optimization leads to a board the size of 128 pieces.



Every element of the array, or square on the board, contains a “piece” represented by one byte. Each piece type is given a unique number: 1 for pawn, 2 for knight, etc. Each color is also given a number: 8 for black, 16 for white. These two values are summed together. For example, a black knight is represented as 2+8=10. Additionally, squares without a piece are assigned piece type 0 for none.

```
struct Piece {
	static const unsigned char BLACK = 8;
	static const unsigned char WHITE = 16;

	static const unsigned char NONE = 0;
	static const unsigned char PAWN = 1;
	static const unsigned char KNIGHT = 2;
	static const unsigned char BISHOP = 3;
	static const unsigned char ROOK = 4;
	static const unsigned char QUEEN = 5;
	static const unsigned char KING = 6;
};
```

These values allow the compiler to easily check the color and type of the piece. The color of the piece is obtained with `piece & 0x18`, and piece type is obtained with `piece & 0x07`. These single-instruction checks are essential for efficient move generation.

Move Generation

The move generation is responsible for determining all legal moves that can be made in a given position. It is typically composed of two algorithms: one to generate pseudo-legal moves, and one to filter out illegal moves. Pseudo-legal moves are moves that follow the rules of how each piece moves, but may or may not leave the king in check. Legal moves are the subset of pseudo-legal moves in which the king is not in check after the move is made.

Generating pseudo-legal moves requires coding every game rule in chess, including special rules such as castling, en passant, and pawn promotion. To say this is a hard task is an understatement, and perfecting the move generation took me multiple weeks. Many bugs can arise in the move generation, but luckily there is a way to pinpoint these bugs which I will discuss later.

Before move generation begins, there needs to be a structure to store the moves. A move is described with two parameters: start position and end position. In my current implementation, I defined a third parameter *move type*, which is useful for updating the board with special moves. To store the moves I simply used a static array of size 218 which is the maximum number of moves that can occur in any position. This is more efficient than dynamically allocating moves as it reduces runtime overhead. These classes are contained in “Move.h” and “MoveList.h”.

Pseudo-legal move generation for non-special moves occurs in the following way with minor modifications for each piece.

1. Generate a list of “directions”. These values depend on the board representation. With a 0x88 board representation, up is +16, left is -1, down-right diagonally is -15, etc.
2. Obtain the next piece on the board that is the same color as the color to move.
3. Obtain the next direction from the direction list.
4. Add the direction value to the start position to obtain the end position.
5. Ensure the new position is a valid square.
6. Ensure the new position does not contain a piece of the same color.
7. Construct a “move” object and push it into the “move list”.
8. If the piece is a sliding piece, and the end position is empty, repeat from step #4.
9. Repeat from step #2.

Once the pseudo-legal moves are obtained, it is necessary to filter out illegal moves. I did this by making the move to update the board, and checking to see if the king is in check. If the king is in check, the move that led to this position is removed from the move list. There are two functions I used to do this.

```
// Returns true if any piece of a certain color can move to a certain squarePos
bool isInCheck(unsigned char squarePos, unsigned char color);
```

The function may be used after a move is made to check if the king is in check. For example if white has just made a move, the function `isInCheck(whiteKingPosition, black)` will return true if the move that was just made is illegal.

```
// Returns 1 if move is legal and 0 if illegal
bool makeMove(Move* move);
```

This function is used to update the board with the given move. It automatically uses `isInCheck()` to check if the move made was legal. To undo a move, simply store the game state before making a move, and return to the stored state.

Debugging the move generation would have been nearly impossible without the assistance of the `Perft()` function. Perft is an abbreviation for performance test, and it recursively generates moves up to a certain depth. It is a dual purpose function that assists in pinpointing bugs and provides a metric for the performance of move generation. Using this function, I can calculate that the program is generating about **eleven million** moves per second.

Evaluation

If chess were a solved game, we would know the exact moves to play in every position to reach a win, loss, or draw. In reality, the number of chess positions is many orders of magnitude greater than the number of atoms in the universe, and chess will most likely never be solved. Because of this, we need to approximate the relative value of a chess position.

In chess programming, there is a standard metric to measure relative value. A positive evaluation means the chess engine predicts that white is winning, while a negative evaluation means black is winning. An evaluation of 0 indicates completely even winning chances. The scale is determined by setting the pawn equal to exactly one point.

The traditional way to evaluate a position is to assign each piece a value, then sum the value of the white pieces and subtract the value of the black pieces. However, it is often not the case that the player with more material is winning, and other considerations must be taken into account to provide a precise evaluation. In developing a heuristic, it is important to note the tradeoff between knowledge and speed. More accurate evaluations tend to take longer to calculate.

After testing various heuristics such as mobility score, king safety, and board control, I landed on using what is known as piece square tables. I found that this method results in a relatively accurate evaluation given the speed at which it is calculated. Each piece type has its own table which is handcrafted to represent which squares the piece performs best on. For example, it is known that controlling the center of the board is a good strategy in chess, and the piece square tables reflect this strategy.

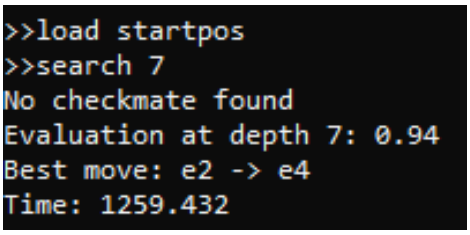
```
static constexpr signed char knightTable[64] = {
      -50,-40,-30,-30,-30,-30,-40,-50,
      -40,-20,  0,  0,  0,  0,-20,-40,
      -30,  0, 10, 15, 15, 10,  0,-30,
      -30,  5, 15, 20, 20, 15,  5,-30,
      -30,  0, 15, 20, 20, 15,  0,-30,
      -30,  5, 10, 15, 15, 10,  5,-30,
      -40,-20,  0,  5,  5,  0,-20,-40,
      -50,-40,-30,-30,-30,-30,-40,-50
};
```

For each piece on the board, my evaluation function simply adds the piece value to the piece value table indexed at its position. This is a very simple form of evaluation, and there is much improvement to be made. Modern chess engines employ neural networks to aid in the accuracy of their evaluation functions. The concept of training a neural network and allowing it to decide which positions are good is a feature that I want to implement in the future.

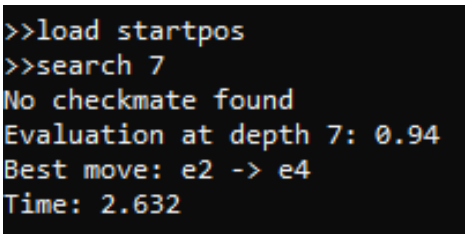
Search Algorithm

The search algorithm, in combination with the evaluation function, is what makes chess engines “artificially intelligent”. It is responsible for exploring the game tree and selecting the best move. All search algorithms work by recursively generating moves and min-maxing the resulting game tree, that is by minimizing the maximum gain of the opponent at every level. The most standard optimization to be made here is alpha-beta pruning. Because this algorithm assumes optimal play, certain branches that are deemed nonoptimal are pruned from the game tree, saving valuable time.

No pruning



With pruning



In this test, with no changes to the code other than including pruning, alpha-beta pruning reduced the computation time by a factor of almost 500. This reduction might be made even more apparent if iterative deepening and move ordering is implemented.

```
// Alpha-beta pruning; stores best move found and returns the evaluation
double alphaBeta(Move* bestMove, int depth, double alpha, double beta, bool
maximizingPlayer);
```

To use this function, initialize alpha to -DBL_MAX, and beta to DBL_MAX. MaximizingPlayer should be true if it is white to move, and a pointer to a Move object needs to be included to store the best move found by the algorithm.

While alpha-beta pruning provides the most notable computational optimization, there are many other optimizations to be made in terms of finding the best move. A common problem that search algorithms encounter is the horizon effect. Because chess engines can only search to a limited depth, there is a possibility of making a detrimental move whose effect is not visible within the search space. In other words, the consequence is just past the search’s “horizon”.

This problem exists within all chess engines, and while its effect can never be fully removed, it can be mitigated.

The purpose of quiescence search is to only evaluate positions that are “quiet”. A quiet position is one that lacks checks and captures. In this manner, the search algorithm will hopefully extend the search past any game winning tactics. Though this method has been shown to reduce the negative effects of limited search depth, there still exists the possibility of game winning tactics that don’t involve a check or capture.