By: Dan Ayzik & Noam Barlin

Chess AI fundamentals and technical details

In this paper we'll discuss the ideas and details behind chess engines, for the report on our journey in this competition, please refer to the report pdf after reading this paper.

Throughout this paper we'll assume the reader has knowledge in the rules of chess and a basic understanding of chess strategy.

Preliminary Technical details

We first discuss some crucial technical details for efficient chess representation and computation, which become ever more necessary due to the strict memory and size restrictions.

Any chess engine must be able to simulate the game of chess itself in order to 'calculate' moves, meaning looking ahead in the position and comparing different move possibilities.
A chess position can be represented in many different ways. Here are some representations and what they are used for:

1) FEN string (Forsyth–Edwards Notation):

This representation compacts the whole game state into 1 string that looks like this:
"rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2"

This string corresponds to the following position:

The combination of letters and numbers each represent a single row, separated by '/' for a total of 8 rows.
The first entry from the left corresponds to the 8th row.
Within the row, characters give information about pieces in the columns from left to right.
A lower case letter corresponds to a black piece.
An upper case letter corresponds to a white piece.
A number indicates a number of consecutive empty cells
Letter mapping:
R -> Rook Q -> Queen
N -> Knight B -> Bishop
P -> Pawn
K -> King
After the cell information, there's general game data.
The 'b' indicates it is currently blacks turn. If 'w' was in its place it would indicate its whites turn.
The sequence KQkq represents castling rights for white and black respectively. For example, a sequence -Q— Would mean white can still castle queen side, while black has lost castling rights. Note, this speaks for the remaining rights to castle, not for the possibility of castling right now.

Next, the '-' indicates there is no en-passant capture currently available. If it was possible, the '-' would be replaced for a cell representation like 'e3', meaning the currently player can do an en-passant capture and his pawn would land on e3, which would capture the pawn on e4. Example:
rnbqkbnr/ppppp1p1/7p/8/3PPp1P/8/PPP2PP1/RNBQKBNR b KQkq e3 0 1

Next is the half move clock, which indicates the number of moves since the last capture or pawn advance. This increments for black and white moves, hence 'half' moves.

Next is the full move clock which counts the number of full moves since the start of the game.

These are relevant for specific rules like the 50 half move rule.

2) PGN (Portable game notation):
PGN encapsulates all details about one complete game of chess by writing the moves that were played in order in algebraic notation. As well as some information about the players, their ratings and date the game was played etc…
Example:

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1.e4 e5 2.Nf3 Nc6 3.Bb5 {This opening is called the Ruy Lopez.} 3...a6
4.Ba4 Nf6 5.O-O Be7 6.Re1 b5 7.Bb3 d6 8.c3 O-O 9.h3 Nb8 10.d4 Nbd7
11.c4 c6 12.cxb5 axb5 13.Nc3 Bb7 14.Bg5 b4 15.Nb1 h6 16.Bh4 c5 17.dxe5
Nxe4 18.Bxe7 Qxe7 19.exd6 Qf6 20.Nbd2 Nxd6 21.Nc4 Nxc4 22.Bxc4 Nb6
23.Ne5 Rae8 24.Bxf7+ Rxf7 25.Nxf7 Rxe1+ 26.Qxe1 Kxf7 27.Qe3 Qg5 28.Qxg5
hxg5 29.b3 Ke6 30.a3 Kd6 31.axb4 cxb4 32.Ra5 Nd5 33.f3 Bc8 34.Kf2 Bf5
35.Ra7 g6 36.Ra6+ Kc5 37.Ke1 Nf4 38.g3 Nxh3 39.Kd2 Kb5 40.Rd6 Kc5 41.Ra6
Nf2 42.g4 Bd3 43.Re6 1/2-1/2
```

The FEN and PGN notation are used to transfer information about a current game state or a past game, and while the FEN notation is efficient memory wise, maintaining it as you play the game is costly. Thus, FEN notation is used to transfer information about a position, like a random starting position for a match, and PGN is used for large datasets of past games.

3) The board can also be represented by an 8x8 array, with a char representing each piece. This naïve approach can be maintained rather nicely when making moves, but there's still a better way.

4) Bitboards:
The actual representation for efficient chess engines are Bitboards. A bitboard is simply a 64-bit unsigned integer, this corresponds nicely with the 64 squares in chess, since we can decide the 'on' bits (1's) represent the Prescence of a piece on a corresponding square. This however, loses information about what type of piece it is and what color it is, so instead we use 12 of these bitboards, each representing locations for a specific piece color combination. The LSB corresponds to a1, while the MSB corresponds to h8.
Example: The starting positions has a white rook on 'a1' and on 'h1'. The corresponding bitboard is the number 129, which is: 0000000000000000000000000000000000000000000000000000000010000001 in 64bit binary
So this would be the bitboard for the white rooks.

Since our representation is now just 12 unsigned long integers, we can use bitwise operations to move pieces, and perform calculations like who's attacking who. Example:
Given the bitboard of white pawns, if we shift left by 7 and bitwise & it with the ~(not) of the H file mask, we get the cells that the white pawns attack to their left. If we do a similar calculation for the right-side attacks, and bitwise | them with the left attacks, we get the cells the white pawns are attacking. This seems cumbersome but is extremely efficient for a computer.

So to sum up, bitboards are our go-to representation for our inner simulation and calculations used for our evaluation heuristics (more on that later). However, they are not ideal as input for a neural network, since the integers themselves don't carry information, rather the information is in their binary representation. So, when working with a neural network, we must turn our bitboards into relevant vectors.

By: Dan Ayzik & Noam Barlin

Chess engine methodologies:

Researching into chess engines lead us to discover that there are 2 main successful ways for a chess engine to play.

1) Static evaluation + search

2) Reinforcement learning with MCTS

We will now dive into the fundamentals of these methods.

Static evaluation:

The first part of the first method is just the static evaluation part. This means, giving a numerical evaluation for the current board position in a static manner without looking ahead and calculating moves.
We give a positive evaluation if the position is in favor of the white player, negative if it's in favor of the black player, 0 if it's dead even or a drawn position, and 'infinity' or negative 'infinity' if the board is in checkmate.
The absolute value of the evaluation represents how 'far ahead' a player is, which can be translated directly to win probability.
This evaluation can be done in multiple ways, we will go into shallow details for two of them.

Heuristic static evaluation:

A simple way to provide a static evaluation is a heuristic, meaning we 'guess' by using our previous knowledge of the game. The most known example of this is the material evaluation.
Each piece is worth 'points' according to the following mapping:
Pawn -> 1 Rook -> 5
Knight -> 3 Queen -> 9
Bishop -> 3

Meaning, all else equal, if I captured a queen while my opponent captured my rook I'm up by 4 'points'.
This is known as material balance which despite being simplistic, it's the biggest deciding factor for an evaluation.
Examples of other heuristics are:
-Center control - It is well established in chess that having your pieces near the center of the board is better.
-King safety
-Piece mobility(number of available moves)
-Piece connectivity(how well your pieces are 'communicating')
And many more which we'll cover later.
When performing a static evaluation, you can calculate these features and give points to white and black based on these. Then, you can determine the final evaluation to be whites score – blacks score.
The challenges here are which features to choose, and what is the number of points that each feature gets in relation to others. Features can be tested for their usefulness simply by trial and error, or tuned using a neural network (more on that later).

Neural network static evaluation:

Since the goal is to have a function that takes the state of the board and outputs a number, a neural network can be used for this task. However, interestingly enough, they were not used successfully until summer 2020, where NNUE (Efficiently updatable neural networks) were implemented into the now strongest engine; Stockfish. Before this time, neural networks were good at static evaluation but were too slow compared to the benefit they provided over a heuristic hand-crafted evaluation. Their significantly lower calculation speed was made even more severe due to the fact that when engines calculate they have to statically evaluate an

extremely high number of positions.
How these networks are trained:
Curiously enough, these networks are mostly trained using supervised learning where the target label is the hand-crafted evaluation. At first glance it seems silly, but it turns out the neural network is able to generalize better than the heuristic evaluation and thus is more successful.
Note: NNUE differ slightly from a normal neural network, we will go into them in greater detail later on.

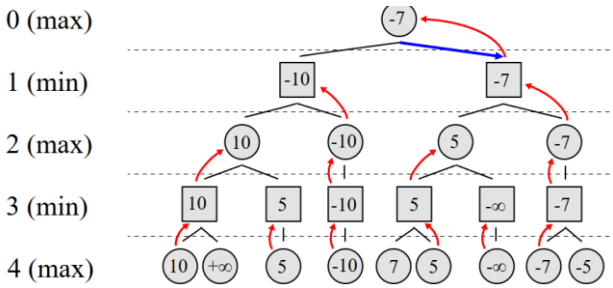
Search:

The 2nd part of the first method is the search part, mainly a minimax search. First, we define a set 'depth': 'd', which is the number of half moves (one for each player) that we will look ahead. We'll explain how the search works using an example. Let d = 2 and assume we are playing white. As white our goal is to maximize the evaluation. Assume we have 2 possible moves and we wish to choose between them. Clearly, we want to take the move which leads to a higher evaluation, the 'max' between the 2 moves. However, we do not analyze these positions statically as we yet to look deep enough. So then, we explore all possible moves from these positions and then since we are now 2 moves deep which is equal to 'd' we evaluate those statically. Since black is the player that 'made' those moves, he will pick the moves leading to a minimal evaluation, after which we will pick the max between those minimums. Hence the name minimax.
Minimax pseudo code:

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

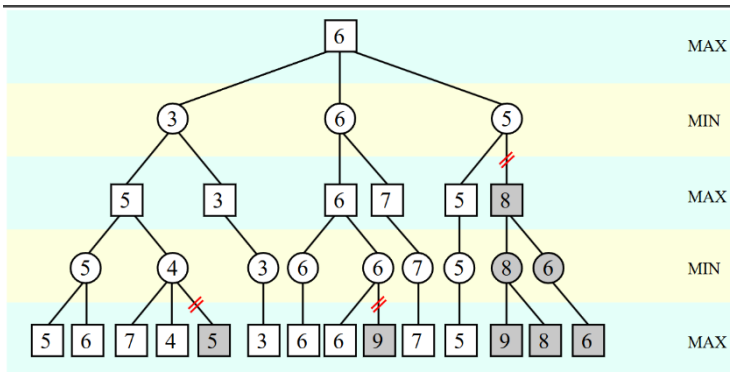
```
(* Initial call *)
minimax(origin, depth, TRUE)
```

Example of the recursion tree post run to depth 4:



As we can see, the search reliably picks the best option assuming our static evaluator is strong. The problem however is that this is extremely costly and has exponential complexity. The average number of legal moves per board position is estimated to be around 30. Meaning at just depth 5 we evaluate $30^5 \sim 2^{25}$ positions. This quickly grows out of hand even for the strongest computers. To improve on this, we introduce a method known as alpha-beta pruning.
Intuition: If a node is trying to maximize and one of its children has returned an evaluation 'e1', it's other children now 'know' that if they find an evaluation 'e2' smaller than e1 there's no need to look at other moves. This is because, since that child is a minimizing node, their evaluation will be upper bounded by 'e2' < 'e1' which means that node is 'uninteresting' to its parent. For a minimizing node we do a similar thing.
Essentially, alpha tracks the highest value seen so far by the maximizing player within a subtree, and beta tracks the smallest value seen so far by the minimizing player.
Illustration:

Grayed out nodes are not actually explored.



This technique greatly reduces the number of explored nodes without hurting accuracy at all.

Beyond alpha-beta pruning there are many possible improvements to add to the search algorithm. We will mention the ones we experimented with and their effects later on.

It's important to mention that even with alpha-beta pruning we still try to check all moves, but we stop early if we pass the alpha/beta cutoff.
Because of this, move ordering becomes very important. If we can search the potentially stronger moves first, we will achieve more cutoffs, thus search less positions overall. For instance, some captures are more likely to yield good results compared to other moves, thus should be looked into first.

MCTS (Monte carlo tree search):

We won't elaborate on this as it's not something we've gone into since the competition constraints don't allow for this to be a successful strategy, however we think it's still worth mentioning.

In a MCTS you use reinforcement learning to directly learn the 'policy', e.g. what move you should play. You learn that by simulating the game all the way to the end and seeing the result: win, loss or draw.
Since you have to simulate to the end, you cannot afford to search all moves. Instead of limit the 'breadth' of the tree by searching simulating only a few lines all the way to the end.
This strategy has been used in engines like Alpha zero (From Google's DeepMind) and in Leela chess 0. In 2017 they overtook stockfish as the best engines, only to later be defeated by him when NNUE were introduced in 2020.

Side notes on opening books and table bases:

Alongside the methods we described, engines use 'opening books' which contains a table of what move to play given a specific early position (If it's a main variant of a known opening). This competition eliminates the need for opening books by setting up the matches starting from a 'random' early game position that is not a main variant of an opening, we do the same in our simulations.

Table bases are slightly different, chess has been solved for any position with 7 pieces or less. Meaning we know the absolute best move as long as there are at most this many pieces on the board in any combination. However, these tables weight around 1 Terabyte and they are impractical for this competition, thus we do not include them.