Chess Engine

Sriramprabhu Sankaraguru[1] and Harshdeep Singh[2]

I. INTRODUCTION

In this project, we are implementing a chess engine that builds a game tree with the set of possible moves and finds the optimal move from the game tree efficiently.

A. Methods

Finding the optimal move from a game tree is a search problem. Especially, in the game of chess, white player is trying to maximize the gain and the black player is trying the opposite. So, the standard algorithm to solve this kind of problem is Minimax algorithm. As we have seen in the lectures, the game tree for chess grows very large. And its impossible to keep everything in memory. For ex. B $\approx$ 35 and M $\approx$ 100. Applying Minimax (time complexity $O(B^M)$) on this space to get the exact solution is impossible. So, we are limiting the depth to 5. Also, we use alpha-beta pruning to avoid searching in suboptimal paths earlier. To evaluate the position of the board at any point of time, we use Piece-value table that associates a value for each piece in the board, and a Piece-square table that gives a small reward for a piece that controls most squares based on its position in the board.

B. Evaluation

To compare/evaluate the performance of our AI, we used chess.com website that uses stockfish chess engine (one of the top most chess AIs). We evaluate our engine with the ELO points. Also, we make sure that our engine doesnt make any obvious mistakes/blunders during the game.

II. USER INTERFACE

Initially, we started implementing the chess engine by using javascript and HTML, but we noticed that as the level (depth level for search) was increased the performance was affected; it was taking more time to perform the computations.So we decided to implement it as a web application and move the search and evaluation logic to the server side which improved the performance. The chess engine web application, uses node.js and express.js on server side and HTML for the visualization of the chess engine.

The chess engine had several different components such as Chess board UI, Move generator,Board evaluation strategy and Search Algorithms. Each of the components is explained below:

[1] MS in CS at NEU. email: Sankaraguru.s@husky.neu.edu
[2] MS in CS at NEU. email: Singh.h@husky.neu.edu

A. UI and Move Generation

Chess Engine UI: For chess engine UI, we have used **chessboard.js** library. The chessboard.js library provides us with functions/properties that allows a player to play using the UI. Below mentioned is the list of functions/properties provided by chessboard.js that are used in our chess engine:

- **draggable**: is a boolean property, which if set to true, makes pieces on the board draggable.Default value of this property is false.
- **position**: is a string property and can be provided with either 'start' or FEN(Forsyth-Edwards Notation). Based on the input provided,it sets the initial position of the board.
- **onDragStart**: is a function that is fired when a piece is picked up from the board. The first argument to the function is the source of the piece, the second argument is the piece, the third argument is the current position on the board, and the fourth argument is the current orientation.
- **onDrop**: is a function that is fired when piece is dropped at a location on the board.Based on the square where the piece is dropped, the function returns a value. If the piece is dropped at a square that is not a valid/legal move/square for that piece then it function returns 'snap-back' and the piece is returned back to its source square. Similarly, when a piece captures an opponents piece, the onDrop function returns 'trash' and the opponent's piece is removed from the board.
- **onMouseoutSquare**: is a function that is fired when the mouse leaves a square. The purpose of this function is to highlight the legal moves for a selected piece. In our chess engine, legal moves are shown as grey squares on the chess board.
- **onMouseoverSquare**: is a function that is fired when the mouse enters a square. The purpose of this function is to highlight the legal moves for a piece in a given square. In our chess engine, legal moves are shown as grey squares on the chess board.
- **onSnapEnd**: is a function that is fired when the piece snap animation is complete.This function updates the position of the board after piece castling, En passant and pawn promotion.

Move Generation: For move generation in our chess engine, we have used **chess.js** library.The chess.js is a standard JavaScript Chess library that is used for chess move generation/validation, piece placement/movement, and check/checkmate detection, basically it performs everything but AI.The chess.js library is used widely by various gaming

Fig. 1. The Chess Board UI generated using chessboard.js

websites like chess.com, lichess, The Internet Chess Club etc. Chess.js API provides us following functions:

- **Chess**: is constructor which takes an optional FEN-string as input. The FEN string passed as an input specifies the board configuration. If no FEN string input is provided, the game is created with board defaulting to its stating position.
- **in_checkmate**: is a function that returns true if the side to move has been checkmated otherwise it returns false.
- **in_draw**: is a function that return true if the game is drawn otherwise returns false.
- **fen**: is a function that returns the current position of the board as an FEN string notation.
- **moves**: is a function that returns a list of valid/legal moves from the current position in the game.The moves function takes an optional parameter which controls the single square move generation. If an invalid square is passed as input,it returns an empty list of legal moves.
- **move**: is a function that takes a move object from list of moves, as an input and makes the move.
- **game_over**: is a function that returns true if the game has ended.Otherwise,, returns false. The game can end via checkmate, stalemate,draw,threefold. repetition, or insufficient material.

III. Board Evaluation

Board Evaluation is the technique to calculate utility or value/score of the players based on the position of the board.It takes into account the number and types of pieces on the board following a move, and evaluates/calculates score. For chess engine we are trying to maximize the score for white pieces and minimize score for the black pieces.

.

Pawn	Knight/Bishop	Rook	Queen	King
10 pts	30 pts	50 pts	90 pts	900 pts

Table 1. Piece-Value table for the chess engine.

A. Basic evaluation

The evaluation function used in our chess engine is using Piece-Value table approach. In Piece-Value table approach, each piece type is assigned some points and the score of a player is determined by summing up the points of the pieces present on the board for that player. The table above shows points for each piece type in our chess engine.

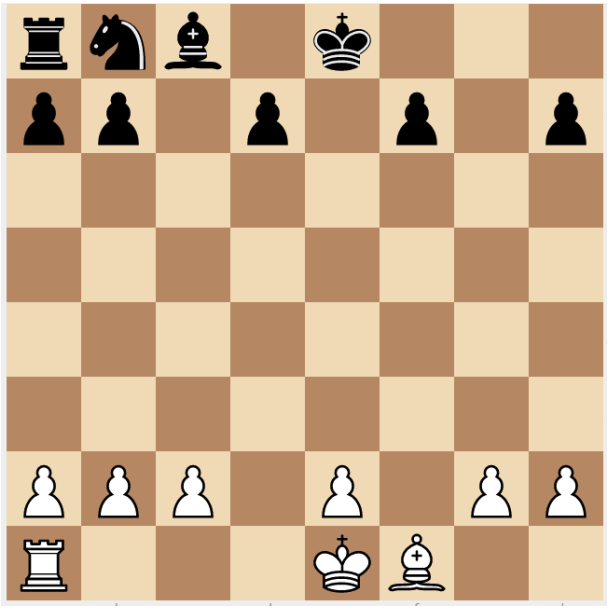

Fig. 2. The score calculated for white player based on piece value table mentioned above is: 50 + 900 + 30 + 60 = 1040 pts.

B. Improved Evaluation

The board evaluation approach described above is a naive approach that does not take into account the board control i.e. which player will have more control over the board after making a move. To bring board control into account while calculating scores or doing board evaluation, we decided to incorporate Piece-Square table approach.

In Piece-Square table approach, instead of just adding up the points of pieces present on the board, we multiply the weights assigned to each square on the board (based on the piece type present in it) with the points of that piece type and then sum over all the pieces present on the board for that player. For each piece type, every square on the board is assigned some weight. These weights are standard and given as below:

In fig. 3 and fig.4, we can see two scenarios. In scenario Fig.3, we can see that the white knight is position at square d4. All the possible moves for knight from square d4 are marked with gray dots. In scenario Fig.4, we see the white knight positioned at a3 and all the legal moves for knight from square a3 are marked with gray dots. Now if we

Pawn Table:

```
[0, 0, 0, 0, 0, 0, 0, 0],
[50, 50, 50, 50, 50, 50, 50, 50],
[10, 10, 20, 30, 30, 20, 10, 10],
[5, 5, 10, 27, 27, 10, 5, 5],
[0, 0, 0, 25, 25, 0, 0, 0],
[5, -5,-10, 0, 0,-10, -5, 5],
[5, 10, 10,-25,-25, 10, 10, 5],
[0, 0, 0, 0, 0, 0, 0, 0]
```

Knight Table:

```
[-50,-40,-30,-30,-30,-30,-40,-50],
[-40,-20, 0, 0, 0, 0,-20,-40],
[-30, 0, 10, 15, 15, 10, 0,-30],
[-30, 5, 15, 20, 20, 15, 5,-30],
[-30, 0, 15, 20, 20, 15, 0,-30],
[-30, 5, 10, 15, 15, 10, 5,-30],
[-40,-20, 0, 5, 5, 0,-20,-40],
[-50,-40,-20,-30,-30,-20,-40,-50]
```

Bishop Table:

```
[-20,-10,-10,-10,-10,-10,-10,-20],
[-10, 0, 0, 0, 0, 0, 0,-10],
[-10, 0, 5, 10, 10, 5, 0,-10],
[-10, 5, 5, 10, 10, 5, 5,-10],
[-10, 0, 10, 10, 10, 10, 0,-10],
[-10, 10, 10, 10, 10, 10, 10,-10],
[-10, 5, 0, 0, 0, 0, 5,-10],
[-20,-10,-40,-10,-10,-40,-10,-20]
```

Queen Table:

```
[-20, -10, -10, -5, -5, -10, -10, -20],
[-10, 0, 0, 0, 0, 0, 0, -10],
[-10, 0, 5, 5, 5, 5, 0, -10],
[ -5, 0, 5, 5, 5, 5, 0, -5],
[ 0, 0, 5, 5, 5, 5, 0, -5],
[-10, 5, 5, 5, 5, 5, 0, -10],
[-10, 0, 5, 0, 0, 0, 0, -10],
[-20, -10, -10, -5, -5, -10, -10, -20]
```

King Table:

```
[-30, -40, -40, -50, -50, -40, -40, -30],
[-30, -40, -40, -50, -50, -40, -40, -30],
[-30, -40, -40, -50, -50, -40, -40, -30],
[-30, -40, -40, -50, -50, -40, -40, -30],
[-20, -30, -30, -40, -40, -30, -30, -20],
[-10, -20, -20, -20, -20, -20, -20, -10],
[ 20, 20, 0, 0, 0, 0, 20, 20],
[ 20, 30, 10, 0, 0, 10, 30, 20]
```

The above tables are for white player. For Black player, we just flip the tables.

calculate score for white player using the piece square tables provided below, we get that for scenario in Fig.3, the score of white player is 100 points and for scenario in Fig.4, the score of the white player is -590 points. We can see that the score for scenario in Fig.4 is less than score for scenario in Fig. 3 because we considered the weights assigned to each square in calculating the score and also it is clear that for scenario in Fig.3, white player has more control over the board(as number of possible moves for knight is more) than for scenario Fig.4).

Also, if we have used Piece-Value table approach for evaluation, then in fig 3 and fig 4 white player would have equal scores since we do not consider square weights in piece value table approach. The score of white player would be 60 + 30 + 50 + 90 +900 = 1130.

Fig. 3. Maximum Board control for white player with knight at center.

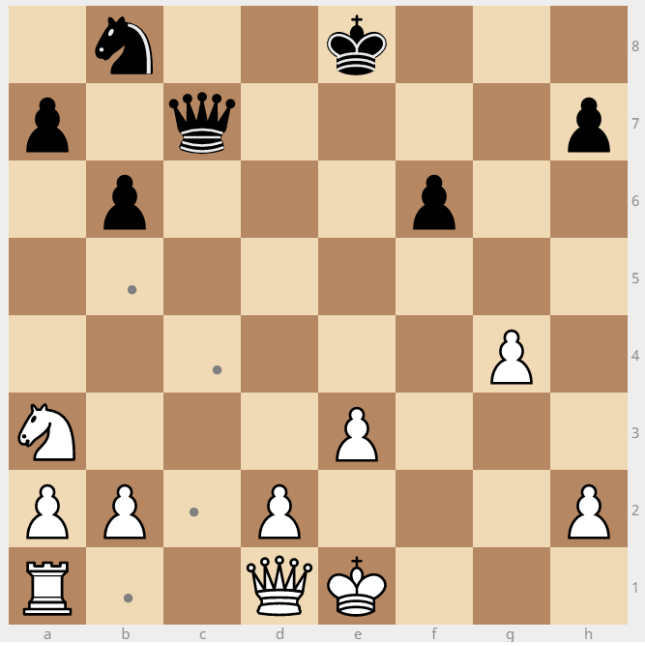

Fig. 4. Board control for white player with knight at corner.

IV. SEARCH

With the UI, move generation and Evaluation functions, we have setup the playground and its time for our AI to make the optimal moves. If we closely look at the evaluation function, the strategy for white player is to maximize the score and the strategy for black player is to minimize the score at any given point of time. As you have guessed, we are going to use Minimax algorithm.

A. Building Search Tree

Chess.js library gives us a set of valid moves to make from any given position. Depth is determined by the input we get from the UI. The recursive search tree is generated using the moves till the given depth. At alternating levels of the tree, black and white player will make moves to maximize of minimize the score. Following figure shows the basic search tree at depth 2.

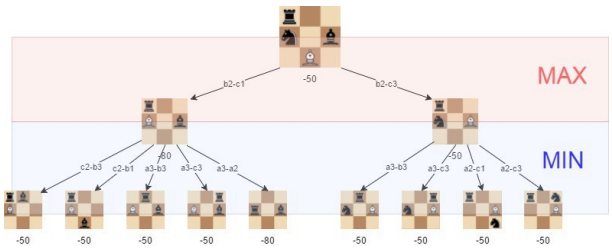

Fig. 5. Simple Game tree at depth 2

B. Search using Minimax

Minimax algorithms find the best possible move from the game tree that maximizes the score if it plays as white player and minimizes otherwise. We will have a look at the pseudo code of the algorithm that we use for this purpose. Given

```
MinValue(depth, game):
  Init moves with set of possible moves from the
       game
  object.
  topScore = MIN
  for each move in moves:
       make the move
       topScore = min(topScore, maxValue(depth - 1,
            game));
       undo the move
 return topScore;

MaxValue(depth, game):
  Init moves with set of possible moves from the
       game object.
  topScore = MAX
  for each move in moves:
       make the move
       topScore = max(topScore, minValue(depth - 1,
            game));
       undo the move
  return topScore;
```

this, the Minimax root is going to find the move that gives topScore based on whether its playing as white or black.

```
MinimaxRoot(depth, game, isWhite)
       Init moves with set of possible moves from
            the game object.
       if isWhite:
             From moves, return the move that gives
                  topScore using MaxValue function
       Else:
             From moves, return the move that gives
                  topScore using MinValue function
```

As we have seen, the depth determines the optimality of the move. For example, lets take a following scenario. Black just played Knight captures pawn on e4. If the engine thinks only at depth 1, white bishop will capture the queen as it maximizes reward. But its a famous trap.

Fig. 6. The famous Legal Trap

This game ends in two more moves. $Bxf2+8.Ke2Bg4-Mate$. Like the following figure.

Fig. 7. The famous Legal Trap - Checkmate

This could be avoided if the engine analyses the position for two more levels say Depth 3 or more. The deeper we search, the better. But, due to the limited resources we have, we are limiting the depth to at most 5. Still, we can improve the performance of the algorithm. We can avoid suboptimal

paths so that we reduce the positions we search. This is achieved with Alpha-beta pruning.

V. ALPHA-BETA PRUNING

In Minimax, is that the number of game states to examine are exponential in the depth of the tree. We wanted to eliminate searching through some branches that are assured to be poor than what we have already. So, we can evaluate deeper with the same resources. As we know, Alpha-beta pruning doesnt change the outcome of the algorithm. Its just an improvement as it reduces the number of game states to examine in search tree by a significant amount. We have implemented Alpha-Beta Pruning for this purpose. The pseudo code for Alpha-Beta pruning is given below:

```
minValue(depth,game,alpha,beta):
   if the node is at depth limit:
        return -score of board position
   else:
      get all legal moves for board position
      set score to min value possible
      for each move in list of legal moves:
       perform move
       score = Min(score, maxValue(depth-1,
            game, alpha, beta)
       undo the move
       beta = min of score and beta
       if beta is less than or equal to alpha:
            return score


maxValue(depth, game, alpha,beta):
   if the node is at depth limit:
      return -score of board position
   else:
      get all legal moves for board position
      set score to max value possible
      for each in list of legal moves:
         perform move
         score = Max(score, minValue(depth-1, game,
               alpha, beta)
         undo the move
         alpha = maximum of score and alpha
         if beta is less than or equal to alpha:
            return score
```

Alpha-beta pruning does not examine the states that do not give better results than what we already have. Therefore, it allows us to go deep in the tree and examine the moves. Alpha-beta pruning identifies the states which will not give better score, at earlier stages or levels in the tree, thus ignoring them early and spending more time in examining the states that we know might give us better score. In the fig.8 below, we can observe the same tree as shown in Fig 5 but with alpha-beta pruning. The difference is that by using alpha-beta pruning, we can ignore the states marked by red cross in fig.8. It is because we know that those states cannot give us better results than what we already have. At bottom level, we have 5 states on the left, out of which 4 have -50 as value and one has -80 and we are supposed to select minimum from these. We know that below this min level is a max level from where we got these values. So, there is no other way we can get other max values than -50 and -80 for this level. Now, we need to select minimum out of these values, which is -80. Therefore, we can ignore the remaining states as they only generate values greater than -80 and we need values less than -80.

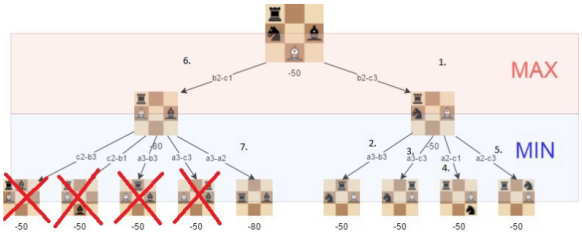

Fig. 8. State of Game tree shown in Fig.5 after implementing Alpha-Beta pruning

This is how Alpha-beta pruning helps in reducing the game states to examine in the tree.

Testing Performance of chess engine after Alpha-beta pruning:For to reach the state shown in fig.9. below, we performed 10 moves. In Table 2, is the count for number of positions evaluated for both the search algorithms. Computations for both the algorithms are performed at depth level 3.

Fig. 9. Chess board position after performing 10 moves.

Algorithm	Positions Evaluated
Minimax	40838
Minimax with Alpha-beta pruning	25287

Table 2. The performance of our chess engine at depth 3 using both the search algorithms.

VI. ENGINE EVALUATION

Firstly, the engine was tested to see if it makes any blunders (eg. Missing Checkmate opportunity, Loosing a piece without purpose, Overlooking checkmate threats from

opponent, etc.). We played numerous games and from the results, we were able to conclude that at Depth 3+, the engine doesn't make any blunders.

Chess engine/players rating is measured as ELO points. Our chess engines performance has been evaluated using Stockfish chess engine. Stockfish is one of the top chess engines rated with 3000+ ELO points. We played some games against the Stockfish chess engine available on www.chess.com. We reduced the difficulty of the Stockfish AI till our chess engine is able to beat it.

We evaluated the performance of our engine at different depth. In depth 1, the engine makes a lot of blunders. From depth 3, it doesn't make any blunders but the ELO point was only at the range of 1400s. We played games with our maximum depth (Level 5) and from the games, we were able to conclude that our chess engine is rated at ELO point range from 1600 to 1800.

1) Few things that are worth mentioning is that, our chess engine evaluates till depth 5. But Stockfish can evaluate till depth 22.
2) Stockfish chess engine has access to comprehensive open theory Database. This database contains list of opening 10-15 moves that are studied thoroughly by experts. So, they can use them to make the first 10-15 moves that are the most optimal moves to achieve maximum development (i.e. maximized Piece-Square value. We dont have that DB.
3) Also, they have end-game specific evaluation. During end-game, they have a special algorithm that kicks in to evaluate the board position to search the optimal move.
   For instance, there may be a situation where we must sacrifice our piece(Knight/Bishop/Rook) to draw the game or to promote the pawn to queen. These methods make Stockfish much better than our chess engine.

VII. FUTURE IMPROVEMENTS

There are a number areas where the engine can be improved.

A. Move ordering for deeper search

As mentioned earlier, the deeper we search the better. As the resources are limited, we use Alpha-Beta pruning to maximize the positions we can evaluate with the given resources. The performance of this algorithm is related to the number of branches/paths that we can prune. The more we prune the better. To prune a branch, we must have seen a better branch earlier. In our example, If we can see a better move that maximizes the topScore, we can prune the suboptimal branches/paths we will see in the future. If we can somehow order the moves such that the best moves comes first, the algorithm will be able to prune a lot of branches. This means we can search at deeper levels using the same resource. If we can search deeper, we are likely to find better moves.

B. Opening Theory DB

Almost all of the top chess engines like Stockfish, Komodo, etc. have access to the opening theory DB. Opening refers to the first 10-15 moves that involves developing the pieces like Bishop, Knight, Rook and Queen to active squares. Openings in chess game are critical as the board position after the piece development dictates the mid-game and end-game strategies. If we have a great opening/piece development, we have the advantage to apply pressure on the opponent. There are a lot of standard openings that are studies thoroughly by experts which are guaranteed to result in optimal piece development. These openings are stored in database which is contacted by those chess engines. This helps them to reach a superior position and play better later on. We don't have the database now. If we can integrate them, it would further increase our chances of making best moves.

C. End game specific evaluation

To win the game, especially on the end-games, we need a special evaluation technique. During end-games, both the players will try to promote the pawn to queen and win the game. Some times, we have to sacrifice the piece to stop the promotion and draw the game. Otherwise, the game will be lost. Similarly, we have to make some clearance sacrifices so that our own pawn's path is cleared and the pawn is promoted to a queen. These sacrifices won't be effective every time. They may be blunders/mistakes during mid-game. They are only applicable at the end-games. So, if we can make the end-game specific evaluation during the end-games, we can improve the chances of winning the game.

D. Self-Learning AI

This is the hot topic on chess engines or overall game AIs right now. Google recently released their self-learning chess engine named Alpha Zero. This engine is based on neural networks and it doesn't have any Opening theory DB or End-game specific evaluations. It is loaded only with the rules of the game. It plays with itself and learns from the mistakes/achievements. From that, it tries to play the most optimal moves over the period of time. But this would be a whole big project that will require comprehensive study and time.

VIII. CONTRIBUTION

A. User Interface

Chess.js and Chessboard.js libraries offered the APIs. We looked into those APIs and integrated the features that we need with our engine which is implemented in node.js.

Visualization using chessboard.js	Harshdeep
Move generation using chess.js	Sriramprabhu

B. Search & Evaluation

Minimax algorithm is used to search for a best move in the search tree. We used Alpha-Beta pruning to optimize the algorithm. At leaf/terminal node, the evaluation function is used to evaluated the position of the board.

We used www.chess.com to evaluate the performance of our search engine. We played games and consolidated the results to reach a conclusion.

Board Evaluation and Improved evaluation	Harshdeep
Search using Minimax and Alpha-Beta pruning	Sriramprabhu
Evaluation and Documentation	Combined effort

IX. ATTRIBUTION

Following libraries and blogs helped us in developing our engine:

- chess.js - https://github.com/jhlywa/chess.js - Javascript library used for Move generation & Validation.
- chessboard.js - http://chessboardjs.com/ - Javascript library for UI and board visualization.
- https://medium.freecodecamp.org/simple-chess-ai-step-by-step-1d55a9266977 - Blog on Writing chess AI.

We had to integrate the chess.js and chessboard.js libraries to our project. Also, we referred the blog to implement our search and evaluation functions.

REFERENCES

[1] www.chess.com
[2] https://stockfishchess.org/
[3] https://en.wikipedia.org/wiki/AlphaGo_Zero