Overview

The goal of this project is to create a chess-engine using AI techniques. To achieve this goal, my method is to first formulate a state-space for searching through game-states with alpha-beta pruning, to improve upon alpha-beta pruning via techniques such as quiescence search, to reduce the amount of computation needed via hashing, and to engineer in domain knowledge through opening tables and piece heuristics. Specifically, I have implemented the following:

1. Chess state-space formulation
2. Alpha-beta pruning
3. Quiescence search
4. Transposition table
5. Iterative deepening
6. Aspiration window
7. Opening book
8. Chess-based heuristics
9. MVV-LVA move-ordering

I shall proceed to describe each in-turn, followed by experiments.

Chess state-space formulation

I implemented the state-space as a 2D grid that contains the class “ChessPiece”. Each “ChessPiece” is a super-class of the base class “Piece” which handles functions common among all pieces such as being captured. An important function of “ChessPiece” is that it is in charge of generating all moves that the chess piece it represents can make. A move that the chess piece can make are moves that do not leave its own king in check etc (i.e. legal moves). Moreover, each piece has unique movesets which are all accounted for. It also keeps track of whether a piece is alive or not, and handles things such as promotion and castling rights.

The class “ChessGame” is used to handle the overall flow of the game, to enforce rules, check for checkmates, and handle captures and moves for either the engine or user. It, along with the class “ChessPiece”, are critical for state-space formulation. Specifically, the 2D grid is contained in this class and each cell of the grid contains the class “ChessPiece”. An empty square is represented by an “empty” piece. By holding references inside a list, we can access the pieces easily from other functions; it also helps to avoid duplicating pieces which is a costly operation and would slow down search. To keep track of the player’s turn, “ChessGame” simply tracks the number of moves made as the white side always move on an even number of moves made. To generate all possible moves that pieces from white or black can make, I simply go through each grid of the 2D grid, access the piece referenced, and call the function that generates all possible moves for that piece. By doing so for each piece, we can generate all possible

moves total. This will be used later for searching. “ChessGame” also handles captures, especially unorthodox captures such as en passant, and also castling, as well as promotion. Human players would also have to moves checked by this class, to make sure that the move is valid. “ChessGame” can also undo a move, which proves to be important for searching.

Finally, the moves each piece can make are contained by a set of functions in “moves.py”. These functions check whether the king is in check, how pieces can move, etc.

The relevant code is in “pieces.py” which contains the class “ChessPiece”, “moves.py” which contain functions to enforce movement of pieces, and “chessgame.py” which contains the class “ChessGame”.

Alpha-beta pruning

The minimax variant I used is alpha-beta pruning which prunes values according to current alpha and current beta. The search always begins on the engine’s turn and iterates between max and min nodes expansion. For each node, be it max or min, we get all possible moves that the player on that turn can make via “ChessGame”, then expand each node by playing that move and continuing with either max or min. After the max or min returns, we undo that move with “ChessGame” and update our alpha and beta values. Specifically, for max nodes we update alpha and our current value if the returned value from the expansion is greater than our current value, and prune by beta if our current value, which could be the newly expanded value, is greater than or equal to beta. Basically, the standard alpha-beta procedure (as I have yet to introduce the additional features).

On terminal state, it first checks whether the game has ended and if it did return the utility of winning. Otherwise we use an evaluation function which will be described below.

One thing to note is that for this code and all methods below, I have made, without loss of generality, the assumption that the engine always plays the black piece. This is to simplify the development process but the techniques and methods are applicable to either side of the game.

The relevant code is in “engine/alphabeta.py”.

Quiescence search

One problem with cutting off the search using an evaluation function for alpha-beta pruning is that the evaluation could have been done for a non-quiescent state. A quiescent state is a game-state where there are no pending moves that would swing the evaluation. In chess, a quiescent state are states where there are no captures available. In contrast, a non-quiescent state is a state where a sequence of captures are available. For example, it would be disastrous if the engine captured a pawn with its queen, thinking it

gained an advantage, just to have that queen captured because the original pawn was guarded. Another instance of this problem is the horizon effect, where the engine might think a move avoids a negative event that is inevitable but postponable, simply because it cuts off search before the event can be fully evaluated. For chess, these states generally begin by captures. Quiescence search deals with these problems by continuing the search if any captures are available. It will only terminate once there are no captures left or if the maximum quiescence search depth is exceeded.

My implementation of this is similar to alpha-beta pruning. Once the minimax algorithm reaches its maximum search depth and decides to cut off search, if the last move made by the engine is a capture, then I call quiescence search to better evaluate the position. On each turn, the player-to-move generates the captures it can do, and for each capture we play that capture, call another quiescence search with that capture played, then undo that capture once it returns. The terminal condition of the recursive calls to quiescence search are either when there are no more captures or the search-depth exceeded a limit. For each state expanded by the search, we compute a "standing pat" which is a static evaluation of the current board position for the engine. At the beginning of the search, this is used as a lower bound on the evaluation for the position.

```
stand_pat = eval_position(game, self.color)

if color == "B":
    if stand_pat >= beta:
        return beta
    if stand_pat > alpha:
        alpha = stand_pat
else:
    if stand_pat <= alpha:
        return alpha
    if stand_pat < beta:
        beta = stand_pat
```

Then, depending on whether we are maximizing the score or minimizing the score, we adjust the bound like we would in alpha-beta pruning. Specifically, as black is the engine and it's trying to maximize the score, it would prune by beta, update alpha. The human player as white, however, is trying to minimize the score so it would prune by alpha, update beta. Then, the search proceeds recursively by playing the sequence of captures until terminal conditions are met.

The first terminal condition is the maximum depth is reached. In this case, we simply return the "standing pat". If, however, we have completed the sequence of captures and arrived at a quiescent state, then, depending on whether the node is max or min, we return alpha and beta respectively. Thus, if the capturing sequence is good for the "max" player, then alpha would reflect this since alpha stands for the highest value we have found so far for any choice point along the path for max (i.e. for this capture, we have at least a value of alpha). If, however, it is not good for the player, then alpha would also reflect this, and subsequently the value returned by the minimax's terminal utility would be alpha (i.e. it would be low).

Thus, we see that quiescence search helps to avoid problems with limited search depth and the horizon effect. The relevant code is in "engine/alphabeta.py" as a function of the class "AlphaBetaSearch".

Transposition table

This technique is essentially a hash table containing mappings from state to (depth of this hash entry, evaluation for this state, best move for that state). It is used because a lot of chess positions are reachable via a sequence of moves, so it makes sense to hash the evaluation of each state and reuse them should the opportunity arise. This component is actually key to a few other components down the line, and it itself provides a lot of uses regarding alpha-beta pruning. First, I will describe its use cases, then the implementation.

Transposition table use cases:

1. Better move ordering (described in later sections).
2. In alpha-beta max node expansion, if the current board configuration is evaluated before and hashed in the transposition table, and if the depth of that evaluation is deeper than or equal to the depth we are about to search to for this board configuration, then use this hashed evaluation and return the evaluation and the corresponding move.
3. For min node expansion, similar to 2.
4. Better quiescence search by using the same trick as 2 and 3, except for node expansion within quiescence search instead of minimax search.
5. Used with iterative deepening (described in later sections).
6. Used with aspiration windows (described in later sections).

Clearly, the transposition table is a critical component and its implementation is as follows. To represent each board configuration as an integer, I used zobrist hashing which transforms each board configuration into a long integer by taking into account features such as availability of castling, pieces on each square, player-turn, etc. Then, to further reduce the zobrist hash to a smaller integer, I simply use the modulo operator to reduce it to within a suitable size — this is the final hash key. This means that the final hash key could have collisions, so for each hash entry, I store the zobrist key that generated the hash key so that if I were to use this hash entry, I first make sure that the entry's zobrist key equals that of the current board configuration. In addition, each entry includes the evaluation of the corresponding board configuration, as well as the depth of the evaluation, and the best move found for that position. The replacement policy for a collision is to keep the entry with the greatest depth. Because a collision could be two different boards due to the possibility of two different boards having the same hash key, I simply keep the entry with the greatest depth, and ignore the fact that it could be a different board configuration.

The transposition table is better during the mid-game and end-game as there are less pieces so there is a higher chance of pieces moving into the same configuration which would create a transposition table hit. Details regarding its performance (i.e. transposition table hit) as well as all these other techniques are described in a later section.

The relevant code is in “engine/transposition.py” which stores the Transposition class, and “engine/zobrist_hashing.py” which handles the zobrist hashing. The usage of the table is in the file “engine/alphabeta.py” which uses the transposition table to improve performance.

Iterative deepening

This technique is used to generate transposition table entries which would improve move ordering for deeper minimax searches. The idea is that we first begin by performing minimax alpha-beta pruning with a max depth of 1, then 2, then all the way to the true maximum depth. In the process, the transposition table will accumulate evaluation of positions from lower depths. Then, as we increase the depth, we can use these accumulated evaluations to form better move orderings which would help us to prune more branches of the game-tree via alpha-beta pruning.

In addition, iterative deepening is used in combination with aspiration windows to improve alpha-beta pruning which will be described below.

Implementation is very simple. For every search with alpha-beta pruning, simply change its max-depth to 1, 2, …, all the way to the actual maximum. Do the search for each max-depth, then use the best move found by the actual maximum depth. Remember that the previous, smaller, max-depths are there to assist the alpha-beta pruning of the later, greater, max-depths.

Relevant code is in “alphabeta.py”, where the function “alpha_beta_search” uses iterative deepening to perform iterative deepening minimax search with alpha beta pruning.

Aspiration window

Yet another technique to heuristically improve alpha-beta pruning. Aspiration window increases the speed of alpha-beta pruning minimax search by reducing the search-space. The idea is that we can use the evaluation generated by the previous iteration of iterative deepening to guess the alpha-beta value of the current iteration. For example, if for iterative deepening with max-depth = 3 generated an evaluation of v, we can create a “window” around this v as a guess of the starting alpha and beta value for max-depth = 4, say $\alpha = v - 100$ and $\beta = v + 100$ where in this case “100” is the aspiration window. In contrast, the original alpha-beta pruning uses $\alpha = -infinity$ and $\beta = infinity$ as starting values. By using aspiration windows, the alpha-beta range is narrower so more beta cutoffs or alpha cutoffs are achieved.

Consequently, minimax search is quicker; however, the caveat is that if the evaluation due to using aspiration windows is less than or equal to α, or if the evaluation is greater than or equal to β, then we would have to redo the alpha-beta pruning minimax search with $\alpha = -infinity$ and $\beta = infinity$. The

reason is that the evaluation should not be less than or equal to alpha, since alpha is the least value the max node can achieve, and the evaluation should not be greater than or equal to beta, since beta is the most the max node can achieve. Violating these conditions means that the evaluation we based our aspiration window on has deviated too much from reality, so we have to redo the search using $\alpha = -infinity$ and $\beta = infinity$ to recover from this deviation.

```
alpha = -np.inf
beta = np.inf

for depth in range(1, self.max_search_depth + 1):
    self.search_depth = depth

    state = {
        'game': deepcopy(game),
        'depth': 0,
        'is_capture': False
    }

    value, move = self.max_value(state, alpha, beta)

    if value <= alpha or value >= beta:
        value, move = self.max_value(state, -np.inf, np.inf)

    alpha = value - self.aspiration_window
    beta = value + self.aspiration_window
```

Implementation is once again very simple. Simply perform iterative deepening as usual, and for each evaluation found for that iteration of iterative deepening, form the aspiration window around that value, then assign the window to alpha and beta values. Now, these new alpha and beta values will be used for the next iteration of iterative deepening. If it happens that the next iteration of iterative deepening diverged (value <= alpha or value >= beta), then redo the search with $\alpha = -infinity$ and $\beta = infinity$. Clearly, the key is to choose a right-sized aspiration window that reduces the amount of deviation (i.e. reduce the number of times we need to redo the search). By choosing the right-sized window, the gain in speed is greater than the cost. However, choosing the right window is a hyperparameter and requires testing.

The relevant code is in file “engine/alphabeta.py” within the function “alpa_beta_search”.

Opening book

The problem with using search-based methods for playing chess is that the opening consists of a lot of moves, and searching through them takes time. In particular, openings are important in chess because it dictates how pieces are developed on the board, and not having a good opening can be devastating. Moreover, it is hard for engines to “discover” openings because doing so requires high search depth. Thankfully, the chess literature is very vast and nowadays there are available databases of chess openings that one can add to their engine so that their engine can avoid searching for an opening. Opening books work by storing opening sequences via hash keys which corresponds to chess board configurations. As long as the current board configuration exists in the database, one can query it for one or more moves that follow well-studied openings in chess. However, if the current board configuration is not within the database, then we would have to resort back to searching. When that happens, usually the opening is near

its end anyways, or the opponent has made such a bad move that the engine could simply search for a good counter-play.

I have implemented an opening book for my engine such that for the opening phase of the game, the engine would check if the current board configuration is in the database. If it is, follow the opening book. Otherwise, do alpha-beta pruning minimax search as usual. To implement the opening book, I have referenced the python module "python-chess" which uses an opening book in polyglot format. This format stores openings which are accessed with zobrist keys and stored in ".bin" files. Each key corresponds to one or more opening moves and by binary searching the entry, I query the database and return one or more moves (for different opening sequences). My engine chooses one of these and plays the move, and this continues until the board configuration has no more entries in the database. This saves computation time, especially in the opening when there are many available moves.

The relevant file for opening book implementation is "engine/opening_book.py" and "engine/zobrist_hashing.py". The usage of the opening book is in file "engine.py" which contains the class "Engine"that switches between using the opening book and searching.

Chess-based heuristics

It is critical for alpha-beta pruning to have a good heuristic. The reason is that the pruning operation is highly dependent on the evaluation of board configuration, and having a good heuristic would, obviously, prune more branches than a bad heuristic. Moreover, a heuristic determines how well your engine plays (unless it is at the end-game with checkmate soon, then the winning utility would suffice) so having a good heuristic is crucial for the performance of minimax search. Therefore, I have hard-coded some chess heuristics as my evaluation function and the implementation are as follows:

For each color, either white, or black, I did a weighted sum of the following features.

1. Material score. This is the sum of the values of each piece that are alive. Each piece has their own value which is also a heuristic. More specifically, the values are

```
if piece.name == "pawn":
    material_score += 100
elif piece.name == "knight":
    material_score += 350
elif piece.name == "bishop":
    material_score += 350
elif piece.name == "rook":
    material_score += 525
elif piece.name == "queen":
    material_score += 1000
else:
    material_score += 10000
```

where the king piece has the greatest value of 10000. These values are heuristics formed by chess players from their experience. The king needs a large value to prevent minimax under-valuing the king piece — if you lose the king you lose!

2. Reward bishop pair. The bishop pair is very powerful so I increase the evaluation if the bishop pair is alive.
3. Penalize rook pair. The rook pair is redundant to have so reduce the evaluation if the rook pair is alive. Obviously, the total sum of the rook pair will be greater than this penalization; this is to make sure that the engine prefers some other pairs with the same value over rook pairs.
4. Penalize knight pair. Similar to 3.
5. Reward for pawns alive. Pawns are important in general for end-games, so I increase the evaluation if there is at least one pawn alive.
6. Pawn structure. Good pawn structure is also important in chess. I increase the evaluation if there are past pawns and reduce the evaluation if there are isolated pawns or doubled pawns.
7. Queen is alive. This is here to ensure that the engine values the queen piece as it is the most important piece in chess next to the king piece.

These features are summed together with weights for each feature. To form the final evaluation, I compute the above weighted sum for both black and white. Then, I minus white's evaluation from black's evaluation — this is the final evaluation fed into the search algorithm.

The relevant file is "engine/heuristic.py" which contains the evaluation function.

MVV-LVA move-ordering

This last technique is used to improve the node selection of alpha-beta pruning so as to prune more nodes. It effectively improves node selection on the left-side of the game-tree so as to prune away more nodes on the right-side of the tree. MVV-LVA is short for Most Valuable Victim, Least Valuable Aggressor.

When choosing moves to play, the first move that is always played first is an entry from the transposition table, if any. This is because the transposition table entry stores a "best move" up to a certain depth for a board configuration, and by playing that move first we will adjust our alpha and beta values to prune away obviously bad moves.

Then, the moves that are played are capture moves, where the aggressor captures the victim; this is because these moves have the largest swing in evaluation so we should try them out first. Among these moves, I order them via MVV - LVA. So, the priority of the capture is dependent on the value of the victim piece and the value of the aggressor piece. For example, a pawn has low value, so it capturing a high value piece will have high MVV-LVA value. In contrast, a queen capturing a pawn will have a low MVV-LVA value. Essentially, the equation that determines the priority is $piece_{value}(victim\ piece) - piece_{value}(aggressor\ piece)$. I order the captures moves according to this heuristic, largest value first.

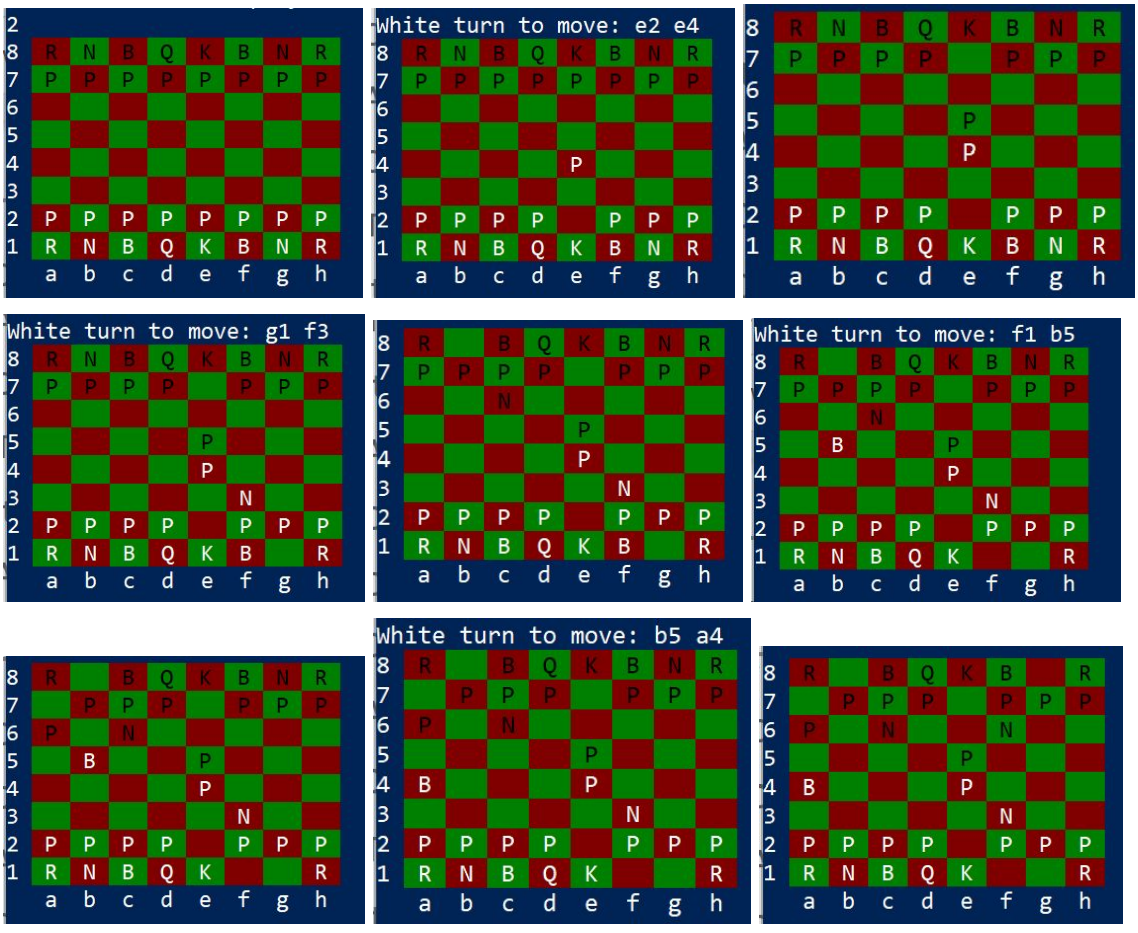
Finally, the remaining moves are all assigned zero priority. These moves are the last moves to test, and are hopefully pruned away.

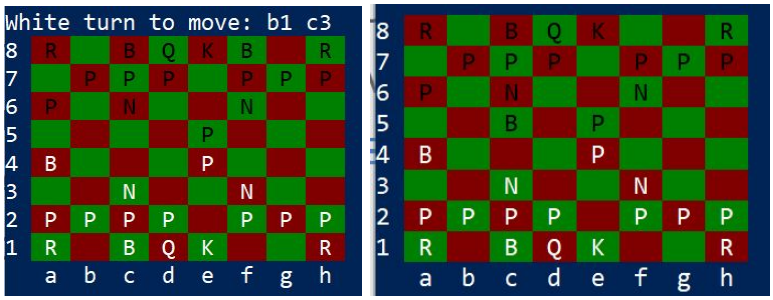
The relevant code is in “engine/alphabeta.py”, in the function “get_move_ordering”.

Experiments

Opening book experiments:

To test the opening book, I played the Ruy Lopez opening (a common chess opening) against the engine to see if it will follow the line. It did follow the lines, no searches were involved, and below are the illustrations (moves are from left to right, top to down). I am white and the engine is black.

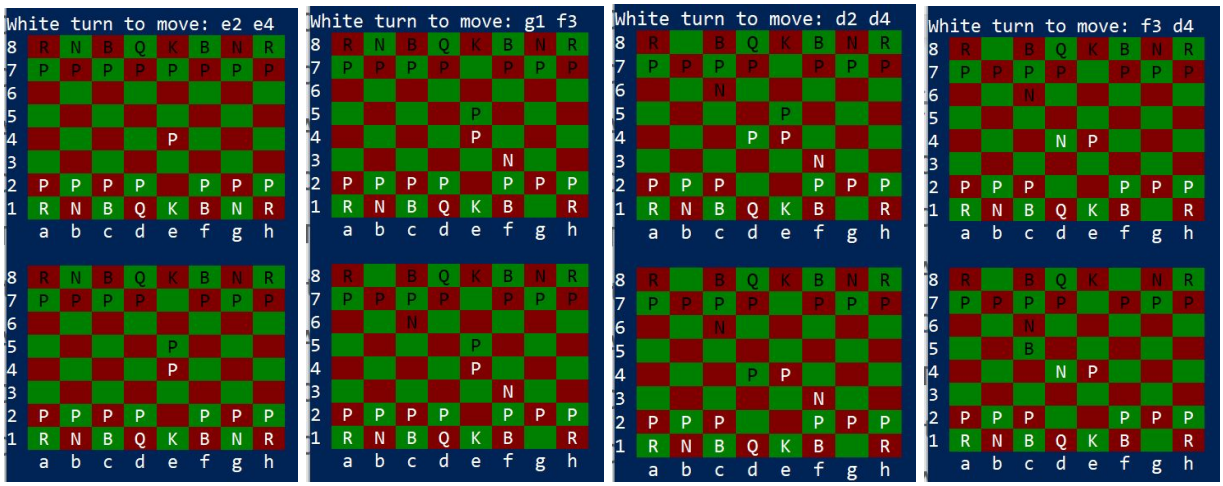




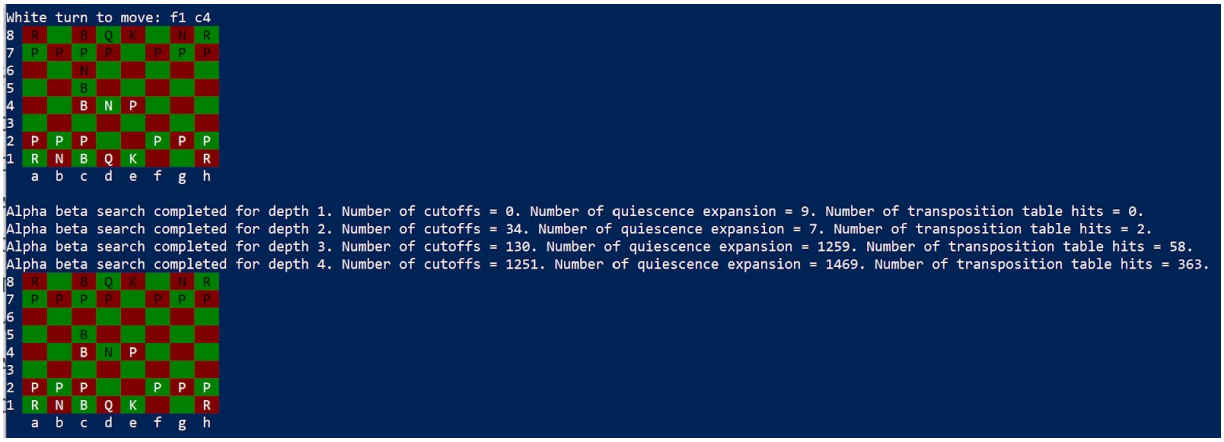
By following an opening book, we effectively increase the search depth without needing actual computation. These moves are instantaneously played by the engine and once the pieces are set, we effectively reduce the number of possible move combinations for later searches. We see that the opening book works, what is left is to test how the search component of the engine performs.

Search techniques experiments:

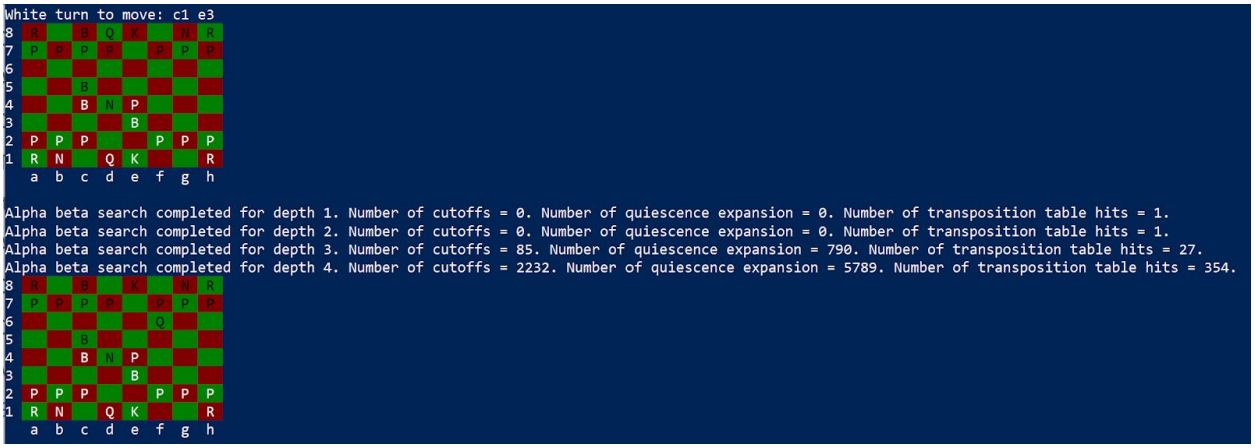
To test the effectiveness of alpha-beta pruning (i.e. the effectiveness of move-ordering), the number of quiescence searches, and the effect of iterative deepening, I played (as white) against the engine to see the computational complexity. I used a minimax search depth of 4, a quiescence search depth of 5, an aspiration window of 1000, and a transposition table of size 100000.



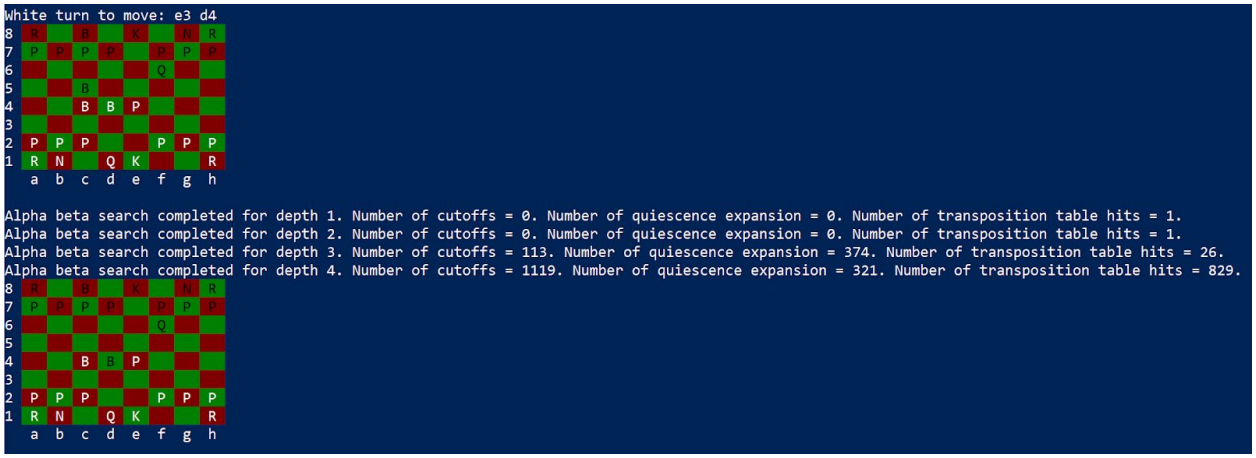
For the first eight moves, the engine followed its opening book. On the ninth move, I deviated from the opening and the engine began its search.



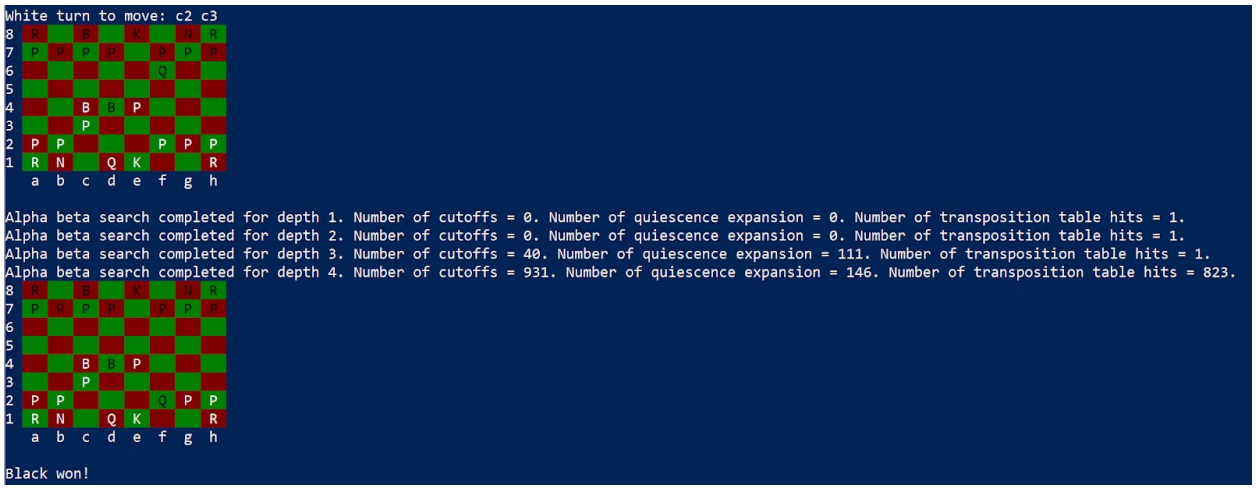
We see the iterative deepening in action as the depth for each alpha-beta pruning minimax search increases from 1 to 4. The number of cutoffs increases non-linearly as depth increases — this is a good sign. We see that the number of quiescence searches increases as the depth increases. Generally, the number of quiescence searches could reach up to ten thousands so we actually have a very good number of quiescence searches. The reason is due to the MVV-LVA move-ordering we used: with a good move-ordering the quiescence search could be pruned via alpha-beta bounds which is what is happening here. As for the number of transposition table hits, we see that it increases with increasing depth — another good sign, as one advantage of using iterative deepening with a transposition table is to fill it up with entries beforehand so as to get more transposition hits later. I then continued the game by playing a move, and the engine made its response:



Again, the search displays similar good properties. I continued the game and so did the engine.



This time, we see that the number of cutoffs are less than before. Thankfully, the number of quiescence searches needed is even less, moreover, the number of transposition table hits has increased. It makes sense that we would have more transposition table hits as the game continues because there would be more entry hashed into the transposition table. Finally, I play a move leading to what is called the "scholar's mate" to see if the engine's search could discover it.



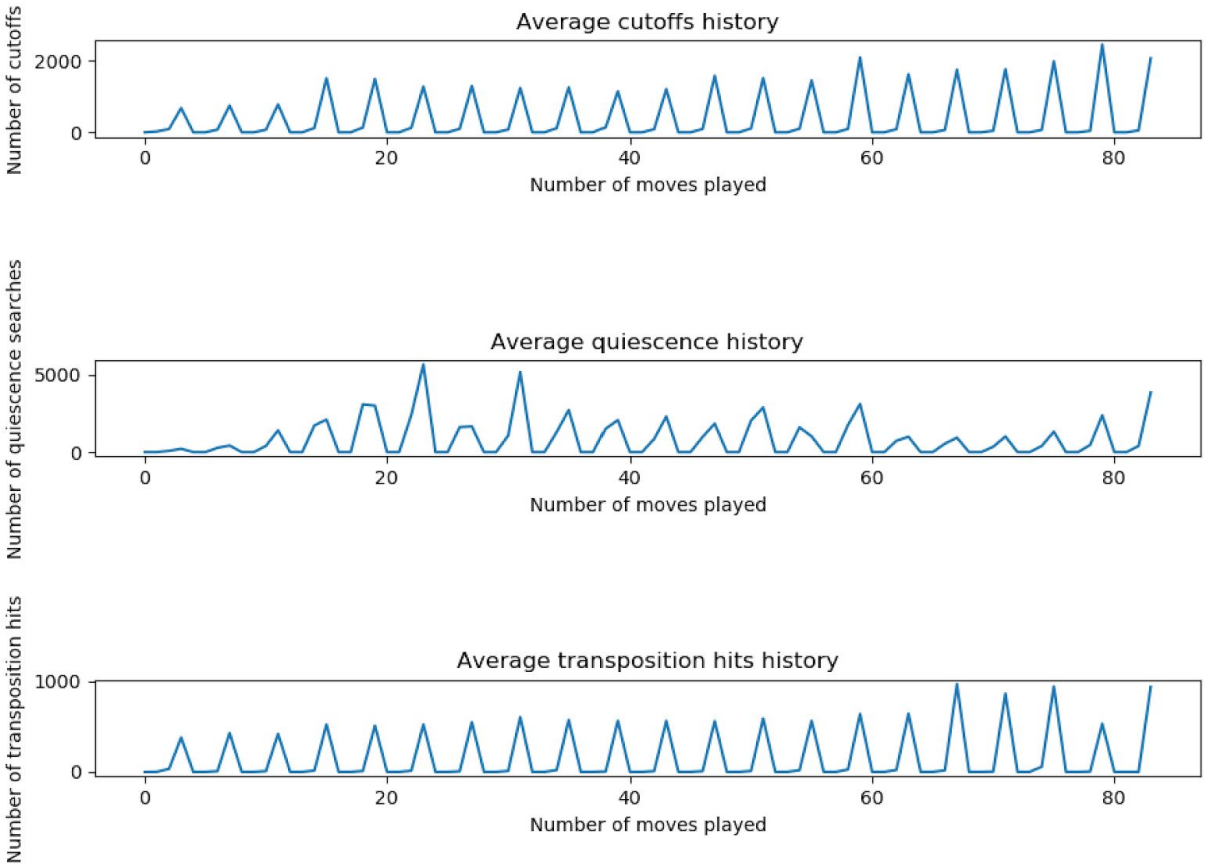
Of course it did. Overall, we observe that the combination of techniques helped us to keep the search computational time low. Each opening move by the engine is instantaneous and each search is around a second or more depending on the position. Furthermore, the number of cutoffs are high, the number of quiescence searches is kept low, and the number of transposition table hits increases as the game progresses. I have also tracked whether the aspiration window was too large, and fortunately, 1000 is big enough to prevent any divergence.

Benchmark against random opponent:

To test how the engine would perform against an opponent that randomizes its strategies, as a baseline benchmark of the engine (it should comfortably defeat a random opponent). The program quickly beat

every single random opponent. To test this, you can read the section below regarding instructions to test my program.

Analysis against random opponent



These plots show the statistics of the engine against a random opponent averaged over 10 games. Overall, we observe that the highest numbers reached by quiescence search is around 5000, highest number of cutoffs are around 2000, and highest number of transposition hits is around 1000. It is interesting how the peaks are periodic with moments in-between where the curve is almost flat. This is actually due to iterative deepening, as the smaller depth searches are collecting info to fill up the transposition tables etc — hence the flatlines, followed by the peaks which utilizes the info and cutoff values from aspiration windows to acquire more cutoffs for the subsequent larger depth searches due to iterative deepening.

Instructions to run the program

1. Enter the directory of the folder “chess”.
2. In your command line, enter “python board.py”.

3. There will be 3 modes: Random play, Human vs AI, and Manual play. Random play is where the engine plays against a random opponent; to select this, enter 1. Human vs AI is where you play against the engine; to select this, enter 2. Manual play is to test the state-space, so the human plays both sides of the board; to select this, enter 3.
4. To make moves, use “from square position” to “another square position” where both positions are space separated. For example, if the king is located in “e1” and you want to move it to “e2”, the command would be “e1 e2”.
5. Enjoy!