Gambit Write Up:

PERFT:

Performance testing. Requires going through a standard set of chess positions at X depth, where X is chosen by the tester although typically will be smaller or bigger for some positions based on their complexity. If your move generation passes PERFT by generating the correct total number of nodes and nodes for each next move, then you can be fairly sure that your move generation is also sound. You can determine this by comparing it to a valid sound move generation of another engine.

From a given position, finds all possible combinations of moves up to a given depth. This depth is usually 5-7 and varies depending on position complexity.

Used split perft to debug 6 positions on webperft, then used regular perft for a large test suite.

Website to help debugging of move gen: webperft. Once it was sound on webperft, most kinks were ironed out so could move onto bigger perft suite.

To test your move generation efficiency, you can run PERFT and keep track of the total number of nodes walked per second. You then compare this to another implementation of well known engines to get a comparison. Note that nodes per second is also highly dependant on your computer architecture. So results from PERFT you see here may not align with your testing.

<https://analog-hors.github.io/site/magic-bitboards/>

Magics help ^

Critical Evaluation of The Concept:

Requires high likelihood or a much higher desired eval

Similar concept: Contempt

Tool used: https://analog-hors.github.io/webperft/

Alternative to the path we chose:

Option 1:

Analyze a massive database of games to evaluate moves based on probability. Say you have a 500 elo player, you would look at all games featuring say 300 – 700 elo players and see what the most common move was in that position. This would be like alpha beta with pruning, where you discard unpromising lines early before wasting lots of time calculating to a given depth on a tree you can be fairly sure will result in a worse position. However, in our case instead of pruning trees with a low evaluated position for the engine, we could prune trees with a low probability of occurring. It would require testing to see what threshold this would be, but for example if a line is under 10% chance of occurring, it will prune that branch. Also, if a line is over 50% chance of occurring, then you could search a higher depth than less likely lines.

Some drawbacks include that it would require a tremendous amount of data, and would need to be trained on the Royal Holloway supercluster. To help you understand the sheer amount of data you would need, there are 10^40 possible legal positions in chess, which is 100 quadrillion times greater than the number of stars in the observable universe. [**https://www.liverpoolmuseums.org.uk/stories/which-greater-number-of-atoms-universe-or-number-of-chess-moves**](https://www.liverpoolmuseums.org.uk/stories/which-greater-number-of-atoms-universe-or-number-of-chess-moves)

Furthermore, eventually it gets to a point where no games have been played in a given position, and therefore would have to rely solely on regular evaluation and decision making.

A possible algorithm to be used here would be the Monte Carlo tree search, however in practice it has consistently been shown to be less efficient than alpha beta pruning.

If I could continue this project:

Id like to use machine learning to analyse data of games from multiple skill levels, to play the probabilistic best line

Would be cool if I could create a gambit UI

How it works:

Board Representation:

Motivation For Bit Boards

- We use 6 piece bit boards and 2 colour bit boards

- To find the white pawns, you find the intersection of the desired colour bit board and piece bit board.

- 64 bit unsigned integers used to represent the chess board.

- Board representation uses LERF standard.

- In binary, the square represents the position in the unsigned integer.

- For example: A rook's bit board at the start of the game would look like this:

- 0b1000000100000000000000000000000000000000000000000000000010000001

- Starting from right to left, as normal with binary, u64[0] is true to represent the rook on A1

- Or also thought of as rook on 00. At u64[7] is another rook, because its on H1, also seen as 07.

- Using b tells the compiler to see this number as a binary instead of decimal number and the beginning 0 prevents it from being seen as a variable name.

Helpful Resources I used:

<https://analog-hors.github.io/site/magic-bitboards/>

Move Generation:

Magic Generation:

We call find\_magic(Piece piece\_type, int square) to find magic numbers for a given sliding piece on a given square. For example, a bishop on a1 will have a different magic number to a rook on a1, and a bishop on a1 will have a different magic number to a bishop on a2.

A screen shot of a computer program

Description automatically generated

Attempts to create a table of attacks using a randomly generated u64. If there are no collisions, this u64 works as a magic and we save the magic.

A screen shot of a computer code

Description automatically generated

Using Magics: