Precision Play: Implementing Adversarial Search in Chess

Anant Maheshwari

April 14, 2024

This report offers the development and performance analysis of an AI-based chess engine implemented in python using the python chess library. The primary goal of this project is to assemble an agent able to accurately evaluate and recommend the most beneficial sequence of moves for a given chess position. The program efficiently explores the vast tree of possibilities using the minimax algorithm further optimized by alpha beta pruning. It also includes a simple move ordering heuristic to prune maximum branches as early as possible. The integration of an opening book as a pattern database further enhances the engines capabilities. The engine utilizes a simple piece table heuristic to evaluate the value of leaf nodes. Through this simplistic model we can get a true understanding of the power of these searching techniques and the massive scope that lies ahead in chess AI.
Keywords: adversarial search, mini-max algorithm, alpha-beta pruning, FEN (Forsyth-Edwards Notation), python chess library, pattern database, piece table heuristic

1 Introduction

The realm of board games offers a captivating arena for exploring the depths of Artificial intelligence (AI) search algorithms and their profound impact on strategic thinking and decision-making techniques. Among these games, chess stands as a timeless classic, imparting an expansive canvas for delving into the intricacies of AI-pushed gameplay. Chess is an iconic recreation that has long tested human intellect, demanding gamers to anticipate their opponent's moves while strategically plotting their direction to victory. With its myriad strategic opportunities and infinite board configurations, chess affords an impressive venture for AI systems.

2 Background and Motivation

The landmark victory of IBM's Deep Blue over international champion Gary Kasparov in 1997 marked a pivotal moment in the history of AI and chess. It not only demonstrated the overwhelming computational power of machines but also shifted the focus from human superiority to expertise and harnessing the capabilities of AI in strategic decision-making. In this context, the study of AI-pushed chess gameplay offers priceless insights into the intersection of technology and strategy, with implications achieving far beyond the confines of the chessboard. The minimax algorithm is a selection-making algorithm commonly utilized in two-participant, 0-sum games like chess, wherein one participant's benefit is another player's loss. However, the exhaustive search area of chess makes it computationally infeasible to discover all possible movements to their terminal states. Hence, the incorporation of alpha-beta pruning will become crucial to reduce the hunt space and enhance the efficiency of the search algorithm. The motivation behind this challenge stems from the choice to create a chess engine capable of competing with human gamers and other present chess engines. By implementing and optimizing search strategies which include minimax with alpha-beta pruning, we aimed to develop a chess engine that can examine positions and suggest the best moves in the given position. [1]

3 Literature Review

Adversarial search techniques are pivotal in AI research, particularly in game playing. The minimax algorithm, introduced by John von Neumann in 1928, is fundamental in developing chess engines. It recursively traverses the game tree to find the optimal move for both players. However, as the game tree depth increases, so does the number of nodes, straining computational resources. To address this, optimizations like alpha-beta pruning, proposed by Donald Knuth and John W. Tukey in the 1950s, are crucial. Alpha-beta pruning cuts down unnecessary branches in the search tree, reducing the search space and enabling deeper exploration within reasonable computational limits. [2] We looked at the documentation of the Python chess library. It has several useful and fast functions based on hashing principles that can easily generate the possible moves in a given position faster than the normal implementation of the same. Also there is an efficient function to convert the FEN notation into the board configuration that we used in our project.[3] We took into account several medium articles and referred some research papers regarding the same. In most of the very basic chess engines developed, they just went with the approach of implementing the alpha-beta pruning and relying on the hardware advancements more to compute the moves efficiently. Implementing the normal alpha-beta pruning is a great advancement over the naive mini-max algorithm but this it is very resource consuming as just for a depth of 4, we need to evaluate tens of thousands of positions.[4] A great advancement in this regard is the implementation of move ordering. Move ordering is just the prioritization of one type of move over the other in the process of calculation of the best move. So, in the process the move ordering is done in the following order: checks, captures and threats. Checks are the immediate attack which can be done by the player on the opponent's king. Captures are the opponent's high value piece that can be captured by the player's comparable or lower value piece. Threat is a positional move that can provide a great position to the player in the next few moves. This prioritization further reduced the positions to a few thousand at depth 4.[5] Also a better advancement to their approach is the implementation of the opening's database. This is also very efficient as in the opening part of the game almost all the popular moves are encoded in that database and according to the player's move a best response is generated from the database and the time taken for the computation is saved.

4 Problem Statement

The project focuses on implementation of a chess position evaluator that evaluates the given board position and generates a line of the best moves upto a particular depth.
Input: A board position is fed into our agent in the form of Forsyth–Edwards Notation (FEN) format. FEN is a compact way to represent the state of a chess game using a single line of text. Also the depth to which the engine must evaluate can be a part of the input with the default evaluation depth being four.
Output: The output consists of two parts. The first part consists of a line of best chess moves comprising of moves which are evaluated by our chess engine. The number of moves depend on the depth that we gave as an input. The second part of the input comprises of the number of positions evaluated that gives us the measure for whether the technique we use are optimising the number of moves evaluated or not.

5 Objective

The objective of our project is to leverage Adversarial Search algorithms, specifically the Mini-Max with AlphaBeta pruning technique, to develop advanced AI systems capable of mastering strategic gameplay in chess. Through the implementation of these algorithms, our AI agents will navigate the complexity of the game boards, strategically analyzing potential moves to determine the most optimal path towards victory. Moreover, our project aims to explore the depths of strategic thinking uncovering the intricacies of risk assessment and reward optimization. Ultimately, our endeavor seeks to push the boundaries of AI in strategic gaming, showcasing its ability to rival and potentially surpass human-level performance in these classic board games while unraveling the mysteries of artificial intelligence-driven strategic decision-making.

6 Solution Approach

6.1 Parsing the position

The standard for communicating a chess position to a computer is the Forsyth Edward Notation or the FEN string. This string provides all relevant information required to recreate the game such as

- The configuration of the board.
- The color of the side to move.
- Availability of castling rights.
- Possibility of en passant captures.
- The number of half-moves since the last pawn move or capture.

Our Engine is built on the **python chess library**. The library has built in functions to parse the FEN string and construct a board object. This board object can now be explored and evaluated by the evaluation functions. [3]

6.2 Evaluation

In order to search for good positions, it is necessary to understand what makes a good position good. The most simplistic way of describing a position's strength is to compare the total value material of each side. But total piece value on its own does not give the full idea about a position. Hence it is important to include the position of pieces in the evaluation as well. For this, we use piece square tables which alter the value of a piece depending on which square it sits on. For example, pawns should progress up the board and knights should be near the center of the board.

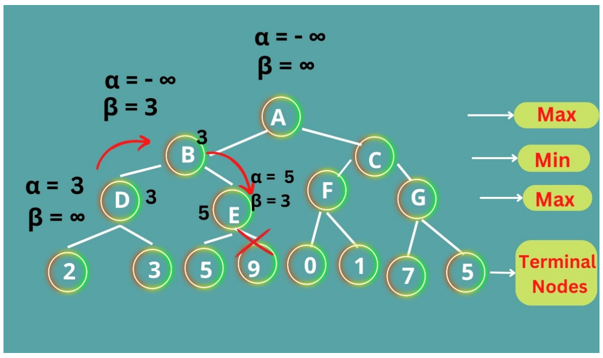

Figure 1: Alpha-Beta Pruning

6.3 Searching With Minimax

Minimax is a search algorithm that finds the next optimal move by minimizing the potential loss in a worst-case scenario. Chess is a game of perfect information, by looking at the board it's possible to know exactly what an opponent is capable of. However, this search for moves is limited by the evaluation function and the depth that computing resources can reach. The search space is a tree of legal moves which grows exponentially at every level (the average branching factor is around 35). By the time the tree is explored, the path to many future boards is known as well as which path restricts the opponent's possible gains the most.

6.4 Alpha-beta pruning

Our engine uses alpha-beta pruning as an improvement over the naive minimax algorithm — which does not fare well against the exponential nature of chess. The idea is simple; branches of the search tree can be eliminated when it is clear that another branch shows more promise. This significantly reduces the number of positions required to be explored. By reducing the depth of branches that will not bear fruit we can search deeper down the better, more fruitful parts of the tree. The speed of alpha-beta pruning can be further increased by applying move ordering. By this the more promising branches of the tree are searched first. This helps in pruning off maximum branches as early as possible. Move ordering cannot be 100% accurate but it's a powerful optimization. For example in chess, we order the exploration of moves by the following priority:

1. Checks.
2. Capturing a higher-value piece with a lower-value piece.
3. Capturing a lower value piece with a higher value piece
4. All other moves.

[2]

6.5 Openings Database

One glaring limitation the engine faced was its inability to look ahead far enough to navigate the tricky opening phase of the game and reach "theoretically" good positions. To address this, we've integrated a text database into our project, containing hundreds of opening board positions in the FEN format, along with their "theoretically" optimal moves. Hence in the opening, we first search the database for the current board position. If a match is found, the engine immediately returns the move associated with that position from the database. However, if the position is not found in the database, the engine proceeds with its Minimax search algorithm to determine the best move. To avoid wasting unnecessary compute time, the engine does not search the text database for the future moves once it does not find the current position. The intuition behind this is simple. Since it is an opening database, if the board position did not exist in the database on move 5, it is very unlikely the position will exist on say move 7.

6.6 Handling incomplete evaluations

Despite all the mentioned optimizations implemented, the engine could only search up to a depth of 4 half moves in reasonable time. However, at such a shallow depth, there remained a risk of returning incomplete evaluations. he inherent volatility of chess positions meant that a seemingly favourable position at the fourth half move could drastically change by the fifth half move due to captures or even checkmates. To handle this, upon reaching depth 0 (the base position being depth 4) the engine searches for an extra 2 negative depths. For these negative depths however, the engine only examines the forcing moves ie checks and captures. This modification handles the issue of incomplete evaluations by smartly finding a middle ground by searching beyond 4 half-moves but faster than the full 6 half-move search.

7 Result and Inferences

The development of the AI chess engine making use of the alpha-beta pruning and minimax algorithm yielded promising outcomes. The engine successfully evaluated positions and suggested optimal moves up to a depth of 6 within reasonable computational time. However, beyond this depth, the evaluation process became significantly time-consuming, indicating computational limitations. Furthermore, the observed computational limitations beyond a depth of 6 suggest the need for further optimizations or hardware enhancements to enable deeper searches without sacrificing computational efficiency. The implementation of move-ordering strategies demonstrates the importance of heuristic-

Algorithm 1 minimax(board, depth, alpha, beta, txt)

```
1:  Initialization:
2:  Set $bestline$ to an empty tuple.
3:
4:  Opening Book Lookup:
5:  if txt is True then
6:      Search for the current board position in the opening book file.
7:      if found then
8:          Return the corresponding move sequence from the opening book.
9:
10: Generate and Order Moves:
11: Generate legal moves for the current board state.
12: Order the moves using the $order\_moves$ function, either considering check or not, based on the ignore parameter.
13:
14: Maximizing Player's Turn (White):
15: for all ordered moves do
16:     Push the move onto the board.
17:     if the move results in a checkmate then
18:         Set $cureval$ to positive infinity and $line$ to ("mate",).
19:     else if the move results in a stalemate then
20:         Set $cureval$ to 0 and $line$ to ("draw",).
21:     else
22:         Recursively call minimax with decreased depth.
23:     Pop the move from the board.
24:     if $cureval$ is greater than alpha or the condition for updating alpha holds then
25:         Update alpha to $cureval$.
26:         Update $bestline$ to the current move and the line obtained from the recursive call.
27:     if alpha is greater than or equal to beta then
28:         Break the loop.
29:
30: Minimizing Player's Turn (Black):
31: Same as step 6 but for the minimizing player (Black), updating beta instead of alpha.
32:
33: Base Case:
34: if depth equals -2 then
35:     Return the evaluation of the current board state by the $boardeval$ function and an empty tuple.
36: if depth is 0 then
37:     Evaluate the current board state using $boardeval$ and store the result in $stopeval$.
38:     if $stopeval$ is greater than or equal to best $boardeval$ returned at -2 depth then
39:         Return $stopeval$ and an empty tuple.
40:     else
41:         Return the $boardeval$.
42:
43: Return the Best Move Sequence:
44: if it's the maximizing player's turn (White) then
45:     Return alpha and $bestline$.
46: else if it's the minimizing player's turn (Black) then
47:     Return beta and $bestline$.
```

based optimizations in improving search efficiency. Prioritizing moves based on checks, captures, and threats showcases the effectiveness of such strategies in reducing computational overhead and enhancing overall performance. Also making use of the already computed data in the form of a database improves the efficiency of our engine in the opening part of the game. The following example further demonstrates the magic of the Alpha-Beta Pruning and Move Ordering. At evaluation depth=4:

Table 1: Summary of Positions Examined

Method	**Number of Possible Positions**
Total	1,865,875
with Alpha-Beta Pruning	57,142
with Move Ordering	2,541

According to the above statistics, the use of alpha-beta pruning and move ordering is much more evident in searching for the best move fast by reducing the search space by almost 99%.

8 Future Work

In terms of future enhancements to this project, the primary focus revolves around the bettering of the board evaluation heuristic to get a more accurate evaluation of the position. Especially on the king safety front. This can be done by incorporating learning strategies to self-compute a heuristic function. Further, we can integrate an endgame table base to make the engine more accurate in the endgames where a normal search may not be able to find the right winning idea. Overall, the area of chess programming is extremely vast and offers many fronts for the improvement of the current model to hopefully play "the perfect game of chess.

9 Conclusion

In the end, the improvement and assessment of the AI chess engine using adversarial search strategies have furnished precious insights into the talents and boundaries of modern-day methodologies in AI-driven chess-playing engines. The implementation of the minimax and alpha-beta pruning algorithms demonstrated their effectiveness in evaluating game positions and suggesting the greatest movements within affordable computational time.

10 References

We have cited all the references in the report, we referred while working on our project.

References

[1] "Chess terms," https://www.chess.com/terms/chess.

[2] O. Healey, "Building my own chess engine," https://healeycodes.com/building-my-own-chess-engine, 2020.

[3] "Python chess library," https://pypi.org/project/chess/, 2010.

[4] "Simple chess ai step-by-step," https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/, 2017.

[5] "Chess programming wiki," https://www.chessprogramming.org/Main_Page.