Technical Specifications for Odyssey

Harry Zhou

My chess engine is better than me in chess. Should I be happy or sad?

Overview

Chess engines can be traced back to the 90s, when IBM's Deep Blue defeated Garry Kasparov, the reigning world champion; with the advancements of modern technology, more chess engines like Stockfish and Alphazero are developed using neural networks, and they can easily defeat any human player.

Naturally, as both a chess player and computer science enthusiast, I was curious of the underlying mechanisms behind those intelligent machines, and decided to create an engine of my own - Odyssey. This was an ambitious goal, given that most of my previous computer science background was focused on competitive programming rather than project development. But I was driven by passion and curiosity, and through relentlessly researching, coming up with new ideas, coding, and debugging, I'm super happy to announce that my chess engine, Odyssey, is estimated to be stronger than myself at chess!

Currently, Odyssey is estimated to have a ELO of 1870 (in comparison, my official rating is around 1400). It has undoubtedly fulfilled my original goal, but I've grown more ambitious, so I will address some potential improvements and plan moving forward in the document. Odyssey is mainly composed of four essential parts: board representation, move generation, evaluation function, and searching/pruning algorithms. The rest of the document will focus on those four parts, and a detailed credits section is at the end.

Board Representation

- Every chess engine needs a board representation to maintain chess positions for move generation, evaluation, and searching
- The most obvious way to maintain such a representation is to use an 8x8 2-D array. While this approach is intuitive, it's inefficient in generating moves later on
- Instead, Odyssey uses **bitboard representation**. Since a chess board has 64 squares, each square can be mapped to a single bit in a `long long` type in C++. For each piece in chess (pawn, knight, bishop, rook, queen, and king), Odyssey maintains such a `Bitboard` type, and for every square

that the piece is on, the corresponding bit would be `1`, and otherwise it'd be `0`. (For example, if one player's queen is on the square A4 and no other square, the queen bitboard would be $2^{(4-1)} = 8$)
- The benefit of bitboard representation is that we can take advantage of fast binary operation. Consider the following case: we want to find out what squares on the board are occupied. With bitboard representation, we don't have to go through each of the 64 squares; instead, we can simply call `occpuied = pawn | knight | bishop | rook | queen | king`, where `|` is the `or` operation in C++
- Odyssey can also read in a position using FEN (Forsyth–Edwards Notation, the standard way of expressing a position)

Move Generation

- Given a position, move generation is the process of generating all legal moves. This step has to be extremely robust and make sure there are no missed/extra moves being generated
- Just like board representation, binary operations are taken advantage of to make move generation faster. Odyssey uses a technique called **magic bitboards** to further speed up the process, especially when it's generating moves for bishops, rooks, or queens (which can be blocked by the placement of other pieces). Essentially, magic bitboards trade some space with a much better time performance
- There are a lot of border cases need to be taken care of as well. There are many "exceptions" in the rule of chess: en passsant (for pawns), castling (for rooks and kings), stalemate etc. Odyssey is tested using "perft", in order to ensure performance and robustness
- Odyssey uses libchess, an open source library to generate moves. See credits section for more information

Evaluation Function

- Ideally, we'd want the engine to search as deeply as possible, but unfortunately, it's impossible to exhaust the search tree simply because there are too many variations. Therefore, when the engine reaches a certain depth, it's important to return an evaluation for that end position
- This step is the most **creative** part of an engine, and I certainly incorporated my personal chess "style" to Odyssey to make it play aggressive, attacking chess! There are many layers to a good evaluation function: material evaluation, piece placements, game phase identification, positional penalties/rewards etc.
  - ◇ Material evaluation: material refers to the number of pieces a side has - usually, the more material the better; after all, if you have a queen and your opponent doesn't, you're likely going to win. Conventionally, a pawn is worth 1 point, knights and bishops are worth 3, rooks are worth 5, queen is worth 9, while the king is ∞ (if you lose the king,

you lose the game!). Odyssey uses a more sophisticated material table, measuring each piece's value in centipawn, and taking account to game phase change (for instance, rooks are usually more valuable in endgames, while knights are more useful in the middlegame)

◇ Piece placement: a knight stuck at a corner is worthless, while a knight at the center, occupying a so-called "outpost" would be an absolute beast; the difference between them is *piece placement*. Odyssey mostly uses piece-square tables to evaluate a piece's placement. An example table for knights during middlegame is shown below:

```
knight_piece_square_table[64] =
{-167, -89, -34, -49,  61, -97, -15, -107,
-73, -41,  72,  36,  23,  62,   7,  -17,
-47,  60,  37,  65,  84, 129,  73,   44,
-9,  17,  19,  53,  37,  69,  18,   22,
-13,   4,  16,  13,  28,  19,  21,   -8,
-23,  -9,  12,  10,  19,  17,  25,  -16,
-29, -53, -12,  -3,  -1,  18, -14,  -19,
-105, -21, -58, -33, -17, -28, -19,  -23};
```

◇ Positional penalties/rewards: there are many chess "principles", such as "king safety", "open files", "connected passed pawns", "bishop pair" etc., and based on them, I assign special rewards or penalties to a position. For instance, a rook is much more active if it's not blocked by pawns - then it's said to be on the "open file", and it receives a bonus:

```
// bonus for semi-open file
bool is_semi_open = ((occupancy(Pawn) & occupancy(White))
                    & file_masks[int_square]).empty();
if (is_semi_open) {
    score_opening += semi_open_file_score;
    score_endgame += semi_open_file_score;
}
```

◇ Giving Odyssey a *style*! Since I'm an attacking player, I'd always like to seize initiatives, and even sacrifice material to checkmate the opponent. Therefore, I made sure Odyssey values initiatives. One way is to give bonus for piece mobilities, so that Odyssey won't be afraid of sacrificing a pawn to open up a long diagonal for the bishop:

```
// bonus for the mobility of the bishop
int bishop_moves = movegen::bishop_moves(square, occupied()).count();
score_opening += (bishop_moves - bishop_unit) * bishop_mobility_opening;
score_endgame += (bishop_moves - bishop_unit) * bishop_mobility_endgame;
```

Searching/Pruning

- An average chess position has 40 moves. With a brute force searching algorithm, an average computer would even struggle with calculating six moves: it needs to consider all $40^6 \approx 4 \times 10^9$ variations. Therefore, a good chess engine would prune the search tree and only calculate the necessary variations

- Odyssey uses the **alpha beta pruning algorithm**. It's based on the observation that "if one already has found a quite good move and search for alternatives, one refutation is enough to avoid it. No need to look for even stronger refutations." By maintaining a lower bound `alpha` and an upper bound `beta`, alpha-beta can optimize a position significantly without the risk of overlooking any move (roughly reduces $O(b^n)$ to $O(b^{\frac{n}{2}})$)
- Odyssey uses the negamax framework to implement alpha-beta
- A quiescence search is implemented to stabilize alpha-beta and avoid any "horizon effect" (the case when the engine stops one move early and falsely evaluates a position)
- Move ordering. In the most ideal case, alpha-beta will pick the best move out of all legal moves, which will significantly reduce the number of nodes it searches. While that's unlikely, there are enhancements that maintain a relatively good move ordering
    - ◇ Principle variation. The principal variation (PV) is a sequence of moves that engine consider best so far, and it's a good guess of the best move in a position. Therefore, PV is always the first to be searched. Odyssey uses a triangular PV table to implement the enhancement
    - ◇ Killer moves. A killer move is a quiet move that causes a beta-cutoff, and should be searched as soon as possible. It's just behind the PV move in move ordering

```cpp
// sort moves based on the priority score of a move
void Position::sort_moves(std::vector<Move>& move_list) {
    // initialize the moves with a score
    std::vector<std::pair<Move, int>> moves_with_score;
    for (auto move : move_list) {
        moves_with_score.push_back(std::make_pair(move, score_move(move)));
    }

    // sorting
    std::sort(moves_with_score.begin(), moves_with_score.end(),
        [](const auto& i, const auto& j) { return i.second > j.second; });

    // get rid of the score
    move_list.clear();
    for (auto p : moves_with_score)
        move_list.push_back(p.first);
}
```

Future Goals/Potential Improvement

- Add a multi-layer neural network evaluation function in addition to the hand-crafted evaluation

- Add an opening book/endgame tablebase specifically designed for those game phases
- Add the Universal Chess Interface to Odyssey, so it can be connected to a GUI

Credits

- libchess
  - ◇ libchess is an open source "C++17 library that [provides] legal move generation"
  - ◇ Originally, I planned to write move generation myself too (see Ophelia, the predecessor of Odyssey). Well, that did not end up well... When I was done with writing code and started testing, I couldn't for the life of me figure out why Ophelia couldn't generate all moves (unit-testing and debugging are indeed two areas that I need to improve on!). Eventually, I realized that there's no point for reinventing the wheels, and I should spend more time on the evaluation function and searching algorithms
  - ◇ When I landed on libchess, I was very happy to see that its code is clean and comprehensible, which allows me to make changes and adjustments easily. Its fast and robust performance provides a solid starting point for Odyssey
- BBC (Bit Board Chess)
  - ◇ BBC is an open source chess engine written by Code Monkey King
  - ◇ BBC was a particularly helpful reference when I was researching effective searching/pruning algorithm
- The Chess Programming Wiki (CPW)
  - ◇ Fabulous wiki on almost every topic on chess programming
  - ◇ I used CPW as a starting point for researching evaluation functions and searching algorithms
- Miscellaneous
  - ◇ CMake for generating Makefiles
  - ◇ Git for version control
  - ◇ Stackoverflow and various C++ forums (how can I not mention them?!) when I came across syntax problems