Name: Pranay Agarwal

UID: 3035557806

AI Chess Mini-Project

Contents

[**Bot Capabilities and Chess Engine** 2](#_Toc57256853)

[**Engine Evaluation** 2](#_Toc57256854)

[**Search Engine** 2](#_Toc57256855)

[**Alpha-Beta Pruning and Negamax** 2](#_Toc57256856)

[**Evaluation** 3](#_Toc57256857)

[**Future Improvements** 4](#_Toc57256858)

[***Move Complexity:*** 4](#_Toc57256859)

[***Opening Position:*** 4](#_Toc57256860)

[***Lack of Game-Time-Specific Moves:*** 4](#_Toc57256861)

[***Horizon Effect:*** 5](#_Toc57256862)

[***Common Pawn Formations:*** 5](#_Toc57256863)

[**Performance Metric** 6](#_Toc57256864)

[**Chess Bot 400 ELO** 6](#_Toc57256865)

[**Chess Bot 1000 ELO** 7](#_Toc57256866)

[**Chess Bot 250 ELO** 8](#_Toc57256867)

[**Chess Bot 700 ELO** 8](#_Toc57256868)

[**References:** 9](#_Toc57256869)

# **Bot Capabilities and Chess Engine**

The AI-bot follows a basic chess-ruleset and uses alpha-beta pruning with a negamax algorithm to calculate the best move.

The AI looks to a **depth of 3** and analyses all valid moves that can be made (accounting for pieces that cannot move due to a potential checkmate. The AI is slightly slower due to having to check if moving a piece leaves the king in check (and therefore using that discard certain moves from the nodes to expand). By removing the no-check safeguard, the AI runs about 8 times faster, however it is more likely to get stuck in the horizon effect and lose material or a position.

The chess ruleset was designed by modelling each piece as a separate class, with a superclass for abstraction of common functionalities. Each piece can move according to the official chess rulebook, however, moves like castling and en-passant are not recognised due to the encoding of each piece’s move history and the excessive memory that would take up during negamax search. A board class was made to keep track of the board state, make sure no side is under check, and to handle moving of pieces (including pawn promotions to a queen). Finally, a Game class was created to generate moves and states of each stage of a game to use in the Engine that controls the AI. The Engine implements the negamax search and position evaluation which is controlled from the main.py file.

# **Engine Evaluation**

## **Search Engine**

The search engine checks for every legal move that does not leave that side’s respective piece in check. This is done by getting all possible moves for each piece. Then, for each move, the board checks if any opposing piece is attacking the king by generating possible moves for that piece. This increases the time complexity by an additional n2 but prevents the AI from accounting for “illegal moves”.

## **Alpha-Beta Pruning and Negamax**

The negamax search was used for this project. The assumption made, was that black’s position evaluation is simply negative of white’s position evaluation since they are competing against each other, and a win for one means a loss for the other. Therefore, the alpha beta function calculates the score as the **negative of the score for the opposing side after each specific move** and always tries to maximise its own score.

If at any time, score exceeds beta, then we know that that score is the best for the given subtree and return that score. Alpha is kept track of as the lower bound, and if any score exceeds alpha, we reset alpha to that score. When calling negamax for the next move, we pass alpha as –(beta), and beta as –(alpha) to reverse the requirements and make sure that both sides are maximising their respective scores.

## **Evaluation**

The current position evaluation is done based on 3 factors:

1. Face value of remaining pieces
2. Value of position occupied by each piece
3. Value of each piece’s utility to the game

**Face value** of remaining pieces refers to the value of each piece remaining on the board. The face value assigned to each piece is detailed below:

* Pawn – 10
* Bishop – 30
* Knight – 30
* Rook – 50
* Queen – 90
* King – 1000

The king is given an absurdly high rating so that the AI knows that the king cannot be sacrificed to get a better position. This is a metric very commonly used in different chess AIs, however the problem with this is that the AI plays only pushes its pawns ahead since it is very cautious about losing its pieces with higher values. This creates a very slow-paced game without many capturing moves since the AI is doing its best to preserve all its pieces. Therefore, another metric was to be added.

In addition to face value, I added another common tactic called **“Pieces-Square” rating**. This is an additional scoring evaluation for each piece based on its position on the board. I referred to the commonly used pieces squares table used on chess.wiki but changed some values around to imply preference for these positions. Some important observations were:

1. Knights are better placed in the centre of the board, rather than on the corners of the board. This encourages the AI to move its knights from the starting position into the gameplay.
2. Rooks are better on the second rank rather than their original position since they can provide more cover and utility. This encouraged the AI to move its pieces to make way for the rooks.
3. Bishops are better placed in the centre of the board rather than the edges. This encourages the AI to move its bishops from their starting positions into gameplay
4. The queen is most useful in the centre of the board, however its centre value is much lower than that of smaller pieces since there is a higher chance of losing the queen. This incentivises the AI to bring out the queen if it poses a valid threat.
5. The pawns are best in the centre of the board (i.e. e4, d4, e5, d5), whereas the pawn poses the biggest threat at the end of the board. This tells the AI that pushing its pawn to the final rank will yield it a very good position (due to the pawn promotion).
6. The king is safest on the last rank. This prevents the AI from moving its king around too much, especially not into the centre of the board where it could get check-mated easily.

These changes accounted for 10% of the scoring evaluation. This made a significant improvement to the gameplay, however, the bot would only move one piece around the board very often since it focused solely on preserving its pieces and yielded a better piece-positioning to me by not developing its other pieces. To fix this issue, I sought to add something that demerited its position for not having movable pieces, called “Piece Utility”.

**Piece Utility** is a metric I introduced that checks how many legal moves a piece can make with respect to the maximum number of moves it should be able to move. For example, at the centre of the board, the queen can make a maximum of 28 moves. However, if the queen is trapped behind the pawns, it will not be able to access any of those moves, and therefore piece utility encourages the AI to make the most use of its pieces. If a piece on the board has 0 available moves, its value is halved since it is not contributing anything to the board. This incentivises the development of pieces to the AI. A piece’s utility (if it is non-zero) contributes to 2.5% of the evaluation. It is calculated by dividing the number of available moves by the maximum number of moves. However, this metric is not used for pawns and kings, since that would leave to unnecessary development of pawns and kings, leading to a worse position.

After the piece utility, the AI bot started playing more typically. The addition of piece utility discouraged it from playing moves like a5 and h5 and instead play moves like e5 since that allows development of multiple pieces at the same time. After this change, the AI played typically better opening moves and did not end up in an underdeveloped position.

## **Future Improvements**

While the AI bot plays averagely, it faces a few problems. Mainly in the form of reduced move complexity, the horizon effect, predictable opening positions, and a lack of game-timing-specific movements. In a future iteration of this project, I would like to implement some of these changes so that the bot can make more informed moves and be a much better player.

### ***Move Complexity:***

The lack of access to moves like castling and en-passant restrict the possible moves for the bot. For example, proper coding of these moves could provide the AI with a better strategy to develop its rooks and safeguard its king. Furthermore, en-passant would give it more options when it pushes the pawns past its half of the board. These added moves could give it a larger roster, and therefore a better chance of winning a better position.

### ***Opening Position:***

The AI bot has a predictable set of opening moves that are caused by the want to develop pieces and get a good position. This allows it to fall into common opening traps like the Stafford gambit. In a future iteration I would like to suggest it some potential counters to common openings and their traps so that it could avoid them and attain a better position during the mid-game.

### ***Lack of Game-Time-Specific Moves:***

The evaluation of each piece’s position is general right now. However, in actual gameplay, the position of pieces becomes much more crucial during the end of a game. However, indicating when the end of a game is to the AI is a very difficult ordeal. In the future I would like to have a dynamically programmed position-squares matrix for each piece. For example, in the endgame, it is better for a king to go to the centre and kill the opponent’s pawns, but the current matrix dissuades the king from going to the centre of the board. This could prove severely problematic in the long run.

### ***Horizon Effect:***

Often, the AI would leave its piece unguarded in front of an attacking piece and shift its focus to another piece. This happens because it thinks it might be able to capture another piece, because it can only look till a depth of 3. This causes the unfortunate circumstance where the opponent prevents that piece from being captured and in the process the bot loses its piece unnecessarily. To prevent this, I tried to ensure that a capturing move’s position was further evaluated, however this led to an exceeded recursive depth and I was not able to debug the exact cause. Therefore, this is a feature I would also like to implement in a future iteration of this project.

### ***Common Pawn Formations:***

The AI is very prone to making common chess mistakes like doubling up its pawns, or having its king unguarded by any pawns, or failing to make a solid chain of pawns to help it in the endgame. This is unfortunate since a strong pawn structure can heavily influence the play of the game. Adding checks for pawn structure would require additional iterations, further increasing the time complexity of getting the best move which would be unfavourable for the scope of this project, and hence I ignored that aspect. In a future release, I would like to deduct position evaluation points for having doubled up pawns, or an unguarded king, or add points to a position if the opponent has a weak, isolated pawn structure. This could help secure the AI’s pawn structure in the mid-game and help it transition into a smoother endgame.

# **Performance Metric**

In the following examples, black (my AI) takes on various chess bots from chess.com of different ratings (playing as white).

## **Chess Bot 400 ELO**

The following is a chess game played between my AI (black) versus the AI on chess.com (white) with an ELO rating of 400. My AI defeated the opponent in 9 moves with an average move time of 11.6 seconds and a standard deviation of 5 seconds.

Background pattern

Description automatically generated

## **Chess Bot 1000 ELO**

The following is a game played between my AI (black) versus the AI on chess.com (white) with an ELO rating of 1000. For much of the game, white was only 1.0 position evaluation points ahead, implying that the game was more or less equally balanced. Unfortunately, the game ended halfway since white played an en-passant move that could not be replicated in the CLI and I had to end the game.

At some points, black conceded a piece to white due to the horizon effect, which is something that I was not able to account for because of an exceeded recursion depth and the added time complexity to the code.

During this game, black spent an average of 19.73 seconds on a move, with a standard deviation of 10.74. The minimum move time was 4 seconds, whereas the maximum was 43 seconds.

Background pattern

Description automatically generated

## **Chess Bot 250 ELO**

Against a chess bot of 250 ELO, black won comfortably in 35 moves with checkmates.

## **Chess Bot 700 ELO**

Against a chess bot of 700 ELO, black took the lead against white by capturing important pieces early on, however the game turned due to a horizon effect loss of piece. Towards the end, black played extremely efficiently and managed to check white’s king and remove all its cover, finally managing to win with one queen and 2 rooks against white’s queens and doubled up pawns

## 

As we can see from the chart above, the time taken for each move is the highest during the middle of the game due to a high level of pieces development. This causes the search tree to have more branches, and thus a higher execution time. As more pieces get captured, we see that the end game is significantly faster execution due to lower number of branches.

# **References:**

The chess ruleset implementation was designed following some ideas from <https://gist.github.com/rsheldiii/2993225>

Ideas like pawn promotion and the changing of board to a matrix rather than a dict were implemented for convenience, hence a large amount of the code had to be refactored to fit my needs.

The basic piece face-values were obtained from the blog <https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/>

The Piece’s Square position were adapted from the chess.wikia page which is now broken. However, a large number of central values were tweaked and adapted to make sure that certain positions had a preference over others.