

computer science chess Engine project

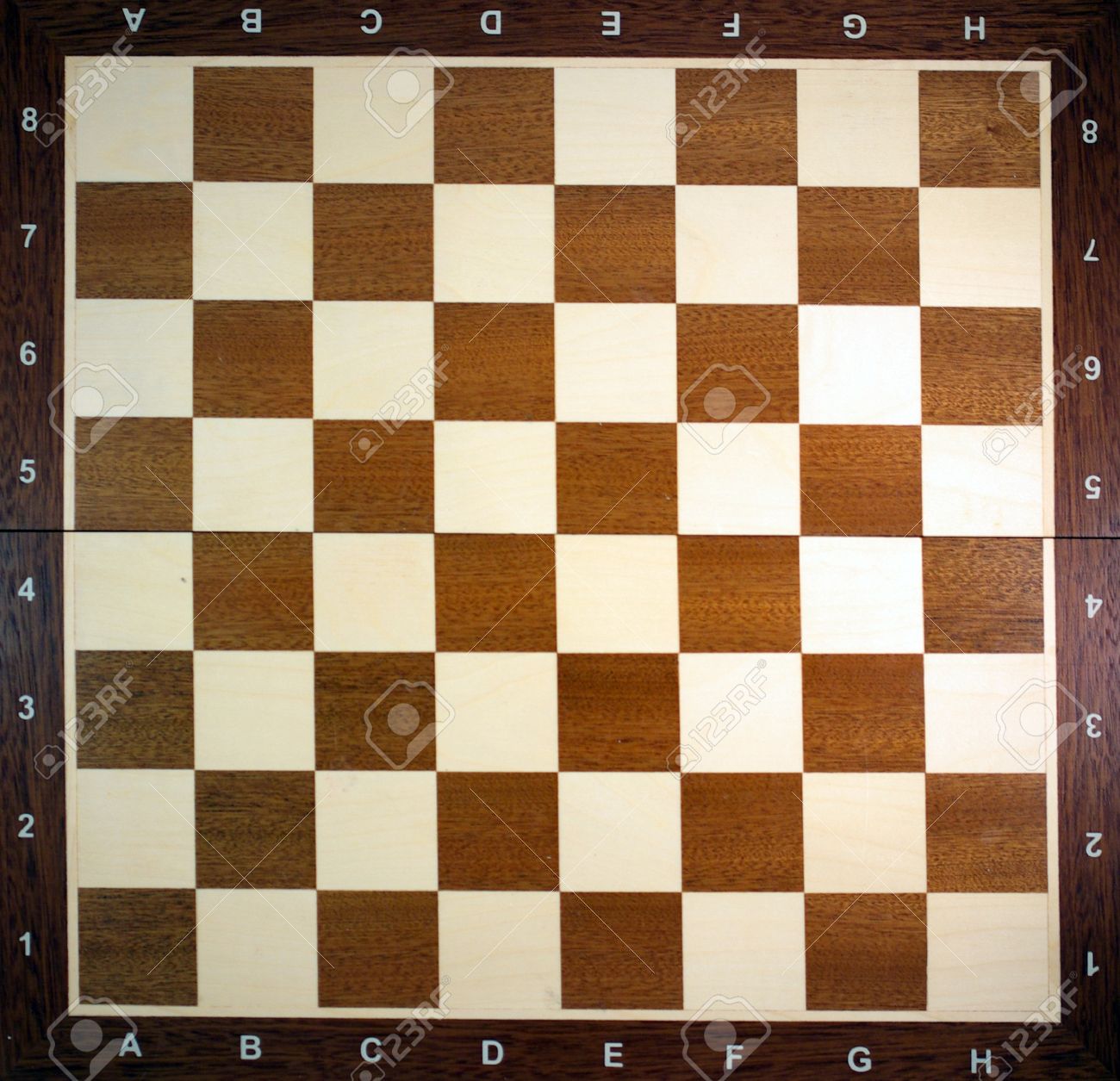
By Pavel Scerbakov



Analysis:

# Game Mechanics / Rules:

Chess is a 2-player game played on a square board made up of 8 squares on each side, totalling 64 squares. Each square is horizontally labelled A-H and vertically numbered 1-8.



Each player plays as either black or white and gets 16 pieces in total.

Chess Piece Values

King, Queen, Bishop, Rook, Pawn

Chess Start Layout

1

5

3

3

9

∞

9

∞



1

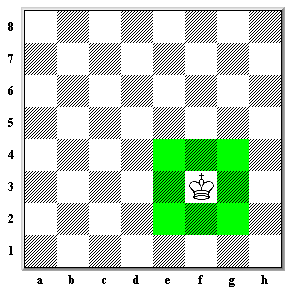
5

3

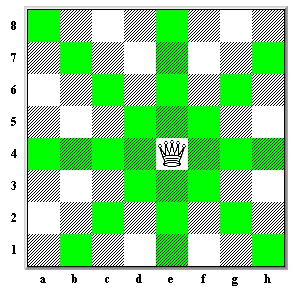
3

Each player has 1 King, 1 Queen, 2 Bishops, 2 Knights and 8 Pawns. They start in the position shown in the image. Each piece has a value set by the chess community throughout the years as shown in the picture. This is because each piece has different utility when compared to the other. Pieces can move differently and wherever they can move, they can capture an enemy piece there, with exception with the pawn. Places where enemy pieces can go are ‘under attack’ meaning a king can not go there. Allied and enemy pieces block where pieces can move, meaning you can’t go past pieces. You can only go as far as just before the piece if it is allied or capture it if it is the enemy, with the exception of the knight.

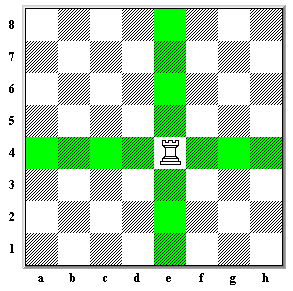
King – The king is the most important piece in the entire chessboard. It is responsible for keeping the player in the game and if the king is put in a position where he can not move and is under check, the game is over for that player. He is able to move to any adjacent square (including diagonal), unless that square can be targeted by an enemy piece. He can also capture adjacent enemy pieces (once again, as long as they’re not protected by enemy pieces).



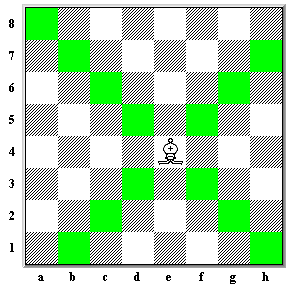
Queen – The queen is the highest worth piece excluding the king. She can move in any direction as far as she wants. Horizontal, vertical or diagonal.



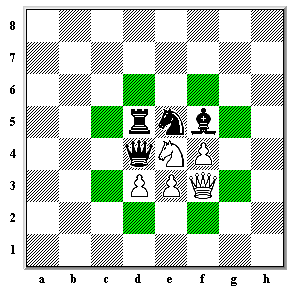
Rook – The rook is similar to the queen in that it can move as far as it wants, however it can only do so on the horizontal and vertical axis (up, down and right, left).



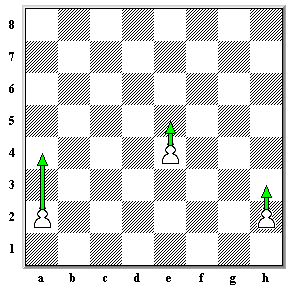
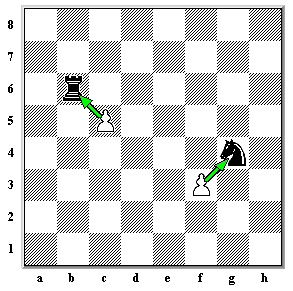
Bishop – The bishop can also move as far as he wants, hoWver it can only do so in the diagonal axis. This means it can never leave the colour of the square that it starts on since all diagonally connected squares are of the same colour.



Knight – The knight is special in that it can ‘jump’ over allied and enemy pieces but it can only move in an L shape. 2 squares horizontally or vertically and then 1 square perpendicular to the direction of the previous 2 squares left or right. The knight can only capture the piece at the end of his L move.

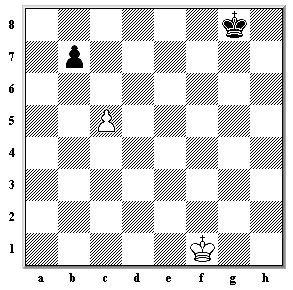
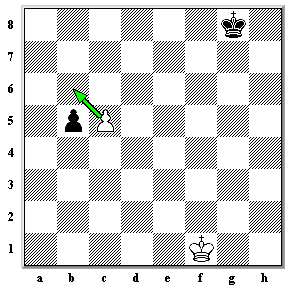


Pawn – The pawn can move forward by 1, forward being in relation to the side of the board. E.g. the black pawn can move towards the white end of the board and vice versa for white pawn. It, however, cannot capture in this way. A pawn can only capture an enemy piece which is diagonally adjacent in front of it. In addition, a pawn’s first move allows it to move 2 forwards instead of one forward if so wished. Finally, another move a pawn can do is referred to as ‘en passant’. This happens when a pawn moves 2 forward past an enemy pawn, ‘skipping’ it’s area of attack. The enemy pawn can take the pawn that passed it by 2 by moving diagonally in front of the passing pawn. En passant can only be performed for the 1 move that it becomes available.



Pawn Move

Pawn Capture



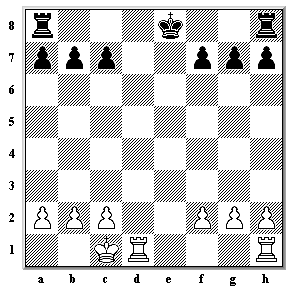
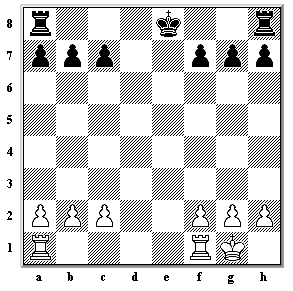
En Passant

There are 2 ways a game of chess can end. A win, a loss or a draw. The way to win in chess is to put the enemy king in a check in such a way where they can not do another move that will free their king from check. A check is where the enemy king is in a position where he is under attack from the enemy piece. A check can be relieved by either capturing the attacking piece, blocking the line of attack or moving the king into a place not under attack by enemy pieces. If player is unable to do one of these three things, the game is lost for him, resulting in a win for the enemy. Another way of winning is by the enemy resigning, forfeiting their game. Now, it is also possible to draw in chess. This is possible in one of many ways. One way of getting a draw in chess is by an agreement between the players. Another is with the threefold repetition rule. This means that if a board layout repeats itself 3 times throughout a match, the match can be called a draw. The three times the position appears, all pieces have to be in the same location with the same number of pieces as before and the same castling & en passant opportunities have to be there. This means that if a rook moves back and forth 2 times for the first time, this will not be a repeating position since in the first one the king would’ve been able to castle with the rook, but since it moved that is no longer possible making it a different position than before.

The third way of drawing in chess is by the 50-move rule. The 50-move rule means that if 50 whole moves in a row happen where a piece is not taken or a pawn is not moved, the game is declared a draw.

The final way of getting a draw is a draw by stalemate. This occurs where the king is put in such a position where no pieces on the board can be moved but the king himself is not in check. This means that the player has no legal moves to make despite it being his turn. This is declared a stalemate and a draw.

In the previous section, castling was mentioned. Castling is the interaction between an unmoved king and an unmoved rook. It occurs when there are no pieces in between the king and his allied rook, the king is not in check and none of the squares in-between and including the rook are under attack. The movement cause them to move in a way the could not before and is used early on in the game to put the king in a safer position. The pictures demonstrate a ‘queen side castle’, castle towards the side of the queen and a ‘king side castle’, castle towards the side of the king.



Finally, there is another important rule of chess. It is the promotion of pawns. If a pawn reaches the end of the board, it can be converted to any piece on the very same turn (Queen, Rook, Bishop, Knight). This converted piece does not have to adhere to the limits of pieces e.g. if the player has 1 queen he can convert the pawn to get a 2nd queen, 3rd queen etc.

# Scoping the Problem

Project Overview

My project will consist of a chess UI that allows the user to play locally on the same host machine against a human player, against a computer opponent chess engine or with another machine.

Main Objectives of the Project

* Options where the user can choose to play against another player on the host machine or to play against a computer.
* Option which will have access to the strength of the engine.
* A chess engine utilizing the **minimax algorithm** with different depths that can be selected.
* An **alpha-beta pruning** algorithm to considerably improve performance of the engine.
* Ability to play either colour in Computer player modes.
* A board with 2D pieces that can be dragged by clicking and let go by clicking again on the board.
* The application will calculate all the possible locations that a selected piece can move to, including castling, double pawn movement and en passant.
* The ability to determine when checkmate, draw and check have occurred.
* Ability to reset the game and board with newly selected settings.
* Squares that highlight when hovering over them whether the move is legal or not, and if clicked when not legal, the piece will be returned to its original position without the turn being cancelled.
* A new window opening when promoting a pawn at the edge of the board to select the piece you would like to promote to.
* Games can be saved with FEN notation and stored as a text file. Stores all relevant information correctly.
* Games can be loaded from any text file with contents written in FEN notation. Loads all information stored properly.
* An undo option that will undo moves. It can undo all the way up to the first move since the start or a load.

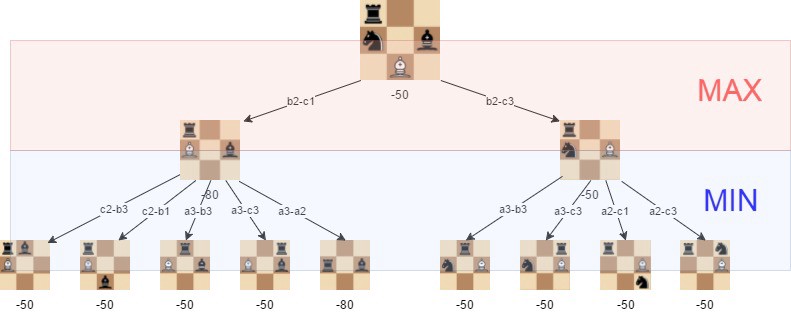
# Research of the Problem

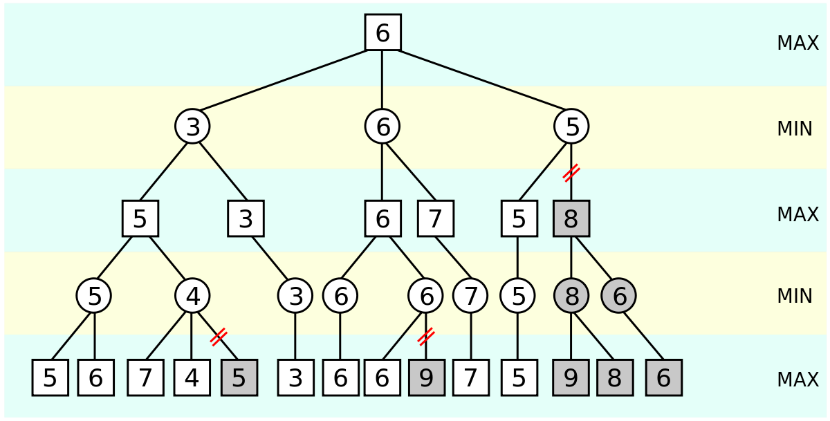
The main 3 points of this project that will need research are the Minimax algorithm, Alpha Beta pruning and FEN notation.

Minimax Algorithm

Simply put, the minimax algorithm is a way for the computer to choose the best possible decision to receive the best possible output for itself. It does this by declaring itself as the ‘Max Player’ and the enemy the ‘Min Player’. I will develop an analysing function to analyse a board and give it a score. The ‘Max Player’ will want a board result with the highest possible score while the ‘Min Player’ would want the board result with the lowest possible score.

So the algorithm begins by the computer (‘Max Player’) choosing a starting initial turn. From there it, it will keep selecting random positions until the depth set has been reached. In the example below, the depth is 3. We can see that the Max player has 2 possible turns. From the first turn, there are 4 possible outcomes, all leading to different scores. The lowest score is -80. This is the best score for the Min player and we assume that the Min player always plays the best they possibly can and therefore chooses -80 as their turn. Despite the other 4 options leading to -50, the best score we can get is -80 since we assumed the Min player picks the -80 option for us. Now in the second path we can see that there are 4 possible outcomes for the Min player to choose. All of them however, lead to -50, leading to this branch leading a guaranteed -50 or better. This leaves the Max player with now 2 options, -50 or -80. The max player would choose the highest score, and therefore chooses -50, the right-hand side of the tree. I have found ‘**Adam Berent’s**’ evaluation scores and piece tables good, and I have decided to use them. As we increase the depth and the possible moves, we obviously get a lot more options that would take a lot more time for the computer to process. To overcome this problem, we implement Alpha-Beta pruning.



Alpha-Beta pruning is, conceptually, quite simple actually. In the diagram below, we present the score values of the different board positions. In the leftmost branch, at depth 4, we calculate the score of 5 for Min. For depth 6, we calculate 7 and 4. After we discover 4, we no longer need to do anymore searching. This is because the min algorithm would now only pick a value of 4 or less at depth 4 meaning it is guaranteed to be lower than the 5 it has already worked out. This means the max algorithm will certainly pick the 5 at depth 3 since the other option is 4 or lower. Similarly, for the 3rd branch at the start, we work out a value of 5 for the Max player. Because of this, we do not need to work out the rest of this branch since the other option that Min could choose over the 5 would be something lower. This means that at depth 2, the options are 3, 6 or <=5. Since the depth 1 max would choose the highest possible score, it would choose 6, rendering the other results of =<5 useless. This already saves quite a lot of options that we need to search and analyze. In much bigger trees, this can save a lot of processing time and power. For chess, this is extremely important as with only a depth of 4, the different positions analyzed without pruning can be 879,750 and with pruning this drops down all the way to 61,721! This is only 7% of the original positions, reducing the processing time by 93%. Therefore, it would clearly be beneficial to implement this algorithm alongside the minimax algorithm.

FEN (Forsyth-Edgeworth Notation) is a standard notation for describing a particular position of a chess game. It stores all the necessary information to restart a game from the position. It consists of 6 fields:

* Piece placement from white’s perspective, where white pieces are denoted by capitals and black ones by lower case letters. Empty squares are denoted by a number, signifying how many empty spaces there are. “/” notate separate ranks.
* The colour who’s turn it is. ‘w’ for white and ‘b’ for black.
* Castling availability. K means white can king’s side castle, Q means white can queen’s side castle, k means black can king’s side castle and q means black can queen’s side castle. No castling availability is denoted by a ‘–‘.
* The en passant square in algebraic notation. Represents the square which an enemy pawn could move to capture the enemy pawn with en passant. This is recorded regardless if a pawn is in position to do the capture. No en passant is denoted by a ‘-‘.
* The half move clock, used for calculating 50 move rule.
* The full move clock, to say what turn it is (starting at 1 for white and incrementing every black turn).

The starting position would look like the following in FEN:  
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

This notation is useful for saving and loading games and would allow users to play the game at a later date after closing the application, load puzzles / scenarios downloaded off the internet or even use the files in another chess application that reads FEN.

A disadvantage of FEN however, is that draw by Threefold Repetition is impossible to recognize from positions before the save. Another method of saving could’ve been used to circumvent this, such as Extended Position Description. In this project however, I did not see the necessity of this and settled with FEN.

## Another Alternative

Another way this task of chess games could’ve been solved, is rather than using a MinMax algorithm, would be to use neural networks (machine learning). This consists of feeding the application sets of games or making the computer play against itself. Every move is recorded and simple evaluation goals are given such as ‘having a piece gives more points’ or ‘checkmate is good’. Overtime, the computer flags certain moves as those that lead to good evaluations and bad evaluations. This is purely from the machine trying out many different games and consolidating its certainty of a move being good. A way to compare it is to compare it to an actual player who would play games over and over and could recognize a position being good or bad from seeing it. The computer acts similar to a brain in this way as a ‘Neural Network’, getting ‘rewarded’ for moves that win the game and ‘punished’ for those that lose the game, causing the computer to play winning moves and not play losing moves.

A limitation of this approach is the capability of me to develop such a system in the time given to me. Another limitation is the time and hardware needed to run the system continuously to get a large enough sample to have a good enough performing chess opponent. This is because the software would need millions if more games to learn properly. This is why I decided to choose the MinMax algorithm since it is reasonable for me to develop and can perform without needing a lot of preparation / a large sample pool of data to function properly.

Documented Design

Key:

**Variable, class or method name.**

**Code**

# Board Design and GUI

I have coded the project consisting of mainly atomic methods to allow me to reuse the same methods as I progress throughout code and base it around 2 main 2D-arrays: The **fieldPiece** array (made up of the ‘Piece’ class objects) and the **movableSpots** Boolean array. **fieldPiece** simply stores the piece on the board relative to their board location and **movableSpots** is the array that stores the fields that can be moved to by a **grabbedPiece** after running the method **MovingRules**. I have used 2D arrays for these since they represent a table, a perfect way to display the board and access its squares with coordinates.

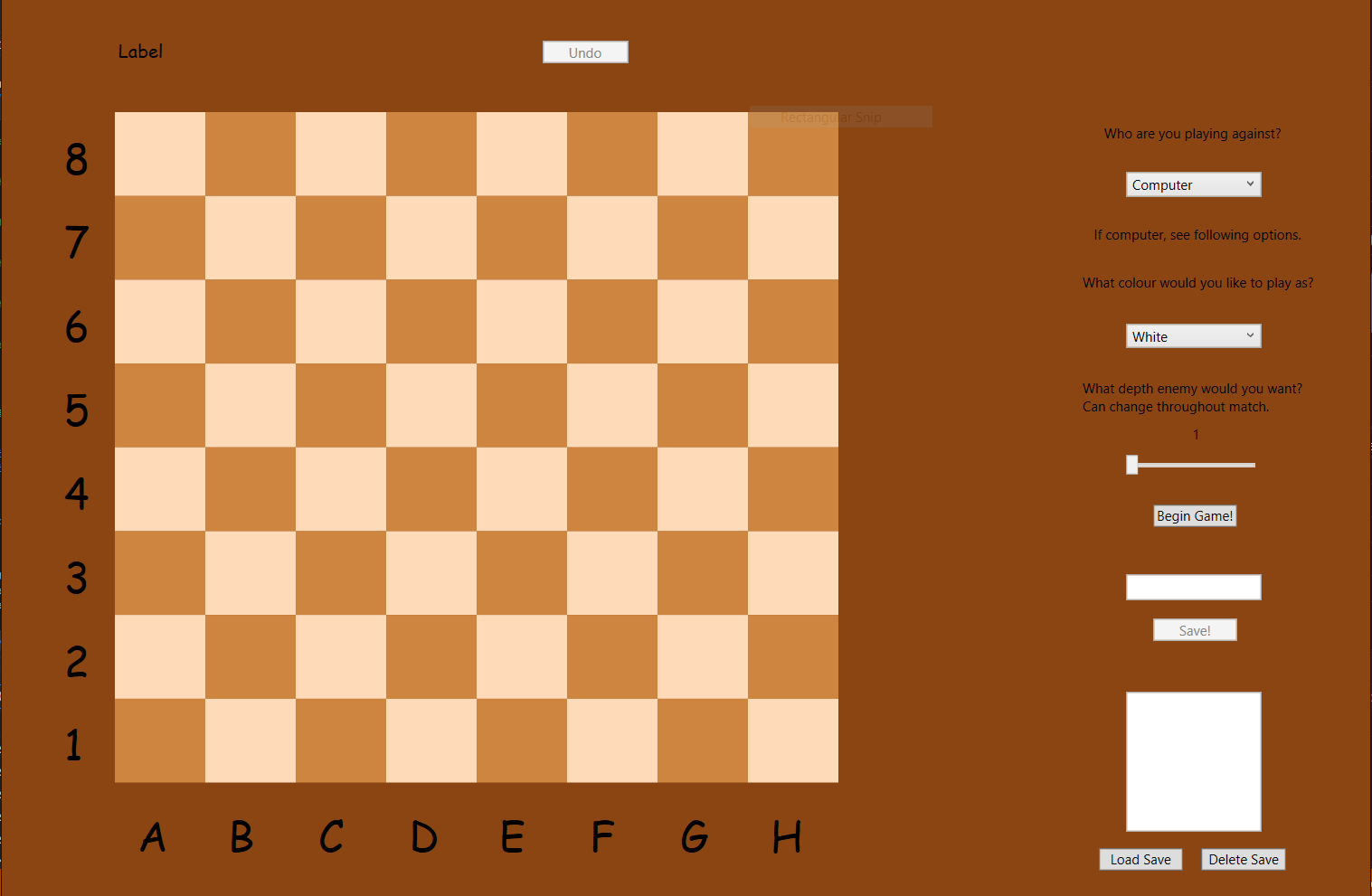


Figure 10

Figure 9

Figure 8

Figure 7

Figure 6

Figure 5

Figure 3

Figure 4

Figure 2

Figure 1

Interface of the program.

The picture above shows the application upon startup.

Figure 1: An undo button that allows the user to undo the move that has just been done by the computer or player. This allows them to change a computer’s move when playing against the computer or changing their own move.

Figure 2: A Combo Box that consists of 2 choices to let the player choose whether they are playing a hotseat game against another player. This option is ‘Player’. To play against the computer, the ‘Computer’ option should be picked.

Figure 3: If ‘Computer’ is selected in Figure 2, this box becomes enabled, allowing the choice of ‘White’ or ‘Black’ as the colour the player would like to play with against the computer.

Figure 4: If ‘Computer’ is selected in Figure 2, this box becomes enabled, allowing the player to slide the bar between 1-8 to change the depth of the computer’s MinMax algorithm. This may be changed during the course of the game with the computer, changing its difficulty.

Figure 5: The ‘Begin Game’ button. It starts the game and resets the game if clicked during play. It uses the parameters selected in the other Combo Box.

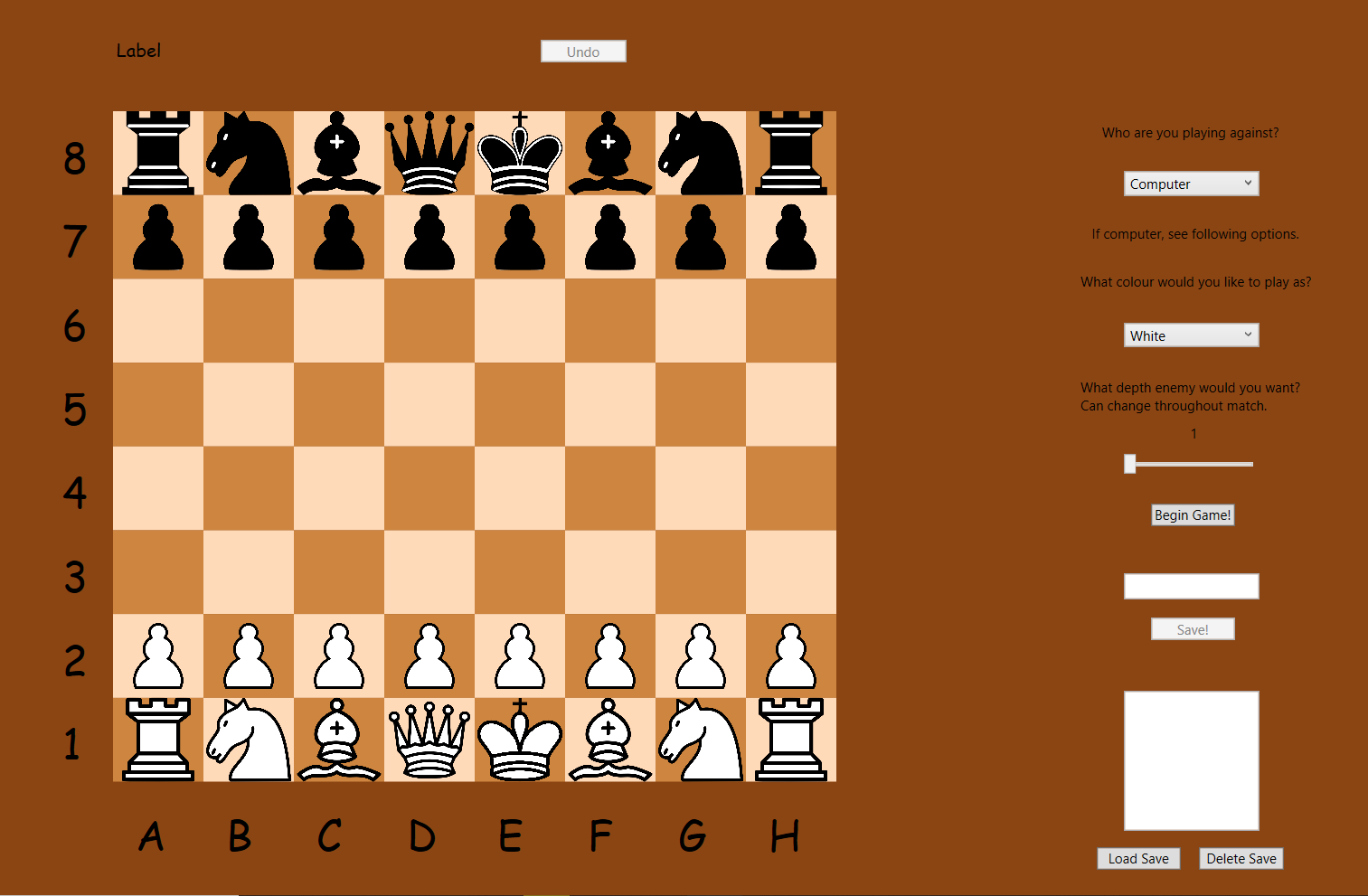
Figure 6: This is the board itself. It is consisted of ‘Rectangle’ classes with dimensions matched to the user’s screen resolution. They use the ‘PeachPuff’ and ‘Peru’ colours.

Figure 7: These are letter and number labels dynamically created to the left and below the chess board to make it easier for players to read board notation.

Figure 8: This is a debug label that displays average time per node searched and total time taken for the computer to do a move. Used for debugging.

Figure 9: This is a save name box and button, where the desired save file name is entered, and upon pressing the button, it is saved.

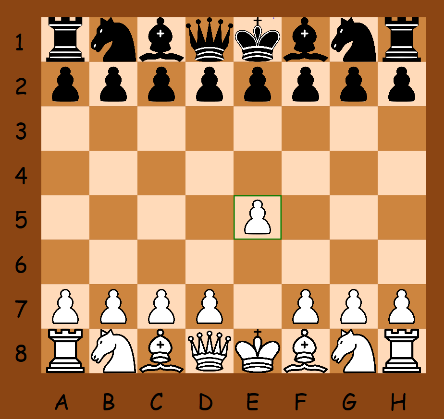
Figure 10: This is the save file section with a multiple-choice box where save files are shown and can be clicked. The load button loads the selected file while the delete button deletes it.



The formed board after pressing ‘Begin Game’

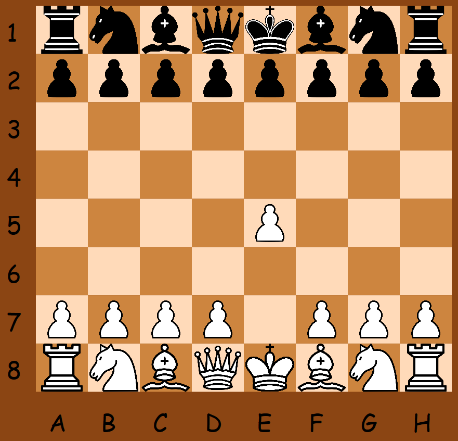
EXECUTEGUITURN

X = x mouse coordinates  
Y = y mouse coordinates  
if x or y not outside board range  
 if field[x,y] is movable to  
 executeturn()  
 if grabbedPiece = “P” PawnConversionCheck  
 SaveMove()  
 PerformGameEnd()  
 ExecutePCTurn()  
 end if  
end if  
AllignPieces()  
grabbedPiece = null

The interface is used by clicking on the pieces, it causes it to be picked and able to be moved around as shown on the right. If the piece is clicked outside the board or on an unmovable spot, it will get put back where it was initially, and won’t pass the turn to the next colour.  
Since the coordinates of the picked-up piece won’t be changed if the turn isn’t executed, all I have to do is realign the board, so the piece will be returned to its original location and set the grabbed piece to nothing, so it is no longer grabbed, because the player put it down, and no longer follows the player’s mouse. This validification is done in the ‘**executeGUITurn’** method. If the turn is valid however, the turn will get executed. All of the methods that manipulate the board are a part of the PlayerBoard class, an instance of which is created in MainWindow.

MAINWINDOW\_MOUSEDOWN

remove highlight from previously highlighted square  
if a piece is grabbed  
 ExecuteGUITurn()  
else  
 FindClickedSquare() // and grab piece if there is one  
end if  
if there is a grabbed piece  
 FindMovesOfGrabbedPiece()  
end if

Clicks are detected by the ‘**MainWindow\_MouseDown**’ event which is executed upon click. It will attempt to execute a turn if a piece is grabbed, since if the player is holding a piece and clicks, they are attempting to place the piece down. If they are not holding a piece, then they could potentially be attempting to pick up a piece, therefore the **FindClickedSquare**() will find what square has been clicked and check whether a piece exists there, if so grab it. After this, if a piece is indeed grabbed, all the movable spots of that piece will be found. This will allow the GUI to showcase whether a square is valid to move to with the green or red highlight by calculating the possible moves when a piece is picked up.

FINDCLICKEDSQUARE

For i = 0 ->boardSize  
 for j=0 ->boardSize  
 if fieldPiece[i,j].sprite = sender and turn = fieldpiece[i,j].colour  
 grabbedPiece = fieldPiece[i,j]  
 end if  
 end if  
end if

For **FindClickedSquare**, I used the ‘sender’ object provided by C# which holds the object that caused the **MainWindow\_MouseDown** event. In this case, it would be a piece or a rectangle. In the method, I compare all the current sprites of the pieces on the board to the one that has triggered to the click event. If a match is found,I then see if the clicked piece colour is the same as the current turn, then it is a valid piece to pick up.

To help with the animation of pieces following your mouse as well as updating highlights of whether a piece can move, I used another event method in C# called **CompositionTarget\_Rendering**. It executes code every frame that the using computer runs the application. At the start, the X and Y values are declared in a way to depict which square the mouse is hovering over. The -100 represents the 100 pixels the chess board starts from the edges. The coordinates will always round down towards 0 due to X and Y being an int, so if each rectangle is 100x100 and the mouse is on (256,181), this would lead to X as 1 and Y as 0 since (256-100)/100 = 1.56-> 1 and (181-100)/100 = 0.81-> 0. The initial conversion of X and Y to -1 when the coordinates are negative is to prevent values of >=-0,5 and <0 of rounding towards 0. If a piece is grabbed, its sprite’s location is set to that of the mouse with some offset not shown in the pseudocode. Then, I see if the mouse is within the chess board by checking its X&Y coordinates. If it is inside the board, I highlight the square the mouse is in with green if the square is a spot the piece can move to according to the **movableSpots** Boolean array. Else, it will be red since you can’t move there. Then, I take advantage of the **previousSquare** that was hovered over. If it is not the same as the current square, that means the mouse is no longer hovering over it, so I should remove the highlight from it. Then I set the **previousSquare** as the current square. Afterwards, I have checks for whether the Undo button should be available. This is done by seeing what the current **turnCount** is. It starts at 0 and goes up every move and down every undo. So at 0, it is the first move so the Undo button should be unavailable. Then I have checks for whether the options for the game should be available depending on which opponent is playing and finally, the value of the information label for the depth slider is updated to its value.

COMPOSITIONTARGET\_RENDERING

X = (x mouse coordinates – 100) / rectangle width   
Y = (y mouse coordinates – 100) / rectangle height  
if x mouse coordinate < 0 X=-1  
if y mouse coordinate < 9 Y=-1  
If there is a grabbedPiece  
 set x-coordinate of grabbedPiece.sprite to x mouse coordinates  
 set y-coordinate of grabbedPiece.sprite to y mouse coordinates  
 if X>0 and X<boardSize and Y>0 and Y<boardSize  
 if movableSpots[X,Y] is true, boardSquares[X,Y].stroke = green  
 else boardSquares[X,Y].stroke = red  
 if previousSquare != boardSquares[X,Y]  
 previousSquare.stroke = nothing  
 previousSquare = boardSquares[X,Y]  
 else previousSquare.stroke = nothing  
end if  
if turnCount = 0 make undo button unavailable  
else make undo button available  
if gametype selected is player  
 turn off playerColour option  
 turn off depth option  
else  
 turn on playerColour option  
 turn on depth option  
end if  
update slider label to depth value

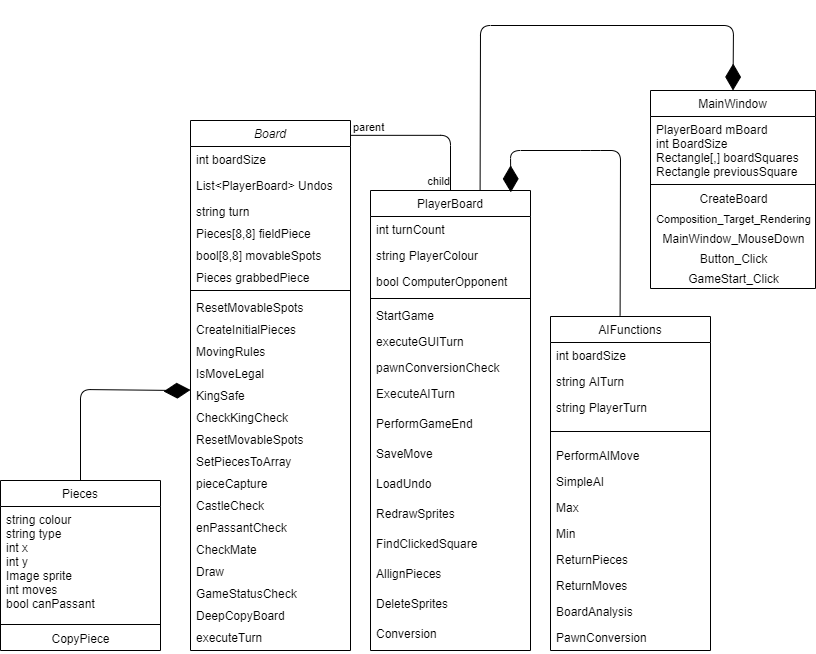
The initial board creation is the only method executed by the code at the beginning of the application in the **MainWindow** method. It loops, creating a rectangle object for each coordinate of the board. hRatio and wRatio are ratios based on the users monitor size in proportion to a 1920/1080 monitor. In a chess board, when both the x and y coordinate starting from the top left are both odd or both even, the square will be white. The other squares will all be black, hence the pseudocode as shown. The rectangles are then added as clickable objects in the **MainWindow\_MouseDown** event. Afterwards, labels that are on the side of the board are created using ASCII and looping. Then The **CompositionTarget\_Rendering** event is compiled. Then finally, the actual pieces themselves are created on the board in the **CreateInitialPieces** method. This is done with loops and the pieces are added to their starting positions with the **Pieces** objects into the **fieldPieces** array. They are not drawn onto the board as of yet. This happens when the start game button is pressed, which causes an event method to run which executes the **startGame** method. **startGame** is a **PlayerBoard** method which adds sprites to all the pieces and draws them, making them interactable and visible on the board. It will also execute the computer’s turn if the player is playing as black, since the computer goes first. From there, the mouse click event is responsible for further interaction with the **mBoard** (main **PlayerBoard** that represents the game the player is playing) by executing turns when clicks are made. All further methods are executed afterwards.

CREATEBOARD

for i=0 -> boardSize  
 for j=0 -> boardSize  
 Create new Rectangle  
 Height = 100 \* hRatio  
 Width = 100 \* wRatio  
 Add Rectangle to ChessBoard  
 BoardSquares[i,j] = Rectangle  
 Rectangle X-coordinate = 100 \* i \*hRatio  
 Rectangle Y-coordinate = 100 \* j \*wRatio  
 if i is even and j is even, Rectangle Colour = white  
 else if i is odd and j is odd, Rectangle Colour = white  
 else Rectangle colour = black  
 Add rectangle as clickable to MainWindow\_MouseDown event  
 (Labels for side letters & numbers code in here)  
 ….  
 (End of labels code)   
 Start CompositionTarget\_Rendering  
 CreateInitialPieces()

# Class Relationships

To illustrate the linkage of the classes, I have developed a class diagram which is a great reference to the origin and relationships of methods and important variables throughout the code.



A note about the diagram is that it does not go into splitting Window with its MainWIndow parent and is instead overall referred to as MainWindow. Technically PlayerBoard declares a MainWindow class inside of it, using the Window’s subroutines, but attempting to display this in the diagram would go against the intent of the diagram.   
Note: *Variables with the name (AI) have either changed to PC or Engine to represent it more accurately.*

# Board Methods

MOVINGRULES

int y = grabbedPiece.y  
int x = grabbedPiece.x  
string colour = grabbedPiece.colour  
string type = grabbedPiece.type  
ResetMovableSpots()  
**if type = “P” && pawn is not on y = 7 or 0**  
 if colour is white  
 if fieldPiece[x,y-1] is empty, movableSpots[x,y-1] = true  
 if grabbedPiece.moves = 0 && [x,y-2] is empty, movableSpots[x,y-2] = true  
 if x>0 && y>0 and (not legalcheck or ([x-1,y-1] is not empty and is a black piece)  
 movableSpots[x-1,y-1] true  
 if x<7 && y>0 and (not legalcheck or ([x+1,y-1] is not empty and is a black piece)  
 movableSpots[x+1,y-1] true  
 if x>0 && y=3 and fieldPiece[x-1,y] is not empty and is black and it canPassant  
 movableSpots[x-1,y-1] true  
 if x>0 && y=3 and fieldPiece[x+1,y] is not empty and is black and it canPassant  
 movableSpots[x+1,y-1] true  
 if colour is black  
 if fieldPiece[x,y+1] is empty, movableSpots[x,y+1] = true  
 if grabbedPiece.moves = 0 && [x,y+2] is empty, movableSpots[x,y+2] = true  
 if x>0 && y<7 and (not legalcheck or ([x-1,y+1] is not empty and is a black piece)  
 movableSpots[x-1,y+1] true  
 if x<7 && y<7 and (not legalcheck or ([x+1,y+1] is not empty and is a black piece)  
 movableSpots[x+1,y+1] true  
 if x>0 && y=4 and fieldPiece[x-1,y] is not empty and is black and it canPassant  
 movableSpots[x-1,y+1] true  
 if x>0 && y=4 and fieldPiece[x+1,y] is not empty and is black and it canPassant  
 movableSpots[x+1,y+1] true  
**if type = “N”**  
 for i = -2 to i = 2  
 if i = 2 && x < boardSize – 2 or I = -2 && x > 1  
 if y>0 && fieldPiece[x+i, y-1] = null or fieldPiece[x+i,y-1] != null and is enemy  
 fieldPiece[x+i, y-1] = true  
 if y<7 && fieldPiece[x+i, y+1] = null or fieldPiece[x+i,y+1] != null and is enemy  
 fieldPiece[x+i, y-1] = true  
 if i = 1 && x < boardSize – 1 or I = -1 && x > 0  
 if y>1 && fieldPiece[x+i, y-2] = null or fieldPiece[x+i,y-2] != null and is enemy  
 fieldPiece[x+i, y-2] = true  
 if y<7 && fieldPiece[x+i, y+2] = null or fieldPiece[x+i,y+2] != null and is enemy  
 fieldPiece[x+i, y+2] = true  
 end for

MOVINGRULES

**if type = “B”** for i = x+1, j= y+1 to i to boardSize and j to boardsize i++, j++  
 if fieldPiece[i,j] is empty, movableSpots[i,j] = true  
 else if fieldPiece[i,j] is an enemy piece  
 movableSpots[i,j] = true  
 break  
 else break  
for i = x+1, j= y-1 to i to boardSize and j to boardsize i++, j--  
 if fieldPiece[i,j] is empty, movableSpots[i,j] = true  
 else if fieldPiece[i,j] is an enemy piece  
 movableSpots[i,j] = true  
 break  
 else break  
for i = x-1, j= y+1 to i to boardSize and j to boardsize i--,j++  
 if fieldPiece[i,j] is empty, movableSpots[i,j] = true  
 else if fieldPiece[i,j] is an enemy piece  
 movableSpots[i,j] = true  
 break  
 else break  
for i = x-1, j= y-1 to i to boardSize and j to boardsize i--, j--  
 if fieldPiece[i,j] is empty, movableSpots[i,j] = true  
 else if fieldPiece[i,j] is an enemy piece  
 movableSpots[i,j] = true  
 break  
 else break

**if type = “R”** for i = x+1 to boardSize i  
 if fieldPiece[i,y] is empty, movableFields = true  
 else if fieldPiece[I,y] is an enemy piece  
 movableSpots[i,y] = true  
 break  
 else break  
for i = x-1, to boardSize i--  
 if fieldPiece[i,y] is empty, movableFields = true  
 else if fieldPiece[i,y] is an enemy piece  
 movableSpots[i,y] = true  
 break  
 else break  
for j= y+1 to boardsize j++  
 if fieldPiece[x,j] is empty, movableFields = true  
 else if fieldPiece[x,j] is an enemy piece  
 movableSpots[x,j] = true  
 break  
 else break  
for j = y-1 to boardSize j--  
 if fieldPiece[x,j] is empty, movableFields = true  
 else if fieldPiece[x,j] is an enemy piece  
 movableSpots[x,j] = true  
 break  
 else break

MOVINGRULES

**If type = “Q”** bishop code + rook code  
**if type = “K”** for i = -1 to 1  
 if i=-1 && x >0 or i = 1 && x < 7 or i = 0  
 if y>0 && (fieldPiece[x+i, y-1] is empty or opposite colour) movableSpots[x+i,y-1] = true  
 if (fieldPiece[x+i, y] is empty or opposite colour) movableSpots[x+i,y] = true  
 if y<7 && (fieldPiece[x+i, y+1] is empty or opposite colour) movableSpots[x+i,y+1] = true  
 end for  
 if (grabbedPiece.moves = 0)   
 if fieldPiece[7,y] !=null && fieldPiece[7,y].type ==”R” && fieldPiece[7,y].moves= 0 and fieldPiece[x+1,y] && fieldPiece[x+2,y] are empty then movableSpots[x+2,y] = true  
 if fieldPiece[0,y] !=null && fieldPiece[0,y].type ==”R” && fieldPiece[0,y].moves= 0 and fieldPiece[x-1,y] && fieldPiece[x-2,y] && fieldPiece[x-3,y] are empty then movableSpots[x+2,y] = true

If legalcheck = true IsMoveLegal()

One main method within the **Board** class is **MovingRules**. It’s responsible for checking the places that the piece stored in **grabbedPiece** can go. It is called throughout the code when the moving positions of a piece is needed. Within the function, it contains another function called **isMoveLegal.** It is called when checking whether the king will be in danger after the move and is not called when the fields the piece is attacking irrelevant of king safety is needed. This is used for finding out whether a move is legal for the king since you can’t move a king into check even if the piece attacking will reveal their own king in to check.

For pawns, I first check that it is not on the first rank (0) or the last rank (7) since that would mean the pawn would be on its last possible move, meaning it wouldn’t have a move and would need to be promoted. I then check the piece’s colour since pawns move differently depending on which colour they are. For white, I check whether the space in front of them going up the board is empty, if so make it a move. I then check the 2 diagonals on the sides. If they have an enemy piece, I set that field as a move. I also check that if the pawn has not moved before, I allow it to move forward 2 squares if both are empty. I also check for en passant. This is done by having the variable **enPassant** from the **Pieces** class be set to true. Just before a turn finishes executing, each pawn is checked and if it has at all moved before, its **enPassant** is set to false. This abides with the rules of chess since if a pawn has moved, it must be captured with en passant on the next turn. By removing the ability to en passant the enemy pawn after it has moved and a turn has passed, it means that it won’t be captured that way later. I also check that the enemy pawn is not only en passantable, but has only moved once and is on the right rank. Otherwise, if a pawn were to move only 1 square forward, ending up next to an enemy pawn, it would still be able to capture it with en passant since it is next to it and **enPassant** is true. The check for the right rank and turn count prevents that. I repeat this process for black, but instead of subtracting from y (moving down the board) I add to y (moving up the board).

For knights, I used iteration to shorten the code rather than checking each one of the 8 possible moves separately. It iterates through -2 to 2 with 1 & 2 representing the knight going horizontally either 1 or 2 to the right or the left. From there, check the vertical displacement by 1 or 2 or -1 or -2 for each of the x coordinates. This would leave in total pairs of displacement – (-2,-1),(-2,1),(-1,-2), (-1,2), (1,-2),(1,2),(2,-1),(2,1). These are all 8 possible knight displacement. In each if statement, the x coordinate and y coordinate is checked so it does not exceed the board array when checking the move. Once again, if empty, it is a possible move and if not, make sure it is an enemy piece by comparing the grabbed knight’s colour to the piece in the way.

For the Bishop, I also utilize iteration, however in its case it can not pass through piece like the Knight, so a set of coordinates is not possible. Instead, x&y are iterated, adding or subtracting to either. Another way of putting this is that as for example x increases, y could also increase causing a South West movement and adding x, subtracting y leads to a North West movement. I iterate these until the edge of the board for each 4 directions. If the spot selected with i,j is empty, it is a movable spot. If a piece is on the spot however, check if it is an enemy. If it is, make it a movable spot and then break the loop since the bishop can’t pass through the piece to the squares behind. If it is not an enemy, just break the loop there.

For the Rook it is nearly the same as the bishop, however only y or x is iterated in each loop, since it only has horizontal and vertical movement (x for the horizontal, y for the vertical).

The Queen is a carbon copy of the Bishop code followed by the Rook code.

The King’s first half of code is coded similarly to that of the knight, with 8 permanent preset possible moves to check, and to save lines of code, iteration is used to cycle through them with i iterating through -1 to 1 to represent displacement of to the left, none or to the right and then checking the vertical displacement for each horizontal one. The location of the piece is checked when checking displaced fields to ensure the check is within the board and for each value of i a -1, +0, +1 value of y is tried to represent all adjacent squares adjacent to the king, including the king itself (however since there is a piece that is not an enemy, it will not count itself as a move). Once again, if a field is empty or an enemy piece is present, the spot is a movable one. For the second half of the king’s code, castling is implemented. It checks that the king has 0 moves with the initial if statement and then checks for a kingside rook (on the right) on the same horizontal. If the rook is in the right place and has not moved either (to adhere to the rules of chess) and all the fields in between the king and rook are empty, then it is a possible castle. Then the same is done for the queenside rook (on the left).

After this, the **IsMoveLegal** method runs if it is the first run of **MovingRules** since legalCheck is set to true at the start.

ISMOVELEGAL

Int X = grabbedPiece.x  
int Y = grabbedPiece.y  
Pieces backupPiece = grabbedPiece  
bool[,] backupSpots = movableSpots.Clone()  
Pieces[,] backupBoard = fieldPiece.Clone()  
for i = 0 to boardSize  
 for j = 0 to boardSize  
 if backupSpots[i,j] = true  
 fieldPiece[i,j] = backupPiece  
 fieldPiece[X,Y] = null  
 foreach piece in fieldPiece  
 if(piece != null and piece.colour =! backuPiece.colour

ISMOVELEGAL

grabbedPiece = piece  
 MovingRules()  
 if KingSafe(backupPiece, i) is not true  
 backupSpots[i,j] = false  
 break  
 fieldPiece = backupBoard.Clone()  
movableSpots = backupSpots.Clone()  
grabbedPiece = backupPiece  
legalCheck = true

**IsMoveLegal checks** each possible move that was calculated within **MovingRules** and checks whether it is, in chess terms, legal. This means that the king is not under check after the move is finished (or for the case of castling, while moving). It does this by firstly setting X and Y parameters of the piece to have a variable for them stored since we will need to clear the original piece’s square, so its coordinates are needed. Next, a copy is made of the board, the grabbedPiece and the movableSpots. This is because these will be now reused with the **MovingRules** function. I am using recursion to make the code more efficient with less lines of code used. Once copies have been made, I iterate i and j for the whole board worth of coordinates and check whether a square is a movable spot from **MovingRules**. If it is, I need to check whether it’s legal. So I move the piece on the board by changing it’s coordinates to the movableSpot’s and setting the previous square as empty. Then, I cycle through each piece on the board, and if it is an enemy piece, I set it as a grabbedPiece and run **MovingRules** to get all the places it threatens. This time, legalCheck is off however, so it will not recurse **MovingRules** again. It also means that pawns will display sideways captures as possible moves since they threaten those squares. We do not need to check whether the enemy moves are legal because these are simply squares that the enemy are threatening and not those that they can go to. This fits the rules of chess since chess does not require the enemy move to be a legal to threaten squares. Once it has generated the possible moves for that piece, **KingSafe** is run where I check whether the king’s location is under check (or for castling move if any of the fields it travels through are). If the king would be under check, this move is illegal and the method returns false. This causes the move to be illegal and is set to false in **backupSpots.** The loop is broken since we no longer need to prove that that move is legal for other enemy pieces. This is to save processing time. After checking one enemy piece, the board state is reversed back to repeat the process for different possible moves. Afterwards, all the variables are returned to their previous states by cloning their backups back into them.

This leaves us with a finished **movableSpots** array with all the fields that the **grabbedPiece** can legally move to. This can be used to tell the user where they can go as the board already does, or for the engine to calculate possible moves.

An important method worth covering before moving onto the engine is the **ExecuteTurn** method. It is a part of the **Board** class and is executed within the **ExecuteGUITurn** method. It is responsible for performing the board coordinate changes.

EXECUTETURN(X,Y,ChessBoard)

clearPassant()  
grabbedPiece.moves++  
if grabbedPiece.type = “P”  
 enPassantCheck(X,Y,ChessBoard)  
 drawBy50Count = 0  
if fieldPiece[X,Y] is not empty pieceCapture(X,Y,ChessBoard)  
if (grabbedPiece.type = “K”) CastleCheck(X,Y)  
if turn = “White” set turn = “Black”  
else turn = “White”  
grabbedPiece.x = X  
grabbedPiece.y = Y  
SetPiecesToArray()

The method starts off by clearing all pawns with at least 1 movement from being able to be capture by enPassant and then adds a move to the piece that is moving. (This was discussed earlier for **MovingRules**). Then, we check if the move is that of a pawn since if it is an en passant move, we would need to clear the enemy pawn that would be behind it. Draw by 50 count is reset to 0 since a pawn move resets the counter for this draw. Then we check for normal captures by seeing if the field the grabbedPiece is moving to taken up. If so, it is a capture and **pieceCapture** resets the drawBy50Count as well as removing the enemy piece completely. Then we check whether the move is a castle if the moving piece is a king. This is done by seeing whether the king is displaced by 2 fields, meaning that it is not a normal move and therefore a castle. Then the rook and king and moved within that subroutine respectively. Then the turn is inverted and the coordinates of the moved piece are updated to the coordinates it’s moving to (X,Y), passed in from before. The final line, **SetPiecesToArray** simple goes through each piece and compares whether it’s stored coordinates as the **Piece x & y** values match up to that of the indexes of the array (coordinates on the board array). If they are not, they are moved to the right index within the array. If this was not run, the grabbedPiece’s coordinates would be the new x & y but it wouldn’t be stored at x & y on the board.

# The Engine

The engine is stored within the **EngineFunctions** class. Here it is made up of 8 methods. An instance of the class is created when **ExecutePCTurn** in **PlayerBoard** is successfully run.

EXECUTEPCTURN(ChessBoard)

If (ComputerOponent is true and PlayerColour != turn)  
 New Instance of EngineFunctions = EngineGame  
 PlayerBoard gameBoard = this  
 EngineGame.PerformPCMove(GameBoard,ChessBoard)  
 SaveMove()  
 AllignPieces()  
 grabbedPiece = null  
 PerformGameEnd()

At the start, we check whether an engine move is necessary, if so, we create an instance of the class calling it EngineGame. Then we make a reference variable of the class the code is running in, **PlayerBoard**, the one where the player is playing against the engine. This will allow the engine to modify the code within its class. Then, we execute the Engine turn. Finally, we Save the move so it the player can undo, align the pieces so they are in the right place visually, set the grabbedPiece to nothing since no piece is grabbed after the engine did its turn and then check if a checkmate or draw is achieved to then execute the necessary code for the user. **ChessBoard** is passed so the Canvas can be accessed to user the slider value of the depth for the engine.   
The class by default has 3 variables set as boardSize = 8, EngineTurn = “Black” and PlayerTurn “White”.

PERFORMPCMOVE(ChessBoard)

PlayerTurn = GameBoard.PlayerColour  
if(PlayerTurn = “Black”) EngineTurn = “White”  
int X = 0  
int Y = 0  
SimpleEngine(GameBoard, ChessBoard,ref X, ref Y)  
PawnConversion(GameBoard, ChessBoard)  
GameBoard.executeTurn(X,Y,ChessBoard)

First, the **PlayerTurn** colour is set to that of which they selected in the start menu. If it is white, then the EngineTurn is already black by default, else it will switch to white. We declare the variables X & Y since these will be the coordinates of the move that the engine will make. Then SimpleEngine is run, with X and Y referenced in so they are changed throughout the code, ending up updated at the end. **PawnConversion** is then executed to convert any pawns at the end and then the turn is executed to the passed in coordinates.

SIMPLEENGINE(GameBoard, ChessBoard, ref X, ref Y)

int nodecount = 0  
int RX = 0  
int RY = 0  
Piece RgrabbedPiece = null  
Board BoardCopy = GameBoard.DeepCopyBoard()  
int depth = sliderValue in ChessBoard  
int maxdepth = depth  
Max(BoardCopy, ChessBoard, ref RX, ref RY, ref, RgrabbedPiece, depth, maxdepth, -1000000000, 1000000000 ref nodeCount)  
X = RX  
Y = RY  
int pY = RgrabbedPiece.x  
int pY = RgrabbedPiece.y  
GameBoard.grabbedPiece = GameBoard.fieldPiece[pX, pY]

First, main variables are declared in this method. **Nodecount** is a debug variable that counts the amount of nodes done. This can be used to modify the code and see how it affects the amount of nodes searched, improving efficiency. **RX** and **RY** are the **X** values and **Y** values that will be modified by the MinMax algorithm to represent the coordinates of the move to be made and the **RgrabbedPiece** will be the piece that will do the move. A copy of the board is made, but not referenced, into the method so it can be recalled at all times and can be edited freely within the algorithm. The depth is set based on what the user selected and that is passed into the method too. The 2 large values represent alpha and beta, used for alpha-beta pruning to reduce processing times by a significant amount. After the method is done, the X and Y values are set to those the engine wants to move to and the coordinates of the selected piece are also set so we can reference them to the equivalent **GameBoard** piece and set it as its **grabbedPiece**.

The method starts by checking whether we have reached final depth, depth 0. If so, I need to return the value of the board state. This is done with the **BoardAnalysis** method which gives a score to the inputted **BoardCopy** **Board** by counting its pieces and giving relevant scores to each piece. Pawn = 100, Knight = 320, Bishop = 325, R = 500, Q = 975, K = 32767 (arbitrary). 5 sets of tables are also used which map all the places on the chess board and add a positive or negative score to a board depending on where the piece is. For example, a knight on the edge would reduce the score. For the king, 2 of these tables exist. One for the main game and one for the endgame, which is used when there are only 2 minor pieces worth of material on the board. A Pawn doesn’t count as a minor piece and a queen counts as 2 within the code.   
Next, we check for a checkmate. If one is accomplished, return a very low score since this means the max has lost.   
Then a backup of the board is made to restore it later.  
Max is set to an extremely low value so any move is better and gets updated as the new max.  
**ReturnPieces** makes a 1D array referencing all the pieces that are on the board of the Max’s colour. These are used to calculate possible moves.  
Then, I iterate through each piece of the Max’s colour. I make the piece the **grabbedPiece** of **BoardCopy** (where the engine simulates the game).   
**ReturnMoves** is then executed for the **BoardCopy** and its **grabbedPiece**. This method calls **MovingRules** but stores the results differently, as to prevent direct piece referencing. I created a 2D array called **moves**, which has 2 values for its first index and however many movable spots there are for the second index. The first index of 0 e.g. [0,x] refers to an X coordinate and a first index of 1 e.g. [1,x] refers to a Y coordinate. This means that I can set the X and Y coordinates of the place we want to directly. Referencing the movable spot could cause the piece to move somewhere else as the code changes, but coordinates ensure that the piece moves to the right square each time.   
Now the turn is executed with the X,Y coordinates and a **PawnConversion** is done. This one is within the **EngineFunctions** class since the engine auto promotes to a queen. An improvement I can make is to make the engine pick the best promotion, however the queen is mostly the best choice.   
Now I call the **Min** method, which is exactly the same as the **Max** method but with inverted values for min and without the if statement at the end to save the best move. However, one difference is that I call the method with a depth decreased by one. This is to prevent the recursion from infinitely looping and eventually reaching a depth of 0, returning a board score. After a depth of 0 is reached, the previous instance of Min/Max on the stack receives its **score**. First thing that it will do now is update **alpha** (or **beta** in **Min**). This is the best achievable score on the branch it is taking. It is used to prevent searching trees that are redundant. This is called alpha-beta pruning. The best Min value (**beta**) is stored and **alpha** (the best **Max** value) is compared to it. If **Min** has a better value, it would be pointless to further search the branch since there is no way that **Min** would pick it since it already has a better score saved (**beta**). Therefore, in the code, if alpha is less than beta, the score is instantly returned, avoiding further redundant recursion, shaving off a lot of nodes that otherwise would be searched.  
Next, the BoardCopy is returned to the previous state that it was within its method (**Backup**) and the **grabbedPiece** is set as the previous one. It uses coordinates rather than direct referencing, once again, since the piece we are directly referencing could have been deleted as a part of **ExecuteTurn** and could no longer exist. Direct coordinates ensure we access the same piece since in the backup, the coordinates of the piece are the same.   
Finally, if the score is better than the current best score (**max** or **min**), set it as the score.

MAX(BoardCopy, ChessBoard, ref RX, ref RY, ref RgrabbedPiece, depth, maxdepth, alpha, beta, ref nodecount)

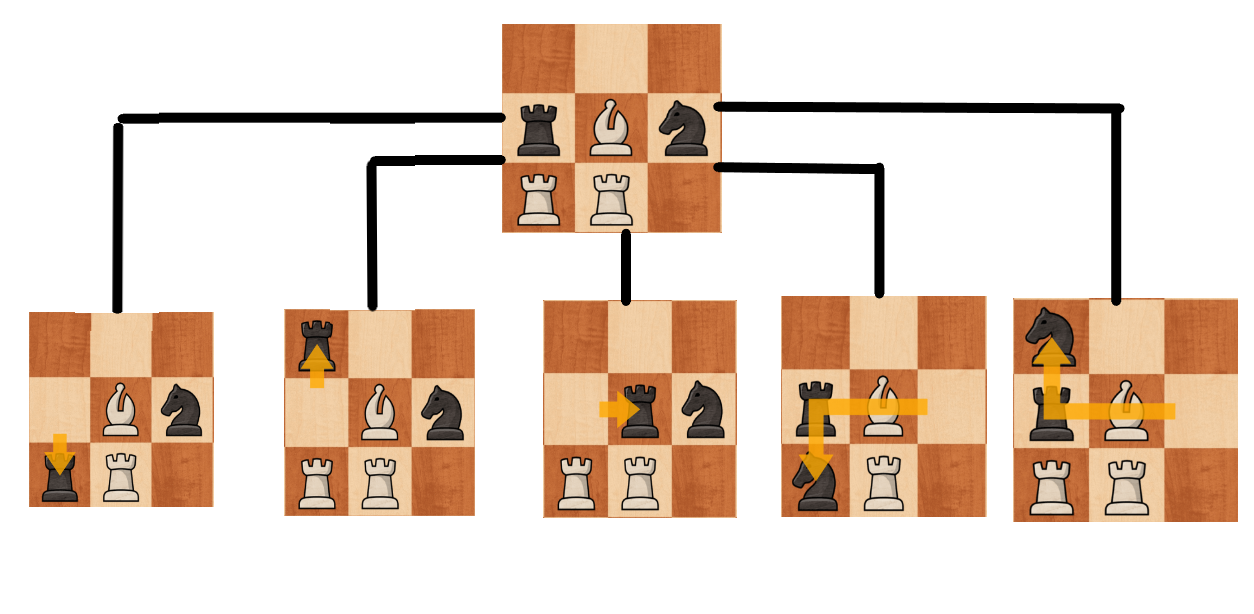
if depth = 0 return BoardAnalysis(BoardCopy, ref nodeCount)  
if BoardCopy.GameStatusCheck() == 2 return -100000000  
Board Backup = BoardCopy.DeepCopy()  
int max = -1000000000  
Pieces[] pieces = returnPieces(BoardCopy, EngineTurn)  
foreach piece in pieces  
 BoardCopy.grabbedPiece = BoardCopy.fieldPiece[piece.x, piece.y]  
 int[,] moves = ReturnMoves(BoardCopy)  
 for i = 0 to (moves.Length / 2) – 1   
 int X = moves[0,i]  
 int Y = moves[1,i]  
 BoardCopy.executeTurn(X,Y,ChessBoard)  
 PawnConversion(GameBoard,ChessBoard)  
 int score = Min(BoardCopy, ChessBoard, ref RX, ref RY, ref RgrabbedPiece, depth, maxdepth, alpha, beta, ref nodecount)  
 if score > alpha alpha = score  
 if beta < alpha return score  
 BoardCopy = Backup.DeepCopy()  
 BoardCopy.grabbedPiece = BoardCopy.fieldPiece[piece.x, piece.y]  
 if score > max  
 max = score  
 if depth = maxdepth  
 RX = X  
 RY = Y  
 RgrabbedPiece = piece.Copy()  
return max

For **Max**, and if statement at the end is there that is executed on the first instance of **Max**. It is the one that decides the best move. Whenever a new **max** is achieved, the coordinates of the move made and the piece that made it are saved as **RX**, **RY** and **RgrabbedPiece**. This is not necessary for **Min** since it does need to output a best move, only the best board positions. Since the 3 variables are referenced, they are updated and will be used in **PerformPCMove** as shown earlier.

To demonstrate this, I will set up a hypothetical board situation and go up to depth 2 explanation of how the algorithm would manage it.

So the tree diagram I have created below starts off at black’s turn and has 5 possible moves. In the current example, the engine is playing at depth 1, and therefore will only execute Max and fetch a score from Min.

Moves are fetched and executed, updating the board state that is being worked with.



-505

-5

-180

-505

-5

The boards are evaluated by a method, and their values are returned.

Initial Board Position

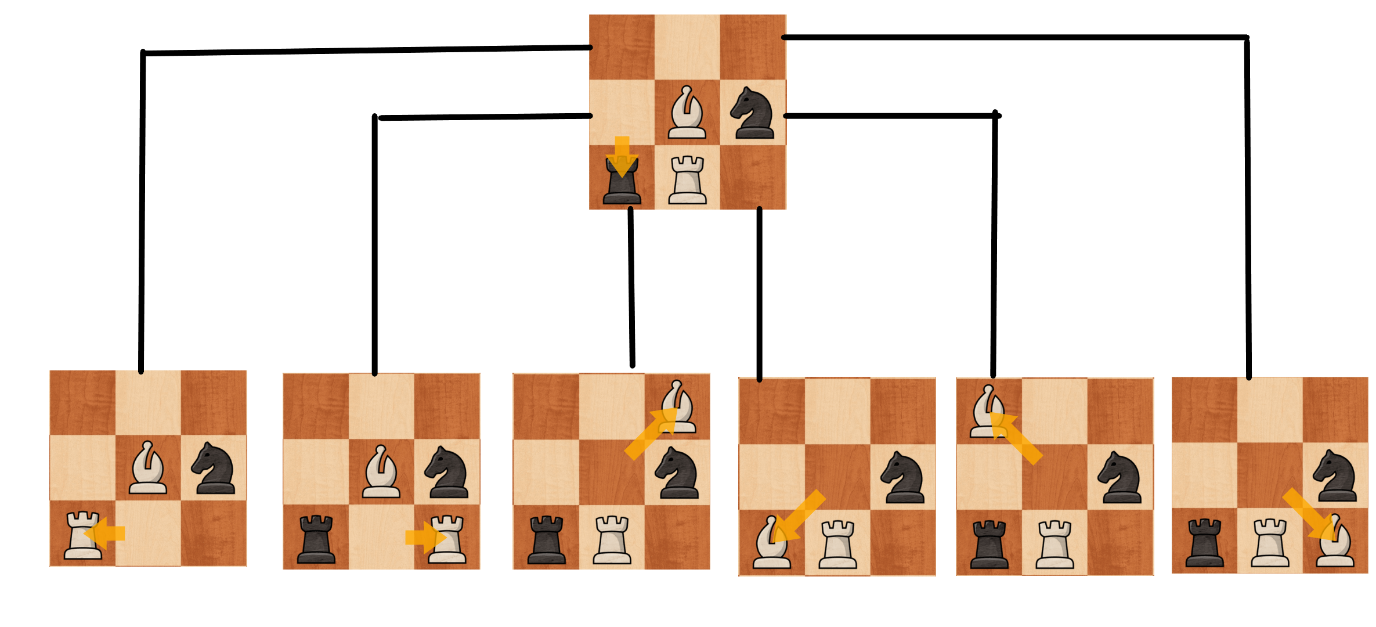
Max is updated to -5, since -5 is better than -10000000 (the value assigned to max, so any move is better than no move). No further board states are better than -5, so it is never changed.

In terms of my code, this would mean that in the Max method, score would change from -5 to -550 to -180 to -5 to -180. The first board’s score is -5, this being updated as the current max. Since the max has been updated of the root Max, the moves are saved and **RgrabbedPiece** becomes the black rook while RX and RY become 0 and 2 (top left is 0,0).

Since **max** is never updated after this board state, the piece to move and its move coordinates are not updated, returning Rook to 0,2 as the best move to **SimpleEngine**.

Max passed in depth of 0 to Min and causes the board to evaluate. As stated before, the pieces’ values are Pawn = 100, Knight = 320, Bishop = 325, R = 500, Q = 975, K = 32767 and the board position tables are irrelevant in this example. So for example, in the first board, the evaluation algorithm would run through the bishop, subtracting 325, then adding 320 for the knight, then adding 500 for the rook then subtracting 500 for the rook, leading to a total of -5.

Now if we decide to add depth 2, matters get a bit more complicated.

Let’s walk through depth 2 from the first board state.

Returned Min: -505

-5

-5

-505

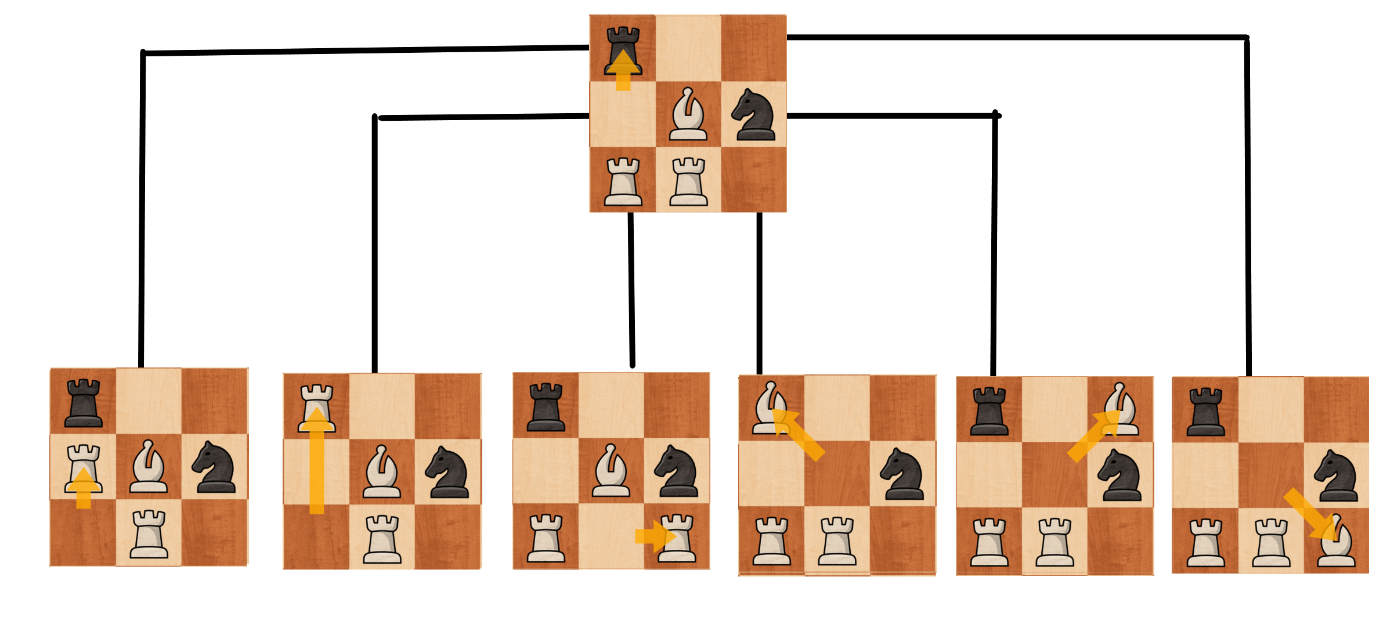
-5

-5

-505

In this tree representation of the code, very similarly to Max, all moves of the Min player are fetched and executed on the board. Since we are at depth 2, when calling Max again, an evaluation is performed. However, this time we are not saving the best moves, since we do not need to store the moves of the Min / Max from this point. We simply update the Min with the best score that can be achieved. In the first position, a score was evaluated of -505, this is new **min** score. **Beta** is also updated to this score but is irrelevant since no further depth is explored. No further board states beat -505, so **min** is updated to -505. This score will be the new returned score to the Max since Min knows they can guarantee themselves this score if Max makes the move.

If we look back, in depth 1 this would usually return -5 but clearly, the compute can make a move that then makes the score worse.This instance of Min now has had its base call performed and Max is returned to. The **max** value is updated to -505 (since that score is better than the initialised **max** score) and **Alpha** is updated to -505 (this CAN have an effect, since another depth is being explored).

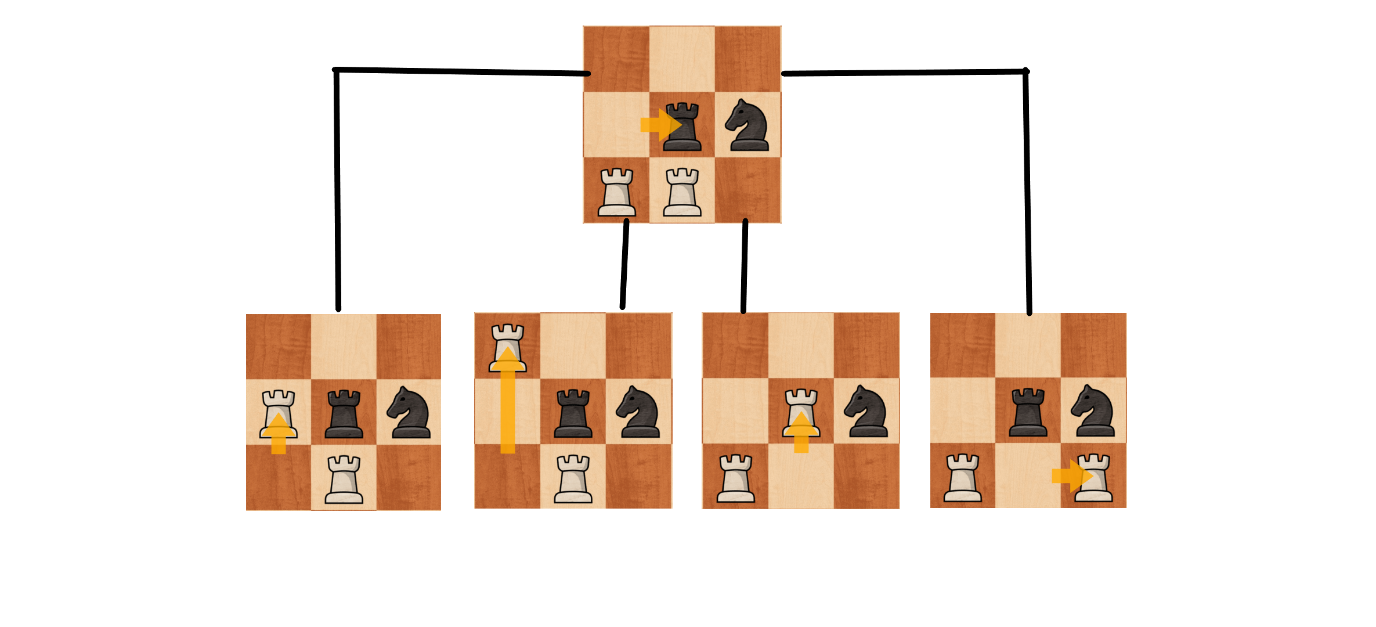
The next move is now passed through to Min:

Returned Min: -1005

-1005

-505

In this example, we can see that some evaluations are simply not calculated. And this is due to Alpha-Beta pruning. Like before, Min finds every move for the first piece it grabs. In this case it is the rook at (0,2). It tries out the first move, getting a score of -505. **Beta** and **min** is updated to -505. However, for the next move of the rook, the score is -1005; this is an even better score. **Beta** is updated to -1005. However, now when **Beta** is compared to **Alpha**, **Beta** is smaller. This is a base call to the current instance of Min, returning the score instantly, without calculating further pieces and their moves. The reason this can be done is because in the previous move calculation for Max, Max calculated that it can get at least -505. The second move for Max here however, allows Min to gain -1005, and it is assumed the Min would take this move in the game. Max would never choose -1005 over -505 and Min will always choose -1005. So no matter how bad or good the further board positions are in this current instance, Max will never pick it since Min is already guaranteed a score worse for Max than Max can already get.



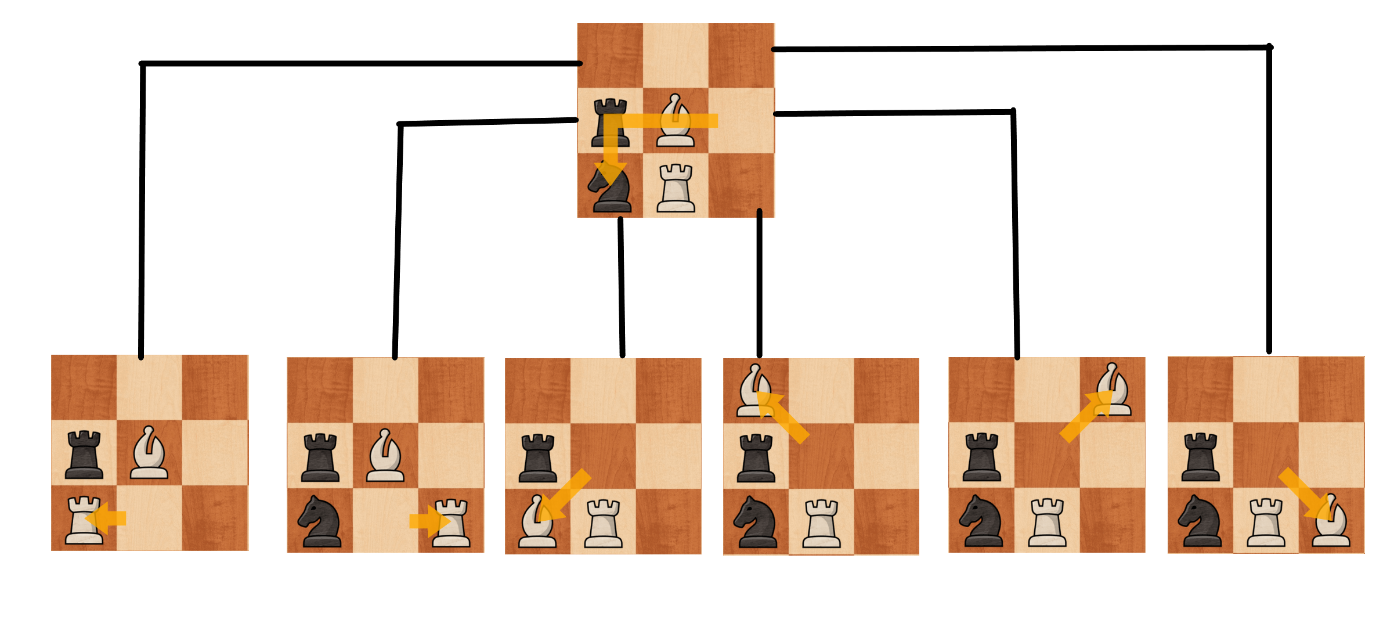
Returned Min: -680

-680

-180

-180

Similarly, here, after calculating moves for the second rook and attempting one, a score worse than Alpha is found, meaning it is redundant to explore this node any further since, once again, Max would never choose this node no matter what.



Returned Min: -325

-5

-5

-5

-325

-5

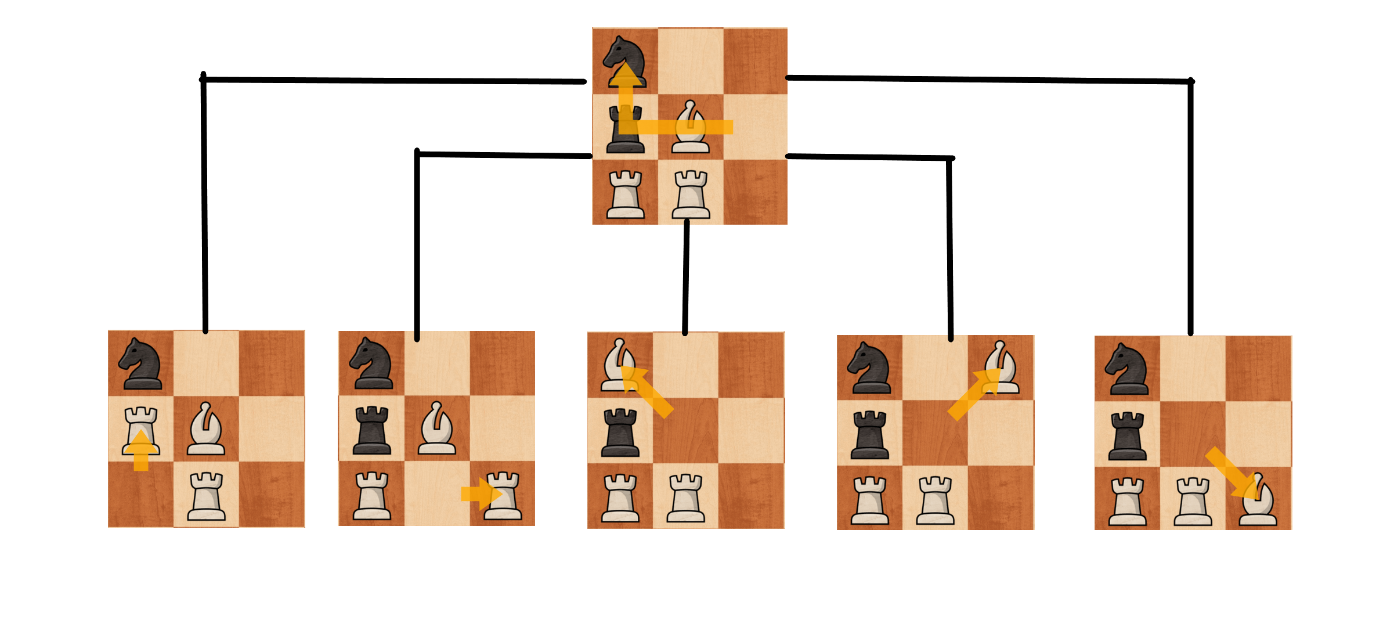
-325

In this tree, the highest score for Min is -325, so this is the score that will be returned. Since none of the scores go below alpha, no pruning can be done here and the whole node is explored. Back in Max, **alpha** and **max** is updated to -325 since it is an improvement on -505.

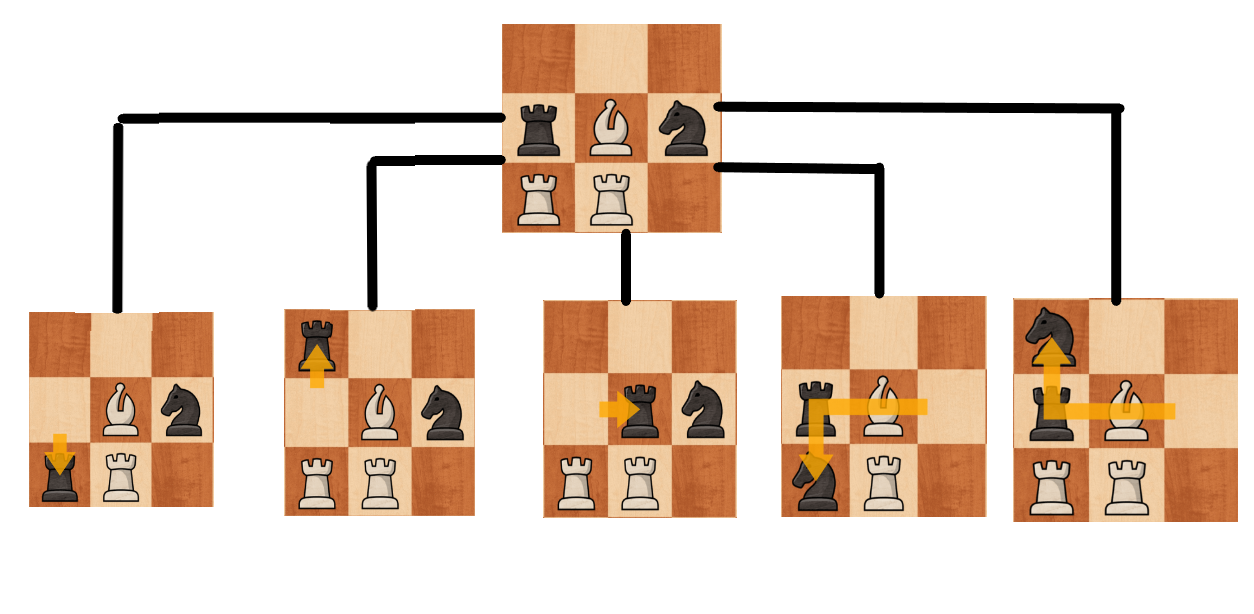
And the final move for Max:

Returned Min: -1005

-1005



We can see here, that due to alpha beta pruning, we have saved a lot of search time, since only 1 move position is needed to be searched in the entire node.

Now if we revisit the original tree, if we use the values returned my Min after assessing its moves, we can see the scores vary much more.

-680

-1005

-1005

-325

-505

In this scenario, Max would’ve updated it’s **max** at the 4th board state to -325 since it is an improvement form -505. The **RgrabbedPiece** would get updated to the knight and **RX** & **RY** will be 0 & 2. As depth is increased, more of the paths can be explored and more accurate scores can be represented as Min & Max get to choose their moves and pass the best score they can achieve to each other. With a simple increase from depth 1 to depth 2 a much better move is found. Alpha Beta’s pruning effect is also demonstrated well with these diagrams since we can see the amount of nodes that have been purged due to the scores. In a bigger scale depth and board, this effect is ever more important and is why it is implemented.

# Board Saving & Loading

Loads and Saves are loaded using FEN notation. Saves are done with the method called **SaveFEN()**.

saveFEN(string fileName)

int emptyCounter = 0  
string FEN = “”  
string pieces = “”  
string saveturn = turn.Substring(0,1).ToLower()  
string whiteCastle = “”  
string blackCastle = “”  
string totalCastle = “”  
string passant = “”  
for y = 7 to 0  
 for x = 0 to 7   
 if fieldPiece[x,y] is empty, emptyCounter++  
 else   
 if(emptyCounter is not 0, pieces += emptyCounter  
 emptyCounter = 0  
 if pieceColour is Black, pieces += pieceType.ToLower()  
 else pieces += pieceType.ToUpper()  
 if pieceType = “K” and it’s moved, and pieceColour is white, whiteCastle = “-“  
 elseif pieceType = “K” and it’s moved, and pieceColour is black, blackCastle = “-“  
 elseif pieceType = “R” and it hasn’t moved and pieceColour is white and whiteCastle != “-“ & X = 0, whiteCastle += “Q”  
 elseif pieceType = “R” and it hasn’t moved and pieceColour is white and whiteCastle != “-“ & X = 7, whiteCastle += “K”  
 elseif pieceType = “R” and it hasn’t moved and pieceColour is black and blackCastle != “-“ & X = 0, blackCastle += “q”   
 elseif pieceType = “R” and it hasn’t moved and pieceColour is black and blackCastle != “-“ & X = 7, blackCastle += “k”  
 if pieceType = “P” and canPassant and hasMoved and y=3 or 4  
 passant += convert x coordinate of char  
 if colour is white, passant += 3  
 else passant +=6  
 endfor  
 if emptyCounter is not 0, pieces += emptyCounter  
 if y is not 0, pieces += “/”  
 emptyCounter = 0  
endfor  
if passant = “”, passant = “-“  
if whiteCastle = “QK”, whiteCastle = “KQ”  
if blackCastle = “qk”, blackCastle = “kq”  
if whiteCastle = “”, whiteCastle = “-“  
if blackCastle = “”, blackCastle = “-“  
if whiteCastle and blackCastle = “-“ then totaCastle = “-“  
else if whiteCastle = “-“, totalCastle = blackCastle  
else if blackCastle = “-“, totalCastle = whiteCastle  
else totalCastle = whiteCastle + blackCastle  
FEN += pieces + “ “ + saveturn + “ “ + totalCastle + “ “ + passant + “ “ + drawBy50Count + “ “ + fullMove  
 Write FEN to file called fileName.

In this pseudocode, we scroll through every square (starting from y = 7 since data is read in from white’s perspective). Every time there is an empty square, 1 is added to the **emptyCounter** since we want to store consecutive empty spaces as a single number representing the spaces (almost like RLE). If there isn’t an empty space, the number of empty spaces stacked from before is attached and then the piece type is attached, case dependant on colour. Then, depending on King moves and Rook moves, castling permissions for King side and Queen side are assigned for both sides. Then, for en passant, I assign the square that can be captured by en passant. I do this by selecting the piece that can be captured by checking if its **enPassant** variable is true and that it lies on a y coordinate 2 away from start (double move) and that it has moved. This guarantees that the piece can be captured by en passant. Then I change the x value of the piece to an ASCII value to represent the x axis notation of the board. If the piece is white, then y of the square getting captured has to be 3 (from chess board notation) and black would have y as 6. After the X loop, in case all 8 squares were empty, empty squares are added on. A dash is added on (unless it’s the last line) to represent a new line. The **emptyCounter** is then reset.

After all pieces have been added, a series of if statements rearrange castle strings into proper order and combines them into a single string. Then all the needed factors for FEN notation are attached to the FEN string and the FEN string is written into a text file with the name as the parameter of the subroutine.

The next function is the **LoadFEN()** function that loads an FEN file onto the board.

loadFEN(string loadName)

string loadData = Read in from File named ‘loadName’.  
string[] loadParts = loadData split  
string[] loadPieces = loadParts[0] split by ‘/’  
int skipCount = 0  
int xCounter = 0  
foreach char letter in loadParts[2]  
 if letter = ‘K’ wKingRook = true  
 if letter = ‘Q’ wQueenRook = true  
 if letter = ‘k’ bKingRook = true  
 if letter = ‘q’ bQueenRook = true  
for int y = 7 to 0  
 for x = 0 to 7  
 if skipCount = 0  
 char[] rowPieces = loadPieces[7-y] into char Array  
 string currentPiece = rowPieces[xCounter].ToString()  
 if trying to parse currentPiece as an int works, output it as skipCount and skipCount --  
 else   
 create new fieldPiece = fieldPiece[x,y]   
 fieldPiece[x,y].type = currentPiece.ToUpper()  
 if currentPiece is uppercase, set fieldPiece[x,y].colour to white  
 else set it to black  
 fieldPiece[x,y].x = x  
 fieldPiece[x,y].y = y  
 if x = 0  
 if fieldPiece is a rook and white and wQueenRook is false, fieldPiece.hasmoved =1  
 if fieldPiece is a rook and black and bQueenRook is false, fieldPiece.hasmoved =1  
 if x = 7  
 if fieldPiece is a rook and white and wKingRook is false, fieldPiece.hasmoved = 1  
 if fieldPiece is a rook and white and wKingRook is false, fieldPiece.hasmoved = 1  
 end if  
 xCounter++  
 else skipCount--  
 end for  
 xCounter = 0  
end for  
if loadParts[3] is not “-“  
 convert ascii into x coordinate  
 if loadParts[3][1] = 6  
 fieldPiece[x,3].canPassant = true  
 fieldPiece[x,3].hasmoved = 1  
 else  
 fieldPiece[x,4].canPassant = true  
 fieldPiece[x,4].hasmoved = 1  
 end if  
end if  
if loadParts[1] = “w”, turn = “White”  
else turn = “Black”  
drawBy50Count = to int loadParts[4]  
fullMove = to int loadParts[5]

In the **LoadFEN()**, I start by splitting apart the read in string by its spaces. This leaves me with the 6 parameters of FEN. The first, the pieces, I split first by the dashes to represent each row. From there, I loop (y starting at 7 again since it’s from white’s perspective) and check for any skips. If there are, adding pieces is skipped. When checking, a separate counter **xCounter** is used for each character in the FEN row splits. This is because there is not always 8 characters used to represent each row. For example, a half empty row with 4 pawns would be 4PPPP which is only 5 characters. Therefore, this is only incremented when a piece is added or a single number of skips is read. It acts as a pointer to the characters while the X in the iteration acts as a pointer to the location on the board. The characters are checked, if it is a number, piece isn’t added and the number is loaded as a **skipCounter**. It is minus oned since it already skipped one cycled while being read. From there, **xCounter** is incremented and the character is reading is skipped until all **skipCounts** have been cycled through. If it isn’t a number, the piece is added and the pieces’ colours are calculated by the fact whether the characters were upper case. Then, if the piece added is a rook, it is checked whether castling is possible with that rook. If it isn’t a move is added to it so castling couldn’t happen.

After the pieces have been added, en passant is added by setting the pawn above the en passant square as en passantable with **canPassant** and **hasmoved** is set to 1 so it will no longer be able to en Passant next turn.

Finally, turns are set to whoever’s turn is stored, the **drawBy50Count** (half move clock) is copied over and the full move clock is copied to **fullMove**.

Testing

<https://youtube.com/playlist?list=PLiPAlBhLM4mcMOfqJLFh6h4f3gF_rMCUH>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Description | Input Data | Expected Outcome | Timestamp | Result |
| 1 | Test that the game starts when the start button is pressed. | Press Start Button. | The board has pieces drawn and set in the start position. | Game Function  00:50 | Works Fully |
| 2 | The settings selected are applied to the game. |  | Enemy opponent active, not active and depth is right. | Game Function  00:50 | Works Fully |
| 3 | When a piece is clicked and picked up, it can only be moved to legal moves, and those are highlighted in green when hovered over. | Click a piece and click it again in a different place. | Piece follows mouse once clicked, and will not make a move if clicked back on an invalid square. Invalid squares outlined red, valid ones green. | Game Function  1:00 | Works Fully |
| 4 | When a check is done, it is written on the screen. |  | Writes ‘Colour is in check’ for whoever is in check. | Game Function  6:50 | Works Fully |
| 5 | When attempting to castle when a relevant field is under attack or the king / rook moved, do not allow. Otherwise, castles. | Attempting to castle with king. | Castling is allowed or not allowed as relevant. | Game Function  7:30 | Works Fully |
| 6 | When a pawn reaches the end, it can promote to another piece in a new window. | Move a pawn to the edge of the board. | A window pops up with pieces, when clicked the pawn turns into clicked piece. | Game Function  10:05 | Works Fully |
| 7 | When stalemate is performed, the type of draw is stated in a pop up and no moves can be made. |  | After a stalemate, the player is informed of it. | Game Function  11:40 | Works Fully |
| 8 | When threefold repetition is performed, the type of draw is stated in a pop up and no moves can be made. |  | After a threefold repetition draw occurs, the player is informed. | Game Function  13:00 | Works Fully |
| 9 | When checkmate is performed, a pop up says so, and no more moves can be made. |  | After a checkmate, the player is informed. | Game Function  14:10 | Works Fully |
| 10 | When a draw by 50 move repetition occurs, a pop up says so and no more moves can be made. |  | After 50 non pawn moves with no captures, a draw is declared. | Game Function  14:40 | Works Fully |
| 9 | Pressing the play button resets the game completely. | Pressing the Play Button. | The board is set to the starting position, with white’s turn. | Game Function  17:25 | Works Fully |
| 10 | When a name is typed in for a save file and save is pressed, the current board position is saved in FEN format. | Save name typed and Save button pressed. | A text file with the typed name is saved in ‘Saves’ and it contains an FEN accurate save format. | Saving and Loading  00:15 | Works Fully |
| 11 | When a file is selected and the load button is pressed, the board is changed to the contents of the selected file. | A save name clicked and Load button pressed. | The board is updated to stores all the details stored in the text file. | Saving and Loading  3:45 | Works Fully |
| 15 | The computer chooses checkmate when it can. |  | If it can foresee a checkmate, it will choose moves to achieve it. | Engine Function  3:00 | Works Fully |
| 16 | When undoing after a computer move, you can change the computer’s move. | Undo button is pressed | After undoing the move, you can click and replay the computer’s move. | Engine function  4:10 | Works Fully |
| 12 | The engine searches every move in the specified depth (without pruning) |  | Debugging of code to demonstrate. | Engine Function 4:30 until the end. | Works Fully |
| 14 | Computer chooses a high evaluation score to strive for. |  | Once searched, the highest found evaluation score move is chosen. | Engine Function 11:10 | Works Fully |
| 13 | Pruning is removed and more nodes should be searched. |  | More nodes should have been searched without pruning. | Engine Function 17:10 | Works Fully |
| 17 | The undo button loads the previous board state, up until load or start of game. It should replicate everything, such as an passant & castling at that time. | Undo button is pressed. | The previous board state is loaded. | Throughout all videos. | Works Fully |

Evaluation

# Objectives

In the analysis section of the document, I have stated the main goals of this project. Now I will use these goals to evaluate the success of my project, and what more work could’ve been done.

* Options where the user can choose to play against another player on the host machine or to play against a computer.
* Option which will have access to the strength of the engine.
* A chess engine utilizing the **minimax algorithm** with different depths that can be selected.
* An **alpha-beta pruning** algorithm to considerably improve performance of the engine.
* Ability to play either colour in Computer player modes.
* A board with 2D pieces that can be dragged by clicking and let go by clicking again on the board.
* The application will calculate all the possible locations that a selected piece can move to, including castling, double pawn movement and en passant.
* The ability to determine when checkmate, draw and check have occurred.
* Ability to reset the game and board with newly selected settings.
* Squares that highlight when hovering over them whether the move is legal or not, and if clicked when not legal, the piece will be returned to its original position without the turn being cancelled.
* A new window opening when promoting a pawn at the edge of the board to select the piece you would like to promote to.
* Games can be saved with FEN notation and stored as a text file. Stores all relevant information correctly.
* Games can be loaded from any text file with contents written in FEN notation. Loads all information stored properly.
* An undo option that will undo moves. It can undo all the way up to the first move since the start or a load.

For the first point:

* Options where the user can choose to play against another player on the host machine or to play against a computer.

This has been successfully completed, as in the main screen you can successfully select ‘Player’ or ‘Computer’. Once game is started, the selected options are the ones taken into effect. A potential improvement that could’ve been added is to have a button that makes the engine perform the current move or provide a hint for the best move the algorithm calculates.

* Option which will have access to the strength of the engine.

This task has been completed well as well, a slider provides a way to change the strength of the engine throughout the game and before loading. The values range from 1 – 8. When selected, the next time the engine makes a move, that depth is successfully used.

* A chess engine utilizing the **minimax algorithm** with different depths that can be selected.

This goal has been performed to a fairly good standard, however there is a lot of room for improvement. First, the algorithm works as intended, searching through each possible node and using recursion, exploring nodes further until depth 0 has been reached where an evaluation (courtesy of Adam Berent) is performed. These scores are correctly managed and Max & Min select the scores properly. The only bottleneck is the performance time due to the move finding algorithm. At depth 4, moves can take an average of 10 seconds or more. This is very slow for depth 4, considering how low the nodes count is. If I had more time, I would redesign the move finding algorithm in a way where previous squares that are movable are not changed unless another piece would affect it. This is especially important when checking the legality of the move, since my move finding algorithm calculates every other piece’s possible move every time moves are calculated to ensure the king is not under check. This ends up being very inefficient in the long term. The less redundant iterations and code executions are performed, the more efficient the search will be since even small amount of time adds up when this many nodes are calculated.

* An **alpha-beta pruning** algorithm to considerably improve performance of the engine.

This point has been completed successfully, alpha beta pruning works completely as intended with a great decrease in nodes searched when it is enabled. A possible improvement is the ability to disable and enable it within the menu of the game. However, I doubt this would have any real need other than for debugging.

* Ability to play either colour in Computer player modes.

This works as intended. The computer is able to play as black or white (vice versa for player). An improvement could be visual display, with the pieces flipping sides to match the perspective of the player.

* A board with 2D pieces that can be dragged by clicking and let go by clicking again on the board.

This works as intended. The pieces are centered with the mouse and clicking again drops the piece back to its original or its new location if it is a legal move. However, a certain improvement to be made is to make the piece place down and align before the engine makes their move, rather than left hanging. I could not find a solution for it; a possible option is multithreading. If I had more time, I would’ve attempted to implement multithreading to allows the board to realign before / slightly during the algorithm.

* The application will calculate all the possible locations that a selected piece can move to, including castling, double pawn movement and en passant.

This works perfectly, all moves are always calculated correctly. Both when undod and loaded from saves. As mentioned before, the efficiency of this algorithm could be improved however.

* The ability to determine when checkmate, draw and check have occurred.

This works completely as intended. Checkmate is detected, check is detected, stalemate, threefold and 50 move rule are all detected. These are all displayed an in the case of everything other than check, a window pops up informing the player as well and pieces are no longer interactable. These can be undoed with the undo button and a different game path can be played. A limitation however, is when a game is loaded, previous game states can not be reviewed and therefore threefold repetition can not be detected. This is a downside of FEN and a different save format would allow me to circumvent this.

* Ability to reset the game and board with newly selected settings.

This works completely as intended. When the ‘Begin Game’ button is pressed, the board is reset and the parameters selected on the menu are applied. A potential GUI change would be to display the current board settings.

* Squares that highlight when hovering over them whether the move is legal or not, and if clicked when not legal, the piece will be returned to its original position without the turn being cancelled.

This feature works as intended. When hovering over legal moves, the square is green and over illegal ones, the square is red. When clicking on illegal moves, the piece is returned back or when clicked off the board. An improvement could be to add an option to enable move highlights, so when you click a piece it shows you all the moves. This is to make the game easier, so you don’t have to visualize moves. I did not do this since I wanted it to be a bit harder.

* A new window opening when promoting a pawn at the edge of the board to select the piece you would like to promote to.

This works as intended. The right colour pieces are displayed. The game is paused while the piece is being picked.

* Games can be saved with FEN notation and stored as a text file. Stores all relevant information correctly.

This works completely as intended. All pieces are saved properly with all the relevant information.

* Games can be loaded from any text file with contents written in FEN notation. Loads all information stored properly.

This works completely as intended. All pieces are loaded properly and all the details of the game are loaded as well. Relevant pieces are updated to match with the save (for example, no queen’s side castle makes the queen’s rook uncastlable. En passant square allows that piece to be capture by en passant.

* An undo option that will undo moves. It can undo all the way up to the first move since the start or a load.

Finally, the undo function also work completely as intended. It undoes moves all the way to the start of the game / load and restores castling rights and en passant squares. A possible improvement would be to allow to redo an undo. So if an undo was made, you could redo it.

# Feedback

I have gathered feedback from a player. They have made a few comments that I will discuss here:

They stated that the time taken for turns was too long. I agree with their feedback. I believe the problem that lies is that further than depth 4, there are simply too many nodes for the current node processing time. This is very likely due to the move finding algorithm which iterates a lot of times to find each move. When literally millions of moves are being processed, each millisecond counts. If I had more time, I would’ve researched or developed a new and more efficient way of saving and reusing moves and not updating them unnecessarily.

Another comment was about switching the board perspective when playing. So when it is black’s turn, black pieces are towards the player. This is an improvement I would also do if I had more time, it has no effect on performance / functionality but would be a good GUI improvement.

Another comment discussed that when loading saves, the default turn of the engine is not selected. I developed loading in mind for saves to be able to be loaded with customization so you are able to select whether you would like to continue the save against an opponent or computer and what colour you’d like the computer to be. I could however, implement hard written information in saves such as original opponent (computer or player) and if computer, what colour they were. This information could be displayed on the screen.

Finally, they suggested whether I could use heuristic functions to improve performance. While I appreciate this feedback, after comparing node exploration counts to common engine exploration node counts, I have very similar numbers. This leads me to believe that rather than node count removal, I need to reduce time processing for nodes. To sacrifice the engine’s functionality for a not very large increase in performance, I think, is not a good trade off.

# Conclusion

Overall the project, I believe, was completed well. I have fulfilled all the things that I wanted to implement and I am able to successfully play against the computer, having a sometimes challenging game and having the engine beat some of my friends. There are improvements to be made that I would add if I had more time to dedicate to the project.