

# FOVEROS Chess Engine

## (Modelling 2)

Lim, Ji Xiong  
Student no: 1327879  
Candidate no: 61135

Variath Paul, Joel  
Student no: 1352347  
Candidate no: 62790

November 28, 2014

Department of Mechanical Engineering  
University of Bristol

### **Abstract**

The FOVEROS Chess Engine project is an engine built in MATLAB that encompasses the interaction, display, legality and computer thought for a chess game. A brute force approach is taken to produce computer thought using tree-search methods, MiniMax algorithm and Alpha Beta Pruning optimisation. The optimisation has shown to reduce the time taken by 90%. A heuristic function was created to evaluate the board for the purposes of selecting the best path.

This Chess Engine successfully runs with minimal issues and demonstrates interesting characteristics such as aggression, the ability to checkmate, to castle and to evade check. The implementation of the rules of chess is near perfect and is able to execute special moves such as En Passant, Pawn Promotion and Castling.

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Modelling Aims</b>	<b>6</b>
<b>3</b>	<b>Overview</b>	<b>7</b>
3.1	Starting The Chess Engine . . . . .	7
<b>4</b>	<b>Back-end</b>	<b>8</b>
4.1	Movement files . . . . .	8
4.1.1	Pawn Movement . . . . .	9
4.1.2	Knight Movement . . . . .	10
4.1.3	Bishop Movement . . . . .	10
4.1.4	Rook Movement . . . . .	10
4.1.5	Queen Movement . . . . .	11
4.1.6	King Movement . . . . .	11
4.2	Special moves . . . . .	11
4.2.1	Castling . . . . .	12
4.2.2	En passant . . . . .	12
4.2.3	Pawn promotion . . . . .	13
4.3	Board Analysis . . . . .	13
4.3.1	Analyseboard Function . . . . .	13
4.3.2	KingCheck Function . . . . .	13
4.3.3	Checkmate Function . . . . .	13
<b>5</b>	<b>Front-end</b>	<b>14</b>
5.1	Click Series of Functions . . . . .	14

---

---

5.2	Graphical User Interface . . . . .	15
<b>6</b>	<b>Artificial Intelligence</b>	<b>17</b>
6.1	Brute Force Approach . . . . .	17
6.1.1	Minimax Algorithm . . . . .	17
6.1.2	Alpha-Beta Pruning . . . . .	17
6.2	Generation of Moves . . . . .	18
6.3	Heuristic Analysis . . . . .	19
6.4	AI Control . . . . .	19
<b>7</b>	<b>Results</b>	<b>20</b>
<b>8</b>	<b>Discussions</b>	<b>22</b>
8.1	Tuning . . . . .	22
8.2	Alpha Beta Pruning Efficiency . . . . .	22
<b>9</b>	<b>Conclusions</b>	<b>23</b>
<b>A</b>	<b>Appendix: The Code</b>	<b>25</b>
A.1	Back-end . . . . .	25
A.1.1	MovementRook.m . . . . .	25
A.1.2	MovementQueen.m . . . . .	26
A.1.3	MovementPawn.m . . . . .	28
A.1.4	MovementKnight.m . . . . .	30
A.1.5	MovementKing.m . . . . .	31
A.1.6	MovementBishop.m . . . . .	33
A.2	Front-end . . . . .	34
A.2.1	ClickPiece.m . . . . .	34

---

---

A.2.2	ClickCapturePiece.m . . . . .	37
A.2.3	ClickCastling.m . . . . .	39
A.2.4	ClickEnpassant.m . . . . .	42
A.2.5	ClickPawnPromo.m . . . . .	44
A.2.6	ClickMovePiece.m . . . . .	47
A.3	AI . . . . .	50
A.3.1	AIControl . . . . .	50
A.3.2	AI_GenerateAllMoves.m . . . . .	52
A.3.3	heuristicalysis.m . . . . .	56
A.4	Board analysis . . . . .	59
A.4.1	analyseboard.m . . . . .	59
A.4.2	KingCheck.m . . . . .	60
A.4.3	checkmate.m . . . . .	60
A.4.4	readchessboard.m . . . . .	61

---

## 1 Introduction

The history of chess spans several centuries back to India, where it is believed to have originated in the 6th century. Chess in its current form came into being around the early 1500's. With the rise of modern technology, chess is naturally the game of choice to demonstrate the implications of increased processing power in computers.

Chess has fascinated game theorists because it is deterministic which means that there can only be 1 winning side. It is also a game of perfect information, meaning that all the information is available to the player to make the best decision to move. This adversarial game also has well defined and rigid rules which limit movements to a relatively small number making it ideal for modelling.[6]

With the development of early computers in 1950's, the race was on to develop the ultimate chess engine that would never be beaten by a human player. Alan Turing wrote the first ever computer chess game in 1950. In 1996, Deep Blue won a match against Garry Kasparov. It was the first time a chess engine had beaten a reigning world champion.

Chess engines adopt a brute force approach. Therefore, development of chess engines tie together a broad range of topics, improve the understanding of tree searching algorithms and its optimisations, logical thought processes behind the game as well as a deeper understanding of computational thought and Artificial Intelligence(AI).

---

## 2 Modelling Aims

1. To create a program that displays a standard chessboard, where the user is able move the pieces legally.
  2. The program should be able to generate all possible future moves up to a specified depth using the brute force approach.
  3. The user must be able to specify game settings such as:
    - Difficulty - Random, Easy and Hard
    - Player choice - Player Vs Player, Player Vs AI and AI Vs AI
  4. The program must incorporate an algorithm that evaluates the chessboard from a certain player's perspective to aid the AI's decision.
  5. The program must display game messages, time taken to execute moves, player scores and a player performance graph.
-

### 3 Overview

The architecture of the system is split into 2 parts, the frontend and the backend. The frontend consists of the Mdp\_Chessboard package provided to us which is the basis for the board graphics that is displayed. The frontend initialises a structured variable called *B* which contains info about the game as well as a 16 by 16 matrix that portrays the location of the chessboard pieces as the game progresses.

The backend consists of 3 important variables that are initialised at the start of the game which are *chessboard*, *piece\_colour*, *num\_moves*. These are all 8 by 8 matrices that display the pieces as numbers and their relevant information. All manipulation and board calculations are carried out on the backend. The calculations of possible moves that can be made are also done in the backend.

Communication between the backend and frontend occurs through a function called *readchessboard*. This function interprets the backend variables and creates a new *B* that represents the backend which is ready to be plotted.

The reason why the backend variables was used in calculations rather than *B* is because *B* is a structure and contains extra information which would be unnecessary for calculation and would slow the system down.

The game control system is linear. The player of the next term, whether it is the user or the AI, is determined by the function called after the end of a move.

#### 3.1 Starting The Chess Engine

In order to run the game, type in "ChessGame" into the MATLAB console.

During gameplay, it is possible to change Player Choice. In shifting to an AI player, a move must be made in order to activate the AI's move process. There are some known issues with shifting players midway. It is also possible to change the AI settings midway and see a different characteristic. The gain values Heuristic Analysis file can be changed to give different characteristics.

In AI Vs AI gameplay, to stop the recurring move process, this can be done by selecting the other 2 options that will halt the AI thought.

## 4 Back-end

The back end of the chess engine works with 8x8 matrices that simulate a chessboard. This increases the speed of the engine, and reduces complexity. Only at the final stages, right before an output is generated, is the chessboard converted from its matrix form to a structure. This reduction in complexity is essential to the programming of basic movements.

Piece	Value
Pawn	1
Knight	3
Bishop	4
Rook	5
Queen	9
King	10

Figure 1: The value of each piece type

The matrix *chessboard* keeps track of the location of the pieces. Each type of piece is assigned a unique number that corresponds to its importance/value(See *Fig. 1*). Matrix *piece\_colour* stores the ASCII value of the piece colours. Another matrix *num\_moves* keeps track of how many moves each piece has made. Whenever a move is made, all three matrices are updated.

5	3	4	9	10	4	3	5
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
5	3	4	9	10	4	3	5

Figure 2: The *chessboard* matrix at game start

98	98	98	98	98	98	98	98
98	98	98	98	98	98	98	98
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
119	119	119	119	119	119	119	119
119	119	119	119	119	119	119	119

Figure 3: The *piece\_colour* matrix at game start

### 4.1 Movement files

Each piece type has its own movement function. This function takes in the position and colour of the piece and the chessboard and generates a matrix that shows all the possible legal moves that piece can make. Empty positions into which the piece can move



are denoted by the number 1. Positions currently occupied by the opponent, where the piece can move and make a capture are denoted by the number 2. (See *Fig. 4,5*)

5	0	0	0	10	0	3	5
1	1	1	0	0	1	1	0
0	0	3	0	0	0	0	0
0	4	0	1	1	4	3	1
0	4	0	0	1	4	0	9
0	0	3	1	0	0	0	0
1	1	1	0	0	1	1	1
0	0	5	9	10	0	0	5

Figure 4: A random board with the queen highlighted

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	2	0
0	0	0	0	0	2	1	0
0	0	0	0	0	0	1	1
0	0	0	0	0	2	0	2
0	0	0	0	0	0	0	0

Figure 5: Possible moves of the queen - "1" indicates empty squares, "2" indicates potential captures

#### 4.1.1 Pawn Movement

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 6: Possible moves of a pawn from the highlighted square

### 4.1.2 Knight Movement

0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0
0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 7: Possible moves of a knight from the highlighted square

### 4.1.3 Bishop Movement

1	0	0	0	0	0	1	0
0	1	0	0	0	1	0	0
0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	0	0	1	0	0
1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1

Figure 8: Possible moves of a bishop from the highlighted square

### 4.1.4 Rook Movement

0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
1	1	1	0	1	1	1	1
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0

Figure 9: Possible moves of a rook from the highlighted square

#### 4.1.5 Queen Movement

1	0	0	1	0	0	1	0
0	1	0	1	0	1	0	0
0	0	1	1	1	0	0	0
1	1	1	0	1	1	1	1
0	0	1	1	1	0	0	0
0	1	0	1	0	1	0	0
1	0	0	1	0	0	1	0
0	0	0	1	0	0	0	1

Figure 10: Possible moves of a queen from the highlighted square

#### 4.1.6 King Movement

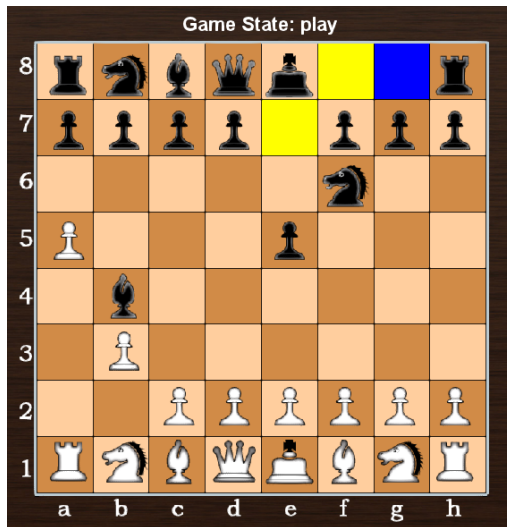
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0
0	0	1	0	1	0	0	0
0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 11: Possible moves of a king from the highlighted square

### 4.2 Special moves

The FOVEROS Chess Engine is programmed to execute special moves such as castling, en passant and pawn promotion. However in the case of en passant, the move is not restricted to the first opportunity of its execution.

### 4.2.1 Castling



(a) Board shows possibility of castling



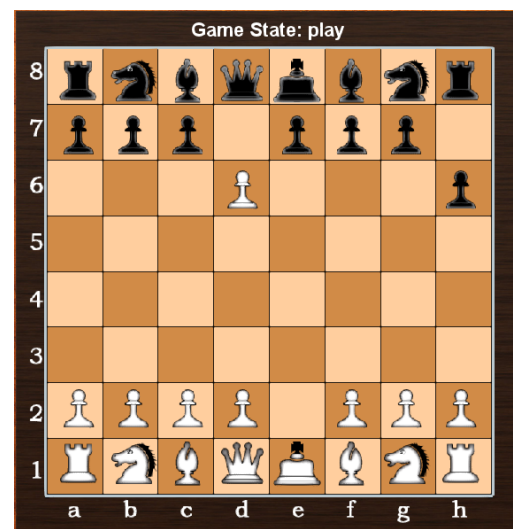
(b) Castling took place

Figure 12: Before and after Castling

### 4.2.2 En passant



(a) Board shows possibility of En passant



(b) En passant took place

Figure 13: Before and after En passant

### 4.2.3 Pawn promotion



(a) Board shows possibility of Pawn promotion

(b) Pawn promotion took place

Figure 14: Before and after Pawn promotion

## 4.3 Board Analysis

### 4.3.1 Analyseboard Function

The function *analyseboard* looks at all the pieces of a specified colour and determines where every piece is able to move. This function generates the matrix *potentialmoves* containing all the possible moves that side can make and a vector *capt\_index* containing the locations of possible captures. This function lends itself to the *KingCheck* and *checkmate* functions.

### 4.3.2 KingCheck Function

The function *KingCheck* uses the results of *analyseboard* to determine if the king is in check. The king is in check when its position overlaps with the opponent's potential moves. This function lends itself to the *checkmate* function.

### 4.3.3 Checkmate Function

The function *checkmate* determines if the current board is a checkmate state for the specified colour. This function generates all the possible moves searching for any legal moves that can be made. Even if one legal move exists, then the board is determined to not be in a state of checkmate.

*checkmate* assumes that the king is already in check and hence must always be paired with the *KingCheck* function.

## 5 Front-end

### 5.1 Click Series of Functions

This section discusses the following functions:

- *ClickPiece*
- *ClickCapturePiece*
- *ClickMovePiece*
- *ClickPawnPromo*
- *ClickCastling*
- *ClickEnPassant*

The purpose of the click series of functions is to enable the user to make selections on the GUI that translate into a new board state. *ClickPiece* is embedded in the `ButtonDown` function of the images on the plot. When a user selects a piece, its moves are highlighted as shown. In essence, *ClickPiece* acts like a switch to embed the other 5 functions into the right squares so that the user is able to make the corresponding move. (See *Fig. 15*)

Line 9 – 17: Determines the colour that is at play.

Line 19 - 31: Determines the coordinates of the user selection and makes the relevant coordinate conversions for the backend.

Line 34 – 51: Determines the piece that is selected and generates its possible moves.

Line 52 – 104: The squares of the board are redrawn with colours corresponding to the matrix of the possible moves. For example: A capture is shown in the possible moves matrix as ‘2’ and the corresponding square is coloured red.

Lines 108 – 134: This section redraws the pieces on the board and embeds “ClickPiece” again so that the user can reselect another piece if necessary.

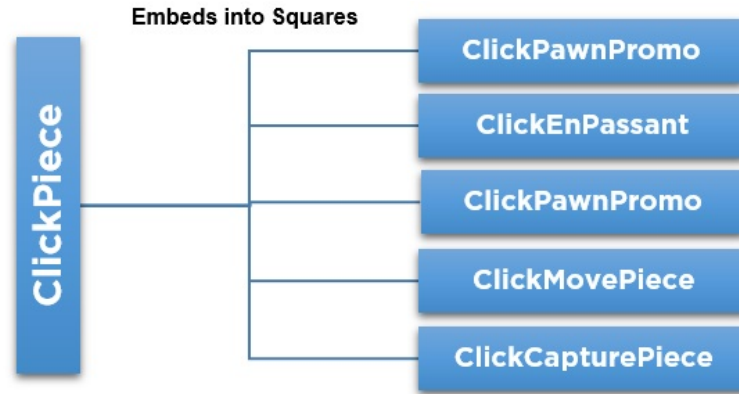


Figure 15: Flow chart showing the activation of *ClickPiece*

This section will detail the code of the 5 click functions for *ClickMovePiece*, *ClickCapturePiece*, *ClickCastling*, *ClickEnPassant* and *ClickPawnPromo*. The code for all of them are very similar, except their function specific parts. The code explanation below details the *ClickMovePiece* function:

Line 8 – 14 : Determines the colour at play

Line 17 – 32: Determines the coordinates that player has chosen to move the piece and makes the relevant coordinate conversions for the backend.

Line 40 – 60: The move is done on a future board state to validate that the King will not left in check as a result of the move which constitutes and illegal move. This is where the different click functions will have function specific coordinate transformations to facilitate their purpose.

Line 73 – 75: If the move is valid, the future board state is saved as the accepted board state.

Line 77 – 94: Validates if the opposing King is in check as a result of the move and also validates of checkmate or stalemate have taken place.

Line 99 – 128: The board is redrawn to reflect the new board state.

Line 131 – 138: Based on the user's player choice, the next move is passed to the relevant control function.

## 5.2 Graphical User Interface

The GUI is designed to show the relevant board stats and game messages. The user is able to select from different settings for the GUI. It also plots the scores on the graph to show the progression of both players with each turn.



Figure 16: FOVEROS Chess Engine GUI



## 6 Artificial Intelligence

### 6.1 Brute Force Approach

The approach to implementing computer thought is through brute force calculations. The strength in computer systems lies in its ability to perform a large amounts of computation in a short period of time. Therefore, it has the ability to generate all future board states up to a certain depth. Depth is the number of turns ahead of the current board state that is generated.

#### 6.1.1 Minimax Algorithm

The generation of board states must be aided by an algorithm to evaluate and save the best board states. This is an adversarial search problem where only 2 players are competing. In order to make the optimal decision for a certain player, the MiniMax strategy is used. The idea is that from a certain player's perspective, the objective is to maximise the player's advantage and to minimise the opponent's advantage. The generation starts from the "Initial State" and branches out until it reaches the "Leaf Nodes" or "Terminal Nodes" where the board is evaluated and the results backed up the tree as the recursion unwinds.[6]

Generation of board states however is a computationally expensive task especially to deeper depths. The branching factor can be defined as the number of children per node. The average branching factor for chess positions is 35 to 38 moves per position. Therefore,  $38^2$  (1,444) game states need to be evaluated at depth 2,  $38^3$  (54,872) game states need to be evaluated at depth 3,  $38^4$  (2,085,136) game states need to be evaluated at depth 4. It is clear that the number of game states will continue to grow at an exponential rate.[1]

#### 6.1.2 Alpha-Beta Pruning

An algorithm that "prunes" the search tree needs to use to eliminate parts of the tree that will have no effect on the final result of the tree. Alpha Beta pruning is a method to prune the search tree. Alpha is the maximum lower bound of the possible solutions and Beta is the minimum upper bound of the possible solutions. The search proceeds down to the first terminal node of the tree, the board score is backed up the tree and becomes either the alpha or beta value of the node above it depending on whether it is a minimum or maximum player.



Figure 17: Figure shows  $\alpha$ - $\beta$  pruning criteria

The search then proceeds and references back to the alpha and beta value before it. If the values are found to be lower than the maximum lower bound or higher than the minimum upper bound, the values of alpha and beta are reassigned to the new values. However, pruning occurs when alpha is greater than beta and the branches leading from the selected node are not evaluated because it will not change the overall result of the tree. It will not change it because there is no overlap between the maximum lower bound and the minimum upper bound as shown in the figure.[7]

By implementing Alpha Beta pruning with random move ordering, the average number of nodes to be evaluated will be dropped to  $b^{3d/4}$  from  $b^d$ . This means that Alpha Beta pruning is able to perform a depth 4 calculation at roughly the same speed as a depth 3 without Alpha Beta pruning.[8]

## 6.2 Generation of Moves

The function *AI\_GenerateAllMoves* generates all nodes up to a certain depth, implements the MiniMax Algorithm and Alpha Beta pruning. The function does this via the recursion method where the function calls itself within its function. This is useful because in the MiniMax algorithm we require the values to be backed up the tree and recursion does that inherently.

Line 1 – 15: Determines the colour at play

Line 20 – 26: At Terminal Node, the board is evaluated

Line 29 – 122: Implementation for the Maximum Player

Line 126 – 222: Implementation for the Minimum Player

The implementation for the Maximum and Minimum player are the same but with slightly different parameters. The general structure is discussed below for the Maximum Player.

Line 33 – 36: The coordinates of the current colour pieces are found. The arrangement of the pieces in the vector is randomly permuted.

Line 42 – 57: For each of the pieces, the individual possible moves are generated

Line 63 – 67: The coordinates of the possible moves are randomly permuted

Line 70 – 87: The potential future board state is generated.

Line 89 – 116: The board is evaluated for legality, if found to be illegal it will be ignored. If it is valid, the function calls itself again. The parameters passed are the new node and the depth is decreased. The alpha-beta pruning condition is also implemented here.

---

### 6.3 Heuristic Analysis

The function *heuristicanalysis* examines the board from a given player's point of view. The goal of this function is to assign a numerical value(boardscore) to each board depending on a given set of conditions. These conditions determine if a move is good or bad.[2][9]

A move is judged to be good:

- If it opens up possibilities to capture opponent's pieces
- If it increases the number of pieces captured
- If it opens up space for other pieces to move
- If it increases control of the centre of the board
- If it enables the king to castle
- If it brings the pawns closer to the end of the board for promotion
- If it leads to checks and checkmates on the opponent

A move is judged to be bad:

- If it causes threats to its own pieces
- If it decreases the number of its own pieces
- If it leads to the opponent's pawns being promoted
- If it leads to its own king being checked or checkmated

Each of these conditions is assigned a specific *gainfactor*. The more important the condition is, the higher the numerical value of the *gainfactor*. Good moves have positive *gainfactor* which encourage the AI to make those moves, while bad moves have negative *gainfactor*, discouraging the AI.

For example, a move that leads to the opponent king being checkmated is assigned the highest possible score of 99999, while a move that leads to its own checkmate is assigned the lowest possible score of -99999.

The values of the Gain factors depend on the level of difficulty the player has chosen. In a 'Random' game the gains are all set to 0 and have no influence on the moves. In an 'Easy' game, the gains are such that it's easy for the player to make threats and capture pieces. In a 'Hard' game, the gains ensure that the AI plays aggressively.

### 6.4 AI Control

The function *AIControl* sets the depth parameters for *AI\_GenerateAllMoves*, plots the board scores as seen on the GUI and validates check/checkmate/stalemate for both colours.

---

7 Results

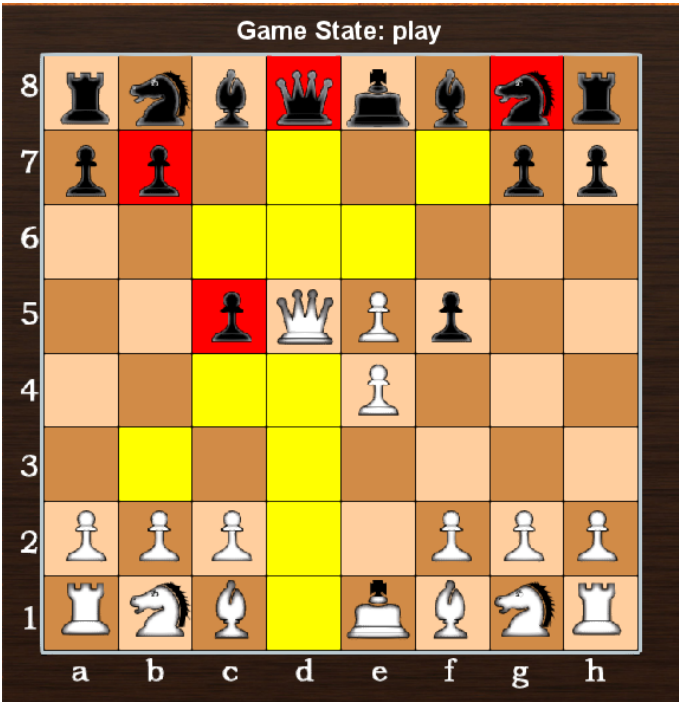


Figure 18: Possible moves of Queen in game



Figure 19: Checkmate by Black



Figure 20: AI has Castled

Depth	Without Alpha Beta Pruning		With Alpha Beta Pruning		Percentage Reduction Between With and Without Pruning	
	Time/s	Nodes	Time/s	Nodes	Time	Node
1	0.046	20	0.05	20	8.70	0
2	1.45	620	0.31	136	-78.6	-78.1
3	46.7	13928	1.98	832	-95.8	-94.0
4	1365	420180	23.4	8535	-98.3	-98.0
5	>2400	>720000	96	33022	-	-

Table 1: Performance results of  $\alpha$ - $\beta$  pruning

## 8 Discussions

### 8.1 Tuning

Tuning is the process of finding the optimum gains for the different scores in the Heuristics Analysis function. The different gains increase or decrease the effect of different parameters on the board score. Tuning is a difficult process because the values are relative and therefore trial and error is to be used to determine the optimum value. By comparison, the programmers of the legendary Deep Blue machine had made plausible initial guesses for values and there was a lot of uncertainty to what the correct values should be.[5]

The first objective for optimum tuning was to reach a high level of aggressiveness for the AI as that is most noticeable feature. This would be demonstrated by making captures when the opportunity presents itself. The second objective was to prevent moves that would jeopardise its own pieces and that also includes encouraging castling to shield the King. The third objective was to increase likelihood of check and checkmate.

The current gains that are set are based on trial and error. Based on the results, some of the features have demonstrated itself rather obviously. *Fig. 19* shows the situation of Player Vs AI and the AI has successfully checkmated the player in a rather creative arrangement that involves 3 different pieces. The graph also shows that the boardscore has been maximised for that colour which is the correct output from the Heuristics Analysis.

*Fig. 20* also shows the AI using Castling to its advantage in the late game by allowing the rook to come out and the King to seek shelter behind the pawns thus increasing its boardscore as a result of the move.

A deficiency that is quite common in the AI was that the AI tended to make captures that would lead to its own piece being captured. The pay off would sometimes be less than the value of the piece being sacrificed. The AI would also pass off captures sometimes but it is rare.

### 8.2 Alpha Beta Pruning Efficiency

*Table 1* shows the various timings and number of nodes at a certain depth. The results is as expected, the number of nodes that need to be evaluated at a certain depth have an exponential relationship with the depth itself. The Alpha Beta Pruning implementation cuts down the time taken by a very substantial amount. This is also as expected as discussed in the Artificial Intelligence section. It should enable the system to go 1 depth deeper with Alpha Beta Pruning at the same speed as without the Alpha Beta Pruning.

---

## 9 Conclusions

The FOVEROS Engine which means “Awesome” in Greek employs basic understanding of the tree search algorithm, MiniMax Theory, Alpha Beta pruning optimisation. It has also met the design aim of being a functional chessboard that has inbuilt rules and able to execute all the special moves without problems.

There are further improvements that can be made for this engine. The first is the integration of a database of opening moves. This will enable the engine to have greater flexibility in its opening moves and open up more possibilities for the engine. This can be done by setting the AI to respond with predefined moves for the first 3 rounds based on the response of the opponent.

Further optimisations to the tree-search algorithm can be looked into. The first is Iterative Deepening which encourages deeper searches until the pre-allocated time is exhausted. It is a time management strategy in depth-first searches and has benefits for move ordering and pruning.[3]

FOVEROS employs random move ordering in the algorithm and so dynamic move ordering techniques should be looked into. The benefits of Alpha Beta pruning are only tangible if the best move is presented as soon as possible so that pruning can take place immediately.[4]

## References

- [1] *Chess Programming*. 2014. <https://chessprogramming.wikispaces.com/Branching+Factor> [Online; last accessed 20-Nov-2014].
  - [2] Chess Central. *Introduction to Chess Strategy*. 2014. [http://www.chesscentral.com/chess\\_strategy\\_a/201.htm](http://www.chesscentral.com/chess_strategy_a/201.htm) [Online; last accessed 20-Nov-2014].
  - [3] ChessProgramming. *Iterative Deepening*. 2014. <https://chessprogramming.wikispaces.com/Iterative+Deepening> [Online; last accessed 12-Nov-2014].
  - [4] Cornell University. *Minimax search and Alpha-Beta Pruning*. 2014. <https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm> [Online; last accessed 13-Nov-2014].
  - [5] Furnkranz, Johannes and Kubat, Miroslav. *Machines That Learn To Play Games*. Nova Science Publishers, Huntington, New York, 2001.
  - [6] Russell, Stuart J. and Norvig, Peter. *Artificial Intelligence - A Modern Approach*. Pearson Education, Upper Saddle River, New Jersey, 2010.
  - [7] UCLA. *Minimax with Alpha Beta Pruning*. 2014. <http://cs.ucla.edu/~rosen/161/notes/alphabeta.html> [Online; last accessed 10-Nov-2014].
  - [8] Wikipedia. *Alpha Beta Pruning*. 2014. [http://en.wikipedia.org/wiki/Alpha%E2%80%93beta\\_pruning](http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning) [Online; last accessed 14-Nov-2014].
  - [9] zaifrun, Chess.com. *Creating a Chess Engine*. 2014. <http://www.chess.com/blog/zaifrun/creating-a-chess-engine-part-4---position-evaluation> [Online; last accessed 12-Nov-2014].
-



## A Appendix: The Code

### A.1 Back-end

#### A.1.1 MovementRook.m

```

1
2 function [possiblemoves] = MovementRook(chessboard,piece_colour,p_x,p_y)
3
4 %Initialisation values -----
5 r_colour = piece_colour(p_x,p_y);
6 possiblemoves = zeros(8,8);
7
8 %This section allows movement in vertical direction -----
9 i = 1;
10 while(p_x+i<9)
11     if(piece_colour(p_x+i,p_y)== r_colour)
12         break
13     end
14     if(piece_colour(p_x+i,p_y)~= r_colour && chessboard(p_x+i,p_y)~=0)
15         possiblemoves(p_x+i,p_y) = 2;
16         break
17     end
18     possiblemoves(p_x+i,p_y) = 1;
19     i = i+1;
20 end
21
22 i = 1;
23 while(p_x-i>0)
24     if(piece_colour(p_x-i,p_y)== r_colour)
25         break
26     end
27     if(piece_colour(p_x-i,p_y)~= r_colour && chessboard(p_x-i,p_y)~=0)
28         possiblemoves(p_x-i,p_y) = 2;
29         break
30     end
31     possiblemoves(p_x-i,p_y) = 1;
32     i = i+1;
33 end
34
35 %This section allows movement in the horizontal direction
36 i = 1;
37 while(p_y+i<9)
38     if(piece_colour(p_x,p_y+i)== r_colour)
39         break
40     end
41     if(piece_colour(p_x,p_y+i)~= r_colour && chessboard(p_x,p_y+i)~=0)
42         possiblemoves(p_x,p_y+i) = 2;
43         break
44     end
45     possiblemoves(p_x,p_y+i) = 1;
46     i = i+1;
47 end
48
49 i = 1;
50 while(p_y-i>0)
51     if(piece_colour(p_x,p_y-i)== r_colour)
52         break

```

```

53     end
54     if(piece_colour(p_x,p_y-i)~= r_colour && chessboard(p_x,p_y-i)~=0)
55         possiblemoves(p_x,p_y-i) = 2;
56         break
57     end
58     possiblemoves(p_x,p_y-i) = 1;
59     i = i+1;
60 end
61
62 %-----
63
64 end

```

### A.1.2 MovementQueen.m

```

1  function [possiblemoves] = MovementQueen(chessboard,piece_colour,p_x,p_y)
2
3  %Initialisation values -----
4  possiblemoves = zeros(8,8);
5  r_colour = piece_colour(p_x,p_y);
6
7  %This section allows movement in / direction -----
8  i=1;
9  while(p_x+i<9 && p_y+i<9)
10     if(piece_colour(p_x+i,p_y+i)== r_colour)
11         break
12     end
13     if(piece_colour(p_x+i,p_y+i)~= r_colour && chessboard(p_x+i,p_y+i)~=0)
14         possiblemoves(p_x+i,p_y+i) = 2;
15         break
16     end
17     possiblemoves(p_x+i,p_y+i) = 1;
18     i = i+1;
19 end
20
21 i=1;
22 while(p_x-i>0 && p_y-i>0)
23     if(piece_colour(p_x-i,p_y-i)== r_colour)
24         break
25     end
26     if(piece_colour(p_x-i,p_y-i)~= r_colour && chessboard(p_x-i,p_y-i)~=0)
27         possiblemoves(p_x-i,p_y-i) = 2;
28         break
29     end
30     possiblemoves(p_x-i,p_y-i) = 1;
31     i = i+1;
32 end
33
34 %This section allows movement in the \ direction-----
35 i=1;
36 while(p_x+i<9 && p_y-i>0)
37     if(piece_colour(p_x+i,p_y-i)== r_colour)
38         break
39     end
40     if(piece_colour(p_x+i,p_y-i)~= r_colour && chessboard(p_x+i,p_y-i)~=0)
41         possiblemoves(p_x+i,p_y-i) = 2;
42         break
43     end

```

```

44     possiblemoves(p_x+i,p_y-i) = 1;
45     i = i+1;
46 end
47
48 i=1;
49 while(p_x-i>0 && p_y+i<9)
50     if(piece_colour(p_x-i,p_y+i)== r_colour)
51         break
52     end
53     if(piece_colour(p_x-i,p_y+i)~= r_colour && chessboard(p_x-i,p_y+i)~=0)
54         possiblemoves(p_x-i,p_y+i) = 2;
55         break
56     end
57     possiblemoves(p_x-i,p_y+i) = 1;
58     i = i+1;
59 end
60
61 %This section allows movement in vertical direction
62 i = 1;
63 while(p_x+i<9)
64     if(piece_colour(p_x+i,p_y)== r_colour)
65         break
66     end
67     if(piece_colour(p_x+i,p_y)~= r_colour && chessboard(p_x+i,p_y)~=0)
68         possiblemoves(p_x+i,p_y) = 2;
69         break
70     end
71     possiblemoves(p_x+i,p_y) = 1;
72     i = i+1;
73 end
74
75 i = 1;
76 while(p_x-i>0)
77     if(piece_colour(p_x-i,p_y)== r_colour)
78         break
79     end
80     if(piece_colour(p_x-i,p_y)~= r_colour && chessboard(p_x-i,p_y)~=0)
81         possiblemoves(p_x-i,p_y) = 2;
82         break
83     end
84     possiblemoves(p_x-i,p_y) = 1;
85     i = i+1;
86 end
87
88 %This section allows movement in the horizontal direction
89 i = 1;
90 while(p_y+i<9)
91     if(piece_colour(p_x,p_y+i)== r_colour)
92         break
93     end
94     if(piece_colour(p_x,p_y+i)~= r_colour && chessboard(p_x,p_y+i)~=0)
95         possiblemoves(p_x,p_y+i) = 2;
96         break
97     end
98     possiblemoves(p_x,p_y+i) = 1;
99     i = i+1;
100 end
101
102 i = 1;
103 while(p_y-i>0)
104     if(piece_colour(p_x,p_y-i)== r_colour)
105         break

```

```

106     end
107     if(piece_colour(p_x,p_y-i)~= r_colour && chessboard(p_x,p_y-i)~=0)
108         possiblemoves(p_x,p_y-i) = 2;
109         break
110     end
111     possiblemoves(p_x,p_y-i) = 1;
112     i = i+1;
113 end
114 %-----
115
116 end

```

### A.1.3 MovementPawn.m

```

1  function [possiblemoves] = MovementPawn(chessboard,piece_colour,num_moves,p_x,p_y)
2
3  %Initialisation values -----
4  r_colour = piece_colour(p_x,p_y);
5  possiblemoves = zeros(8,8);
6
7  %This section allows all movements after checking whether it exceeds the board or not -----
8  switch r_colour
9      case 119 %White case
10         %En passant-----
11         if (p_x==4)
12
13             if(p_x-1>0 && p_y-1 >0) %Capture left
14                 if(piece_colour(p_x,p_y-1)~=r_colour && chessboard(p_x,p_y-1)==1 && num_moves==0)
15                     possiblemoves(p_x-1,p_y-1) = 3;
16                 end
17             end
18
19             if(p_x-1>0 && p_y+1<9) %Capture right
20                 if(piece_colour(p_x,p_y+1)~=r_colour && chessboard(p_x,p_y+1)==1 && num_moves==0)
21                     possiblemoves(p_x-1,p_y+1) = 3;
22                 end
23             end
24
25         end
26
27         if(p_x-1>0) %Forward movement
28             if(chessboard(p_x-1,p_y)==0)
29                 possiblemoves(p_x-1,p_y) = 1;
30             end
31         end
32
33         %Initial forward movement
34         if(p_x==7 && chessboard(p_x-2,p_y)==0 && chessboard(p_x-1,p_y)==0)
35             possiblemoves(p_x-2,p_y) = 1;
36         end
37
38         if(p_x-1>0 && p_y-1 >0) %Capture left
39             if(piece_colour(p_x-1,p_y-1)~=r_colour && chessboard(p_x-1,p_y-1)~=0)
40                 possiblemoves(p_x-1,p_y-1) = 2;
41                 if(p_x==2) %Capture and pawn promotion
42                     possiblemoves(p_x-1,p_y-1) = 5;
43                 end
44             end

```

```

45         end
46
47         if(p_x-1>0 && p_y+1<9) %Capture right
48             if(piece.colour(p_x-1,p_y+1)~=r.colour && chessboard(p_x-1,p_y+1)~=0)
49                 possiblemoves(p_x-1,p_y+1) = 2;
50                 if(p_x==2) %Capture and pawn promotion
51                     possiblemoves(p_x-1,p_y+1) = 5;
52                 end
53             end
54         end
55
56         %Pawn promotion-----
57         if(p_x==2)
58             if(chessboard(p_x-1,p_y)==0)
59                 possiblemoves(p_x-1,p_y) = 5;
60             end
61         end
62
63     case 98 %Black Case
64
65         %En passant-----
66         if (p_x==5)
67
68             if(p_x-1>0 && p_y-1 >0) %Capture left
69                 if(piece.colour(p_x,p_y-1)~=r.colour && chessboard(p_x,p_y-1)==1 && num_move
70                     possiblemoves(p_x+1,p_y-1) = 3;
71                 end
72             end
73
74             if(p_x-1>0 && p_y+1<9) %Capture right
75                 if(piece.colour(p_x,p_y+1)~=r.colour && chessboard(p_x,p_y+1)==1 && num_move
76                     possiblemoves(p_x+1,p_y+1) = 3;
77                 end
78             end
79
80         end
81
82         if(p_x+1<9) %Forward movement
83             if(chessboard(p_x+1,p_y)==0)
84                 possiblemoves(p_x+1,p_y) = 1;
85             end
86         end
87
88         %Initial Forward movement
89         if(p_x==2 && chessboard(p_x+2,p_y)==0 && chessboard(p_x+1,p_y)==0)
90             possiblemoves(p_x+2,p_y) = 1;
91         end
92
93         if(p_x+1<9 && p_y-1>0) %Capture left
94             if(piece.colour(p_x+1,p_y-1)~=r.colour && chessboard(p_x+1,p_y-1)~=0)
95                 possiblemoves(p_x+1,p_y-1) = 2;
96                 if(p_x==7) %Capture and pawn promotion
97                     possiblemoves(p_x+1,p_y-1) = 5;
98                 end
99             end
100         end
101
102         if(p_x+1<9 && p_y+1<9) %Capture right
103             if(piece.colour(p_x+1,p_y+1)~=r.colour && chessboard(p_x+1,p_y+1)~=0)
104                 possiblemoves(p_x+1,p_y+1) = 2;
105                 if(p_x==7) %Capture and pawn promotion
106                     possiblemoves(p_x+1,p_y+1) = 5;

```

```

107             end
108         end
109     end
110
111     %Pawn promotion
112     if(p_x==7)
113         if(chessboard(p_x+1,p_y)==0)
114             possiblemoves(p_x+1,p_y) = 5;
115         end
116     end
117
118 end
119 %
120 end

```

#### A.1.4 MovementKnight.m

```

1 function [possiblemoves] = MovementKnight(chessboard,piece_colour,p_x,p_y)
2
3 %Initialisation values
4 r_colour = piece_colour(p_x,p_y);
5 possiblemoves = zeros(8,8);
6
7 %This sections allows L shaped movements for knight
8 if(p_x-2>0 & p_y-1>0)
9     if (piece_colour(p_x-2,p_y-1)~= r_colour && chessboard(p_x-2,p_y-1)~=0)
10         possiblemoves(p_x-2,p_y-1) = 2;
11     elseif (piece_colour(p_x-2,p_y-1)== r_colour)
12         ;
13     else
14         possiblemoves(p_x-2,p_y-1) = 1;
15     end
16 end
17
18 if(p_x-2>0 & p_y+1<9)
19     if (piece_colour(p_x-2,p_y+1)~= r_colour && chessboard(p_x-2,p_y+1)~=0)
20         possiblemoves(p_x-2,p_y+1) = 2;
21     elseif (piece_colour(p_x-2,p_y+1)== r_colour)
22         ;
23     else
24         possiblemoves(p_x-2,p_y+1) = 1;
25     end
26 end
27
28 if(p_x-1>0 & p_y-2>0)
29     if (piece_colour(p_x-1,p_y-2)~= r_colour && chessboard(p_x-1,p_y-2)~=0)
30         possiblemoves(p_x-1,p_y-2) = 2;
31     elseif (piece_colour(p_x-1,p_y-2)== r_colour)
32         ;
33     else
34         possiblemoves(p_x-1,p_y-2) = 1;
35     end
36 end
37
38 if(p_x-1>0 & p_y+2<9)
39     if (piece_colour(p_x-1,p_y+2)~= r_colour && chessboard(p_x-1,p_y+2)~=0)
40         possiblemoves(p_x-1,p_y+2) = 2;
41     elseif (piece_colour(p_x-1,p_y+2)== r_colour)

```

```

42         ;
43     else
44         possiblemoves(p_x-1,p_y+2) = 1;
45     end
46 end
47
48 if(p_x+1<9 & p_y-2>0)
49     if (piece_colour(p_x+1,p_y-2)~= r_colour && chessboard(p_x+1,p_y-2)~=0)
50         possiblemoves(p_x+1,p_y-2) = 2;
51     elseif (piece_colour(p_x+1,p_y-2)== r_colour)
52         ;
53     else
54         possiblemoves(p_x+1,p_y-2) = 1;
55     end
56 end
57
58 if(p_x+1<9 & p_y+2<9)
59     if (piece_colour(p_x+1,p_y+2)~= r_colour && chessboard(p_x+1,p_y+2)~=0)
60         possiblemoves(p_x+1,p_y+2) = 2;
61     elseif (piece_colour(p_x+1,p_y+2)== r_colour)
62         ;
63     else
64         possiblemoves(p_x+1,p_y+2) = 1;
65     end
66 end
67
68 if(p_x+2<9 & p_y-1>0)
69     if (piece_colour(p_x+2,p_y-1)~= r_colour && chessboard(p_x+2,p_y-1)~=0)
70         possiblemoves(p_x+2,p_y-1) = 2;
71     elseif (piece_colour(p_x+2,p_y-1)== r_colour)
72         ;
73     else
74         possiblemoves(p_x+2,p_y-1) = 1;
75     end
76 end
77
78 if(p_x+2<9 & p_y+1<9)
79     if (piece_colour(p_x+2,p_y+1)~= r_colour && chessboard(p_x+2,p_y+1)~=0)
80         possiblemoves(p_x+2,p_y+1) = 2;
81     elseif (piece_colour(p_x+2,p_y+1)== r_colour)
82         ;
83     else
84         possiblemoves(p_x+2,p_y+1) = 1;
85     end
86 end
87
88 %-----
89
90 end

```

### A.1.5 MovementKing.m

```

1 function [possiblemoves] = MovementKing(chessboard,piece_colour,num_moves,potential_moves,p
2
3 %Initialisation values -----
4 r_colour = piece_colour(p_x,p_y);
5 possiblemoves = zeros(8,8);
6

```

```

7  %This section allows all movements after checking whether it exceeds the board or not
8  %and ensures that the king is not moving into square that is in check
9
10 %-----
11 %                               Movement (8 Directions)
12 %-----
13
14
15     if(p_x+1<9)
16         if (piece_colour(p_x+1,p_y)~= r_colour && chessboard(p_x+1,p_y)~=0)
17             possiblemoves(p_x+1,p_y) = 2;
18         elseif (piece_colour(p_x+1,p_y)== r_colour)
19             ;
20         else
21             possiblemoves(p_x+1,p_y) = 1;
22         end
23     end
24
25
26     if(p_x+1<9 && p_y+1<9)
27         if (piece_colour(p_x+1,p_y+1)~= r_colour && chessboard(p_x+1,p_y+1)~=0)
28             possiblemoves(p_x+1,p_y+1) = 2;
29         elseif (piece_colour(p_x+1,p_y+1)== r_colour)
30             ;
31         else
32             possiblemoves(p_x+1,p_y+1) = 1;
33         end
34     end
35
36
37     if(p_x+1<9 && p_y-1>0)
38         if (piece_colour(p_x+1,p_y-1)~= r_colour && chessboard(p_x+1,p_y-1)~=0)
39             possiblemoves(p_x+1,p_y-1) = 2;
40         elseif (piece_colour(p_x+1,p_y-1)== r_colour)
41             ;
42         else
43             possiblemoves(p_x+1,p_y-1) = 1;
44         end
45     end
46
47
48     if(p_y+1<9)
49         if (piece_colour(p_x,p_y+1)~= r_colour && chessboard(p_x,p_y+1)~=0)
50             possiblemoves(p_x,p_y+1) = 2;
51         elseif (piece_colour(p_x,p_y+1)== r_colour)
52             ;
53         else
54             possiblemoves(p_x,p_y+1) = 1;
55         end
56     end
57
58
59     if(p_y-1>0)
60         if (piece_colour(p_x,p_y-1)~= r_colour && chessboard(p_x,p_y-1)~=0)
61             possiblemoves(p_x,p_y-1) = 2;
62         elseif (piece_colour(p_x,p_y-1)== r_colour)
63             ;
64         else
65             possiblemoves(p_x,p_y-1) = 1;
66         end
67     end
68

```



```

69
70     if(p_x-1>0)
71         if (piece_colour(p_x-1,p_y)~= r_colour && chessboard(p_x-1,p_y)~=0)
72             possiblemoves(p_x-1,p_y) = 2;
73         elseif (piece_colour(p_x-1,p_y)== r_colour)
74             ;
75         else
76             possiblemoves(p_x-1,p_y) = 1;
77         end
78     end
79
80
81     if(p_x-1>0 && p_y+1<9)
82         if (piece_colour(p_x-1,p_y+1)~= r_colour && chessboard(p_x-1,p_y+1)~=0)
83             possiblemoves(p_x-1,p_y+1) = 2;
84         elseif (piece_colour(p_x-1,p_y+1)== r_colour)
85             ;
86         else
87             possiblemoves(p_x-1,p_y+1) = 1;
88         end
89     end
90
91
92     if(p_x-1>0 && p_y-1>0)
93         if (piece_colour(p_x-1,p_y-1)~= r_colour && chessboard(p_x-1,p_y-1)~=0)
94             possiblemoves(p_x-1,p_y-1) = 2;
95         elseif (piece_colour(p_x-1,p_y-1)== r_colour)
96             ;
97         else
98             possiblemoves(p_x-1,p_y-1) = 1;
99         end
100     end
101
102 %-----
103 %                               Castling
104 %-----
105
106
107 %----- For white king -----
108
109 possiblemoves(p_x,p_y)=0;
110 %-----
111 end

```

### A.1.6 MovementBishop.m

```

1 function [possiblemoves] = MovementBishop(chessboard,piece_colour,p_x,p_y)
2
3 %Initialisation values -----
4 r_colour = piece_colour(p_x,p_y);
5 possiblemoves = zeros(8,8);
6
7 %This section allows movement in / direction -----
8 i=1;
9 while(p_x+i<9 && p_y+i<9)
10     if(piece_colour(p_x+i,p_y+i)== r_colour)
11         break
12     end

```

```

13     if(piece_colour(p_x+i,p_y+i)~= r_colour && chessboard(p_x+i,p_y+i)~=0)
14         possiblemoves(p_x+i,p_y+i) = 2;
15         break
16     end
17     possiblemoves(p_x+i,p_y+i) = 1;
18     i = i+1;
19 end
20
21 i=1;
22 while(p_x-i>0 && p_y-i>0)
23     if(piece_colour(p_x-i,p_y-i)== r_colour)
24         break
25     end
26     if(piece_colour(p_x-i,p_y-i)~= r_colour && chessboard(p_x-i,p_y-i)~=0)
27         possiblemoves(p_x-i,p_y-i) = 2;
28         break
29     end
30     possiblemoves(p_x-i,p_y-i) = 1;
31     i = i+1;
32 end
33
34 %This section allows movement in the \ direction-----
35 i=1;
36 while(p_x+i<9 && p_y-i>0)
37     if(piece_colour(p_x+i,p_y-i)== r_colour)
38         break
39     end
40     if(piece_colour(p_x+i,p_y-i)~= r_colour && chessboard(p_x+i,p_y-i)~=0)
41         possiblemoves(p_x+i,p_y-i) = 2;
42         break
43     end
44     possiblemoves(p_x+i,p_y-i) = 1;
45     i = i+1;
46 end
47
48 i=1;
49 while(p_x-i>0 && p_y+i<9)
50     if(piece_colour(p_x-i,p_y+i)== r_colour)
51         break
52     end
53     if(piece_colour(p_x-i,p_y+i)~= r_colour && chessboard(p_x-i,p_y+i)~=0)
54         possiblemoves(p_x-i,p_y+i) = 2;
55         break
56     end
57     possiblemoves(p_x-i,p_y+i) = 1;
58     i = i+1;
59 end
60
61 %-----
62
63 end

```

## A.2 Front-end

### A.2.1 ClickPiece.m

```

1 %ClickPiece Obtains all the data from a user's click, highlights possible

```

```

2  %moves and allows the user to make that move.
3  function [varargout]=ClickPiece(var1,var2,B,piece_colour,chessboard,...
4      num_moves,parameters,potentialmoves,handles,varargin )
5
6  set(handles.gameconsole,'String','')
7
8  %-----Determines which colour is able to be selected-----
9  if(mod(B.info.turn,2)==1)
10     colourturn = 119;
11     oppositecolour = 98;
12 else
13     colourturn = 98;
14     oppositecolour = 119;
15 end
16
17 onlyAIOption = 0;
18 %-----
19 clickP = get(gca,'CurrentPoint');
20     x = ceil(clickP(1,2));
21     y = ceil(clickP(1,1));
22 %----- Conversion from Graph Grid to B.top grid -----
23     x = 13-x;
24     y = y + 4;
25 %-----
26 %This is the board
27     piecetype = B.top(x,y).name;
28
29 %-----Conversion from B.Top grid to Chessboard grid-----
30     p_x = x - 4;
31     p_y = y - 4;
32
33 if(piece_colour(p_x,p_y) == colourturn)
34 %-----Generates Possible Moves-----
35
36 switch piecetype
37     case 'pawn'
38         [possiblemoves] = MovementPawn(chessboard,piece_colour,num_moves,p_x,p_y);
39     case 'rook'
40         [possiblemoves] = MovementRook(chessboard,piece_colour,p_x,p_y);
41     case 'knight'
42         [possiblemoves] = MovementKnight(chessboard,piece_colour,p_x,p_y);
43     case 'bishop'
44         [possiblemoves] = MovementBishop(chessboard,piece_colour,p_x,p_y);
45     case 'queen'
46         [possiblemoves] = MovementQueen(chessboard,piece_colour,p_x,p_y);
47     case 'king'
48         [possiblemoves] = MovementKing(chessboard,piece_colour,num_moves,...
49             potentialmoves,p_x,p_y);
50 end
51
52 %-----
53 %             REDRAWS THE BOARD BUT HIGHLIGHTS POSSIBLE MOVES
54 %-----
55 %-----Draws Rectangles-----
56 icount=0;
57 for i=1:71
58     icount=icount+1;
59     if mod(i,2)==1
60         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
61             parameters.dx ,parameters.dx],'Curvature',[0,0],...
62             'FaceColor',[0.82 0.545 0.278])
63     else

```

```

64         rectangle('Position', [parameters.xx(icount), parameters.yy(icount)], ...
65                 parameters.dx , parameters.dx], ...
66                 'Curvature', [0,0], 'FaceColor', [1 0.808 0.62])
67     end
68 end
69
70 %----- Highlights possible moves-----
71 for r=1:parameters.rows
72     for c=1:parameters.cols
73         switch possiblemoves(r,c)
74 %----- Highlights movable squares-----
75         case 1
76             rectangle('Position', [parameters.xx(9-r,c), parameters.yy(9-r,c)], ...
77                     parameters.dx , parameters.dx], 'Curvature', [0,0], 'FaceColor', 'y', ...
78                     'ButtonDownFcn', {@ClickMovePiece, x, y, B, piece.colour, chessboard...
79                     , num_moves, parameters, possiblemoves, handles, onlyAIOption, 0, 0})
80 %----- Highlights capturable squares-----
81         case 2
82             rectangle('Position', [parameters.xx(9-r,c), parameters.yy(9-r,c)], ...
83                     parameters.dx , parameters.dx], 'Curvature', [0,0], 'FaceColor', 'r')
84 %----- Highlights Enpassant Squares-----
85         case 3
86             rectangle('Position', [parameters.xx(9-r,c), parameters.yy(9-r,c)], ...
87                     parameters.dx , parameters.dx], 'Curvature', [0,0], 'FaceColor', 'r', ...
88                     'ButtonDownFcn', {@ClickEnpassant, x, y, B, piece.colour, chessboard...
89                     , num_moves, parameters, possiblemoves, handles, onlyAIOption, 0, 0})
90 %----- Highlights Castling Squares-----
91         case 4
92             rectangle('Position', [parameters.xx(9-r,c), parameters.yy(9-r,c)], ...
93                     parameters.dx , parameters.dx], 'Curvature', [0,0], 'FaceColor', 'b', ...
94                     'ButtonDownFcn', {@ClickCastling, x, y, B, piece.colour, chessboard...
95                     , num_moves, parameters, possiblemoves, handles, onlyAIOption, 0, 0})
96 %----- Highlights Pawn Promotion Square-----
97         case 5
98             rectangle('Position', [parameters.xx(9-r,c), parameters.yy(9-r,c)], ...
99                     parameters.dx , parameters.dx], 'Curvature', [0,0], 'FaceColor', 'c', ...
100                     'ButtonDownFcn', {@ClickPawnPromo, x, y, B, piece.colour, chessboard...
101                     , num_moves, parameters, possiblemoves, handles, onlyAIOption, 0, 0, 0})
102         end
103     end
104 end
105 %-----
106 % Redraws images
107 %-----
108 for r=1:parameters.rows
109     for c=1:parameters.cols
110         if ~isempty(B.top(r+B.info.pad/2, c+B.info.pad/2).image)
111             % load the image
112             [X, map, alpha] = imread(B.top(r+B.info.pad/2, c+B.info.pad/2).image);
113             % draw the image
114             %If Statement enables capture move
115             if possiblemoves(r,c) == 2
116                 imHdls(r,c) = image(c+[0 1]-1, [parameters.rows-1 parameters.rows]-r+1, ...
117                                     mirrorImage(X), 'AlphaData', mirrorImage(alpha), ...
118                                     'ButtonDownFcn', {@ClickCapturePiece, x, y, B, piece.colour, chessboard...
119                                     , num_moves, parameters, possiblemoves, handles, onlyAIOption, 0, 0});
120                 %Enables Pawn Promotion
121             elseif possiblemoves(r,c) == 5 && chessboard(r,c)~=0
122                 imHdls(r,c) = image(c+[0 1]-1, [parameters.rows-1 parameters.rows]-r+1, ...
123                                     mirrorImage(X), 'AlphaData', mirrorImage(alpha), ...
124                                     'ButtonDownFcn', {@ClickPawnPromo, x, y, B, piece.colour, chessboard...
125                                     , num_moves, parameters, possiblemoves, handles, onlyAIOption, 0, 0});

```

```

126         %Else enable click piece
127     else
128         imHdls(r,c) = image(c+[0 1]-1,[parameters.rows-1 parameters.rows]-r+1,...
129             mirrorImage(X),'AlphaData',mirrorImage(alpha),...
130             'ButtonDownFcn',{@ClickPiece,B,piece_colour,chessboard,...
131                 num.moves,parameters,potentialmoves,handles,onlyAIOption,0,0});
132     end
133 end
134 end
135 end
136 drawnow;
137 end
138 end
139 %
140 %

```

### A.2.2 ClickCapturePiece.m

```

1 %CapturePiece Part of the Click Series of Functions – Enables capture
2 function [chessboard,piece_colour, num.moves,allowscheck]=ClickCapturePiece(v1,v2,x_ori,y_ori,
3     num.moves,parameters,PM,handles,onlyAIOption,move_x,move_y,varargin)
4
5 %
6 %             Init values,conversions and click location
7 %
8 if(mod(B.info.turn,2)==1)
9     colourturn = 119;
10    oppositecolour = 98;
11 else
12     colourturn = 98;
13     oppositecolour = 119;
14 end
15
16 if onlyAIOption == 0
17     clickP = get(gca,'CurrentPoint');
18     x = ceil(clickP(1,2));
19     y = ceil(clickP(1,1));
20 %----- Conversion from Graph grid to B.top grid -----
21     x = 13-x;
22     y = y + 4;
23 %-----Conversion from B.Top grid to Chessboard grid-----
24     p_x = x - 4; %p_x is necessary because it is the current clicked position
25     p_y = y - 4;
26     ori_x = x_ori - 4; %The difference is that ori_x is for chessboard,
27     ori_y = y_ori - 4; %x_ori is for B.top
28 else
29     p_x = move_x; %Where is it moving to
30     p_y = move_y;
31     ori_x = x_ori; %Where was it originally
32     ori_y = y_ori;
33 end
34
35 %
36 %             Checks if King is exposed to check in any way
37 %
38 %The method used is to create a future chessboard based on the move
39 %requested
40

```

---

```

41 fboard = chessboard;
42 f_p_colour= piece_colour;
43 f_num_moves = num_moves;
44 %This step officially moves the piece
45 fboard(p_x,p_y) = chessboard(ori_x,ori_y);
46 f_p_colour(p_x,p_y) = piece_colour(ori_x,ori_y);
47 f_num_moves(p_x,p_y) = num_moves(ori_x,ori_y) + 1;
48 %This step empties the previous box
49 fboard(ori_x,ori_y) = 0;
50 f_p_colour(ori_x,ori_y) = 0;
51 f_num_moves(ori_x,ori_y) = 0;
52
53 %Analyses the future board
54 [potentialfuturemoves,capt_index_future] = analyseboard(fboard,...
55     f_p_colour,f_num_moves,oppositecolour);
56 [allowscheck]=KingCheck(fboard,f_p_colour,colourturn,...
57     capt_index_future,potentialfuturemoves);
58 if allowscheck==1 && onlyAIoption == 0
59     set(handles.gameconsole,'String','King will be left in check, move invalid')
60 end
61 %
62 %Ensures it can only move legally
63 if PM(p_x,p_y)==2 && chessboard(p_x,p_y) ~= 10 && allowscheck==0
64
65 B.info.turn = B.info.turn + 1;
66 %
67 %           This is to edit the backend chessboard matrix
68 %
69 %This step officially moves the piece
70 chessboard = fboard;
71 piece_colour = f_p_colour;
72 num_moves = f_num_moves;
73
74
75 %-----To Check Opposing Side-----
76 [potentialmoves,capt_index] = analyseboard(chessboard,piece_colour,num_moves,colourturn);
77 [checkopp]=KingCheck(chessboard,piece_colour,oppositecolour,capt_index,potentialmoves);
78 if checkopp == 1 && onlyAIoption == 0
79     set(handles.checkstat,'String','Check')
80     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
81     if ischeckmate
82         set(handles.checkstat,'String','Checkmate, White Wins')
83     end
84 elseif checkopp == 0 && onlyAIoption ==0
85     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
86     if ischeckmate
87         set(handles.checkstat,'String','Stalemate')
88     else
89         set(handles.checkstat,'String','')
90     end
91 end
92
93 if onlyAIoption == 0
94     [B] = readchessboard(B,chessboard,piece_colour);
95 %
96 %           Redraws the Board
97 %
98 icount=0;
99 for i=1:71
100     icount=icount+1;
101     if mod(i,2)==1
102         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...

```

---

```

103         parameters.dx ,parameters.dx], 'Curvature',[0,0],...
104         'FaceColor',[0.82 0.545 0.278])
105     else
106         rectangle('Position',[parameters.xx(icount),parameters.yy(icount)],...
107         parameters.dx ,parameters.dx],...
108         'Curvature',[0,0], 'FaceColor',[1 0.808 0.62])
109     end
110 end
111
112 for r=1:parameters.rows
113     for c=1:parameters.cols
114         if ~isempty(B.top(r+B.info.pad/2,c+B.info.pad/2).image)
115             % load the image
116             [X, map, alpha] = imread(B.top(r+B.info.pad/2,c+B.info.pad/2).image);
117             % draw the image
118             imHdls(r,c) = image(c+[0 1]-1,[parameters.rows-1 parameters.rows]-r+1,...
119             mirrorImage(X), 'AlphaData',mirrorImage(alpha),...
120             'ButtonDownFcn',{@ClickPiece,B,piece_colour,chessboard,...
121             num_moves,parameters,potentialmoves,handles});
122         end
123     end
124 end
125 drawnow;
126 if (get(handles.choice2, 'Value')==1)
127     AIControl(B,piece_colour,chessboard,num_moves,parameters, handles);
128 end
129 if (get(handles.choice3, 'Value')==1)
130     AIvsAI(B,piece_colour,chessboard,num_moves,parameters, handles)
131 end
132 if (get(handles.choice1, 'Value')==1)
133     PlayerVsPlayer( B,piece_colour,chessboard,num_moves,parameters, handles )
134 end
135 end
136 %
137 end
138 end

```

### A.2.3 ClickCastling.m

```

1 %Castling Enables frontend implementation of castling
2 function [chessboard,piece_colour, num_moves,allowscheck]=ClickCastling(v1,v2,x_ori,y_ori,B,
3     num_moves,parameters,PM,handles,onlyAIOption,move_x,move_y,varargin)
4
5 %
6 %             Init values,conversions and click location
7 %
8 if(mod(B.info.turn,2)==1)
9     colourturn = 119;
10    oppositecolour = 98;
11 else
12     colourturn = 98;
13     oppositecolour = 119;
14 end
15
16 if onlyAIOption == 0
17     clickP = get(gca, 'CurrentPoint');
18     x = ceil(clickP(1,2));
19     y = ceil(clickP(1,1));

```

```

20 %—— Conversion from Graph grid to B.top grid ——
21     x = 13-x;
22     y = y + 4;
23 %—— Conversion from B.Top grid to Chessboard grid——
24     p_x = x - 4; %p_x is necessary because it is the current clicked position
25     p_y = y - 4;
26     ori_x = x_ori - 4; %The difference is that ori_x is for chessboard,
27     ori_y = y_ori - 4; %x_ori is for B.top
28 else
29     p_x = move_x; %Where is it moving to
30     p_y = move_y;
31     ori_x = x_ori; %Where was it originally
32     ori_y = y_ori;
33 end
34
35 %
36 %             Checks if King is exposed to check in any way
37 %
38 %The method used is to create a future chessboard based on the move
39 %requested
40
41 fboard = chessboard;
42 f_p_colour= piece_colour;
43 f_num_moves = num_moves;
44 %This step officially moves the piece
45 fboard(p_x,p_y) = chessboard(ori_x,ori_y);
46 f_p_colour(p_x,p_y) = piece_colour(ori_x,ori_y);
47 f_num_moves(p_x,p_y) = num_moves(ori_x,ori_y) + 1;
48 %This step empties the previous box
49 fboard(ori_x,ori_y) = 0;
50 f_p_colour(ori_x,ori_y) = 0;
51 f_num_moves(ori_x,ori_y) = 0;
52
53 %Analyses the future board
54 [potentialfuturemoves,capt_index_future] = analyseboard(fboard,...
55     f_p_colour,f_num_moves,oppositecolour);
56 [allowscheck]=KingCheck(fboard,f_p_colour,colourturn,...
57     capt_index_future,potentialfuturemoves);
58 if allowscheck ==1 && onlyAIoption == 0
59     set(handles.gameconsole,'String','King will be left in check, move invalid')
60 end
61 %
62 %Ensures it can only move legally
63 if PM(p_x,p_y)==4 && allowscheck==0
64
65 %
66 %             B.top
67 %
68 %Coordinate system is X_rook = [B.top Chessboard]
69 if ( p_x == 8 && p_y == 7)
70     x_rook = [12 8]; %Initial Rook Position
71     y_rook = [12 8];
72     move_x = [12 8]; %Final Rook Position
73     move_y = [10 6];
74 elseif ( p_x == 8 && p_y == 3 )
75     x_rook = [12 8];
76     y_rook = [5 1];
77     move_x = [12 8];
78     move_y = [8 4];
79 elseif ( p_x == 1 && p_y == 7)
80     x_rook = [5 1];
81     y_rook = [12 8];

```



```

82     move_x = [5 1];
83     move_y = [10 6];
84 elseif ( p_x == 1 && p_y == 3)
85     x_rook = [5 1];
86     y_rook = [5 1];
87     move_x = [5 1];
88     move_y = [8 4];
89 end
90
91 B.info.turn = B.info.turn + 1;
92 %
93 %             This is to edit the backend chessboard matrix
94 %
95 %-----King-----
96 %This step officially moves the piece
97 chessboard(p_x,p_y) = chessboard(ori_x,ori_y);
98 piece_colour(p_x,p_y) = piece_colour(ori_x,ori_y);
99 num_moves(p_x,p_y) = num_moves(ori_x,ori_y) + 1;
100
101 %This step empties the previous box
102 chessboard(ori_x,ori_y) = 0;
103 piece_colour(ori_x,ori_y) = 0;
104 num_moves(ori_x,ori_y) = 0;
105
106 %-----Rook-----
107 %This step officially moves the piece
108 chessboard(move_x(2),move_y(2)) = chessboard(x_rook(2),y_rook(2));
109 piece_colour(move_x(2),move_y(2)) = piece_colour(x_rook(2),y_rook(2));
110 num_moves(move_x(2),move_y(2)) = num_moves(x_rook(2),y_rook(2)) + 1;
111
112 %This step empties the previous box
113 chessboard(x_rook(2),y_rook(2)) = 0;
114 piece_colour(x_rook(2),y_rook(2)) = 0;
115 num_moves(x_rook(2),y_rook(2)) = 0;
116
117 %-----Analyses for potential checks & provides game stats-----
118 [potentialmoves,capt_index] = analyseboard(chessboard,piece_colour,num_moves,colourturn);
119 [checkopp]=KingCheck(chessboard,piece_colour,oppositecolour,capt_index,potentialmoves);
120 if checkopp == 1 && onlyAIoption == 0
121     set(handles.checkstat,'String','Check')
122     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
123     if ischeckmate
124         set(handles.checkstat,'String','Checkmate, White Wins')
125     end
126 elseif checkopp == 0 && onlyAIoption ==0
127     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
128     if ischeckmate
129         set(handles.checkstat,'String','Stalemate')
130     else
131         set(handles.checkstat,'String','')
132     end
133 end
134
135 if onlyAIoption ==0
136     [B] = readchessboard(B,chessboard,piece_colour);
137 %
138 %             Redraws the Board
139 %
140 icount=0;
141 for i=1:71
142     icount=icount+1;
143     if mod(i,2)==1

```

```

144         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
145             parameters.dx ,parameters.dx], 'Curvature',[0,0],...
146             'FaceColor',[0.82 0.545 0.278])
147     else
148         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
149             parameters.dx ,parameters.dx],...
150             'Curvature',[0,0], 'FaceColor',[1 0.808 0.62])
151     end
152 end
153
154 for r=1:parameters.rows
155     for c=1:parameters.cols
156         if ~isempty(B.top(r+B.info.pad/2,c+B.info.pad/2).image)
157             % load the image
158             [X, map, alpha] = imread(B.top(r+B.info.pad/2,c+B.info.pad/2).image);
159             % draw the image
160             imHdls(r,c) = image(c+[0 1]-1,[parameters.rows-1 parameters.rows]-r+1,...
161                 mirrorImage(X), 'AlphaData',mirrorImage(alpha),...
162                 'ButtonDownFcn',{@ClickPiece,B,piece_colour,chessboard,...
163                 num_moves,parameters,potentialmoves,handles});
164         end
165     end
166 end
167 drawnow;
168 if (get(handles.choice2, 'Value')==1)
169     AIControl(B,piece_colour,chessboard,num_moves,parameters, handles);
170 end
171 if (get(handles.choice3, 'Value')==1)
172     AIVsAI(B,piece_colour,chessboard,num_moves,parameters, handles)
173 end
174 if (get(handles.choice1, 'Value')==1)
175     PlayerVsPlayer( B,piece_colour,chessboard,num_moves,parameters, handles )
176 end
177 end
178 %-----
179 end
180 end

```

#### A.2.4 ClickEnpassant.m

```

1  %Enpassant Enables frontend implementation of En Passant
2  function [chessboard,piece_colour, num_moves,allowscheck]=ClickEnpassant(v1,v2,x_ori,y_ori,B
3      num_moves,parameters,PM, handles,onlyAIOption,move_x,move_y,varargin)
4
5  %-----
6  %             Init values,conversions and click location
7  %-----
8  if(mod(B.info.turn,2)==1)
9      colourturn = 119;
10     oppositecolour = 98;
11 else
12     colourturn = 98;
13     oppositecolour = 119;
14 end
15
16 if onlyAIOption == 0
17     clickP = get(gca, 'CurrentPoint');
18     x = ceil(clickP(1,2));

```

```

19     y = ceil(clickP(1,1));
20 %----- Conversion from Graph grid to B.top grid -----
21     x = 13-x;
22     y = y + 4;
23 %----- Conversion from B.Top grid to Chessboard grid -----
24     p_x = x - 4; %p_x is necessary because it is the current clicked position
25     p_y = y - 4;
26     ori_x = x_ori - 4; %The difference is that ori_x is for chessboard,
27     ori_y = y_ori - 4; %x_ori is for B.top
28 else
29     p_x = move_x;    %Where is it moving to
30     p_y = move_y;
31     ori_x = x_ori;    %Where was it originally
32     ori_y = y_ori;
33 end
34
35 %-----
36 %           Checks if King is exposed to check in any way due to move
37 %-----
38 %The method used is to create a future chessboard based on the move
39 %requested
40
41 fboard = chessboard;
42 f_p_colour= piece_colour;
43 f_num_moves = num_moves;
44 %This step officially moves the piece
45 fboard(p_x,p_y) = chessboard(ori_x,ori_y);
46 f_p_colour(p_x,p_y) = piece_colour(ori_x,ori_y);
47 f_num_moves(p_x,p_y) = num_moves(ori_x,ori_y) + 1;
48 %This step empties the previous box
49 fboard(ori_x,ori_y) = 0;
50 f_p_colour(ori_x,ori_y) = 0;
51 f_num_moves(ori_x,ori_y) = 0;
52
53 %Analyses the future board
54 [potentialfuturemoves,capt_index_future] = analyseboard(fboard,...
55     f_p_colour,f_num_moves,oppositecolour);
56 [allowscheck]=KingCheck(fboard,f_p_colour,colourturn,...
57     capt_index_future,potentialfuturemoves);
58 if allowscheck==1 && onlyAIoption == 0
59     set(handles.gameconsole,'String','King will be left in check, move invalid')
60 end
61 %-----
62 %Ensures it can only move legally
63 if PM(p_x,p_y)==3 && allowscheck==0
64 %-----
65 %           Moves Data in B.TOP & deletes previous cell
66 %-----
67 %Coordinates of the captured piece
68 if (piece_colour(ori_x,ori_y)==98)
69     del_x = [p_x+3 p_x-1];
70     del_y = [p_y+4 p_y];
71 end
72
73 if (piece_colour(ori_x,ori_y)==119)
74     del_x = [p_x+5 p_x+1];
75     del_y = [p_y+4 p_y];
76 end
77
78 B.info.turn = B.info.turn + 1;
79 %-----
80 %           This is to edit the backend chessboard matrix

```

```

81 %-----
82 %This step officially moves the piece
83 chessboard(p_x,p_y) = chessboard(ori_x,ori_y);
84 piece_colour(p_x,p_y) = piece_colour(ori_x,ori_y);
85 num_moves(p_x,p_y) = num_moves(ori_x,ori_y) + 1;
86
87 %This step empties the previous box
88 chessboard(ori_x,ori_y) = 0;
89 piece_colour(ori_x,ori_y) = 0;
90 num_moves(ori_x,ori_y) = 0;
91
92 %This step deletes the captured piece
93 chessboard(del_x(2),del_y(2)) = 0;
94 piece_colour(del_x(2),del_y(2)) = 0;
95 num_moves(del_x(2),del_y(2)) = 0;
96
97
98 %-----Analyses for potential checks & provides game stats-----
99 [potentialmoves,capt_index] = analyseboard(chessboard,piece_colour,num_moves,colourturn);
100 [checkopp]=KingCheck(chessboard,piece_colour,oppositecolour,capt_index,potentialmoves);
101 if checkopp == 1 && onlyAIOption == 0
102     set(handles.checkstat,'String','Check')
103     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
104     if ischeckmate
105         set(handles.checkstat,'String','Checkmate, White Wins')
106     end
107 elseif checkopp == 0 && onlyAIOption ==0
108     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
109     if ischeckmate
110         set(handles.checkstat,'String','Stalemate')
111     else
112         set(handles.checkstat,'String','')
113     end
114 end
115
116 if onlyAIOption == 0
117     [B] = readchessboard(B,chessboard,piece_colour);
118 %-----
119 %                                     Redraws the Board
120 %-----
121 icount=0;
122 for i=1:71
123     icount=icount+1;
124     if mod(i,2)==1
125         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
126             parameters.dx ,parameters.dx],'Curvature',[0,0],...
127             'FaceColor',[0.82 0.545 0.278])
128     else
129         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
130             parameters.dx ,parameters.dx],...
131             'Curvature',[0,0],'FaceColor',[1 0.808 0.62])
132     end
133 end
134 end
135 end

```

### A.2.5 ClickPawnPromo.m

```

1  %PawnPromo Enables Front End Implementation of Pawn Promo
2  function [chessboard,piece_colour, num_moves,allowscheck]=ClickPawnPromo(v1,v2,x_ori,y_ori,B
3      num_moves,parameters,PM,handles,onlyAIOption,move_x,move_y,promo,varargin)
4
5  %-----
6  %                               Init values,conversions and click location
7  %-----
8  if(mod(B.info.turn,2)==1)
9      colourturn = 119;
10     oppositecolour = 98;
11 else
12     colourturn = 98;
13     oppositecolour = 119;
14 end
15
16 if onlyAIOption == 0
17 clickP = get(gca,'CurrentPoint');
18     x = ceil(clickP(1,2));
19     y = ceil(clickP(1,1));
20 %----- Conversion from Graph grid to B.top grid -----
21     x = 13-x;
22     y = y + 4;
23 %----- Conversion from B.Top grid to Chessboard grid -----
24     p_x = x - 4; %p_x is necessary because it is the current clicked position
25     p_y = y - 4;
26     ori_x = x_ori - 4; %The difference is that ori_x is for chessboard,
27     ori_y = y_ori - 4; %x_ori is for B.top
28 else
29     p_x = move_x;    %Where is it moving to
30     p_y = move_y;
31     ori_x = x_ori;    %Where was it originally
32     ori_y = y_ori;
33 end
34
35 %-----
36 %                               Checks if King is exposed to check in any way
37 %-----
38 %The method used is to create a future chessboard based on the move
39 %requested
40
41 fboard = chessboard;
42 f_p_colour= piece_colour;
43 f_num_moves = num_moves;
44 %This step officially moves the piece
45 fboard(p_x,p_y) = chessboard(ori_x,ori_y);
46 f_p_colour(p_x,p_y) = piece_colour(ori_x,ori_y);
47 f_num_moves(p_x,p_y) = num_moves(ori_x,ori_y) + 1;
48 %This step empties the previous box
49 fboard(ori_x,ori_y) = 0;
50 f_p_colour(ori_x,ori_y) = 0;
51 f_num_moves(ori_x,ori_y) = 0;
52
53 %Analyses the future board
54 [potentialfuturemoves,capt_index_future] = analyseboard(fboard,...
55     f_p_colour,f_num_moves,oppositecolour);
56 [allowscheck]=KingCheck(fboard,f_p_colour,colourturn,...
57     capt_index_future,potentialfuturemoves);
58 if allowscheck==1 && onlyAIOption == 0
59     set(handles.gameconsole,'String','King will be left in check, move invalid')
60 end
61 %-----
62 %Ensures it can only move legally

```

---

```

63 if PM(p_x,p-y)==5 && allowscheck==0
64 %
65 %             Moves Data in B.TOP & deletes previous cell
66 %
67 %Allows user to input desired piece. Checks legality.
68 if ~onlyAIOption
69 set(handles.gameconsole, 'String', 'Pawn has been promoted');
70     flags=0;
71     while(flags==0)
72         flags=1;
73         v=0;
74         while v == 0
75             [pawn_prom,v] = listdlg('PromptString','Select a piece:',...
76                 'SelectionMode','single',...
77                 'ListString',{'Rook','Queen','Knight','Bishop'});
78         end
79         switch pawn_prom
80             case 1
81                 chessboard(p_x,p-y)= 5;
82             case 2
83                 chessboard(p_x,p-y)= 9;
84             case 3
85                 chessboard(p_x,p-y)= 3;
86             case 4
87                 chessboard(p_x,p-y)= 4;
88             otherwise
89                 disp('Invalid input');
90                 flags=0;
91         end
92     end
93 else
94         switch promo
95             case 'rook'
96                 chessboard(p_x,p-y)= 5;
97             case 'queen'
98                 chessboard(p_x,p-y)= 9;
99             case 'knight'
100                 chessboard(p_x,p-y)= 3;
101             case 'bishop'
102                 chessboard(p_x,p-y)= 4;
103         end
104 end
105
106 B.info.turn = B.info.turn + 1;
107
108 %
109 %             This is to edit the backend chessboard matrix
110 %
111 %This step officially moves the piece
112 num_moves(p_x,p-y) = num_moves(ori_x,ori_y) + 1;
113 piece_colour(p_x,p-y)= colourturn;
114
115 %This step empties the previous box
116 chessboard(ori_x,ori_y) = 0;
117 piece_colour(ori_x,ori_y) = 0;
118 num_moves(ori_x,ori_y) = 0;
119
120 %-----Analyses for potential checks & provides game stats-----
121 [potentialmoves,capt_index] = analyseboard(chessboard,piece_colour,num_moves,colourturn);
122 [checkopp]=KingCheck(chessboard,piece_colour,oppositecolour,capt_index,potentialmoves);
123 if checkopp == 1 && onlyAIOption == 0
124     set(handles.checkstat, 'String', 'Check')

```

---

```

125     [ischeckmate]=checkmate(B,chessboard,piece-colour, num-moves);
126     if ischeckmate
127         set(handles.checkstat,'String','Checkmate, White Wins')
128     end
129 elseif checkopp == 0 && onlyAIOption ==0
130     [ischeckmate]=checkmate(B,chessboard,piece-colour, num-moves);
131     if ischeckmate
132         set(handles.checkstat,'String','Stalemate')
133     else
134         set(handles.checkstat,'String','')
135     end
136 end
137
138 if onlyAIOption == 0
139     [B] = readchessboard(B,chessboard,piece-colour);
140 %-----
141 %                               Redraws the Board
142 %-----
143 icount=0;
144 for i=1:71
145     icount=icount+1;
146     if mod(i,2)==1
147         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
148             parameters.dx ,parameters.dx],'Curvature',[0,0],...
149             'FaceColor',[0.82 0.545 0.278])
150     else
151         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
152             parameters.dx ,parameters.dx],...
153             'Curvature',[0,0],'FaceColor',[1 0.808 0.62])
154     end
155 end
156
157 for r=1:parameters.rows
158     for c=1:parameters.cols
159         if ~isempty(B.top(r+B.info.pad/2,c+B.info.pad/2).image)
160             % load the image
161             [X, map, alpha] = imread(B.top(r+B.info.pad/2,c+B.info.pad/2).image);
162             % draw the image
163 end

```

### A.2.6 ClickMovePiece.m

```

1  %Movepiece Part of the Click Series of Functions – Enables movement
2  function [chessboard,piece-colour, num-moves,allowscheck]=ClickMovePiece(v1,v2,x-ori,y-ori,B
3      num-moves,parameters,PM,handles,onlyAIOption,move-x,move-y,varargin)
4
5  %-----
6  %                               Init values,conversions and click location
7  %-----
8  if(mod(B.info.turn,2)==1)
9      colourturn = 119;
10     oppositecolour = 98;
11 else
12     colourturn = 98;
13     oppositecolour = 119;
14 end
15
16 if onlyAIOption == 0

```

```

17 clickP = get(gca, 'CurrentPoint');
18     x = ceil(clickP(1,2));
19     y = ceil(clickP(1,1));
20 %----- Conversion from Graph grid to B.top grid -----
21     x = 13-x;
22     y = y + 4;
23 %----- Conversion from B.Top grid to Chessboard grid -----
24     p_x = x - 4; %p_x is necessary because it is the current clicked position
25     p_y = y - 4;
26     ori_x = x_ori - 4; %The difference is that ori_x is for chessboard,
27     ori_y = y_ori - 4; %x_ori is for B.top
28 else
29     p_x = move_x;    %Where is it moving to
30     p_y = move_y;
31     ori_x = x_ori;    %Where was it originally
32     ori_y = y_ori;
33 end
34
35 %-----
36 %             Checks if King is exposed to check in any way
37 %-----
38 %The method used is to create a future chessboard based on the move
39 %requested
40
41 fboard = chessboard;
42 f_p_colour= piece_colour;
43 f_num_moves = num_moves;
44 %This step officially moves the piece
45 fboard(p_x,p_y) = chessboard(ori_x,ori_y);
46 f_p_colour(p_x,p_y) = piece_colour(ori_x,ori_y);
47 f_num_moves(p_x,p_y) = num_moves(ori_x,ori_y) + 1;
48 %This step empties the previous box
49 fboard(ori_x,ori_y) = 0;
50 f_p_colour(ori_x,ori_y) = 0;
51 f_num_moves(ori_x,ori_y) = 0;
52
53 %Analyses the future board
54 [potentialfuturemoves,capt_index_future] = analyseboard(fboard,...
55     f_p_colour,f_num_moves,oppositecolour);
56 [allowscheck]=KingCheck(fboard,f_p_colour,colourturn,...
57     capt_index_future,potentialfuturemoves);
58 if allowscheck ==1 && onlyAIoption == 0
59     set(handles.gameconsole,'String','King will be left in check, move invalid')
60 end
61 %-----
62 %-----
63
64 if PM(p_x,p_y)==1 && allowscheck==0 %Ensures it can only move legally
65
66 %Iterates the turn
67 B.info.turn = B.info.turn + 1;
68
69 %-----
70 %             This is to edit the backend chessboard matrix
71 %-----
72 %This step officially moves the piece
73 chessboard = fboard;
74 piece_colour = f_p_colour;
75 num_moves = f_num_moves;
76
77 %-----Analyses for potential checks & provides game stats-----
78 [potentialmoves,capt_index] = analyseboard(chessboard,piece_colour,num_moves,colourturn);

```



```

79 [checkopp]=KingCheck(chessboard,piece_colour,oppositecolour,capt_index,potentialmoves);
80
81 if checkopp == 1 && onlyAIOption ==0
82     set(handles.checkstat,'String','Check')
83     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
84     if ischeckmate
85         set(handles.checkstat,'String','Checkmate, White Wins')
86     end
87 elseif checkopp == 0 && onlyAIOption ==0
88     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
89     if ischeckmate
90         set(handles.checkstat,'String','Stalemate')
91     else
92         set(handles.checkstat,'String','')
93     end
94 end
95
96 %-----
97 if onlyAIOption == 0
98     [B] = readchessboard(B,chessboard,piece_colour);
99 %-----
100 %                               Redraws the Board
101 %-----
102 icount=0;
103 for i=1:71
104     icount=icount+1;
105     if mod(i,2)==1
106         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
107             parameters.dx ,parameters.dx],'Curvature',[0,0],...
108             'FaceColor',[0.82 0.545 0.278])
109     else
110         rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
111             parameters.dx ,parameters.dx],...
112             'Curvature',[0,0],'FaceColor',[1 0.808 0.62])
113     end
114 end
115
116 %-----
117 for r=1:parameters.rows
118     for c=1:parameters.cols
119         if ~isempty(B.top(r+B.info.pad/2,c+B.info.pad/2).image)
120             % load the image
121             [X, map, alpha] = imread(B.top(r+B.info.pad/2,c+B.info.pad/2).image);
122             % draw the image
123             imHdls(r,c) =image(c+[0 1]-1,[parameters.rows-1 parameters.rows]-r+1,...
124                 mirrorImage(X),'AlphaData',mirrorImage(alpha),...
125                 'ButtonDownFcn',{@ClickPiece,B,piece_colour,chessboard,...
126                     num_moves,parameters,potentialmoves,handles});
127         end
128     end
129 end
130 drawnow;
131 if(get(handles.choice2,'Value')==1)
132     AIControl(B,piece_colour,chessboard,num_moves,parameters, handles);
133 end
134 if(get(handles.choice3,'Value')==1)
135     AIVsAI(B,piece_colour,chessboard,num_moves,parameters, handles)
136 end
137 if(get(handles.choice1,'Value')==1)
138     PlayerVsPlayer( B,piece_colour,chessboard,num_moves,parameters, handles )
139 end
140 end

```

```

141 %
142 end
143 end

```

## A.3 AI

### A.3.1 AIControl

```

1  function [B,piece_colour,chessboard,num_moves,parameters, handles]=AIControl(B,piece_colour,
2      num_moves,parameters, handles)
3  %AIControl Enables AI to be in action
4
5  %-----
6  %                               Init Values
7  %-----
8  if(mod(B.info.turn-1,2)==1)
9      colourturn = 119;
10     oppositecolour = 98;
11 else
12     colourturn = 98;
13     oppositecolour = 119;
14 end
15
16 [userboardscore] = heuristicanalysis(B,chessboard, piece_colour,num_moves,119,handles);
17 set(handles.UPS,'String',userboardscore)
18 handles.userboardscore = [handles.userboardscore userboardscore];
19 depth = 2;
20 set(handles.depth,'String',depth)
21
22 %----- Stops Game Execution if White Wins -----
23 % [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
24 % if ischeckmate
25 %     return
26 % end
27
28 [oppcolourpotentialmoves,oppcolourcapt.index] = analyseboard(chessboard, piece_colour,num_moves);
29
30 [ischeck]=KingCheck(chessboard,piece_colour,oppositecolour, oppcolourcapt.index,oppcolourpotentialmoves);
31 if ischeck == 1
32     set(handles.checkstat,'String','Check')
33     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
34     if ischeckmate
35         set(handles.checkstat,'String','Checkmate, White Wins')
36     end
37 elseif ischeck == 0
38     [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
39     if ischeckmate
40         set(handles.checkstat,'String','Stalemate')
41     else
42         set(handles.checkstat,'String','')
43     end
44 end
45 %-----Plot UserBoardScore-----
46
47 handles.turnforwhite = [handles.turnforwhite B.info.turn];
48 plot(handles.graph,handles.turnforwhite,handles.userboardscore,'-b',...
49     handles.turnforblack,handles.AIBoardscore,'-r','LineWidth',2)

```

---

```

50  set(handles.graph,'XColor','w','YColor','w')
51  xlabel(handles.graph,'Turn')
52  ylabel(handles.graph,'Score')
53
54  %-----
55  set(handles.AIMsgs,'String','Thinking Really Hard')
56
57  %Produces AI's decision
58  tic
59  [boardscore,chessboard,piece_colour,num_moves]=...
60      AI.GenerateAllMoves(B,chessboard,piece_colour,num_moves,depth,1,-99999,99999,handles);
61  time =toc;
62
63  set(handles.AIMsgs,'String',['Time Taken To Think Was: ' num2str(time) ' seconds'])
64
65  %Translates the results into B.top
66  [B] = readchessboard(B,chessboard,piece_colour);
67  %Iterates turn
68  B.info.turn = B.info.turn + 1;
69
70  %----- Shows AI Board Score-----
71  [AIBoardScore] = heuristicaanalysis(B,chessboard, piece_colour,num_moves,98,handles);
72  set(handles.APS,'String',AIBoardScore)
73  handles.AIBoardscore = [handles.AIBoardscore AIBoardScore];
74
75  %-----Plots AI Board Score-----
76
77  handles.turnforblack = [handles.turnforblack B.info.turn];
78  plot(handles.graph,handles.turnforwhite,handles.userboardscore,'-b',...
79      handles.turnforblack,handles.AIBoardscore,'-r','LineWidth',2)
80  set(handles.graph,'XColor','w','YColor','w')
81  xlabel(handles.graph,'Turn')
82  ylabel(handles.graph,'Score')
83
84  %----- Checks if AI has checkmated User -----
85  [oppcolourpotentialmoves,oppcolourcapt_index] = analyseboard(chessboard, piece_colour,num_moves);
86
87  [ischeck]=KingCheck(chessboard,piece_colour,colourturn, oppcolourcapt_index,oppcolourpotentialmoves);
88  if ischeck == 1
89      set(handles.checkstat,'String','Check')
90      [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
91      if ischeckmate
92          set(handles.checkstat,'String','Checkmate, Black Wins')
93      end
94  elseif ischeck == 0
95      [ischeckmate]=checkmate(B,chessboard,piece_colour, num_moves);
96      if ischeckmate
97          set(handles.checkstat,'String','Stalemate')
98      else
99          set(handles.checkstat,'String','')
100      end
101  end
102  %-----
103
104  %-----
105  % Redraws the Board
106  %-----
107  icount=0;
108  for i=1:71
109      icount=icount+1;
110      if mod(i,2)==1
111          rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...

```

---

```

112             parameters.dx ,parameters.dx], 'Curvature',[0,0],...
113             'FaceColor',[0.82 0.545 0.278])
114         else
115             rectangle('Position',[parameters.xx(icount),parameters.yy(icount),...
116             parameters.dx ,parameters.dx],...
117             'Curvature',[0,0], 'FaceColor',[1 0.808 0.62])
118         end
119     end
120
121     %-----
122     for r=1:parameters.rows
123         for c=1:parameters.cols
124             if ~isempty(B.top(r+B.info.pad/2,c+B.info.pad/2).image)
125                 % load the image
126                 [X, map, alpha] = imread(B.top(r+B.info.pad/2,c+B.info.pad/2).image);
127                 % draw the image
128                 imHdls(r,c) = image(c+[0 1]-1,[parameters.rows-1 parameters.rows]-r+1,...
129                 mirrorImage(X), 'AlphaData',mirrorImage(alpha),...
130                 'ButtonDownFcn',{@ClickPiece,B,piece.colour,chessboard,...
131                 num_moves,parameters,oppcolourpotentialmoves,handles});
132             end
133         end
134     end
135     drawnow;
136     %-----
137     end

```

### A.3.2 AI\_GenerateAllMoves.m

```

1  %AI - Generates moves and stores them for 1 PLY (Only for DATA Tree)
2  function [boardscore,bchessboard,bpiece.colour,bnum.moves,handles]=...
3      AI_GenerateAllMoves(B,chessboard,piece.colour,num.moves,depth,maxormin,alpha,beta,handles)
4  %-----
5  %                               Init Values
6  %-----
7  TmpB = B;
8
9  if(mod(TmpB.info.turn,2)==1)
10     colour = 119;
11     oppcolour = 98;
12 else
13     colour = 98;
14     oppcolour = 119;
15 end
16
17 TmpB.info.turn = TmpB.info.turn +1;
18 %-----
19
20 if depth == 0
21     TmpB.info.turn = TmpB.info.turn-1;
22     [boardscore] = heuristicalysis(TmpB,chessboard, piece.colour,num.moves,colour,handles);
23     bchessboard = chessboard;
24     bpiece.colour = piece.colour;
25     bnum.moves = num.moves;
26 else
27
28     if maxormin == 1 %Maximizing Player
29         %===== Generates Future Nodes or Leafs =====

```

---

```

30 %
31 %      Loop that generates all possible moves
32 %
33 [p_x,p_y] = find(piece_colour == colour);
34 perm_index = randperm(length(p_x));
35 p_x = p_x(perm_index);
36 p_y = p_y(perm_index);
37 n_remaining = length(p_x);
38 [potentialmoves] = analyseboard(chessboard, piece_colour,num_moves,oppcolour);
39 previousboardscore = -99999;
40 %In essence, we are going through each piece, looking at it's possible
41 %moves, make those possible moves, evaluate, save bestboard.
42 for i=1:n_remaining
43     p_type = chessboard(p_x(i),p_y(i));
44     switch p_type
45         case 1
46             [move] = MovementPawn(chessboard,piece_colour,num_moves,p_x(i),p_y(i));
47         case 5
48             [move] = MovementRook(chessboard,piece_colour,p_x(i),p_y(i));
49         case 4
50             [move] = MovementBishop(chessboard,piece_colour,p_x(i),p_y(i));
51         case 3
52             [move] = MovementKnight(chessboard,piece_colour,p_x(i),p_y(i));
53         case 9
54             [move] = MovementQueen(chessboard,piece_colour,p_x(i),p_y(i));
55         case 10
56             [move] = MovementKing(chessboard,piece_colour,num_moves,potentialmoves,p_x(i),p_y(i));
57     end
58
59 %
60 %      Individual Piece Moves That Generate New Game States
61 %      Recursion is also added in each loop
62 %
63     [move_x,move_y] = find(move ~= 0);
64     perm_index2 = randperm(length(move_x));
65     move_x = move_x(perm_index2);
66     move_y = move_y(perm_index2);
67     n_move = length(move_x);
68     pruneflag = 0;
69 %This loop generates all the game states from 1 piece
70     for j = 1:n_move
71         switch move(move_x(j),move_y(j))
72             case 1
73                 [pchessboard, ppiece_colour, pnum_moves,kingincheck]=ClickMovePiece(0,0,p_x(i),p_y(i),
74                     num_moves,0,move,0,1,move_x(j),move_y(j));
75             case 2
76                 [pchessboard, ppiece_colour, pnum_moves,kingincheck]=ClickCapturePiece(0,0,p_x(i),p_y(i),
77                     num_moves,0,move,0,1,move_x(j),move_y(j));
78             case 3
79                 [pchessboard, ppiece_colour, pnum_moves,kingincheck]=ClickEnpassant(0,0,p_x(i),p_y(i),
80                     num_moves,0,move,0,1,move_x(j),move_y(j));
81             case 4
82                 [pchessboard, ppiece_colour, pnum_moves,kingincheck]=ClickCastling(0,0,p_x(i),p_y(i),
83                     num_moves,0,move,0,1,move_x(j),move_y(j));
84             case 5
85                 [pchessboard,ppiece_colour, pnum_moves,kingincheck]=ClickPawnPromo(0,0,p_x(i),p_y(i),
86                     num_moves,0,move,0,1,move_x(j),move_y(j),'queen');
87         end
88 %-----A node has been generated, what do you want to do with it?-----
89         if kingincheck
90             %ignore because move not valid
91             if ~exist('boardscore','var')

```

---

```

92         boardscore = -99999;
93         bchessboard = 0;
94         bpiece.colour = 0;
95         bnum_moves = 0;
96     end
97 else
98     %Generate another layer with recursive parameters
99     [boardscore,~,~,~,handles]=...
100     AI_GenerateAllMoves(TmpB,pchessboard,ppiece.colour,pnum_moves,depth-1,-maxormin,handles);
101
102     if boardscore > previousboardscore
103         previousboardscore = boardscore;
104         bchessboard = pchessboard;
105         bpiece.colour = ppiece.colour;
106         bnum_moves = pnum_moves;
107     end
108     if boardscore>alpha
109         alpha = boardscore;
110     end
111     % disp([depth alpha beta boardscore previousboardscore i j n_remaining n_move])
112     if alpha>beta
113         pruneflag = 1;
114         break
115     end
116 end
117 %-----
118 end
119 if pruneflag
120     break
121 end
122 end
123 %=====
124
125
126 elseif maxormin == -1 %Minimizing Player
127 %===== Generates Future Nodes or Leafs =====
128 %-----
129 % Loop that generates all possible moves
130 %-----
131 [p_x,p_y] = find(piece.colour == colour);
132 perm_index = randperm(length(p_x));
133 p_x = p_x(perm_index);
134 p_y = p_y(perm_index);
135 n_remaining = length(p_x);
136 [potentialmoves] = analyseboard(chessboard, piece.colour,num_moves,oppcolour);
137 previousboardscore = 99999;
138 %In essence, we are going through each piece, looking at it's possible
139 %moves, make those possible moves, evaluate, save bestboard.
140 for i=1:n_remaining
141     p_type = chessboard(p_x(i),p_y(i));
142     switch p_type
143     case 1
144         [move] = MovementPawn(chessboard,piece.colour,num_moves,p_x(i),p_y(i));
145     case 5
146         [move] = MovementRook(chessboard,piece.colour,p_x(i),p_y(i));
147     case 4
148         [move] = MovementBishop(chessboard,piece.colour,p_x(i),p_y(i));
149     case 3
150         [move] = MovementKnight(chessboard,piece.colour,p_x(i),p_y(i));
151     case 9
152         [move] = MovementQueen(chessboard,piece.colour,p_x(i),p_y(i));
153     case 10

```

```

154         [move] = MovementKing(chessboard,piece.colour,num_moves,potentialmoves,p_x(i),p_y(i));
155     end
156
157     %-----
158     %           Individual Piece Moves That Generate New Game States
159     %           Recursion is also added in each loop
160     %-----
161     [move_x,move_y] = find(move ~= 0);
162     perm_index2 = randperm(length(move_x));
163     move_x = move_x(perm_index2);
164     move_y = move_y(perm_index2);
165     n_move = length(move_x);
166     pruneflag = 0;
167     %This loop generates all the game states from 1 piece
168     for j = 1:n_move
169         switch move(move_x(j),move_y(j))
170             case 1
171                 [pchessboard, ppiece.colour, pnum_moves,kingincheck]=ClickMovePiece(0,0,p_x(j),p_y(j),
172                     num_moves,0,move,0,1,move_x(j),move_y(j));
173             case 2
174                 [pchessboard, ppiece.colour, pnum_moves,kingincheck]=ClickCapturePiece(0,0,p_x(j),p_y(j),
175                     num_moves,0,move,0,1,move_x(j),move_y(j));
176             case 3
177                 [pchessboard, ppiece.colour, pnum_moves,kingincheck]=ClickEnpassant(0,0,p_x(j),p_y(j),
178                     num_moves,0,move,0,1,move_x(j),move_y(j));
179             case 4
180                 [pchessboard, ppiece.colour, pnum_moves,kingincheck]=ClickCastling(0,0,p_x(j),p_y(j),
181                     num_moves,0,move,0,1,move_x(j),move_y(j));
182             case 5
183                 [pchessboard,ppiece.colour, pnum_moves,kingincheck]=ClickPawnPromo(0,0,p_x(j),p_y(j),
184                     num_moves,0,move,0,1,move_x(j),move_y(j),'queen');
185         end
186         %-----A node has been generated, what do you want to do with it?-----
187         if kingincheck
188             %ignore because move not valid
189             if ~exist('boardscore','var')
190                 boardscore = 99999;
191                 bchessboard = 0;
192                 bpiece.colour =0;
193                 bnum_moves =0;
194             end
195         else
196             %Generate another layer with recursive parameters
197             [boardscore,~,~,~,handles]=...
198             AI.GenerateAllMoves(TmpB,pchessboard,ppiece.colour,pnum_moves,depth-1,-maxormin,handles);
199
200             if boardscore < previousboardscore
201                 previousboardscore = boardscore;
202                 bchessboard = pchessboard;
203                 bpiece.colour = ppiece.colour;
204                 bnum_moves = pnum_moves;
205             end
206             if boardscore<beta
207                 beta = boardscore;
208             end
209
210     % disp([depth alpha beta boardscore previousboardscore i j n_remaining n_move])
211     if alpha>beta
212         pruneflag = 1;
213         break
214     end
215 end

```

```

216         end
217         if pruneflag
218             break
219         end
220     %-----
221 end
222 %=====
223 end % For if maxormin
224 end % For DEPTH if condition
225 end %For Function

```

### A.3.3 heuristicanalysis.m

```

1  function [boardscore] = heuristicanalysis(B,chessboard, piece.colour,num.moves,currentcolour)
2  %Colour should be the side in which it is being analysed for
3
4  %-----
5  %                               Init Values
6  %-----
7  if currentcolour == 119
8      oppcolour = 98;
9  else
10     oppcolour = 119;
11 end
12 %Generates potential moves of the currently investigated game state colour
13 [potentialmoves,capt_index] = analyseboard(chessboard, piece.colour,num.moves,currentcolour)
14 %Generates potential moves of the opponent
15 [oppcolourpotentialmoves, oppcolourcapt_index] = analyseboard(chessboard, piece.colour,num.moves,oppcolour)
16
17 %Finds the locations of own pieces and opponent's pieces
18 piece_index = find(piece.colour==currentcolour);
19 opp_piece_index = find(piece.colour==oppcolour);
20
21 %-----
22
23 %-----Capture Analysis-----
24 %A move is good because it opens up capture possibilities
25 num_pot_capture = length(capt_index); %Number of potential Captures
26 capt_value_sum = sum(chessboard(capt_index)); %The total capture value
27
28 %A move is good if it increases the number of capture
29 capt_value_diff = 51 - sum(chessboard(opp_piece_index));
30
31 %----- Moves Analysis -----
32 %A move is good because it opens up space for other pieces to move
33 nocapture = potentialmoves;
34 nocapture(capt_index) = 0;
35 num.moves.available = sum(sum(nocapture));
36
37 %----- Threats -----
38 %If the move causes other pieces to be under threat, the move is worse.
39 opp_num_pot_capture = length(oppcolourcapt_index);
40 opp_capt_value_sum = sum(chessboard(oppcolourcapt_index));
41
42 %----- Number of own pieces -----
43 %A move is good if it prevents the number of own pieces from decreasing.
44 own_piece_sum_diff = 51 - sum(chessboard(piece_index));
45

```



```

46
47 %----- Control of centre space -----
48 %A move is good if it increases control of the centre of the board
49 centre_piece=zeros(8,8);
50 centre_piece([28 29 36 37])=chessboard([28 29 36 37]);
51 centre_piece = centre_piece~=0;
52 centre_space_sum = sum(centre_piece(piece_index));
53
54
55 %----- Own King Checked? -----
56 %Checks if own king is in check. If in check, also checks if its a checkmate
57 own_ischeck = KingCheck(chessboard,piece.colour,currentcolour,oppcolour,capt_index,oppcolour);
58 if own_ischeck==1
59     own_ischeckmate = checkmate(B,chessboard,piece.colour,num.moves);
60     else own_ischeckmate = 0;
61 end
62
63 %----- Castling? -----
64 %Checks if castling has taken place
65 rook_pos = find(chessboard==5 & piece.colour==currentcolour);
66 king_pos = find(chessboard==10 & piece.colour==currentcolour);
67 castle = 0;
68
69 if currentcolour == 98 %Black case
70     if (king_pos==49 && ismember(41,rook_pos) && num.moves(41)==1 && num.moves(49)==1)
71         castle = 1;
72     elseif (king_pos==17 && ismember(25,rook_pos) && num.moves(25)==1 && num.moves(17)==1)
73         castle = 1;
74     end
75
76 else %White case
77     if (king_pos==56 && ismember(48,rook_pos) && num.moves(48)==1 && num.moves(56)==1)
78         castle = 1;
79     elseif (king_pos==24 && ismember(32,rook_pos) && num.moves(32)==1 && num.moves(24)==1)
80         castle = 1;
81     end
82 end
83
84 %----- Opponent Checkmate? -----
85 %Checks if opponent king is in check. If in check, also checks if its a checkmate
86 opp_ischeck = KingCheck(chessboard,piece.colour,oppcolour,capt_index,potentialmoves);
87 if opp_ischeck==1
88     opp_ischeckmate = checkmate(B,chessboard,piece.colour,num.moves);
89     else opp_ischeckmate = 0;
90 end
91
92 %----- Possibility of opponenet's promotion? -----
93 %A move is bad if it brings opponent's pawn closer to the end of the board for promotion.
94 pawn_index = find(chessboard==1 & piece.colour==oppcolour);
95 if oppcolour == 98 %Black case
96     end_dist = 8-rem(pawn_index,8);
97     sum_opp_pawn_dist = sum(end_dist==0) + 0.5*sum(end_dist==1);
98 else %White case
99     end_dist = rem(pawn_index,8)-1;
100     sum_opp_pawn_dist = sum(end_dist==0) + 0.5*sum(end_dist==1);
101 end
102
103 %----- Possibility of own promotion? -----
104 %A move is good if it brings own pawn closer to the end of the board for promotion.
105 pawn_index = find(chessboard==1 & piece.colour==currentcolour);
106 if currentcolour == 98 %Black case
107     end_dist = 8-rem(pawn_index,8);

```

```

108     sum_own_pawn_dist = sum(end_dist==0) + 0.5*sum(end_dist==1);
109 else %White case
110     end_dist = rem(pawn_index,8)-1;
111     sum_own_pawn_dist = sum(end_dist==0) + 0.5*sum(end_dist==1);
112 end
113
114 %----- Gain Factor for Hard -----
115 if (get(handles.setHard, 'Value')==1)
116     gainCapture = 3; %Encourages AI to position a piece such that it can capture more pieces in
117     gainMoves = 10; %Encourages AI to position such that it opens space for other pieces
118     gainThreats = -4; %Discourages AI to make moves that will lead to threats
119     gainOpppieces = 25; %Encourages to make moves that decrease opponents pieces
120     gainOwnpieces = -5; %Discourages AI from making moves that decrease own pieces
121     gainCentre = 1; %Encourages AI to increase control of centre space
122     gainOwnprom = 1; %Encourages AI to promote own pawns close to the end of the board
123     gainOppprom = -10; %Discourages AI to promote opponent's pawns
124 end
125 %----- Gain Factor for Easy -----
126 if (get(handles.setEasy, 'Value')==1)
127     gainCapture = 2; %Encourages AI to position a piece such that it can capture more pieces in
128     gainMoves = 10; %Encourages AI to position such that it opens space for other pieces
129     gainThreats = -2; %Discourages AI to make moves that will lead to threats
130     gainOpppieces = 3.5; %Encourages to make moves that decrease opponents pieces
131     gainOwnpieces = 1; %Discourages AI from making moves that decrease own pieces
132     gainCentre = 10; %Encourages AI to increase control of centre space
133     gainOwnprom = 10; %Encourages AI to promote own pawns close to the end of the board
134     gainOppprom = -1; %Discourages AI to promote opponent's pawns
135 end
136 %----- Gain Factor for Random -----
137 if (get(handles.setRandom, 'Value')==1)
138     gainCapture = 0; %Encourages AI to position a piece such that it can capture more pieces in
139     gainMoves = 0; %Encourages AI to position such that it opens space for other pieces
140     gainThreats = 0; %Discourages AI to make moves that will lead to threats
141     gainOpppieces = 0; %Encourages to make moves that decrease opponents pieces
142     gainOwnpieces = 0; %Discourages AI from making moves that decrease own pieces
143     gainCentre = 0; %Encourages AI to increase control of centre space
144     gainOwnprom = 0; %Encourages AI to promote own pawns close to the end of the board
145     gainOppprom = 0; %Discourages AI to promote opponent's pawns
146 end
147 %----- Final Score Calculation -----
148 boardscore = gainCapture * capt.value.sum...
149             + gainMoves * num_moves_available...
150             + gainThreats * opp_capt.value.sum...
151             + gainOpppieces * capt.value.diff...
152             + gainOwnpieces * own.piece.sum.diff...
153             + gainCentre * centre.space.sum...
154             + gainOwnprom * sum_own_pawn_dist...
155             + gainOppprom * sum_opp_pawn_dist;
156 %Checks if castling has occurred
157 if castle == 1
158     boardscore = boardscore + 250;
159 end
160 %If a checkmate has occurred, new boardscores are assigned
161 if (get(handles.setHard, 'Value')==1 || get(handles.setEasy, 'Value')==1 )
162     if opp_isccheckmate == 1
163         boardscore = 99999;
164     end
165
166     if own_isccheckmate == 1
167         boardscore = -99999;
168     end
169 end

```

```

170
171 if (get(handles.setRandom, 'Value')==1)
172     boardscore=rand* 2000;
173 end
174 end

```

## A.4 Board analysis

### A.4.1 analyseboard.m

```

1 %Analyseboard Looks at one colour, sees where each piece is able to
2 %move. This is to allow for the Check function and castling.
3 %Colour in this case can be either current one or opposing one
4 %Use oppositecolour to generate threats and threat captures
5 function [potentialmoves,capt_index] = analyseboard(chessboard, piece_colour,num_moves,colour)
6
7 %Initialisation -----
8 [p_x, p_y] = find(piece_colour == colour);
9 n_remaining = length(p_x);
10 potentialmoves = zeros(8,8);
11
12 %Loop to look at every piece's moves -----
13 for i=1:n_remaining
14     %Determines what piece is selected
15     p_type = chessboard(p_x(i),p_y(i));
16
17     %Based on the type of piece, its movement is calculated
18     switch p_type
19         case 1
20             [move] = MovementPawn(chessboard,piece_colour,num_moves,p_x(i),p_y(i));
21             %disp('Pawn');
22         case 5
23             [move] = MovementRook(chessboard,piece_colour,p_x(i),p_y(i));
24             %disp('Rook');
25         case 4
26             [move] = MovementBishop(chessboard,piece_colour,p_x(i),p_y(i));
27             %disp('Bishop');
28         case 3
29             [move] = MovementKnight(chessboard,piece_colour,p_x(i),p_y(i));
30             %disp('Knight');
31         case 9
32             [move] = MovementQueen(chessboard,piece_colour,p_x(i),p_y(i));
33             %disp('Queen');
34         case 10
35             [move] = MovementKing(chessboard,piece_colour,num_moves,potentialmoves,p_x(i),p_y(i));
36             %disp('King');
37     end
38
39     %Sums up all possible moves of 1 colour.
40     potentialmoves = potentialmoves+move;
41 end
42 %-----
43 %           Analysis of potentialmoves
44 %-----
45 %-----Capture Analysis-----
46 potentialcaptures = potentialmoves ~= 0 & chessboard~= 0;
47 capt_index = find(potentialcaptures==1);

```

```

48 %num_pot_capture = length(capt_index);
49 %capt_value_sum = sum(chessboard(capt_index));
50
51 %----- Moves Analysis -----
52 %nocapture = potentialmoves;
53 %nocapture(capt_index) = 0;
54 %num_moves_available = sum(sum(nocapture));
55
56 end

```

#### A.4.2 KingCheck.m

```

1 %KingCheck Checks if the king is in check, checkmate or stalemate
2 %Colour in this case must be the current colour
3 %King Colour must be contrary to CAPT_INDEX & POTENTIAL MOVES
4 function [value]=KingCheck(chessboard,piece_colour,ownkingcolour, oppcolour,capt_index,oppco
5
6 %----- King In Check -----
7 king_index = find(chessboard == 10 & piece_colour == ownkingcolour);
8 kingincheck = ismember(king_index,oppcolour,capt_index);
9 if(kingincheck)
10     value = 1;
11 %Otherwise not in check
12 else
13     value = 0;
14 end
15 end

```

#### A.4.3 checkmate.m

```

1 %Checkmate Determines if the currentboard is a checkmate state for
2 %specified colour
3 %Gives 1 for Checkmate, 0 for not checkmate
4 function [result]=checkmate(B,chessboard,piece_colour, num_moves)
5
6 if(mod(B.info.turn,2)==1)
7     colour = 119;
8     oppcolour = 98;
9 else
10     colour = 98;
11     oppcolour = 119;
12 end
13 result = 1;
14 %-----
15 %           Loop that generates all possible moves
16 %-----
17 [p_x,p_y] = find(piece_colour == colour);
18 n_remaining = length(p_x);
19 [potentialmoves] = analyseboard(chessboard, piece_colour,num_moves,oppcolour);
20
21 %In essence, we are going through each piece, looking at it's possible
22 %moves, make those possible moves, evaluate, save bestboard.
23 for i=1:n_remaining
24     p_type = chessboard(p_x(i),p_y(i));

```

```

25     switch p-type
26         case 1
27             [move] = MovementPawn(chessboard,piece_colour,num_moves,p_x(i),p_y(i));
28         case 5
29             [move] = MovementRook(chessboard,piece_colour,p_x(i),p_y(i));
30         case 4
31             [move] = MovementBishop(chessboard,piece_colour,p_x(i),p_y(i));
32         case 3
33             [move] = MovementKnight(chessboard,piece_colour,p_x(i),p_y(i));
34         case 9
35             [move] = MovementQueen(chessboard,piece_colour,p_x(i),p_y(i));
36         case 10
37             [move] = MovementKing(chessboard,piece_colour,num_moves,potentialmoves,p_x(i),p_y(i));
38     end
39
40     %-----
41     %           Individual Piece Moves That Generate New Game States
42     %           Recursion is also added in each loop
43     %-----
44     [move_x,move_y] = find(move ~= 0);
45     n_move = length(move_x);
46     %This loop generates all the game states from 1 piece
47     for j = 1:n_move
48         switch move(move_x(j),move_y(j))
49             case 1
50                 [pchessboard, ppiece_colour, pnum_moves,kingincheck]=ClickMovePiece(0,0,p_x(j),p_y(j),
51                     num_moves,0,move,0,1,move_x(j),move_y(j));
52             case 2
53                 [pchessboard, ppiece_colour, pnum_moves,kingincheck]=ClickCapturePiece(0,0,
54                     num_moves,0,move,0,1,move_x(j),move_y(j));
55             case 3
56                 [pchessboard, ppiece_colour, pnum_moves,kingincheck]=ClickEnpassant(0,0,p_x(j),p_y(j),
57                     num_moves,0,move,0,1,move_x(j),move_y(j));
58             case 4
59                 [pchessboard, ppiece_colour, pnum_moves,kingincheck]=ClickCastling(0,0,p_x(j),p_y(j),
60                     num_moves,0,move,0,1,move_x(j),move_y(j));
61             case 5
62                 [pchessboard,ppiece_colour, pnum_moves,kingincheck]=ClickPawnPromo(0,0,p_x(j),p_y(j),
63                     num_moves,0,move,0,1,move_x(j),move_y(j),'queen');
64         end
65
66         result = min(kingincheck, result);
67         if result == 0
68             break
69         end
70
71     end
72     if result == 0
73         break
74     end
75 end

```

#### A.4.4 readchessboard.m

```

1  %readchessboard takes in chessboard and creates B
2  function [B] = readchessboard(B,chessboard,piece_colour)
3
4  X = struct(NewPiece([]));

```

```
5 % build the initial board with everything non-playable at first
6 % add paddings to the non-playable areas of 4 squares and place pieces
7 for i=1:size(chessboard,1)+B.info.pad
8     for j=1:size(chessboard,2)+B.info.pad
9         X(i,j) = NewPiece([]);
10    end
11 end
12
13
14 % now place pieces and playable areas
15 for i=1:size(chessboard,1)
16     for j=1:size(chessboard,2)
17         if chessboard(i,j) == 0
18             pName = []; pColour = 0;
19         else
20             switch chessboard(i,j)
21                 case 1
22                     pName = 'pawn';
23                 case 3
24                     pName = 'knight';
25                 case 4
26                     pName = 'bishop';
27                 case 5
28                     pName = 'rook';
29                 case 9
30                     pName = 'queen';
31                 case 10
32                     pName = 'king';
33             end
34
35             switch piece-colour(i,j)
36                 case 119
37                     pColour = 1;
38                 case 98
39                     pColour = -1;
40             end
41             X(i+B.info.pad/2,j+B.info.pad/2) = NewPiece(pName,pColour);
42         end
43     end
44 end
45
46 B.top = X;
47 end
```