Iain Sheerin
HW3
CS76

Introduction

The purpose of this assignment was to create an AI that would return the best possible move in a chess match, given the position. I had to use Minimax and Alphabeta algorithms to quickly traverse the possible states of the positions at deeper and deeper depths, using an evaluation function to estimate the value of a given position.

Minimax

I implemented the minimax algorithm in the MinimaxAI file. When creating the AI, I give its color (white is True, black is False as given by python chess package) and another Boolean for whether Minimax should print its best move at each depth as it uses an iterative deepening search. In terms of ideal depth, I found that 3 is best for a quick response. At depth 4, after playing e2e4 as an openning move, the AI called Minimax 433,585 times, recommending g8h6 after 75 seconds. However, if the depth is set to 3, the AI called Minimax only 14,420 times, recommending g8h6 after 3 seconds. Chess has such a high branching factor so Minimax isn't able to go into much depth without taking long computational times. Ultimately though, a depth is 3 is ideal for actual play because a quick response time is needed for finishing games in a timely manner. I also included a time "penalty", subtracting 0.01 from the score at each depth. This forces the AI to pick the move that reaches a given score "quicker" than other moves. This is most noticeable in end game situations. For example, if there was a checkmate in 2 and a checkmate in 1, without the time penalty, the AI would pick one arbitrarily. However, with the time penalty, the AI picks the checkamate in one.

Evaluation Function

My evaluation function calculated the total material value of the board, multiplied by a color multiplier (1 for white, -1 for black) such that the evaluation would return positive values if the AI had a better material score. Pawns were assigned 1/-1, Knight/Bishops 3, Rooks 5, Queen 9, and King 100. Besides the king, these values are commonly used in quick chess evalutions. I created the file "test_minimax" to test my minimax/evaluation function. I gave my function/AI three tests. In the first test, the AI is given a checkmate in one as white. In the second test, after the human moves, the AI is given a checkmate in one as black. In the third test, the AI has the move and can either take an unprotected queen or protect a knight. The caveat is that if the AI takes the queen, the opposing player can take the knight and checkmate the AI. With the evaluation function and minimax algorithm, the AI passed all of these tests.

Iterative Deepening

In the third test of the minimax testing, mentioned earlier, the effect of iterative deepening is evident. At the first level, the AI recommends taking the unprotected queen because the AI cannot see ahead that the opposing player will beat AI. However at the second and third levels, the AI recommends blocking the mate and not taking the unprotected queen.

Alpha Beta Pruning

Without any more reordering, Alpha Beta at depth 4 seems to perform at the same speed as Minimax at depth 3. In the example 1 in test_alphabeta.py, At depth 4, alpha beta recommends a checkmate after 2,284 calls. As shown in example 2, at depth 4, minimax recommends a checkmate after 2,412 calls. Alphabeta return a move as with the same score as Minimax but was able to visit less nodes. In example 3, At depth 4, Alphabeta returns a move of score -10.03 after visiting 153,330 nodes. In example 3 of minimax.py, At depth 4, Minimax returns a score of -10.03 after 1,095,338 visiting nodes. This shows just how Alpha Beta pruning is efficient at reducing the number of nodes visited. After reordering the moves to have captures looked at first, in example 3 at depth 4, Alphabeta returns the same move of score -10.03 after visited only 35,305 nodes. Overall, these tests showed that Alphabeta is equivalent to minimax at the same depth level, but Alphabeta is able to return moves of the same score while visiting less nodes.

Transposition Table

I implemented a transposition table in a class called Wrapper in the file, TranspositionTable.py. The transposition table greatly decreases the number of nodes visited by AlphaBeta and Minimax. In test_alphabeta.py, in example 1, the number of nodes visited by Alphabeta reduced from 2,284 to 457. In example 2, the number of nodes visited by minimax reduced from 2,412 to 469. In example 3, the number of nodes visited by Alphabeta reduced from 153,330 to 8993. In test_minimax, in example 3, the number of nodes visited by minimax is reduced from 1,095,338 to 57,852. These example show the effectiveness of the Transposition table on reducing the number of nodes visited.

Extensions:

Note that I implemented all of the extensions in the file "AlphabetaAIQ_search.py". Zobrist hashing, was implemented with the original AlphabetaAI and MinimaxAI because it was a part of the wrapper used to make the transposition table.

Recent Work

In the paper "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm" (2017) by David Silver, Thomas Hubert, Julian Schrittwieser, et. al, they discuss a generic self-learning algorithm called Alpha Zero and applied it to games such as Chess. Instead of an alpha beta search, Alpha zero uses a Monte-Carlo tree search that tranverses a tree based on simulations of self play. After 4 hours, Alpha Zero was able to outperform Stockfish, the top chess engine, in a series of chess matches.

Quiescence Search

In order to mitigate horizon effects, I implemented a quiescence search that continued searching move if there was still a capture to be made on the board. In test_qsearch.py, example 1 shows the regular AlphaBeta evaluate the position to a depth level of 4, before moving the rook. The search reaches a depth of 4 and makes 18,175 calls. In example 2, the AlphaBeta with quiescence search evalutes the normal position to a depth level of 4, but with does to a depth level of 18 with th quiescence search, making 372,958 calls, before moving the queen. By going to a depth of 18, the quiescence search is able to evaluate the position fully.

Zobrist Hash Function

I implemented a Zobrist Hash Function within my TranspositionTable.py file. It uses random 64 bit strings corresponding to squares on the chess board to create a hash value based on the XORing the strings corresponding to pieces on the board. This evaluation is quick and has a low probability of collisions, making it ideal as a hash function.

Opening Book

Since the beginning of a chess game has no possible captures, the AI typically conducted random moves while not hanging pieces for the opponent. However, these moves were often bad, as they did not set up a good pawn structure or develop pieces that would be essential for a good mid game. To combat this, I implemented an opening book. Using polyglot from the chess package and downloading an opening book file from user Steve.R at http://www.chess2u.com/t5834p15-gm-polyglot-book, the AI was able to play "smart" looking moves at the beginning of the match, such as the sicilian defense as black against e2e4. The openning book was based on hundreds of games played by grandmasters. One limitation of this implementation, if the opponent plays a move not entered into the opening book, the AI will rely on its AlphaBeta searching to find the best move.

Advance Move reordering

To implement advance move reordering, I created a dictionary which would store the scores of the positions at each level of the iterative deepening search. While conducting iterative deepening search, if a board had been previously explored, it would reorder the next moves to explore. Moves with the highest scores would be looked at first, then moves with lowers scores, followed by unexplored moves. Using the board in example 3 of file test_qsearch.py, at depth 4, the AI explored 146,522 nodes compared to 148,095 that would have been explored, had no reordering been done. Ultimately though, looking at capture moves first was a better reordering for this position and depth level; the AI only explored 34,956 nodes.