A Mini-Project Report on

CHESS ENGINE

by

1. Aayush Singh
2. Deepanshu Sonparote
3. Nitish Talekar
4. Amey Waghmode

under the guidance of

Prof. Dilip. S. Kale

MCT

MANJARA CHARITABLE TRUST

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI

Department of Computer Engineering

University of Mumbai

April - 2019

MANJARA CHARITABLE TRUST

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI

Juhu-Versova Link Road Versova, Andheri(W), Mumbai-53.

Certificate

Department of Computer Engineering

This is to certify that

1. Aayush Singh
2. Deepanshu Sonparote
3. Nitish Talekar
4. Amey Waghmode

Have satisfactory completed this mini-project entitled

“CHESS ENGINE”

Towards the complete fulfillment of the

BACHELOR OF ENGINEERING IN (COMPUTER ENGINEERING)

as laid by University of Mumbai.

Guide	Head of Department
Prof. Dilip. S. Kale	Dr. Satish Y. Ket

Principal

Dr. Sanjay Bokade

Internal Examiner	External Examiner

DECLARATION

We wish to state that the work embodied in this synopsis titled ”CHESS ENGINE” forms our own contribution to the work carried out under the guidance of ”Prof. Dilip. S. Kale” at the Rajiv Gandhi Institute of Technology.
We declare that this written submission represents our ideas in our own words and where others’ ideas or words have been included, we have adequately cited and referenced the original sources. we also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. we understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

(Students Signatures)

1. Aayush Singh	(TE - B - 658)
2. Deepanshu Sonparote	(TE - B - 659)
3. Nitish Talekar	(TE - B - 662)
4. Amey Waghmode	(TE - B - 668)

ACKNOWLEDGEMENT

We wish to express our sincere gratitude to Dr. Sanjay Bokade, Principal and Dr. Satish. Y. Ket , H.O.D of Computer Department of Rajiv Gandhi Institute of Technology for providing us an opportunity to do our project work on "CHESS ENGINE".

This project bears on imprint of many peoples. We sincerely thank our project guide Prof. Dilip. S. Kale for his guidance and encouragement in carrying out this synopsis work.

Finally, we would like to thank our colleagues and friends who helped using completing the Project (Synopsis) work successfully.

1. Aayush Singh	(TE - B - 658)
2. Deepanshu Sonparote	(TE - B - 659)
3. Nitish Talekar	(TE - B - 662)
4. Amey Waghmode	(TE - B - 668)

ABSTRACT

The game of chess has many times been a benchmark for testing the advancement in fields of Artificial Intelligence (AI) and Machine Learning (ML). A landmark for artificial intelligence was achieved in 1997 when Deep Blue defeated the human world champion. Although the technique of brute-forcing through all the possible positions is possible, the use of these methods within chess is not feasible, neither physically nor computationally. This report describes the implementation of a chess move generator with the help of decision tree algorithms. An evaluation function is used to compute a heuristic value of a board position and used to find the possible move in a given board position. To measure the performance of the program, it was put to challenge against some of famous game lines. Further tests against a traditional chess program as well as positional tests have been done. The results and the impact of the different logic parts is presented and discussed.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

EPD	Extended Position Description
FEN	Forsyth–Edwards Notation
UCI	Universal Chess Interface
TPU	Tensor Processing Unit

Chapter 1
Introduction

1.1 Overview

Chess is a two-player board game played on a chessboard, a square-checkerboard with 64 squares arranged in an eight-by-eight grid. Each player begins the game with sixteen pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The object of the game is to checkmate the opponent's king, whereby the king is under immediate attack (in "check") and there is no way to remove or defend it from attack on the next move.

The game's present form emerged in Europe during the second half of the 15th century, an evolution of an older Indian game, Shatranj. Theoreticians have developed extensive chess strategies and tactics since the game's inception. Computers have been used for many years to create chess-playing programs, and their abilities and insights have contributed significantly to modern chess theory. One, Deep Blue, was the first machine to beat a reigning World Chess Champion when it defeated Garry Kasparov in 1997.

Chess engine is a computer program that decides what move to make during the game. It makes some calculations based on the current position to decide the next move. In our project, we use machine learning (specifically decision trees) to identify elements of positional play that can be incorporated in chess engines.We model chess board positions as networks of interacting pieces and predict best possible outcome when the engine evaluates both sides to be of comparable strength. Our findings indicate that we can make such predictions with reasonable accuracy and thus, this technique can be augmented with current chess engines to improve their performance.

The primary goal of the project was to create a chess engine capable of giving an interesting game to the casual chess player. This means the engine must be able to adequately open and close chess games, and have knowledge of at least basic chess strategies and formations. This meant that our engine had to be fast enough to maintain the interest of the player, whilst still giving itself time to compute a good move to play.

1.2 Motivation

Many chess playing programs, or “engines” , are available, some available free over the Internet, others serious commercial ventures. Given the volume of chess engines, the motivation behind this project may not be immediately apparent.

However, the world of computer chess is highly competitive, and consequently, a lot of information and research in the area is kept secret so as to maintain an advantage against competitors. Though there are now a small number of tutorials on the Internet with regards to programming chess engines, these generally do not cover the entirety of the process, meaning that significant portions of the problem are completely undocumented publicly.

Clearly the lack of documentation makes the project a challenging one, allowing us both to practice our software development skills, but also to learn more about computer chess, specifically the mathematics that lie behind it. This understanding also allows us to get a better grasp of how other, similar, games such as Checkers or go are implemented on a computer.

Though there are a few open source chess programs, the vast majority of these are written in the C language for maximum efficiency. This not only makes them harder to read and follow, but also means that these engines are more tricky to make truly modular, and cannot take advantages of the modern features of object orientation.

To our knowledge there are no open source chess engines that are written in an object oriented language, and consequently, the decision to implement our program in Python differentiates us from the vast majority of open source chess engines. Admittedly this decision was taken for other pragmatic reasons such as the ease and speed of implementation, but it does allow us to design our program in a way that differs significantly from the other open source chess engines.

1.3 Organization of Report

Chapter 1 - Introduction

This section provides an overview of chess, engines and tactics involved in chess.
It also mentions the motivation towards developing this project.

Chapter 2 - Literature Review

Various research papers were reviewed during development of this project.
In review section, the papers are summarized to provide brief idea of the topic.

Chapter 3 - Design and Implementation

Design and Implementation techniques are discussed in this section. An overview to all modules and their working is provided in this section.

Chapter 4 - Simulation, Result and Analysis

The model developed is run and tested for a set of test positions and the results are analysed in this section

Conclusion and Future Enhancement

The overall scope of project and learning from this project is mentioned in last section.

Chapter 2
Literature Review

2.1 Technical paper reviews

2.1.1 Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm - Deepmind

The paper review the working and implementation of Deepmind's AlphaZero.
The AlphaZero totally discards the move calculation part involved in chess engines, however, it focuses more on tactics and having a better position in the future. It is said to have used 5,000 first-generation TPUs to generate self-play games and 64 second-generation TPUs to train the neural networks. Instead of an alpha-beta search with domain-specific enhancements, *AlphaZero* uses a general-purpose Monte-Carlo tree search (MCTS) algorithm. Each search consists of a series of simulated games of self-play that traverse a tree from root to leaf. As it uses reinforcement learning instead of alpha-beta search it drastically affects the time required to search a move. During a game being played, the board is filtered into various appropriate arrays and concatenated together which forms the input to the neural network, hence producing the output move.

2.1.2 How Stockfish Works: An Evaluation of the Databases Behind the Top Open-Source Chess Engine

It is important to distinguish between computer chess research and research using chess as a test bed. The latter expands the user community past chess enthusiasts to AI developers. In AI, the real-time evaluation and categorization of a dynamic environment are crucial to maintaining functionality while performing a task. This environment can be categorized using preset labels and rules or adapted to based purely on experience. Stockfish's evaluation of chess positions, based not only on material properties (i.e. "I have more pieces than you") but on the relationships and structures that exist between said material (i.e. "My piece is protected by two other pieces, and is threatening your unprotected piece."), is useful in the domain of autonomous systems which must quickly evaluate the stability of groups of objects interacting within set rules.

While the human method of analyzing alternatives seems to involve selecting a few promising lines of play and exploring them, computers are necessarily exhaustive rather than selective, so refinement techniques have been (and continue to be) developed.

2.1.3 Improvising The Performance Of A Chess Engine By Fine Tuning The Evaluation Function

An evaluation function needed to be developed to evaluate the present board position.
This function takes into consideration the current piece position for both sides and the threats and risks in the position.
The usage of opening book in the initial phase of games reduces the calculation effort. A bin file of theoretical chess openings is already predefined to provide a better position out of openings.

2.1.4 The Kopec-Bratko Experiment

Kopec and Bratko published a systematic test for assessing the strength of chess-playing programs. Despite its age, it can still be fairly challenging, even for modern chess engines. Of course, 24 test positions are not going to be enough for a definite result, especially given that most positions are of a specific type: fairly closed and the solution often involves some kind of sacrifice.

For each of the 24 positions, a unique best move is known and to cross-check, we have evaluated the positions with Stockfish. The engine is given to solve each position. 1 point is awarded for the correct best move. The positions are given as EPD (FEN Format).

2.2 Methodology

Using decision tree algorithms to select best possible move amongst available moves.

Using Minimax algorithm to find move based on score .Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally.

Alpha-Beta pruning is also used to help speed up the decision making process.Alpha-Beta pruning is not a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree.

Quiescence Search algorithm helps in limiting search parameters.The purpose of this search is to only evaluate "quiet" positions, or positions where there are no winning tactical moves to be made. This search is needed to avoid the horizon effect.

2.3 Problem Formulation

The main focus of the project is to create an application that can tactically and strategically think for itself and play a complex game like chess in a manner that shows its thinking capabilities. Thus the problem question in our project is
“Whether a machine can be created to replicate human thinking in complex strategic games like chess?”

2.4 Problem Statement

To create an application which is able to recreate Human-level Intelligence in the game of chess using suitable algorithms and techniques.
To use various algorithms to improve the response of the application in order to play an efficient game of chess.

2.5 Proposed System

Our project is based on a specific point system. A chess board at any given time in a game will produce a certain score based on its components and strategic placement. Our proposed system calculates the score of every future possible move from the board. This score is generated by a evaluation function that considers numerous factors on the board to analyse the situation and give a valid score. The application searches for the move that gives the board maximum points in order to win. The search is done via decision tree algorithm that will eventually generate the best move among all possible moves. Hence by subjecting the application to search the best move for itself, the application will imitate human intelligence.

Chapter 3
Design and Implementation

3.1 Design

The Board

Chess is a two-player strategy board game played on a chessboard, a checkered game board with 64 squares arranged in a 8 x 8 grid.

In a chess board, files are labelled by the letters *a* to *h* from left to right from the white player's point of view, and the ranks by the numbers *1* to *8*, with 1 being closest to the white player, thus providing a standard notation called algebraic chess notation. Each square on the board has a name from a1 to h8.

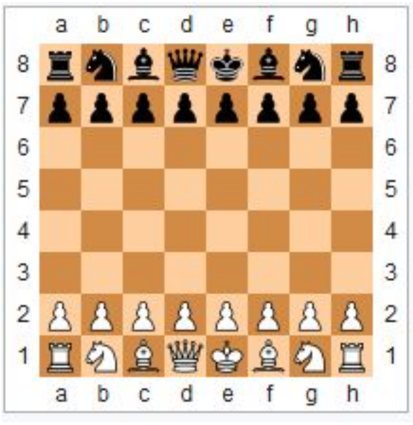


Fig 3.1. Board

Pieces

The game pieces are divided into white and black sets, and the players are referred to as White and Black, respectively. Each player begins the game with 16 pieces of the specified color, consisting of one king, one queen, two rooks, two bishops, two knights, and eight pawns. The pieces are set out as shown in the diagram and photo, with each queen on a square of its own color (the white queen on a light square; the black queen on a dark square).

- The king moves one square in any direction. The king also has a special move called *castling* that involves also moving a rook.
- A rook can move any number of squares along a rank or file, but cannot leap over other pieces. Along with the king, a rook is involved during the king's castling move.
- A bishop can move any number of squares diagonally, but cannot leap over other pieces.
- The queen combines the power of a rook and bishop and can move any number of squares along a rank, file, or diagonal, but cannot leap over other pieces.
- A knight moves to any of the closest squares that are not on the same rank, file, or diagonal. (Thus the move forms an "L"-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically.) The knight is the only piece that can leap over other pieces.
- A pawn can move forward to the unoccupied square immediately in front of it on the same file, or on its first move it can advance two squares along the same file, provided both squares are unoccupied (black dots in the diagram); or the pawn can capture an opponent's piece on a square diagonally in front

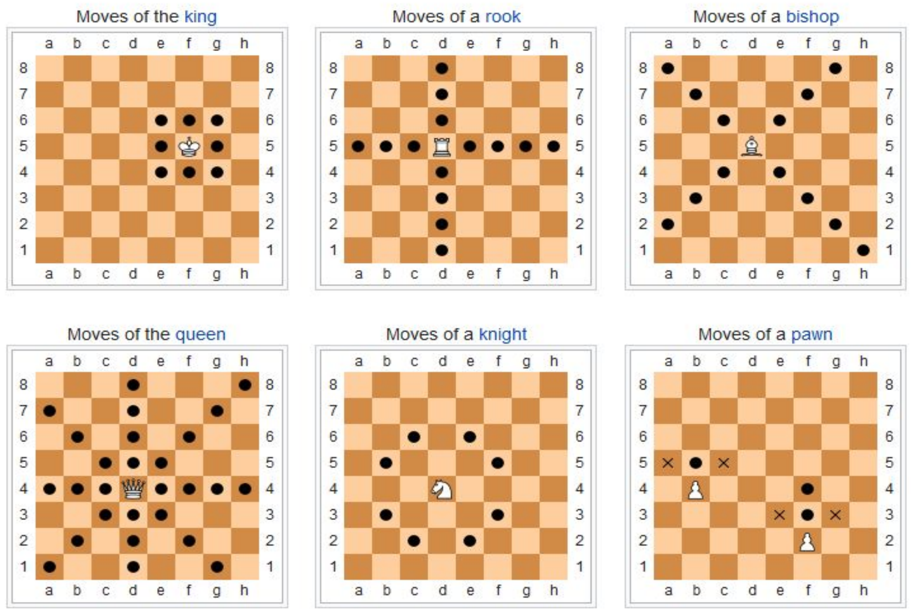


Fig 3.2. Movement of Pieces

3.1.1 Chess - SVG

The board being displayed during gameplay is obtained using ‘chess.svg’ module developed under python-chess. SVG stands for Scalable Vector Graphics. It is an XML based vector image format used for 2D graphics display. This is a static display of board and hence cannot be interacted with.

The embedding of SVG Images in output is done using IPython Notebooks. We use the display() function defined in the module to display the FEN-obtained bitboards to an image of defined size.

3.2. Implementation

3.2.1. Python-Chess

Python-chess is a pure Python chess library with move generation, move validation and support for common formats.

3.2.2. Minimax Algorithm

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally.

In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

For Example:-

Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move, i.e. you are at the root and your opponent at next level.

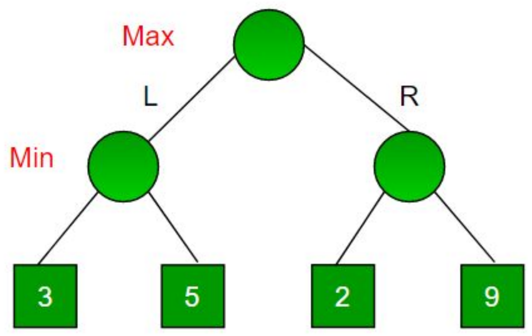


Fig 3.3. Minimax(E1)

Since this is a backtracking based algorithm, it tries all possible moves, then backtracks and makes a decision.

- Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3

- Maximizer goes RIGHT: It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.

Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

Now the game tree looks like below :

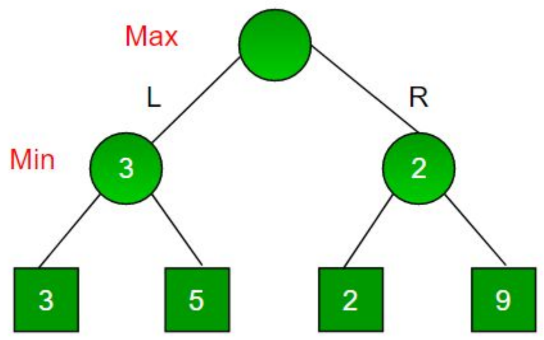


Fig 3.4. Minimax(E2)

The above tree shows two possible scores when maximizer makes left and right moves. The maximizer then chooses the maximum value.

3.2.3. Alpha-Beta Pruning

The process of traversing the entire decision tree is too long and it has a time complexity of O(V+E) i.e. Depth First Search

Alpha-Beta pruning is not a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available.

It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.
Alpha is the best value that the **maximizer** currently can guarantee at that level or above.
Beta is the best value that the **minimizer** currently can guarantee at that level or above.

For Example:-

- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the maximizer must choose max of **B** and **C**, so **A** calls **B** first.

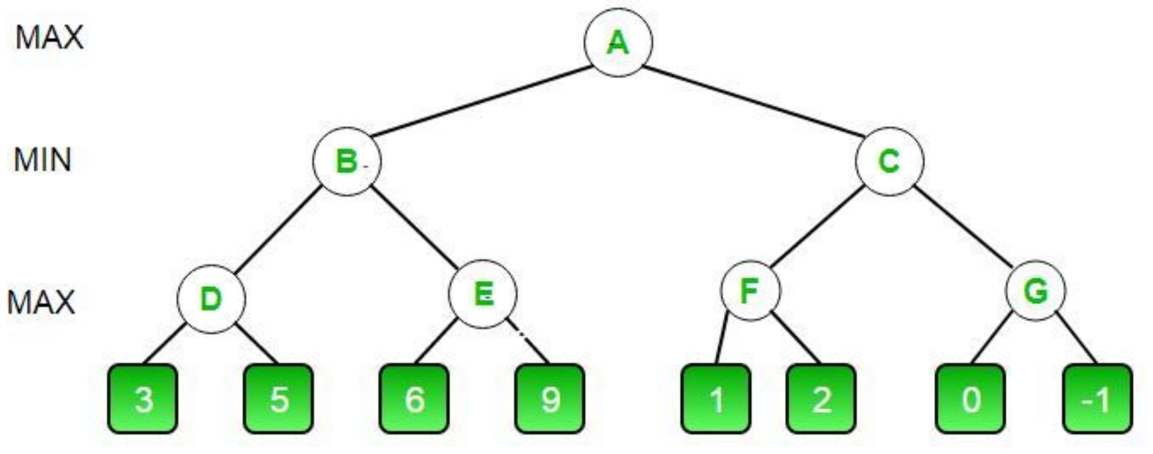


Fig 3.5. Alpha-Beta(E1)

- At **B,** it the minimizer must choose min of **D** and **E** and hence calls **D** first.
- At **D**, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at **D** is max(-INF, 3) which is 3.
- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.
- **D** now looks at its right child which returns a value of 5.At **D**, alpha = max(3, 5) which is 5. Now the value of node **D** is 5
- **D** returns a value of 5 to **B**. At **B**, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. **B** now calls **E** to see if he can get a lower value than 5.
- At **E** the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at **B** and that is what **B** passed down to **E**
- Now **E** looks at its left child which is 6. At **E**, alpha = max(-INF, 6) which is 6. Here the condition becomes true. beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and **E** returns 6 to **B**
- Note how it did not matter what the value of **E**'s right child is. It could have been +INF or -INF, it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of **B**. This way we didn't have to look at that 9 and hence saved computation time.

- **E** returns a value of 6 to **B**. At **B**, beta = min(5, 6) which is 5.The value of node **B** is also 5

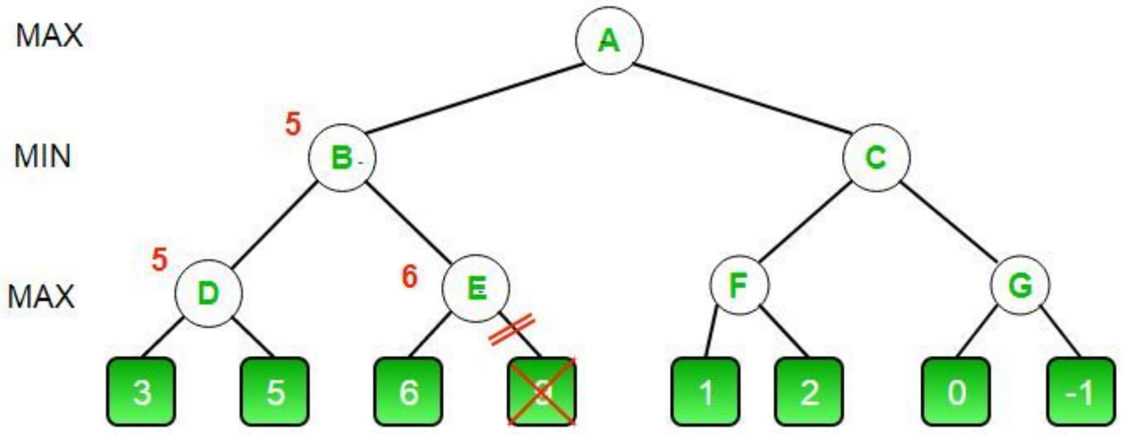


Fig 3.6. Alpha-Beta(E2)

- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.
- At **C**, alpha = 5 and beta = +INF. **C** calls **F**
- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
- **F** returns a value of 2 to **C**. At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes true as beta = 2 and alpha = 5. So it breaks and it does not even have to compute the entire sub-tree of **G**.
- The intuition behind this break off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.
- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5.

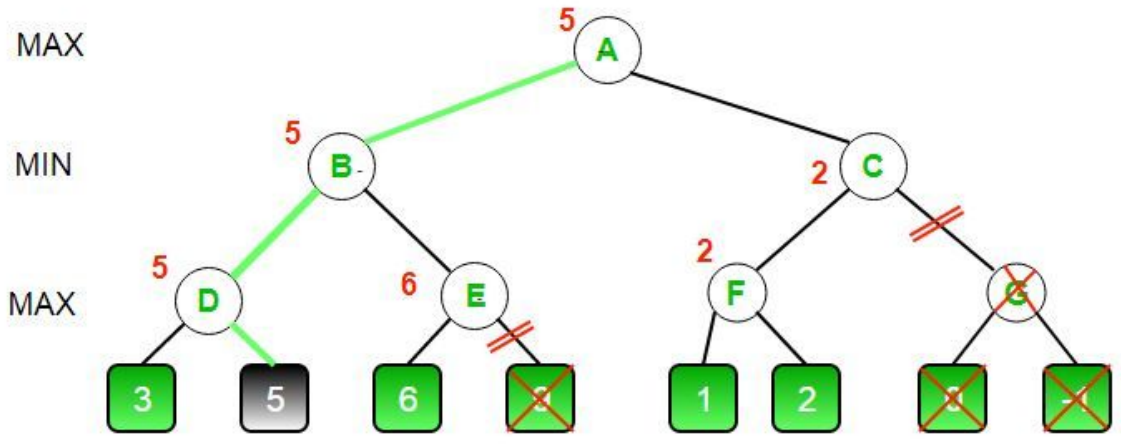


Fig 3.7. Alpha-Beta(E2)

3.2.4. Quiescence Search

Most chess programs, at the end of the main search perform a more limited quiescence search, containing fewer moves. The purpose of this search is to only evaluate "quiet" positions, or positions where there are no winning tactical moves to be made. This search is needed to avoid the horizon effect. Simply stopping your search when you reach the desired depth and then evaluate, is very dangerous. Consider the situation where the last move you consider is QxP. If you stop there and evaluate, you might think that you have won a pawn. But what if you were to search one move deeper and find that the next move is PxQ? You didn't win a pawn, you actually lost a queen. Hence the need to make sure that you are evaluating only quiescent (quiet) positions. This is continued until a quiet position is encountered and no further capture can be done.

3.2.5. Evaluation Function

Chess, being a mind-powered game requires intense attention of players. The professional players can recognize if the position is favourable to them or to their opponents. This is used to calculate moves which are beneficial to oneself rather to the opponent. Therefore it quite clear that the game of chess being played can be in favour of any one of the players.

But how do we determine who exactly is the one with the initiative?

In the project developed, we used 2 important parameters to evaluate a score for position on board: Piece Count & Piece Position.

Each piece has an associated value. These values are thoroughly calculated with the help of Top-Level Grandmasters.

Piece	Value (Centi-pawns)
Pawn	100
Knight	320
Bishop	330
Rook	500
Queen	900
King	INFINITY

Table 3.1. Piece Values

The position evaluation is done with the help of a heuristic function which evaluates the board position and returns a value associated with the position. The returned value is positive when White has the advantage and negative when Black has the advantage.

The Piece count advantage can be determined by the difference in the number of respective pieces on each side. This result affects the total evaluation on a higher scale.

The Piece Position is an important part of evaluation as the square occupied by the pieces indirectly define the activity of the pieces.

A bishop on the a1-h8 diagonal is more valuable than one on d1-h5 diagonal. A bishop on a1-h8 diagonal is directly attacking black's king-side, hence being more active.

Similarly, all the pieces have an optimal square which when occupied leads to a better position and an advantageous game for the player.

- The King is supposed to be either behind the pawns on the first rank, or on the 7th rank in opposition’s camp.
- The queen and the knights have an optimal position in the center of the board.
- The Rooks must occupy open or semi-open files.

The advantage is usually calculated in centipawns which is later reduced to pawn-scale advantage.
For comparison, a Grandmaster with an advantage(position evaluation) of 2 will win the game 90/100 times.

3.2.6. Functioning

The engine developed uses all the fore-mentioned algorithms and techniques to function properly and calculate moves.

At present, we allow 4 ways of gameplay :-
1. Play As White
2. Play As Black
3. Play From Position
4. Find Best Move

The move calculation method in all gameplay is same.

The flow of move calculation can be described as follow :-

```
1. Initialize Depth, Bestscore, BestMove
2. Board initialized or Current Board position in play
3. If depth is not 0
    a. Get all legal moves possible in current position
    b. For each Move in legal moves :
        i. Play Move to get new position
        ii. If board_evaluation better than Bestscore
            1. Set Bestscore = board_evaluation
            2. Set BestMove = Move
        iii. Find best move in current position for depth=depth-1
        iv. Undo Move
4. Best Move Found.
```

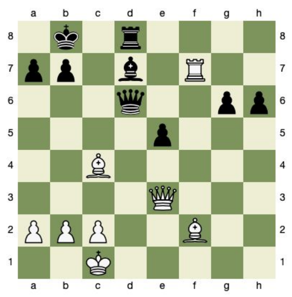
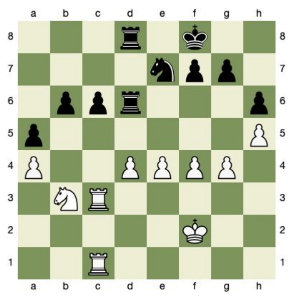
Chapter 4
Results & Analysis

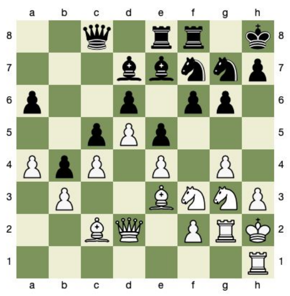
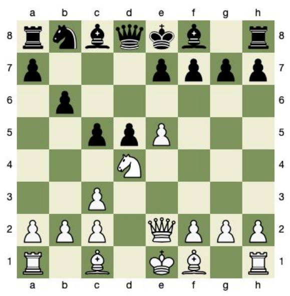
4.1 Results

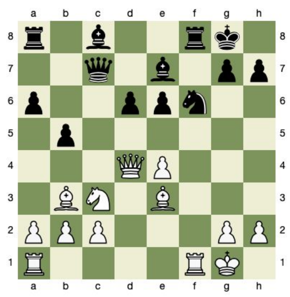
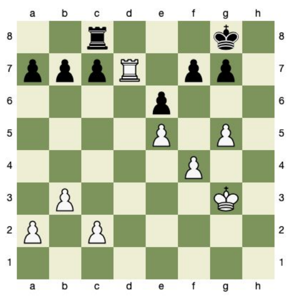
4.1.1 Test Cases :-

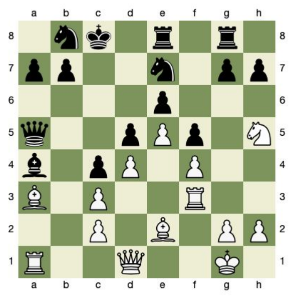
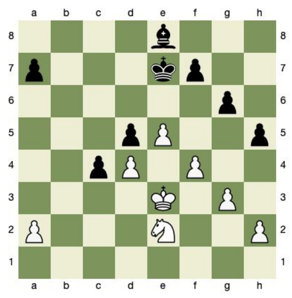
The comparisons made below are on the basis of our CHESS ENGINE and STOCKFISH which is one of the best chess engines in the world.
In the following test cases, evaluation is done as following

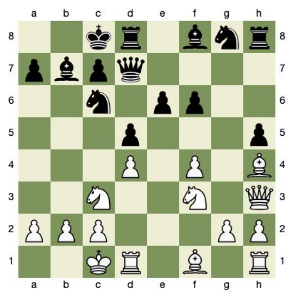
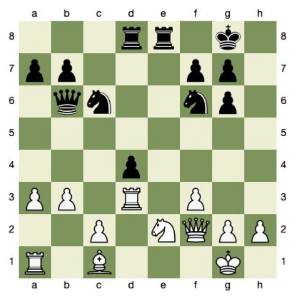
0 - 2	GREAT
2 - 6	GOOD
6 < eval	BLUNDER

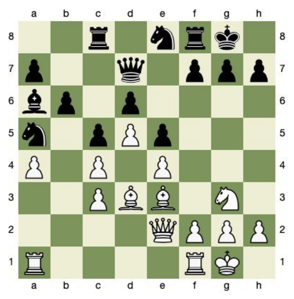
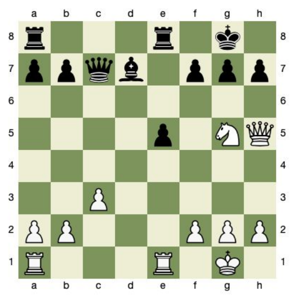
Test Position 1 Black to Move  Best Move :- Qd1+ Predicted Move :- Bc6 Stockfish Eval of Predicted move :- BLUNDER	Test Position 2 White to Move  Best Move :- d5 Predicted Move :- e5 Stockfish Eval of Predicted move :- GREAT
As the search takes place till depth 4, our engine is unable to predict the threat of Qd1+ which leads to checkmate in next 3 moves.	Although our engine does not predict the BEST move but it predicts a move which keeps both sides evenly balanced.

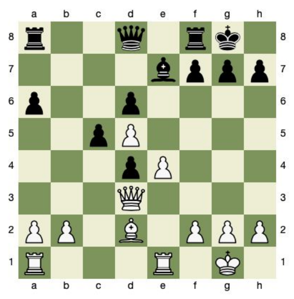
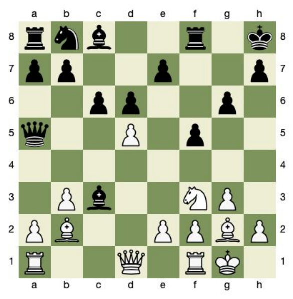
Test Position 3 Black to Move  Best Move :- f5 Predicted Move :- f5 Stockfish Eval of Predicted move :- GREAT	Test Position 4 White to Move  Best Move :- e6 Predicted Move :- Nf3 Stockfish Eval of Predicted move :- GOOD
Here as all the pieces are present on the board and all functions run accordingly so BEST move f5 is predicted.	Here as the move predicted saves a piece but unstables white's position in longer run of the game which is missed due to depth 4 search.

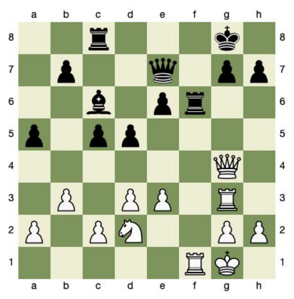
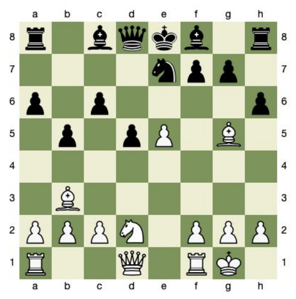
Test Position 5 White to Move  Best Move :- a4 Predicted Move :- Rfe1 Stockfish Eval of Predicted move :- GREAT	Test Position 6 White to Move  Best Move :- g6 Predicted Move :- Kg2 Stockfish Eval of Predicted move :- GREAT
Here the best move was a4 or Nd4 but due to the threat of losing the center ,our engine predicts such a move which is great but not best.	As the engine does not consider detailed depths, white predicts a rather good move than the best one and loses its advantage .

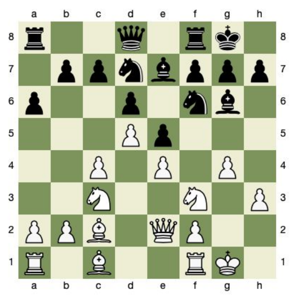
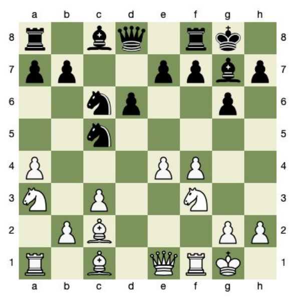
Test Position 7 White to Move  Best Move :- Nf6 Predicted Move :- Bxe7 Stockfish Eval of Predicted move :- GREAT	Test Position 8 White to Move  Best Move :- f5 Predicted Move :- Kf2 Stockfish Eval of Predicted move :- GREAT
As the engine does not consider detailed depths, white predicts a rather good move than the best one and loses its advantage .	Here f5 gains a good advantage for white but the engine decides to play a safer move and continues for a draw.

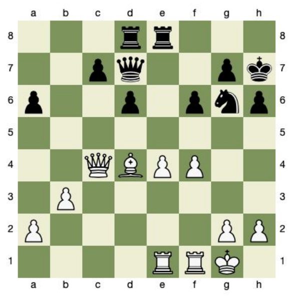
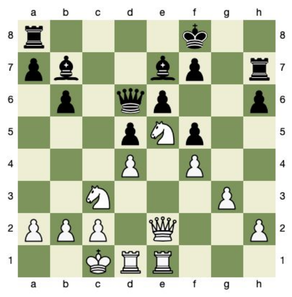
Test Position 9 White to Move  Best Move :- f5 Predicted Move :- Bd3 Stockfish Eval of Predicted move :- GREAT	Test Position 10 Black to Move  Best Move :- Ne5 Predicted Move :- Qc5 Stockfish Eval of Predicted move :- GOOD
Here our engine equally shows a good response compared to stockish. Engine:bd3 a6 f5 exf5 Stockfish:f5 exf5 bd3 Kb8 else loses queen.	Here it's about playing more aggressively as stockfish works at a greater depth it learns future possible moves and plays a more aggressive move.

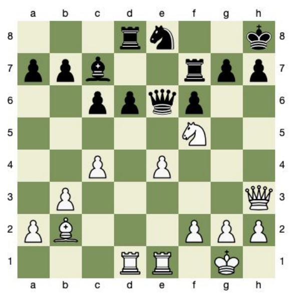
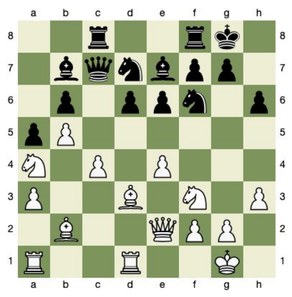
Test Position 11 White to Move  Best Move :- f4 Predicted Move :- Nf5 Stockfish Eval of Predicted move :- GREAT	Test Position 12 Black to Move  Best Move :- Bf5 Predicted Move :- Bf5 Stockfish Eval of Predicted move :- GREAT
Here our engine plays a move creating mating threat on king's side but stockfish tries to open up the game by trying to open the center.	Here the search till depth 4 works because the only move that saves the mate is Bf5 which is correctly predicted.

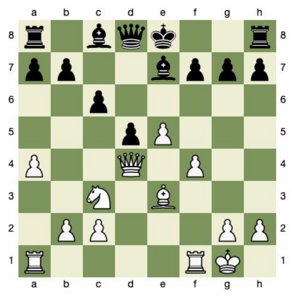
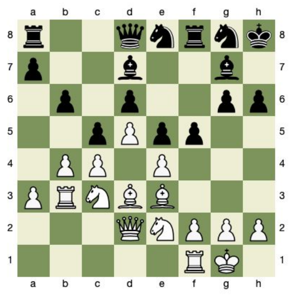
Test Position 13 White to Move  Best Move :- b4 Predicted Move :- Rab1 Stockfish Eval of Predicted move :- GREAT	Test Position 14 White to Move  Best Move :- Qd2 Qe1 Predicted Move :- Qe1 Stockfish Eval of Predicted move :- GREAT
Stockfish plays a move which either makes black's position unstable or loses a pawn. Our engine comparatively plays a decent move which maintains balance of the game.	Here search for the best move is quite easy as Qe1 wins a piece without any compensation to black.

Test Position 15 White to Move  Best Move :- Qxg7+ Predicted Move :- Rxf6 Stockfish Eval of Predicted move :- GREAT	Test Position 16 White to Move  Best Move :- Ne4 Predicted Move :- Be3 Stockfish Eval of Predicted move :- GREAT
Here our engine finds a move which is quite good but fails to gain any advantage, stockfish plays an attacking move. Qxg7+ Qxg7 Rxf6 Qxg3 hxg3 Advantage White.	As stockfish tries to attack by Ne4(next move is critical for black) our engine plays a simple Be3 move which saves our bishop.

Test Position 17 Black to Move  Best Move :- h5 Predicted Move :- Nc5 Stockfish Eval of Predicted move :- GREAT	Test Position 18 Black to Move  Best Move :- Nb3 Predicted Move :- e5 Stockfish Eval of Predicted move :- BLUNDER
h5 is the best move which saves the bishop from getting trapped on g6 (in longer run of the game black would eventually gain advantage).Nc5 is decent as it develops the knight to gain some important squares.	Here Nb3 is a drawish line which is indeed the best move but our engine miscalculates and plays e5 which is poor and black should have a tough game ahead.

Test Position 19 Black to Move  Best Move :- Rxe4 Predicted Move :- d5 Stockfish Eval of Predicted move :- GREAT	Test Position 20 White to Move  Best Move :- g4 Predicted Move :- Nb5 Stockfish Eval of Predicted move :- GREAT
Here both the engines identify the position as white is winning but stockfish still plays for a win in minimum moves with Rxe4 Rxe4 d5.	Here both engines play good moves but the g4 being strongest because it breaks the strong pawn structure of black in the center.(missed due to depth 4 search)

Test Position 21 White to Move  Best Move :- Nh6 Predicted Move :- Nd4 Stockfish Eval of Predicted move :- GREAT	Test Position 22 Black to Move  Best Move :- Bxe4 Predicted Move :- Ne5 Stockfish Eval of Predicted move :- BLUNDER
Here depth 4 calculates well till some point and sees couple of good sequences Nd4 Qxh3 gxh3 (doesn't gain any advantage) whereas Nh6 Qxh3 Nxf7+ Kg8 gxh3 gains advantage.	Bxe4 gains a slight advantage whereas Ne5 loses center for black (missed due to depth 4 search).

Test Position 23 Black to Move  Best Move :- f6 Predicted Move :- Bf5 Stockfish Eval of Predicted move :- GREAT	Test Position 24 White to Move  Best Move :- f4 Predicted Move :- f4 Stockfish Eval of Predicted move :- GREAT
The motto is to stop center pawn march, which is identified by both the engines and according to their depths produces the moves (stockfish being better.)	Here f4 is a natural move as it creates multiple chances for white to evade black's king side.This prediction is done accurately.

4.2 Analysis

The engine was able to calculate a move which produced a deviation in evaluation between 0 - 2 for 19 positions out of 24, which clearly signifies that the moves are equally competitive to an average human chess player.

The moves for only 2 out of 24 positions were found to be causing evaluation variance between 2 to 6. This change of evaluation can drastically change the direction of the game towards the opponent as he may get a big advantage for the inaccurate move made.

The engine also predicted 3 blunder moves. But this was highly caused by the limited depth and the heuristic function not defined to evaluate positional threats. The positions in these blunder cases were mate-in-5 and mate-in-6 threats.

However, when compared to actual stockfish predicted move in test positions, we have only achieved an accuracy of 4/24 (17%).

Although when compared to an amateur human player, we have achieved an accuracy of 19/24 (79%), which implies the engine to be satisfactory but competitive to other engines available.

4.2.1 Comparison

The comparison between the engine and Stockfish is not justifiable as the engine with limited depth and a heuristic function always fails to consider many of the possible threats, pins, skewers and tactics.

The reliance on evaluation function to find better move imitates the amateur player calculation, however, fails when tried to predict a move in a complex position with multiple threats.

These problems within the engine can be overcome by adding more parameters to check when evaluating and possibly trying to evaluate dynamically rather a static method being used currently.

Conclusions and Future Enhancement

On reflection, one of the first design decisions taken upon embarking on this project, to use the Python language for implementation, was a good one. Though coding in a lower level language such as C/C++ could have resulted in greater performance, the choice of Python kept our code relatively clean, maintainable and uncomplex. This allowed us to develop our implementation at a great pace and co-ordinate easily between team members.
Our first aim was to ensure that the engine gave an interesting game against a casual chess player. Tests against human players showed that most players claimed they found the game an enjoyable experience. The personal experience of the developers also reflects this as the engine played at a level that was good enough to provide a serious, though not insurmountable challenge.

The engine is capable of opening chess games in a strategic manner, without foolishly sacrificing pieces early in the game. This is due to evaluation functions added, which calculates the positions. However, the engine fares worse in the endgame scenarios where it has the advantage.
Also due to CPU speed and performance constraints, the Engine works slower than expected. If more time was allotted to this project,resources could have been allocated for speeding up both the board representation and the analyser thus allowing searches to greater depths. An alternative, or complementary solution, would be to implement a simpler endgame evaluator.

The heuristics contained within the evaluator encourage the engine to adopt strategic formations of pieces. There are further heuristics that could be added to increase the engine's ability, such as giving knights and rooks a bonus if the engine brings them into play.
However, despite these flaws, the engine can be deemed an overall success. It implements the vast majority of chess rules and plays a game that is considered challenging and enjoyable by most to beginner and intermediate players who have faced it. Though the engine is certainly no match for other computer engines given the development time invested, its results are impressive.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”, Science, vol. 362, no. 6419, pp. 1140–1144, 2018.

[2] G. V. S. Prasad, P. Ch. N. V. Sairam, M. Bhavana Sri, M. Siva Krishna, “Improvising The Performance Of A Chess Engine By Fine Tuning The Evaluation Function”, Engineering Technology, Vol. 9., pp. 542-555, Sp– 18 / 2017

[3] D. Knuth and R. Moore, “An analysis of alpha-beta pruning”, Artificial Intelligence, vol. 6, no. 4, pp. 293-326, 1975. Available: 10.1016/0004-3702(75)90019-3.

[4] D. Kopec and I. Bratko, “The Bratko-Kopec Experiment: A Comparison Of Human And Computer Performance In Chess,” Advances in Computer Chess, pp. 57–72, 1982

[5] P. Dodgers, "AlphaZero Crushes Stockfish In New 1,000-Game Match", Chess.com, 2018. [Online]. Available: https://www.chess.com/news/view/updated-alphazero-crushes-stockfish-in-new-1-000-game-match.

[6] P. Dodgers, "Machine-Learning Lc0 Joins 'Big 3' Engines Atop Computer Chess Championship At Half", Chess.com, 2019. [Online]. Available: https://www.chess.com/news/view/machine-learning-lc0-joins-big-3-engines-atop-computer-chess-championship-half. [Accessed: 06- Apr- 2019].

[7] V. Sangam, “Artificial Intelligence - MiniMax Algorithm - Theory of Programming”, Theoryofprogramming.com, 2017. [Online]. Available: http://theoryofprogramming.com/2017/12/12/minimax-algorithm/.

[8] G. Bartel, “Playing Strategy Games With The Minimax Algorithm”, freeCodeCamp.org, 2017. [Online]. Available: https://medium.freecodecamp.org/playing-strategy-games-with-minimax-4ecb83b39b4b.

[9] L. Hartikka, “A step-by-step guide to building a simple chess AI,” freeCodeCamp.org, 30-Mar-2017. [Online]. Available: https://medium.freecodecamp.org/simple-chess-ai-step-by-step-1d55a9266977.