Chess Artificial Agent

Daniel John Ellis

Analysis

Introduction:

Chess is a strategy game derived from the Indian game *Chaturanga* sometime before the 7th century. Chaturanga is believed to predate the Eastern strategy games *Xiangqui, Janggi,* and *Shogi*. What we now call Chess, was standardised in the 19th century and is a game of perfect information, which means that at any point in the game, both players can see every element of the board. Chess is made up of thirty-two pieces (sixteen on each team), all in play on an 8x8 checkerboard where the tiles are used as movement spaces and they are referenced as grid 'coordinates' made up of Files and Ranks. Files are denoted A-H and represent the column, Ranks are denoted 1-8 and represent the row. Each team has eight Pawns, two Rooks, two Knights, two Bishops, one King, and one Queen. Each of the six piece types has a unique movement pattern.

A Pawn can only move forward (toward the opponent's side of the board) one space at a time, unless it is on its first move, or it's attacking another piece, in which case it can move two spaces forward or one space diagonally forward, respectively. A Rook can move any amount of spaces in a straight line along the board. A Knight can move in an 'L' pattern only, whereby it must move two spaces in a straight line, and one space in a straight line perpendicular to its first direction. A Bishop can move any amount of spaces diagonally along the board. A King can only move one space, but can move in any chosen direction. And a Queen can move any amount of spaces either diagonally or in a straight line. Because of these movement patterns, it's commonly thought that the Queen is the most powerful piece, and the Pawn is the least powerful piece.

Whilst these are the standard moves in Chess, there are many other rules which alter a piece's movement. The most commonly used rules are *En Passant* and *Castling*. *En Passant* is a rule where a Pawn on rank 5 (or rank 4 from Black's perspective) may attack a Pawn that is in a space next to it on the same rank, provided the Pawn under siege has just made its first move and moved two spaces rather than one. *Castling* is a move that either team can make only once per game where the player can move their King two files along the first rank towards a rook, and then placing the subject Rook on the last space passed by the King. *Castling* is only allowed, providing neither the King or subject Rook have moved before in the game, there are no pieces between the King and Rook, the King isn't in Check, and the space that the King passes through isn't under attack by the opposition.

A game of Chess starts with White's move, and then the move alternates between the two players. The game can end in any number of ways, the standard methods of reaching end game are through *Checkmate, Stalemate,* or *Resignation*. *Checkmate* is achieved when a

player's King is in Check and they can make no legal moves, the player in Check loses in this scenario. *Stalemate* is where the current player can make no legal moves, yet their King is not in Check, this results in a draw. *Resignation* is where a player concedes, and his opponent automatically wins. Other commonly used endgame scenarios include: *Win on Time* where if a match is time controlled, the player who runs out of time first, loses. *Draw by Agreement* where players agree on a draw. And *Draw by Threefold Repetition* whereby if a player consecutively makes the same move pattern three times, the game is classed as a tie.

Description of Current System:

In recent years, people have programmed Chess Engines which have had increasing success when playing games. IBM made a Chess engine called *Deep Blue* which was the first computer to overcome a reigning Chess Champion by defeating Garry Kasparov in 1997. Chess seems simple and ordinary, however given the amount of possible moves that a player can make on their turn, it is thought that every time you play a game, it's unlikely to play out the same as any other game in the past. After each player has made one move, there are 400 possible board configurations, after the second move each, there are 197,742 configurations, and after three moves, there are approximately 121 million. Given that with each move, the number of possible configurations increases exponentially, it's impractical to try and program an opponent which can respond to each configuration individually.

The most common method of programming a Chess opponent is by use of the MiniMax algorithm with Alpha-Beta Pruning. A MiniMax algorithm can only be used on a game of perfect information. It's essentially a large decision tree of all possible moves. The algorithm copies the current state of the board and checks all moves that can be made, then is applied to each of the boards which those moves would result in and does the same. MiniMax is an algorithm based on Indirect Recursion, because the *Min* function calls the *Max* function, and vice versa until either a terminal node is reached or until a specified recursion depth is reached. *Min* and *Max* are functions designed to represent the thought processes of both players. In Chess, *Max* could represent White, this would mean that *Max* is trying to give White the advantage, and *Min* is trying to give Black the advantage. In any Chess program, a *Heuristic Value* must be used to calculate the strength of a move. For this, you must create a function that will look at the board and assess it. A basic Heuristic for Chess would be to allocate a value to each piece and add up the total piece value of a board. Standard values used tend to be: *Pawn = 1, bishop/knight = 3, rook = 5, and queen = 9*. The King is sometimes included as a value tending to infinity because of how invaluable it is, however given that a King cannot be removed from either side, it becomes redundant unless you have *Bitboards*$_{(qv)}$ in use.

Minimax algorithms without pruning:

```
Function Max(Node, Depth, CurrentDepth)
	if Depth = CurrentDepth or Node is terminal then
		Return Heuristic Value
	Value = −∞
	for each child of Node
		MinVal = Min(child, Depth, CurrentDepth + 1)
		if Value < MinVal then
			Value = MinVal
	Return Value

Function Min(Node, Depth, CurrentDepth)
	if Depth = CurrentDepth or Node is terminal then
		Return Heuristic Value
	Value = ∞
	for each child of Node
		MaxVal = Max(child, Depth, CurrentDepth + 1)
		if Value > MaxVal then
			Value = MaxVal
	Return Value
```

Whilst this is an accurate way of getting on the path to your best outcome, it's slow and takes a lot of time. One of these algorithms has to be applied to every possible board combination, so you end up still analysing a few million boards individually which takes time and memory. The more efficient way of searching is by pruning the search tree with an extension on MiniMax called Alpha-Beta Pruning.

Alpha-Beta Pruning assumes both players are playing perfectly, ergo always making the best move possible. The algorithm utilises two values *Alpha* and *Beta* which represent the minimum score that the *Max* player can get and the maximum score that the *Min* player can get, respectively. When the maximum score that *Min* can get is lower than the minimum score that *Max* can get, *Max* can prune that entire branch, because those nodes will never be reached in play.

Minimax Algorithms with Pruning:

```
Function Max(Node, Depth, CurrentDepth, Alpha, Beta)
	if Depth = CurrentDepth or Node is terminal then
		Return Heuristic Value
	Value = −∞
	for each child of Node
		Temp = Min(child, Depth, CurrentDepth + 1, Alpha, Beta)
		if Value < Temp	then
			Value = Temp
		if Temp >= Beta then
			Return Value
		if Temp > Alpha then
			Alpha = Temp
	Return Value
```

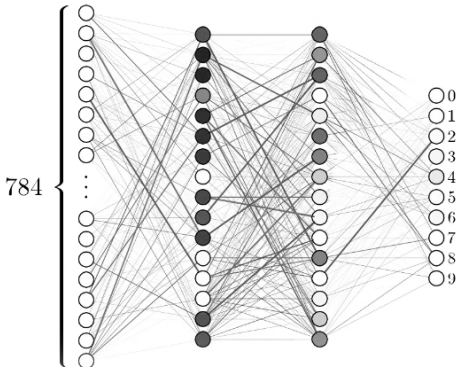
```
Function Min(Node, Depth, CurrentDepth, Alpha, Beta)
	if Depth = CurrentDepth or Node is terminal then
		Return Heuristic Value
	Value = ∞
	for each child of Node
		Temp = Max(child, Depth, CurrentDepth + 1, Alpha, Beta)
		if Value >  Temp    then
			Value = Temp
		if Temp <= Alpha then
			Return Value
		if Temp < Beta then
			Beta = Temp
	Return Value
```

Bitboards are essentially small arrays that represent what's on the board as 1's or 0's. You might have a bitboard for all the individual pieces (One for Kings, one for Queens, etc). When using bitboards, the entire board is subdivided into many layers, however with each layer being 1 bit per tile, the board representation comes down to a 8 bytes per layer that you divide the board into. A common way of dealing with it is to have eight Bitboards, one to show all Pawns (Black and White), one to show all: Rooks, Knights, Bishops, Queens, and Kings separately. Then you have two more Bitboards, one to show all White positions, and one to show all Black positions. This shows every detail needed at any given time, however for processing special moves, you might also need a board to show what pieces have moved already. Bitboards can help when calculating your *Heuristic Value* as you can store a value for positions on the board that could represent how strong they are. The use of Bitboards this way allows a designer to create an opponent with a certain playstyle. For example, a Bitboard might be created that favours tiles closest to the opponent's starting rank, this would create a defensive opponent rather than an offensive one.

The higher level chess opponents are created using Neural Networks and Deep Learning. A Neural Network is a complex computer model designed to replicate the functionality of the brain, they can be applied to many fields, such as Speech Recognition, Social Network Filtering, and Face Recognition. A Neural Network is a series of nodes that are subdivided into layers; to create a Neural Network, you must have a series of input nodes, and a series of output nodes, separated by nodes in *Hidden Layers*. The idea behind a Neural Network is that given a series of inputs, it should be able to provide an output based on what the *Hidden Layers* process. The most common starting idea for a Neural Network comes in the form of Digit Recognition. This is where a hand-drawn digit is processed through the input layer, and there are 10 output nodes (0-9) which display what digit the network believes it has been shown. This is commonly used because of the availability of the MNIST Database (Modified National Institute of Standards and Technology Database), which is a large database of 70,000 hand-drawn digits (60,000 for training, 10,000 for testing). These hand-drawn digits come as 28x28px grayscale images, which are then applied to a Network with 784 input nodes (one node per pixel), and 10 output nodes. The hidden layers of a Neural Network are made up of multiple layers of nodes, with no preset amount of nodes per layer. These nodes are in effect recognising patterns, and then constructing them together to provide an idea as

to what the output should be. This principle has been applied to Chess in the form of engines like *AlphaZero* which defeated world-champion programs *Stockfish, Elmo,* and the 3-day version of *AlphaGo Zero*. It successfully made its name after only 24 hours of training. The reason that Neural Networks are less commonly used is because of how much training data is needed for them, and how complex a Neural Network is to create. This is why MiniMax is standard, alongside the *Monte Carlo Tree Search* algorithm.

Number Recognition Neural Network Structure:



Source: 3Blue1Brown, Youtube

Objectives:

Primary Objectives:

My Program must:

- Understand the basic rules of Chess and only allow valid moves
- Make a calculated move rather than just a random one
- Understand and enforce Checkmate and Stalemate

Possible extensions:

- Implement graphical user input rather than a console based input
- Create algorithms to understand more advanced endgame scenarios such as *Draw by Threefold Repetition*
- Allow complex moves such as *En Passant* and *Castling*

Proposed Solution:

I'm going to have my program create a game tree of all possible moves that can be made by either side, then it will use a MiniMax algorithm with Alpha-Beta pruning to determine which path is best to take. For my *Heuristic Value* I plan to count the amount of moves that can be made by either side, and because my algorithm will play as Black, it will subtract the total number of moves that White can make, from the number of moves that Black can make. This is a rudimentary Heuristic, but I believe it will allow for a more open game as Black won't want to stay defensive.

Design

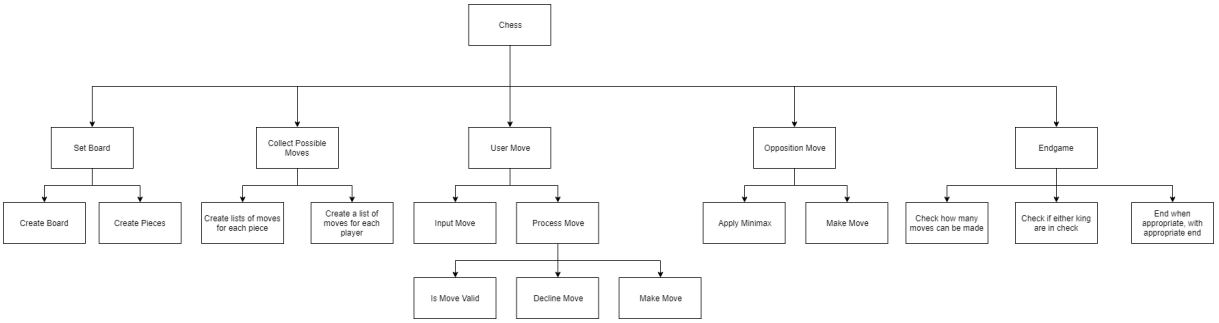
Pseudocode written in Notepad++: https://notepad-plus-plus.org/
Diagrams created on Draw.io: https://www.draw.io/

Overall Design:

Building a Chess opponent can be a complex task, so I've subdivided it into sections which I can design, create, and test. This allows me to be certain of the progress I am making at any given time in the project.

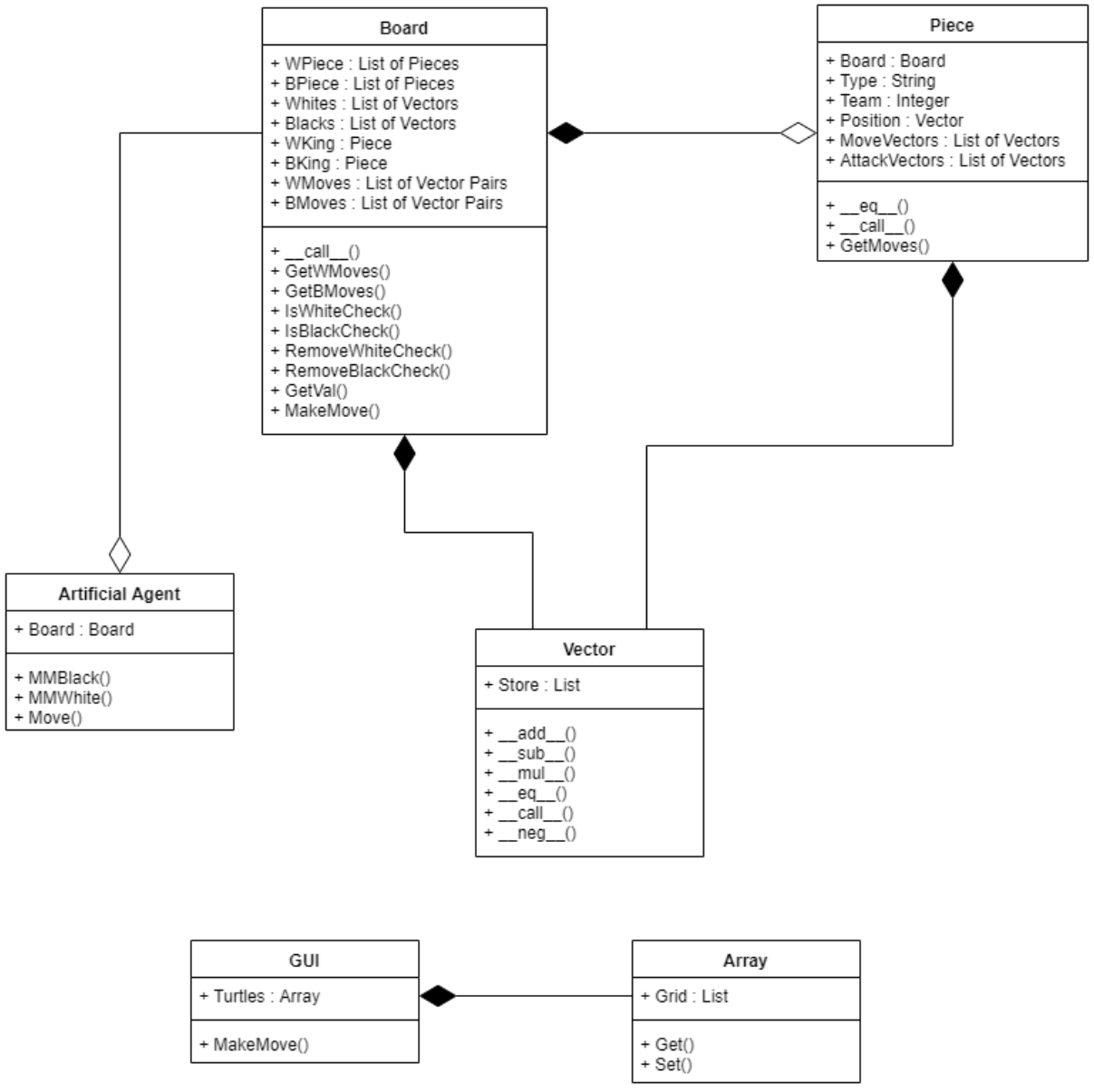
1. Set The Board
    a. Create The Board
    b. Create The Piece Objects
    c. Place The Piece Objects
2. Collect Possible Moves
    a. Create Lists of Moves for Each Piece
        i. Create Algorithm to make all possible piece moves
        ii. Remove any illegal moves
    b. Collect Moves into different teams
3. User Move
    a. Set up input space for move input
    b. Run algorithm in step 2
    c. Process selected move
        i. Compare Move with moves legal moves found in step 2
            1. If illegal move entered, reject move with error message
            2. Else, Make move on board and update Vectors
4. Opposition Move
    a. Run algorithm in step 2
    b. Apply MiniMax on found moves
    c. Make Move
5. End Game
    a. Run algorithm in step 2
    b. Check if either king in Check
    c. Check if any legal moves allowed for current player
        i. If no legal moves and not check, Stalemate
        ii. If no legal moves and check, Checkmate
    d. Display winner

Hierarchy Representation:



A hierarchy created to represent the information shown above in a graphical manor

Class Diagram:



Here I have created a class diagram to help represent how I plan to use Object Oriented Programming in my solution.

The *'Array'* class is from my *'Data_Structure_Library'* library, and is the most basic class I have. Its purpose will be to hold a 2-Dimensional list of Turtle references for the *'GUI'* class

The *'GUI'* class is from my main *'VectorChess'* library, and will handle all the graphical functions needed in my code. It will be the class which accesses the *'Chess_Turtle'* library and all its functions.

The *'Vector'* class is from my *'Maths_Library'* library and handles all the complicated vector functions that are needed throughout the course of the program.

The *'Artificial Agent'* class is from the main *'VectorChess'* library, and is acting as the opponent in the game. This class is where the MiniMax algorithm will be based from.

The *'Board'* class is from the main *'VectorChess'* library and is the class which will store the current state of any board. This is where all moves will be processed.

The *'Piece'* class is from the main *'VectorChess'* library and is the class which will hold all details on a piece on the board. This is where the moves are gathered from.

Proposed Methods:

Function Name	Description
Board.__call__()	Returns a deep copy of the board
Board.GetWMoves()	Collects all moves that White Pieces can make
Board.GetBMoves()	Collects all moves that Black Pieces can make
Board.IsWhiteCheck()	Checks if White's King is in Check
Board.IsBlackCheck()	Checks if Black's King is in Check
Board.RemoveWhiteCheck()	Removes all moves that *'Board.GetWMoves'* produces where White is left in Check
Board.RemoveBlackCheck()	Removes all moves that *'Board.GetBMoves'* produces where Black is left in Check
Board.GetVal()	Returns the Heuristic Value of the board in its current state
Board.MakeMove()	Moves a piece from one space to another, provided it's a valid move from the list refined by *'Board.RemoveBlackCheck'* or

	'Board.RemoveWhiteCheck'
Piece.__eq__()	Checks if a piece is in the same space as a position vector fed to the function
Piece.__call__()	Returns the piece's type
Piece.GetMoves()	Collects all possible moves that the piece in question can make
ArtificialAgent.MMBlack()	This will be the Maximising player of the MiniMax algorithm
ArtificialAgent.MMWhite()	This will be the Minimising player of the MiniMax algorithm
ArtificialAgent.Move()	This function will call *'ArtificialAgent.MMBlack'* to get the best move, and then will proceed to make that move
Vector.__add__()	This function will allow two vectors to be added together correctly
Vector.__sub__()	This function will allow vectors to be subtracted from one another
Vector.__mul__()	This will allow a vector to be multiplied by a scalar amount
Vector.__eq__()	This will check if two vectors have the same ***i***, ***j***, and ***k*** components
Vector.__call__()	This will return a specific value from the vector based on the passed in parameter. If no parameter passed, will return whole vector
Vector.__neg__()	This will return the negative equivalent of a vector
GUI.MakeMove()	This will trigger the board representation to change piece layouts
Array.Get()	Will return value at specified position in the array
Array.Set()	Will update value at specified position in the array

Piece.GetMoves() Pseudocode:

```
GetMoves():
  If Type = Pawn:
    If Team = White:
      For Both AttackVector:
        If Position + AttackVector in PosBlacks:
          Moves = Moves + [Position + AttackVector]

      For MoveVector:
        If Position + MoveVector NOT in PosBlacks OR PosWhites:
          Moves = Moves + [Position + MoveVector]
          If Position + 2 * MoveVector NOT in PosBlacks OR PosWhites AND FirstMove = True:
            Moves = Moves + [Position + 2 * MoveVector]

    If Team = Black:
      For Both AttackVector:
        If Position - AttackVector in PosWhites:
          Moves = Moves + [Position - AttackVector]

      For MoveVector:
        If Position - MoveVector NOT in PosBlacks OR PosWhites:
          Moves = Moves + [Position - MoveVector]
          If Position - 2 * MoveVector NOT in PosBlacks OR PosWhites AND FirstMove = True:
            Moves = Moves + [Position - 2 * MoveVector]

  If Type = Knight OR King:
    If Team = White:
      If Position + MoveVector NOT in PosWhites:
        Moves = Moves + [Position + MoveVector]

      If Position - MoveVector NOT in PosWhites:
        Moves = Moves + [Position - MoveVector]

    If Team = Black:
      If Position + MoveVector NOT in PosBlacks:
        Moves = Moves + [Position + MoveVector]

      If Position - MoveVector NOT in PosBlacks:
        Moves = Moves + [Position - MoveVector]

  If Type = Rook OR Bishop OR Queen:
    If Team = White:
      For Each MoveVector:
        For i in range(1, 7):
          If Position + i * MoveVector in PosBlacks:
            Moves = Moves + [Position + i * MoveVector]
            BreakLoop

          If Position + i * MoveVector in PosWhites:
            BreakLoop

          If Position + i * MoveVector NOT on Board:
            BreakLoop

          If Position + i * MoveVector NOT in PosWhites OR PosBlacks:
            Moves = Moves + [Position + i * MoveVector]

    If Team = Black:
      For Each MoveVector:
        For i in range(1, 7):
          If Position + i * MoveVector in PosWhites:
            Moves = Moves + [Position + i * MoveVector]
            BreakLoop

          If Position + i * MoveVector in PosBlacks:
            BreakLoop

          If Position + i * MoveVector NOT on Board:
            BreakLoop

          If Position + i * MoveVector NOT in PosWhites OR PosBlacks:
            Moves = Moves + [Position + i * MoveVector]

  Return Moves
```

Main Game Pseudocode:

```
StartGame():
  Create Board
  Create Opponent
  Create GUI
  Loop until EndGame:
    Board.GetWMoves()
    Loop until UserMove is Valid
      UserMove = Input
    Board.MakeMove(UserMove)
    GUI.MakeMove(UserMove)
    Board.GetBMoves()
    AI.Move()
  If NoWhiteMoves AND WhiteInCheck:
    Black Win
  If NoBlackMoves AND BlackInCheck:
    White Win
  If NoBlackMoves AND !BlackInCheck:
    StaleMate
  If NoWhiteMoves AND !WhiteInCheck:
    StaleMate
```

A basic rundown of how the primary game algorithm will work. It needs to loop between the player and opposition moves, then when an endgame scenario is reached, it will check the criteria and figure out how the game has ended

Initialise Pieces Pseudocode:

```
InitialisePieces():
  Vectors = SetVectors()
  Whites = List of White Position Vectors
  Blacks = List of Black Position Vectors

  For i in range(0, 7):
    WPiece = WPiece + New Pawn
    BPiece = BPiece + New Pawn

  WPiece = WPiece + New Rook + New Rook
  WPiece = WPiece + New Knight + New Knight
  WPiece = WPiece + New Bishop + New Bishop
  WPiece = WPiece + New King
  WPiece = WPiece + New Queen

  BPiece = BPiece + New Rook + New Rook
  BPiece = BPiece + New Knight + New Knight
  BPiece = BPiece + New Bishop + New Bishop
  BPiece = BPiece + New King
  BPiece = BPiece + New Queen

  Return WPiece, BPiece, Whites, Blacks, WhiteKingPos, BlackKingPos
```

In this stage, I'll need to manually create the pieces with all their individual positions, types, and teams. The 'SetVectors' algorithm will create a reference to all the possible movement vectors and attack vectors. When creating the individual pieces, I'll need to give them their respective vectors as parameters

Create Piece Pseudocode:

```
Piece.__init__(MyBoard, Type, Team, Position, MoveVectors, AttackVectors):

  Ensure following types:
    MyBoard = Board
    Type = String
    Team = Integer
    Position = Vector
    MoveVectors = List of Vectors
    AttackVectors = List of Vectors

  Assign all Parameters to relevant local fields
```

Whilst this isn't complex pseudocode, it does help highlight what fields a Piece object contains and what form each must take

Calculating Attack and Movement Vectors:

I have decided to use Vectors to represent movement and positions because Vectors are defined as a measure of direction and magnitude. Considering the fact that any piece can only move in a set direction, but many can move any magnitude of tiles in that direction, it seemed smart to use vectors to easily represent this. Because of this, rather than considering all eight of the directions in which a Queen can move, I only need to consider four of them. I can create base vectors for every piece, and then when called upon in the '*GetMoves'* algorithm, I will simply apply a scalar multiple to each vector needed.

Application of MiniMax in Pseudocode:

```
Max():
  GetBlackMoves()
  Utility = -∞

  If No Black Moves:
    Return Utility

  If MaxDepth = CurrentDepth:
    Return Heuristic

  For Every Move:
    Clone = Copy of Board
    Clone.MakeMove(Move)
    Temp = Min(Clone, MaxDepth, CurrentDepth + 1, Alpha, Beta)

    If Temp >= Utility:
      Utility = Temp

      If CurrentDepth = 0:
        BestMove = Move

    If Temp >= Beta:
      Return Utility

    If Temp >= Alpha:
      Alpha = Temp

  If CurrentDepth = 0:
    Return BestMove

  Return Utility
```

```
Min():
  GetWhiteMoves()
  Utility = ∞

  If No White Moves:
    Return Utility

  If MaxDepth = CurrentDepth:
    Return Heuristic

  For Every Move:
    Clone = Copy of Board
    Clone.MakeMove(Move)
    Temp = Max(Clone, MaxDepth, CurrentDepth + 1, Alpha, Beta)

    If Temp <= Utility:
      Utility = Temp

      If CurrentDepth = 0:
        BestMove = Move

    If Temp <= Alpha:
      Return Utility

    If Temp < Beta:
      Beta = Temp

  If CurrentDepth = 0:
    Return BestMove

  Return Utility
```

Technical Solution

Code formatted for easier reading using PlanetB facilities:
http://www.planetb.ca/syntax-highlight-word

'Vector_Chess' program:

```
1. import Maths_Library as MyMaths
2. import Chess_Turtle as Graphics
3. import copy
4.
5.
6. _BLACK = 1
7. _WHITE = 0
8. NUMS = ["1", "2", "3", "4", "5", "6", "7", "8"]
9. CHARS = ["A", "B", "C", "D", "E", "F", "G", "H"]
10.
11. def VectToRef(VectA: MyMaths.Vector):
12.     #Parameter Checking
13.     #I plan to use lots of Error checking in my code so that if a bug occurs,
    I can easily find out what it was and where it occurred
14.     if not isinstance(VectA, MyMaths.Vector):
15.         print("Bad Value 'VectA':", VectA)
16.         print("Function can only take Vector Parameter")
17.         raise TypeError
18.     if VectA(2) != 0:
19.         print("Bad Value 'VectA':", VectA)
20.         print("3-Dimensional Vector passed into 2-Dimensional function")
21.         raise TypeError
22.     for i in range(2):
23.         if not isinstance(VectA(i), int):
24.             print("Bad Value 'VectA':", VectA)
25.             print("Function can only accept Vectors with integer components")
26.             raise ValueError
27.     i = VectA(0)
28.     j = VectA(1)
29.     i = chr(ord(str(i)) + 17)
30.     j = str(j+1)
31.
32.     return [i, j]
33.
34. def RefToVect(BoardRef: str):
35.     if not isinstance(BoardRef, str):
36.         print("Bad Value 'BoardRef':", BoardRef)
37.         print("Board Reference must be a list of 2 values")
38.         raise TypeError
39.     if len(BoardRef) != 2:
```

```
40.         print("Bad Value 'BoardRef':", BoardRef)
41.         print("Board Reference must be a list of 2 values")
42.         raise ValueError
43.     if BoardRef[0] not in CHARS or BoardRef[1] not in NUMS:
44.         print("Bad Value 'BoardRef':", BoardRef)
45.         print("Board Reference must be a capital letter (A-H) followed by a
    number (1-8)")
46.         raise ValueError
47.     i = BoardRef[0]
48.     j = BoardRef[1]
49.     i = ord(i) -65
50.     j = ord(j) -49
51.
52.     return MyMaths.Vector(i, j)
53.
54. def IsValidSpace(Move):
55.     EndPos = Move
56.     if EndPos(0) < 0 or EndPos(0) > 7 or EndPos(1) < 0 or EndPos(1) > 7:
57.         return False
58.     return True
59.
60. class Board:
61.     def __init__(self):
62.         self.WPiece, self.BPiece, self.Whites, self.Blacks, self.WKing,
    self.BKing = InitialisePieces(self)
63.         #I created a reference to the Kings so that I can easily check for
    instances of Check later in the code
64.         self.GetWMoves()
65.         self.GetBMoves()
66.         #I collect all moves when the board is initialised to make it easier
    to check a move's eligibility#
67.
68.     def __call__(self):
69.         ##I made the board callable so that I can just return a deepcopy of
    it with ease##
70.         return copy.deepcopy(self)
71.
72.     def GetWMoves(self):
73.         WMoves = []
74.         for Piece in self.WPiece:
75.             Pos, Moves = Piece.GetMoves(self.Whites, self.Blacks)
76.             for Move in Moves:
77.                 WMoves.append([Pos, Move])
78.         self.WMoves = [Move for Move in WMoves if IsValidSpace(Move[1]) ==
    True]
79.
80.     def GetBMoves(self):
81.         BMoves = []
82.         for Piece in self.BPiece:
83.             Pos, Moves = Piece.GetMoves(self.Whites, self.Blacks)
84.             for Move in Moves:
85.                 BMoves.append([Pos, Move])
86.         self.BMoves = [Move for Move in BMoves if IsValidSpace(Move[1]) ==
    True]
```

```
87.
88.     def IsWhiteCheck(self):
89.         self.GetBMoves()
90.         for Element in self.BMoves:
91.             if self.WKing == Element[1]:
92.                 return True
93.         return False
94.
95.     def IsBlackCheck(self):
96.         self.GetWMoves()
97.         for Element in self.WMoves:
98.             if self.BKing == Element[1]:
99.                 return True
100.            return False
101.
102.        def RemoveWhiteCheck(self):
103.            ##The Easiest way to find if a piece is in check is by
    duplicating the board, playing the move, then checking the state##
104.            GoodMoves = []
105.            for Move in self.WMoves:
106.                Clone = self()
107.                Clone.MakeMove(Move[0], Move[1])
108.                Clone.GetBMoves()
109.                if Clone.IsWhiteCheck() != True:
110.                    GoodMoves.append(Move)
111.            self.WMoves = GoodMoves
112.
113.        def RemoveBlackCheck(self):
114.            GoodMoves = []
115.            for Move in self.BMoves:
116.                Clone = self()
117.                Clone.MakeMove(Move[0], Move[1])
118.                Clone.GetWMoves()
119.                if Clone.IsBlackCheck() != True:
120.                    GoodMoves.append(Move)
121.            self.BMoves = GoodMoves
122.
123.        def GetVal(self):
124.            '''''Returns Heuristic Value'''
125.            return len(self.BMoves) - len(self.WMoves)
126.
127.        def PrintVectorLists(self):
128.            '''''DEBUG FUNCTION
129.            Prints all current positions'''
130.            print("Board.Whites")
131.            for Vect in self.Whites:
132.                print(Vect())
133.            print()
134.            print("Board.Blacks")
135.            for Vect in self.Blacks:
136.                print(Vect())
137.
138.        def PrintAllMoves(self):
139.            '''''DEBUG FUNCTION
```

```
140.            Prints all possible moves at current time'''
141.            self.GetWMoves()
142.            self.RemoveWhiteCheck()
143.            self.GetBMoves()
144.            self.RemoveBlackCheck()
145.            print("Whites:")
146.            for Move in self.WMoves:
147.                print(Move[0](), "to", Move[1]())
148.            print()
149.            print("Blacks:")
150.            for Move in self.BMoves:
151.                print(Move[0](), "to", Move[1]())
152.
153.        def PrintAllPieces(self):
154.            '''''DEBUG FUNCTION
155.            Prints all pieces currently on the board and where they are'''
156.            print("Whites:")
157.            for Piece in self.WPiece:
158.                print(Piece(), "at", Piece.Position())
159.            print()
160.            print("Blacks:")
161.            for Piece in self.BPiece:
162.                print(Piece(), "at", Piece.Position())
163.
164.        def MakeMove(self, Start: MyMaths.Vector, End: MyMaths.Vector):
165.            #This function will take move details and convert that to game
    logic
166.            if not isinstance(Start, MyMaths.Vector):
167.                print("Bad Value 'Start':", Start)
168.                print("Position must be in Vector format")
169.                raise TypeError
170.            if not isinstance(End, MyMaths.Vector):
171.                print("Bad Value 'End':", End)
172.                print("Position must be in Vector form")
173.                raise TypeError
174.
175.            IsValid = False
176.
177.            if Start in self.Whites:
178.                for Move in self.WMoves:
179.                    if Move[0] == Start and Move[1] == End:
180.                        IsValid = True
181.                        for Pos in self.Whites:
182.                            if Pos == Start:
183.                                Pos.Store = End()
184.                                for Piece in self.BPiece:
185.                                    ##If the move is attacking another piece,
    delete the piece under attack##
186.                                    if Piece.Position == End:
187.                                        self.Blacks.remove(End)
188.                                        self.BPiece.remove(Piece)
189.                                        break
190.                        break
191.
```

```
192.            elif Start in self.Blacks:
193.                for Move in self.BMoves:
194.                    if Move[0] == Start and Move[1] == End:
195.                        IsValid = True
196.                        for Pos in self.Blacks:
197.                            if Pos == Start:
198.                                #print("Before:", Start(), End())
199.                                Pos.Store = End()
200.                                #print("After:", Start(), End())
201.                                for Piece in self.WPiece:
202.                                    ##If the move is attacking another piece,
    delete the piece under attack##
203.                                    if Piece.Position == End:
204.                                        self.Whites.remove(End)
205.                                        self.WPiece.remove(Piece)
206.                                        break
207.                        break
208.
209.            if IsValid == False:
210.                return False
211.
212.            return True
213.
214.
215.
216.
217.
218.
219.
220.    class Piece:
221.        def __init__(self, MyBoard: Board, Type: str, Team: int, Position:
    MyMaths.Vector, MoveVectors: list, AttackVectors: list):
222.            '''''Makes sure all parameters are valid and then creates a
    piece'''
223.            #I plan to use lots of Error checking in my code so that if a bug
    occurs, I can easily find out what it was and where it occurred
224.            if not isinstance(MyBoard, Board):
225.                print("Bad Value 'MyBoard':", MyBoard)
226.                print("Board must use 'Board' class as datatype")
227.                raise TypeError
228.            if not isinstance(Type, str):
229.                print("Bad Value 'Type':", Type)
230.                print("Piece type must be a string")
231.                raise TypeError
232.            if not isinstance(Team, int) or (Team != 1 and Team !=0):
233.                print("Bad Value 'Team':", Team)
234.                print("Team must be an Integer. Either 0 or 1")
235.                raise TypeError
236.            if not isinstance(Position, MyMaths.Vector):
237.                print("Bad Value 'Position':", Position)
238.                print("Position must be in vector form")
239.                raise TypeError
240.            if not isinstance(MoveVectors, list):
241.                print("Bad Value 'MoveVectors':", MoveVectors)
```

```
242.                print("Move Vectors must be presented as a list")
243.                raise TypeError
244.            if not isinstance(AttackVectors, list):
245.                print("Bad Value 'AttackVectors':", AttackVectors)
246.                print("Attack Vectors must be presented as a list")
247.                raise TypeError
248.            self.Board = MyBoard
249.            self.Type = Type
250.            self.Team = Team
251.            self.Position = Position
252.            self.MoveVectors = MoveVectors
253.            self.AttackVectors = AttackVectors
254.
255.        def __eq__(self, other):
256.            '''''Checks if piece is on the position vector that it's being
    equated to'''
257.            if not isinstance(other, MyMaths.Vector):
258.                return False
259.            if self.Position != other:
260.                return False
261.            return True
262.
263.        def __call__(self):
264.            '''''Returns the Piece's type'''
265.            return self.Type
266.
267.        def GetMoves(self, Whites, Blacks):
268.            '''''This will collect all moves that this piece can make and
    return them in a list'''
269.            if not isinstance(Whites, list):
270.                print("Bad Value 'Whites':", Whites)
271.                print("Piece Locations must be in list form")
272.                raise TypeError
273.            if not isinstance(Blacks, list):
274.                print("Bad Value 'Blacks':", Blacks)
275.                print("Piece Locations must be in list form")
276.                raise TypeError
277.            ##'MyMoves' is to store all moves that this piece determines it
    can make##
278.            ##'ShouldBreak' is going to be used for the Rook, Bishop, and
    Queen. It will allow-##
279.            ##-me to track whether the last move was on a piece or not and
    will break-##
280.            ##-in next loop cycle##
281.            MyMoves = []
282.            ShouldBreak = False
283.            if self() == "Pawn":
284.                if self.Team == _WHITE:
285.                    ##Attack##
286.                    for Vect in self.AttackVectors:
287.                        CurrentMove = self.Position + Vect
288.                        if CurrentMove in Blacks:
289.                            MyMoves.append(CurrentMove)
290.                    ##Move##
```

```
291.                     CurrentMove = self.Position + self.MoveVectors[0]
292.                     if CurrentMove not in Whites:
293.                         if CurrentMove not in Blacks:
294.                             MyMoves.append(CurrentMove)
295.                             if self.Position(1) == 1:
296.                                 CurrentMove = self.Position + 2 *
    self.MoveVectors[0]
297.                                 if CurrentMove not in Whites:
298.                                     if CurrentMove not in Blacks:
299.                                         MyMoves.append(CurrentMove)
300.
301.                 if self.Team == _BLACK:
302.                     ##Attack##
303.                     for Vect in self.AttackVectors:
304.                         CurrentMove = self.Position - Vect
305.                         if CurrentMove in Whites:
306.                             MyMoves.append(CurrentMove)
307.                     ##Move##
308.                     CurrentMove = self.Position - self.MoveVectors[0]
309.                     if CurrentMove not in Whites:
310.                         if CurrentMove not in Blacks:
311.                             MyMoves.append(CurrentMove)
312.                             if self.Position(1) == 6:
313.                                 CurrentMove = self.Position - 2 *
    self.MoveVectors[0]
314.                                 if CurrentMove not in Whites:
315.                                     if CurrentMove not in Blacks:
316.                                         MyMoves.append(CurrentMove)
317.
318.             elif self() == "King" or self() == "Knight":
319.                 ##-Knights and Kings have the same logic as both have attack
    vectors-##
320.                 ##-identical to their move vectors, and both can only move a
    magnitude-##
321.                 ##-of one of their vector/s##
322.                 if self.Team == _WHITE:
323.                     for Vect in self.MoveVectors:
324.                         CurrentMove = self.Position + Vect
325.                         if CurrentMove not in Whites:
326.                             MyMoves.append(CurrentMove)
327.                         CurrentMove = self.Position - Vect
328.                         if CurrentMove not in Whites:
329.                             MyMoves.append(CurrentMove)
330.
331.                 if self.Team == _BLACK:
332.                     for Vect in self.MoveVectors:
333.                         CurrentMove = self.Position + Vect
334.                         if CurrentMove not in Blacks:
335.                             MyMoves.append(CurrentMove)
336.                         CurrentMove = self.Position - Vect
337.                         if CurrentMove not in Blacks:
338.                             MyMoves.append(CurrentMove)
339.
340.             else:
```

```
341.                 if self.Team == _WHITE:
342.                     for Vect in self.MoveVectors:
343.                         ##I use 7 because it's the smallest value that will
    always ensure that-##
344.                         ##-the whole board is covered in all directions##
345.                         for i in range(1, 7):
346.                             CurrentMove = self.Position + i*Vect
347.                             if CurrentMove in Whites:
348.                                 ##A##
349.                                 ##We can't take or pass over our own pieces##
350.                                 break
351.                             elif ShouldBreak == True:
352.                                 ##B##
353.                                 ##If our last move was on an opposing piece,
    we can make no further move this way##
354.                                 ShouldBreak = False
355.                                 break
356.                             elif CurrentMove not in Blacks:
357.                                 ##C##
358.                                 ##This must be a blank tile and we can't have
    gone over any pieces else the-##
359.                                 ##-loop would have broken##
360.                                 MyMoves.append(CurrentMove)
361.                             elif CurrentMove in Blacks:
362.                                 ##D##
363.                                 ##Add move to list because it will take a
    piece, but then break the loop next time-##
364.                                 ##-before adding another move##
365.                                 MyMoves.append(CurrentMove)
366.                                 ShouldBreak = True
367.
368.                         ##This for loop is used for the opposites of the
    moves calculated above##
369.                         for i in range(1, 7):
370.                             CurrentMove = self.Position - i*Vect
371.                             if CurrentMove in Whites:
372.                                 ##A##
373.                                 break
374.                             elif ShouldBreak == True:
375.                                 ##B##
376.                                 ShouldBreak = False
377.                                 break
378.                             elif CurrentMove not in Blacks:
379.                                 ##C##
380.                                 MyMoves.append(CurrentMove)
381.                             elif CurrentMove in Blacks:
382.                                 ##D##
383.                                 MyMoves.append(CurrentMove)
384.                                 ShouldBreak = True
385.                 ##The move for blacks is the same as for whites, just
    searching opposite lists##
386.                 if self.Team == _BLACK:
387.                     for Vect in self.MoveVectors:
388.                         for i in range(1, 7):
```

```
                            CurrentMove = self.Position + i*Vect
                            if CurrentMove in Blacks:
                                ##A##
                                break
                            elif ShouldBreak == True:
                                ##B##
                                ShouldBreak = False
                                break
                            elif CurrentMove not in Whites:
                                ##C##
                                MyMoves.append(CurrentMove)
                            elif CurrentMove in Whites:
                                ##D##
                                MyMoves.append(CurrentMove)
                                ShouldBreak = True

                        ##Again, this for loop calculates the opposite
    moves##
                        for i in range(1, 7):
                            CurrentMove = self.Position - i*Vect
                            if CurrentMove in Blacks:
                                ##A##
                                break
                            elif ShouldBreak == True:
                                ##B##
                                ShouldBreak = False
                                break
                            elif CurrentMove not in Whites:
                                ##C##
                                MyMoves.append(CurrentMove)
                            elif CurrentMove in Whites:
                                ##D##
                                MyMoves.append(CurrentMove)
                                ShouldBreak = True

                ##DEBUG##
                #This small function is used to show moves before the illegal
    ones are removed#
            #for Move in MyMoves:
                #print(self.Position(), Move())

            return self.Position, MyMoves



        def PrintSelf(self):
            '''''DEBUG FUNCTION
            Displays all information about the piece'''
            print(self.Type, self.Team, self.Position())


    class ArtificialAgent:
        def __init__(self, Board):
            self.Board = Board
```

```
441.
442.       def MMBlack(self, Board, Depth, CurrentDepth, Alpha, Beta):
443.           '''''The maximising player of the MiniMax Algorithm'''
444.           Board.GetBMoves()
445.           Board.RemoveBlackCheck()
446.           Utility = -10000
447.           Current = len(Board.BMoves)
448.
449.           ##DEBUG##
450.           #print("Depth of search:", CurrentDepth)
451.           #print("Alpha:", Alpha)
452.           #print("Beta:", Beta)
453.
454.           ##If Black Can make no more moves, this node is terminal##
455.           if Current == 0:
456.               if not isinstance(Utility, int):
457.                   print("Utility:", Utility, "is not an integer")
458.                   return TypeError
459.               return Utility
460.
461.           ##If we've reached our max depth, we return the heuristic value
   of this move##
462.           if Depth == CurrentDepth:
463.               RVal = Board.GetVal()
464.               if not isinstance(RVal, int):
465.                   print("RVal:", RVal, "is not an integer")
466.                   return TypeError
467.               return RVal
468.
469.           for Move in Board.BMoves:
470.               ##I create a copy of the board to test moves on for the
   algorithm##
471.               Check = Board()
472.               Check.MakeMove(Move[0], Move[1])
473.               Temp = self.MMWhite(Check, Depth, CurrentDepth + 1, Alpha,
   Beta)
474.
475.               if not isinstance(Temp, int):
476.                   print("Temp:", Temp, "is not an integer")
477.                   print("Error at:")
478.                   print("Depth:", CurrentDepth)
479.                   print("Move:", Move[0](), "to", Move[1]())
480.                   print("Pieces:")
481.                   Board.PrintAllPieces()
482.                   return TypeError
483.
484.               ##If the Found value is greater than the Utility, Max will
   prefer this path##
485.               if Temp >= Utility:
486.                   Utility = Temp
487.                   ##This means we will always have at least one move that
   we can make##
488.                   ##I only change this on the top level because we don't
   need to worry about##
```

```
489.                    ##remembering future moves until we get to them##
490.                    if CurrentDepth == 0:
491.                        BestMove = Move
492.                ##If Temp is larger than Beta, it means that Min will prefer
    a different path, this is part-##
493.                ##of the 'Alpha-Beta' pruning algorithm##
494.                if Temp >= Beta:
495.
496.                    if not isinstance(Utility, int):
497.                        print("Utility:", Utility, "is not an integer")
498.                        return TypeError
499.
500.                    return Utility
501.                ##If Temp is larger than Alpha, it means we have found a new
    best alternative for Max##
502.                if Temp >= Alpha:
503.                    Alpha = Temp
504.            ##If we have finished our search, return the best move we have
    found##
505.            if CurrentDepth == 0:
506.                return BestMove[0], BestMove[1]
507.
508.                ##If we haven't finished our search, pass the Utility value
    back up the algorithm##
509.                if not isinstance(Utility, int):
510.                    print("Utility:", Utility, "is not an integer")
511.                    return TypeError
512.                return Utility
513.
514.        def MMWhite(self, Board, Depth, CurrentDepth, Alpha, Beta):
515.            '''''The Minimising function of the MiniMax algorithm'''
516.            Board.GetWMoves()
517.            Board.RemoveWhiteCheck()
518.            Utility = 10000
519.            Current = len(Board.WMoves)
520.
521.            ##DEBUG##
522.            #print("Depth of search:", CurrentDepth)
523.            #print("Alpha:", Alpha)
524.            #print("Beta:", Beta)
525.
526.            ##If no moves can be made, node is terminal##
527.            if Current == 0:
528.                if not isinstance(Utility, int):
529.                    print("Utility:", Utility, "is not an integer")
530.                    return TypeError
531.                return Utility
532.
533.            ##If we've reached max depth, return heuristic##
534.            if Depth == CurrentDepth:
535.                RVal = Board.GetVal()
536.                if not isinstance(RVal, int):
537.                    print("RVal:", RVal, "is not an integer")
538.                    return TypeError
```

```
539.                return RVal
540.
541.            for Move in Board.WMoves:
542.                Check = Board()
543.                Check.MakeMove(Move[0], Move[1])
544.                Temp = self.MMBlack(Check, Depth, CurrentDepth + 1, Alpha,
    Beta)
545.
546.                if not isinstance(Temp, int):
547.                    print("Temp:", Temp, "is not an integer")
548.                    print("Error at:")
549.                    print("Depth:", CurrentDepth)
550.                    print("Move:", Move[0](), "to", Move[1]())
551.                    print("Pieces:")
552.                    Board.PrintAllPieces()
553.                    return TypeError
554.
555.                ##If found value is better than utility, we have a new best
    value##
556.                if Temp <= Utility:
557.                    Utility = Temp
558.                    ##If we're at the top level, we have found a best move##
559.                    if CurrentDepth == 0:
560.                        BestMove = Move
561.                ##If Temp is less than Alpha, it means that Max will prefer
    another route##
562.                if Temp <= Alpha:
563.
564.                    if not isinstance(Utility, int):
565.                        print("Utility:", Utility, "is not an integer")
566.                        return TypeError
567.                    return Utility
568.
569.                ##If Temp is less than Beta, we've found a best alternative
    for Min##
570.                if Temp < Beta:
571.                    Beta = Temp
572.            ##If we've reached the end of our search, return best Move
    found##
573.            if CurrentDepth == 0:
574.                return BestMove[0], BestMove[1]
575.
576.            if not isinstance(Utility, int):
577.                print("Utility:", Utility, "is not an integer")
578.                return TypeError
579.
580.            return Utility
581.
582.        def Move(self, MyGUI):
583.            '''''Figure out best move, then apply it'''
584.            From, To = self.MMBlack(self.Board, 1, 0, -10000, 10000)
585.            MyGUI.MakeMove(From, To)
586.            self.Board.MakeMove(From, To)
587.
```

```
588.
589.
590.
591.
592.
593.
594.    class GUI:
595.        def __init__(self):
596.            self.Turtles = Graphics.LoadGame()
597.
598.        def MakeMove(self, From, To):
599.            '''''Make requested move graphically'''
600.            if not isinstance(From, MyMaths.Vector):
601.                print("Bad Value 'From':", From)
602.                print("Positions must be in Vector Form")
603.                return TypeError
604.            if not isinstance(To, MyMaths.Vector):
605.                print("Bad Value 'To':", To)
606.                print("Positions must be in Vector Form")
607.                return TypeError
608.            LFrom = From()
609.            LTo = To()
610.            Graphics.MovePiece(self.Turtles.Get(LFrom[0], LFrom[1]), LTo[0],
    LTo[1])
611.            if self.Turtles.Get(LTo[0], LTo[1]) != None:
612.                self.Turtles.Get(LTo[0], LTo[1]).hideturtle()
613.            self.Turtles.Set(self.Turtles.Get(LFrom[0], LFrom[1]), LTo[0],
    LTo[1])
614.            self.Turtles.Set(None, LFrom[0], LFrom[1])
615.
616.
617.    def InitialisePieces(Board):
618.        '''''Initialise all board pieces with their starting position'''
619.        Vectors = SetVectors()
620.        #Create the list of all occupied spaces for each team so that I can
    easily use them in the creation of pieces.
621.        #This way, I only need to change the values in this list and it will
    change the value in the piece's variable as well.
622.        Whites = [MyMaths.Vector(0, 0), MyMaths.Vector(1, 0),
    MyMaths.Vector(2, 0), MyMaths.Vector(3, 0), MyMaths.Vector(4, 0),
    MyMaths.Vector(5, 0), MyMaths.Vector(6, 0), MyMaths.Vector(7, 0),
    MyMaths.Vector(0, 1), MyMaths.Vector(1, 1), MyMaths.Vector(2, 1),
    MyMaths.Vector(3, 1), MyMaths.Vector(4, 1), MyMaths.Vector(5, 1),
    MyMaths.Vector(6, 1), MyMaths.Vector(7, 1)]
623.        Blacks = [MyMaths.Vector(0, 2), MyMaths.Vector(1, 2),
    MyMaths.Vector(2, 2), MyMaths.Vector(3, 2), MyMaths.Vector(4, 2),
    MyMaths.Vector(5, 2), MyMaths.Vector(6, 2), MyMaths.Vector(7, 2),
    MyMaths.Vector(0, 7), MyMaths.Vector(1, 7), MyMaths.Vector(2, 7),
    MyMaths.Vector(3, 7), MyMaths.Vector(4, 7), MyMaths.Vector(5, 7),
    MyMaths.Vector(6, 7), MyMaths.Vector(7, 7)]
624.        WPiece = []
625.        BPiece = []
626.        for i in range(8):
```

```
627.            WPiece.append(Piece(Board, "Pawn", _WHITE, Whites[8+i],
    Vectors[0], Vectors[1]))
628.            BPiece.append(Piece(Board, "Pawn", _BLACK, Blacks[i], Vectors[0],
    Vectors[1]))
629.
630.        ##White Pieces##
631.        WPiece.append(Piece(Board, "Rook", _WHITE, Whites[0], Vectors[2],
    Vectors[2]))
632.        WPiece.append(Piece(Board, "Rook", _WHITE, Whites[7], Vectors[2],
    Vectors[2]))
633.        WPiece.append(Piece(Board, "Knight", _WHITE, Whites[1], Vectors[3],
    Vectors[3]))
634.        WPiece.append(Piece(Board, "Knight", _WHITE, Whites[6], Vectors[3],
    Vectors[3]))
635.        WPiece.append(Piece(Board, "Bishop", _WHITE, Whites[2], Vectors[4],
    Vectors[4]))
636.        WPiece.append(Piece(Board, "Bishop", _WHITE, Whites[5], Vectors[4],
    Vectors[4]))
637.        WPiece.append(Piece(Board, "King", _WHITE, Whites[4], Vectors[5],
    Vectors[5]))
638.        WPiece.append(Piece(Board, "Queen", _WHITE, Whites[3], Vectors[6],
    Vectors[6]))
639.
640.        ##Black Pieces##
641.        BPiece.append(Piece(Board, "Rook", _BLACK, Blacks[8], Vectors[2],
    Vectors[2]))
642.        BPiece.append(Piece(Board, "Rook", _BLACK, Blacks[15], Vectors[2],
    Vectors[2]))
643.        BPiece.append(Piece(Board, "Knight", _BLACK, Blacks[9], Vectors[3],
    Vectors[3]))
644.        BPiece.append(Piece(Board, "Knight", _BLACK, Blacks[14], Vectors[3],
    Vectors[3]))
645.        BPiece.append(Piece(Board, "Bishop", _BLACK, Blacks[10], Vectors[4],
    Vectors[4]))
646.        BPiece.append(Piece(Board, "Bishop", _BLACK, Blacks[13], Vectors[4],
    Vectors[4]))
647.        BPiece.append(Piece(Board, "King", _BLACK, Blacks[12], Vectors[5],
    Vectors[5]))
648.        BPiece.append(Piece(Board, "Queen", _BLACK, Blacks[11], Vectors[6],
    Vectors[6]))
649.
650.        return WPiece, BPiece, Whites, Blacks, WPiece[14], BPiece[14]
651.
652.    def SetVectors():
653.        '''''Create the vectors that the pieces can move across and assign
    them to lists'''
654.        #I don't need every possible move represented as it's own vector
    because vectors are bi-directional
655.        #I've re-used lots of vectors by making copies in the other lists,
    this is because these vectors should never be altered-
656.        #- by the program and therefore can be used all throughout as many
    pieces can move in the same directions as others
657.        PawnVector = [MyMaths.Vector(0, 1)]
658.        PawnAttackVectors = [MyMaths.Vector(-1, 1), MyMaths.Vector(1, 1)]
```

```
659.        RookVectors = [PawnVector[0], MyMaths.Vector(1, 0)]
660.        KnightVectors = [MyMaths.Vector(-2, 1), MyMaths.Vector(-1, 2),
    MyMaths.Vector(1, 2), MyMaths.Vector(2, 1)]
661.        BishopVectors = [PawnAttackVectors[0], PawnAttackVectors[1]]
662.        KingVectors = [PawnAttackVectors[0], PawnVector[0],
    PawnAttackVectors[1], RookVectors[1]]
663.        QueenVectors = KingVectors
664.        #^^I can safely make the King and Queen movement vectors the same
    because the king will be
665.        #restricted to a scalar multiple of 1 for movements
666.
667.        return [PawnVector, PawnAttackVectors, RookVectors, KnightVectors,
    BishopVectors, KingVectors, QueenVectors]
668.
669.    def GetMoveVector(From: MyMaths.Vector, To: MyMaths.Vector):
670.        '''''Gets a vector which represents the move being made on the
    board'''
671.        if not isinstance(From, MyMaths.Vector):
672.            print("Bad Value 'From':", From)
673.            print("Postions must be in vector form")
674.            raise TypeError
675.        if not isinstance(To, MyMaths.Vector):
676.            print("Bad Value 'To':", To)
677.            print("Postions must be in vector form")
678.            raise TypeError
679.
680.        return MyMaths.GetVector(From, To)
681.
682.    def GetPlayerMove():
683.        FromInput = RefToVect(input("From: "))
684.        ToInput = RefToVect(input("To: "))
685.        if IsValidSpace(FromInput):
686.            if IsValidSpace(ToInput):
687.                return FromInput, ToInput, True
688.        print("Illegal Move")
689.        return FromInput, ToInput, False
690.
691.
692.
693.
694.
695.
696.    def PrintAllPieces():
697.        '''''DEBUG FUNCTION
698.        Prints all pieces on the board'''
699.        print("WHITES")
700.        for Piece in Game.WPiece:
701.            Piece.PrintSelf()
702.        print("BLACKS")
703.        for Piece in Game.BPiece:
704.            Piece.PrintSelf()
705.
706.
707.
```

```
708.
709.   ##GAME##
710.
711.   Game = Board()
712.   GameGUI = GUI()
713.   AI = ArtificialAgent(Game)
714.   i = 0
715.   while i == 0:
716.       Game.GetWMoves()
717.       Game.RemoveWhiteCheck()
718.       Valid = False
719.       while Valid == False:
720.           StartPos, EndPos, Valid = GetPlayerMove()
721.       Game.MakeMove(StartPos, EndPos)
722.       if Game.IsBlackCheck():
723.           print("Black in Check!")
724.       GameGUI.MakeMove(StartPos, EndPos)
725.       AI.Move(GameGUI)
726.       if Game.IsWhiteCheck():
727.           print("White in Check!")
728.
```

My *'Chess_Turtle'* library:

```
1. import Data_Structure_Library as ds
2. import turtle
3.
4. TILESIZE = 65
5. ORIGIN = -4*TILESIZE + 1, -4*TILESIZE + 1
6. BLACKCOLOUR = '#59260B'
7. WHITECOLOUR = '#CBA135'
8.
9. turtle.setup(width = 8*TILESIZE +50, height = 8*TILESIZE +50)
10.turtle.speed(0)
11.turtle.hideturtle()
12.
13.def BlackTile():
14.    turtle.pencolor(BLACKCOLOUR)
15.    turtle.fillcolor(BLACKCOLOUR)
16.    turtle.pendown()
17.    turtle.begin_fill()
18.    for i in range(4):
19.        turtle.forward(TILESIZE-1)
20.        turtle.right(90)
21.    turtle.end_fill()
22.    turtle.penup()
23.    turtle.right(90)
24.    turtle.forward(TILESIZE)
25.    turtle.left(90)
26.
27.def WhiteTile():
28.    turtle.penup()
29.    turtle.right(90)
30.    turtle.forward(TILESIZE)
31.    turtle.left(90)
32.
33.def ReturnToOrigin():
34.    turtle.penup()
35.    turtle.goto(ORIGIN)
36.    turtle.setheading(0)
37.    turtle.forward(1)
38.    turtle.left(90)
39.
40.def DrawBorder():
41.    turtle.setheading(180)
42.    turtle.forward(1)
43.    turtle.left(90)
44.    turtle.forward(1)
45.    turtle.pendown()
46.    turtle.fillcolor(WHITECOLOUR)
47.    turtle.begin_fill()
48.    turtle.setheading(0)
49.    for i in range(4):
50.        turtle.forward(8*TILESIZE + 1)
51.        turtle.left(90)
52.    turtle.end_fill()
53.    turtle.forward(1)
```

```
54.     turtle.left(90)
55.     turtle.forward(1)
56.
57. def DrawBlankBoard():
58.     for i in range(4):
59.         ReturnToOrigin()
60.         turtle.forward(2*TILESIZE*i)
61.         for j in range(4):
62.             BlackTile()
63.             WhiteTile()
64.         ReturnToOrigin()
65.         turtle.forward(TILESIZE+2*TILESIZE*i)
66.         for k in range(4):
67.             WhiteTile()
68.             BlackTile()
69.         ReturnToOrigin()
70.         turtle.forward(2*TILESIZE+2*TILESIZE*i)
71.
72.
73. def MoveTurtle(x, y):
74.     x -= 4*TILESIZE
75.     y -= 4*TILESIZE
76.     turtle.penup()
77.     turtle.goto(x, y)
78.
79. def MovePiece(Turtle, x, y):
80.     '''0 ≤ x, y ≤ 7'''
81.     x = x*TILESIZE - (3*TILESIZE +TILESIZE/2) + 1
82.     y = y*TILESIZE - (3*TILESIZE +TILESIZE/2) + 1
83.     Turtle.penup()
84.     Turtle.goto(x, y)
85.
86. def RegisterShapes():
87.     turtle.register_shape("WPawn.gif")
88.     turtle.register_shape("BPawn.gif")
89.     turtle.register_shape("WRook.gif")
90.     turtle.register_shape("BRook.gif")
91.     turtle.register_shape("WKnight.gif")
92.     turtle.register_shape("BKnight.gif")
93.     turtle.register_shape("WBishop.gif")
94.     turtle.register_shape("BBishop.gif")
95.     turtle.register_shape("WQueen.gif")
96.     turtle.register_shape("BQueen.gif")
97.     turtle.register_shape("WKing.gif")
98.     turtle.register_shape("BKing.gif")
99.
100.   def LoadTurtles():
101.       TurtleList = ds.Array(8, 8)
102.       for i in range(8):
103.           a = turtle.Turtle()
104.           b = turtle.Turtle()
105.           c = turtle.Turtle()
106.           d = turtle.Turtle()
107.
```

```
108.            TurtleList.Set(a, i, 0)
109.            TurtleList.Set(b, i, 1)
110.            TurtleList.Set(c, i, 2)
111.            TurtleList.Set(d, i, 7)
112.        for i in range(8):
113.            for j in range(8):
114.                if TurtleList.Get(i, j) != None:
115.                    TurtleList.Get(i, j).penup()
116.                    TurtleList.Get(i, j).speed(9)
117.                    MovePiece(TurtleList.Get(i, j), i, j)
118.                    TurtleList.Get(i, j).setheading(90)
119.                    TurtleList.Get(i, j).speed(1)
120.
121.        RegisterShapes()
122.        ##Pawns
123.        for i in range(8):
124.            TurtleList.Get(i, 1).shape("WPawn.gif")
125.            TurtleList.Get(i, 2).shape("BPawn.gif")
126.        ##Rooks
127.        TurtleList.Get(0, 0).shape("WRook.gif")
128.        TurtleList.Get(7, 0).shape("WRook.gif")
129.        TurtleList.Get(0, 7).shape("BRook.gif")
130.        TurtleList.Get(7, 7).shape("BRook.gif")
131.        ##Knights
132.        TurtleList.Get(1, 0).shape("WKnight.gif")
133.        TurtleList.Get(6, 0).shape("WKnight.gif")
134.        TurtleList.Get(1, 7).shape("BKnight.gif")
135.        TurtleList.Get(6, 7).shape("BKnight.gif")
136.        ##Bishops
137.        TurtleList.Get(2, 0).shape("WBishop.gif")
138.        TurtleList.Get(5, 0).shape("WBishop.gif")
139.        TurtleList.Get(2, 7).shape("BBishop.gif")
140.        TurtleList.Get(5, 7).shape("BBishop.gif")
141.        ##Kings
142.        TurtleList.Get(4, 0).shape("WKing.gif")
143.        TurtleList.Get(4, 7).shape("BKing.gif")
144.        ##Queens
145.        TurtleList.Get(3, 0).shape("WQueen.gif")
146.        TurtleList.Get(3, 7).shape("BQueen.gif")
147.        return TurtleList
148.
149.
150.    def LoadGame():
151.        ReturnToOrigin()
152.        DrawBorder()
153.        DrawBlankBoard()
154.        return LoadTurtles()
155.
```

My *'Maths_Library'* library:

```
1. import math
2. import copy
3. class Matrix:
4.     '''A Matrix of size NxM'''
5.     def __init__(self, Rows, Columns):
6.         self.RowCount = Rows
7.         self.ColumnCount = Columns
8.         self.Contents = [[0 for i in range(Columns)] for j in range(Rows)]
9.
10.    def __add__(MatA, MatB):
11.        '''Add 2 Matrices (of the same size)'''
12.        if (MatA.RowCount == MatB.RowCount) and (MatA.ColumnCount ==
   MatB.ColumnCount):
13.            MatC = Matrix(MatA.RowCount, MatA.ColumnCount)
14.            for i in range(MatA.RowCount):
15.                for j in range(MatA.ColumnCount):
16.                    Val = MatA.GetVal(i, j) + MatB.GetVal(i, j)
17.                    MatC.SetVal(i, j, Val)
18.            return MatC
19.
20.    def __sub__(MatA, MatB):
21.        '''Subract a Matrix from another Matrix (of the same size)'''
22.        if (MatA.RowCount == MatB.RowCount) and (MatA.ColumnCount ==
   MatB.ColumnCount):
23.            MatC = Matrix(MatA.RowCount, MatA.ColumnCount)
24.            for i in range(MatA.RowCount):
25.                for j in range(MatA.ColumnCount):
26.                    Val = MatA.GetVal(i, j) - MatB.GetVal(i, j)
27.                    MatC.SetVal(i, j, Val)
28.            return MatC
29.
30.    def __mul__(MatA, MatB):
31.        '''Multiply matrices'''
32.        if type(MatB) == Matrix:
33.            if MatA.ColumnCount != MatB.RowCount:
34.                return None
35.            else:
36.                MatC = Matrix(MatA.RowCount, MatB.ColumnCount)
37.
38.                for ARow in range(MatA.RowCount):
39.                    for BColumn in range(MatB.ColumnCount):
40.                        for BRow in range(MatB.RowCount):
41.                            #A really ugly way of incrementing the value in
   the Answer Matrix#
42.                            MatC.SetVal(ARow, BColumn, MatC.GetVal(ARow,
   BColumn) + MatA.GetVal(ARow, BRow) * MatB.GetVal(BRow, BColumn))
43.
44.                return MatC
45.
46.
47.
48.    def Fill(self):
49.        '''Set Values via the Console'''
```

```
50.         for i in range(self.RowCount):
51.             for j in range(self.ColumnCount):
52.                 self.Contents[i][j] = float(input("(" + str(i) + ", " +
    str(j) + "): "))
53.
54.     def GetDet(self):
55.         '''Returns the Determinant of any Square Matrix'''
56.         if self.RowCount == self.ColumnCount:
57.
58.             if self.RowCount == 1:
59.                 Det = self.Contents[0][0]
60.                 return Det
61.
62.             Det = 0
63.             #Create Matrix of Minors
64.             for Column in range(self.ColumnCount):
65.
66.                 MatA = Matrix(self.RowCount -1, self.ColumnCount -1)
67.
68.                 H = 0
69.                 for i in range(self.RowCount):
70.                     if i != 0:
71.                         W = 0
72.                         for j in range(self.ColumnCount):
73.                             if j  != Column:
74.                                 MatA.SetVal(H, W, self.GetVal(i, j))
75.
76.
77.                                 W += 1
78.
79.                         H += 1
80.                 if float(self.GetVal(0, Column)) != 0:
81.                     if Column % 2 == 0:
82.                         Det += (float(self.GetVal(0,
    Column))*float(MatA.GetDet()))
83.                     else:
84.                         Det -= (float(self.GetVal(0,
    Column))*float(MatA.GetDet()))
85.                 else:
86.                     Det += 0
87.
88.                 del MatA
89.
90.
91.
92.             return Det
93.
94.     def GetVal(self, Row, Column):
95.         '''Returns the value in a specified point of the matrix'''
96.         return self.Contents[Row][Column]
97.
98.     def SetVal(self, Row, Column, Value):
99.         '''Sets the value at a specified point of the matrix'''
100.            self.Contents[Row][Column] = Value
```

```
101.
102.       def DisplaySelf(self):
103.           '''Prints matrix'''
104.           for Row in self.Contents:
105.               print(Row)
106.
107.       def Exponent(self, Exp):
108.           '''Returns matrix after exponential multiplication'''
109.           Exp = int(Exp)
110.           if Exp < 2 or (self.RowCount != self.ColumnCount):
111.               return None
112.           else:
113.               Result = self
114.               for i in range(Exp-1):
115.                   Result = MultiplyMat(Result, self)
116.
117.               return Result
118.
119.       def SetRandom(self):
120.           '''Creates random matrix values'''
121.           import random
122.           for i in range(self.RowCount):
123.               for j in range(self.ColumnCount):
124.                   self.SetVal(i, j, random.randint(-5, 5))
125.           del random
126.
127.       def SetIdentity(self):
128.           '''Sets values of an identity matrix'''
129.           if self.RowCount == self.ColumnCount:
130.               for i in range(self.RowCount):
131.                   for j in range(self.ColumnCount):
132.                       if i == j:
133.                           self.SetVal(i, j, 1)
134.                       else:
135.                           self.SetVal(i, j, 0)
136.           else:
137.               return "Only Square Matrix Can Be Identity"
138.
139.
140.
141.   '''Matrix Maths'''
142.
143.   def MultiplyMat(MatA, MatB):
144.       if MatA.ColumnCount != MatB.RowCount:
145.           return None
146.       else:
147.           MatC = Matrix(MatA.RowCount, MatB.ColumnCount)
148.
149.           for ARow in range(MatA.RowCount):
150.                   for BColumn in range(MatB.ColumnCount):
151.                       for BRow in range(MatB.RowCount):
152.                           #A really ugly way of incrementing the value in
    the Answer Matrix#
```

```
153.                             MatC.SetVal(ARow, BColumn, MatC.GetVal(ARow,
    BColumn) + MatA.GetVal(ARow, BRow) * MatB.GetVal(BRow, BColumn))
154.
155.             return MatC
156.
157.
158.
159.     class Vector:
160.         def __init__(self, i, j, k = 0):
161.             self.Store = [i, j, k]
162.             self.Mod = math.sqrt((i**2) + (j**2) + (k**2))
163.
164.         def __add__(self, b):
165.             c = Vector(self(0), self(1), self(2))
166.             for i in range(3):
167.                 c.Store[i] += b.GetVal(i)
168.             return c
169.
170.         def __sub__(self, b):
171.             c = Vector(self(0), self(1), self(2))
172.             for i in range(3):
173.                 c.Store[i] -= b.GetVal(i)
174.             return c
175.
176.         def __mul__(Vect, Scalar):
177.             '''Multiplies a vector by a scalar'''
178.             i = Vect.GetVal(0) * Scalar
179.             j = Vect.GetVal(1) * Scalar
180.             k = Vect.GetVal(2) * Scalar
181.             Result = Vector(i, j, k)
182.             return Result
183.
184.         def __rmul__(Vect, Mult):
185.             '''Multiplies Vector by either a scalar or a Matrix'''
186.             if type(Mult) ==(int or float):
187.                 return Vect * Mult
188.
189.             if type(Mult) == Matrix:
190.                 if Mult.GetColumns == 3:
191.                     MatV = Matrix(3, 1)
192.                     MatV.SetVal(0, 0, Vect.GetVal(0))
193.                     MatV.SetVal(1, 0, Vect.GetVal(1))
194.                     MatV.SetVal(2, 0, Vect.GetVal(2))
195.                     return Mult * MatV
196.
197.                 if Mult.GetColumns == 2:
198.                     MatV = Matrix(2, 1)
199.                     MatV.SetVal(0, 0, Vect.GetVal(0))
200.                     MatV.SetVal(1, 0, Vect.GetVal(1))
201.                     return Mult * MatV
202.
203.         def __eq__(self, Other):
204.             '''Checks if a vector is equal to another vector'''
205.             if not isinstance(Other, Vector):
```

```
206.                 return False
207.             if self.GetVal(0) != Other.GetVal(0):
208.                 return False
209.             if self.GetVal(1) != Other.GetVal(1):
210.                 return False
211.             if self.GetVal(2) != Other.GetVal(2):
212.                 return False
213.             return True
214.
215.         def __call__(self, x = -1):
216.             '''Returns value in Store[x],
217.             if no value entered or x = -1, returns Store as list'''
218.             if x == -1:
219.                 return self.Store
220.             for i in range(3):
221.                 if x == i:
222.                     return self.Store[i]
223.
224.         def __neg__(self):
225.             '''Returns a negative version of self'''
226.             return Vector(-self.GetVal(0), -self.GetVal(1), -self.GetVal(2))
227.
228.         def GetMod(self):
229.             '''Get the magnitude of the vector'''
230.             return self.Mod
231.
232.
233.     '''Vector Maths'''
234.
235.     def Dot(VectA, VectB):
236.         '''Applies the dot product to two vectors'''
237.         a = VectA.GetSelf()
238.         b = VectB.GetSelf()
239.         Total = 0
240.         for i in range(3):
241.             Total += a[i]*b[i]
242.         return Total
243.
244.     def Scalar(VectA, VectB):
245.         '''Will find the Angle in Radians between two Vectors'''
246.         #Definition: a.b = |a||b|Cos(Theta)#
247.         DotProd = Dot(VectA, VectB)
248.         ModA = VectA.GetMod()
249.         ModB = VectB.GetMod()
250.         return math.acos(DotProd/(ModA*ModB))
251.
252.
253.     def Cross(VectA, VectB):
254.         '''Applies the cross product'''
255.         MatI = Matrix(2, 2)
256.         MatI.SetVal(0, 0, VectA.GetVal(1))
257.         MatI.SetVal(0, 1, VectA.GetVal(2))
258.         MatI.SetVal(1, 0, VectB.GetVal(1))
259.         MatI.SetVal(1, 1, VectB.GetVal(2))
```

```
260.
261.       MatJ = Matrix(2, 2)
262.       MatJ.SetVal(0, 0, VectA.GetVal(0))
263.       MatJ.SetVal(0, 1, VectA.GetVal(2))
264.       MatJ.SetVal(1, 0, VectB.GetVal(0))
265.       MatJ.SetVal(1, 1, VectB.GetVal(2))
266.
267.       MatK = Matrix(2, 2)
268.       MatK.SetVal(0, 0, VectA.GetVal(0))
269.       MatK.SetVal(0, 1, VectA.GetVal(1))
270.       MatK.SetVal(1, 0, VectB.GetVal(0))
271.       MatK.SetVal(1, 1, VectB.GetVal(1))
272.
273.       I = MatI.GetDet()
274.       J = -MatJ.GetDet()
275.       K = MatK.GetDet()
276.
277.       VectC = Vector(I, J, K)
278.
279.       return VectC
280.
281.   def IsCollinear(a, b):
282.       '''Checks if a vector is collinear with another vector'''
283.       #If any of these 3 cases are true, the vectors are not collinear
284.       #Checking this now means we don't risk dividing by zero in the next
    step
285.       if (a.Store[2] == 0 and b.Store[2] !=0) or (a.Store[2] != 0 and
    b.Store[2] == 0):
286.           return False
287.       if (a.Store[1] == 0 and b.Store[1] !=0) or (a.Store[1] != 0 and
    b.Store[1] == 0):
288.           return False
289.       if (a.Store[0] == 0 and b.Store[0] !=0) or (a.Store[0] != 0 and
    b.Store[0] == 0):
290.           return False
291.
292.       #If the vectors are the same, then they are collinear
293.       if a == b:
294.           return True
295.
296.       #Loop through to make sure I don't divide by zero at any point
297.       for i in range(2):
298.           if b.Store[i] != 0:
299.               Scalar = a.Store[i] / b.Store[i]
300.       #Once a scalar has been established, we can check if the vectors are
    collinear
301.       for i in range(1,2):
302.           if a.Store[i] / Scalar != b.Store[i]:
303.               return False
304.       return True
305.
306.   def GetVector(PosA, PosB):
307.       '''Get Vector connecting A to B'''
308.       i = PosB.Store[0] - PosA.Store[0]
```

```
309.        j = PosB.Store[1] - PosA.Store[1]
310.        k = PosB.Store[2] - PosA.Store[2]
311.        return Vector(i, j, k)
312.
313.
314.    class Line:
315.        def __init__(self, Position, Direction):
316.            self.Pos = Position
317.            self.Dir = Direction
318.
319.        def DoesMeet(self, Point):
320.            Range = Point.Store[0] - self.Pos.Store[0]
321.            Check = self.Pos + Range * self.Dir
322.            if Check.Store == Point.Store:
323.                return True
324.            return False
325.
326.    def GetSmallest(Value1, Value2):
327.        '''Returns the smallest of two numbers'''
328.        if not isinstance(Value1, int) and not isinstance(Value1, float):
329.            print("Bad Value 'Value1':", Value1)
330.            print("Can only compare numbers")
331.            raise TypeError
332.        if not isinstance(Value2, int) and not isinstance(Value2, float):
333.            print("Bad Value 'Value2':", Value2)
334.            print("Can only compare numbers")
335.            raise TypeError
336.        if Value1 <= Value2:
337.            return Value1
338.        else:
339.            return Value2
340.
341.    def GetLargest(Value1, Value2):
342.        '''Returns the largest of two numbers'''
343.        if not isinstance(Value1, int) and not isinstance(Value1, float):
344.            print("Bad Value 'Value1':", Value1)
345.            print("Can only compare numbers")
346.            raise TypeError
347.        if not isinstance(Value2, int) and not isinstance(Value2, float):
348.            print("Bad Value 'Value2':", Value2)
349.            print("Can only compare numbers")
350.            raise TypeError
351.        if Value1 >= Value2:
352.            return Value1
353.        else:
354.            return Value2
```

Testing

Test	Pass/Fail	Reference to Image
Draw the Board	Pass	Image 1
Create Piece Objects	Pass	Image 2
Draw Pieces on Board	Pass	Image 3
Collect all possible moves for a piece, legal or not	Pass	Images 4 & 5
Remove illegal moves	Pass	Images 6 & 7
Compile Moves into a list for each team	Pass	Images 8
Allow move input	Pass	Images 9 & 10
Display error message if user enters illegal move	Fail	Images 11 & 12
Move piece on the board GUI	Pass	Images 13 & 14
Change vectors in the piece object	Pass	Images 15 & 16
Change vectors in the position list	Pass	Images 17 & 18
Agent applies MiniMax to found moves	See Notes	Images 19-21
Agent moves and updates relevant values and GUI	Pass	Images 22-24
Check if Either King in Check	Pass	Image 25 & 26
Display Stalemate when relevant	Fail	N/A
Display Checkmate when relevant	Fail	N/A
Display Winner	Fail	N/A

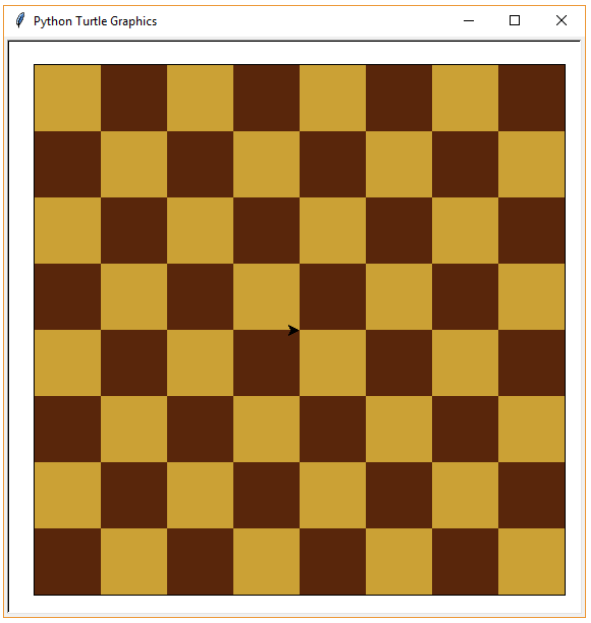


Image 1

Chess Board has been successfully drawn

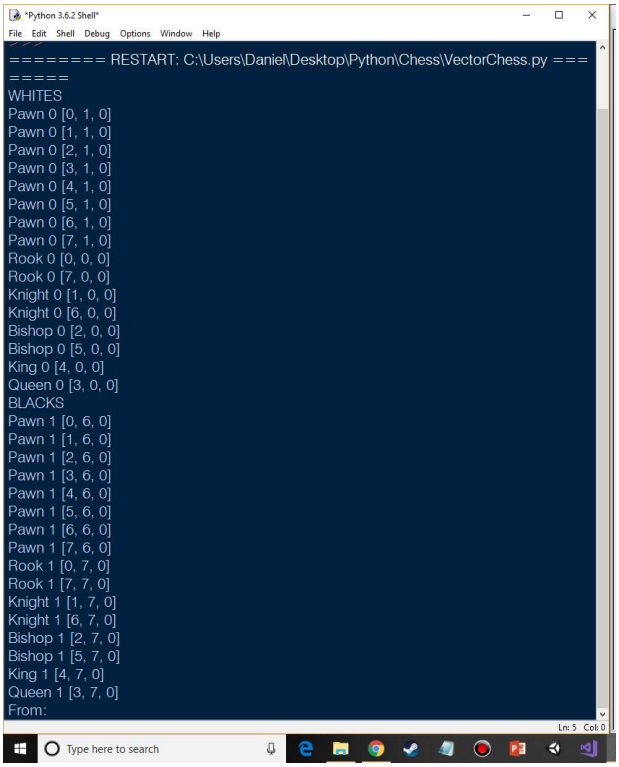


Image 2

Piece objects have been initialised with correct values (Type, Team, Position as Vector)

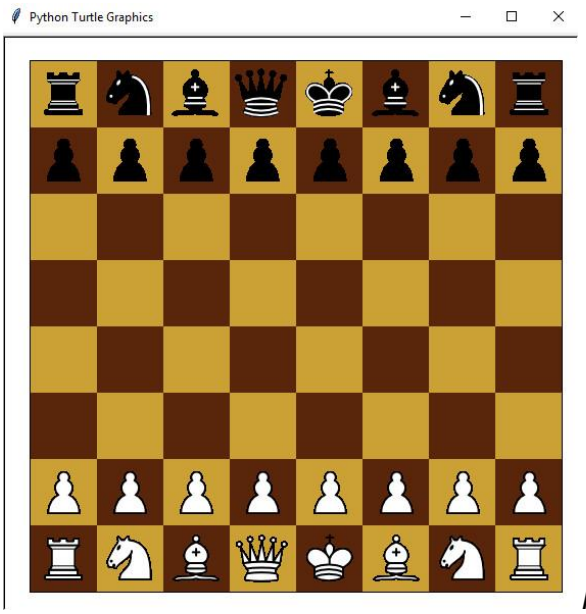


Image 3

Pieces have been drawn on to their correct positions

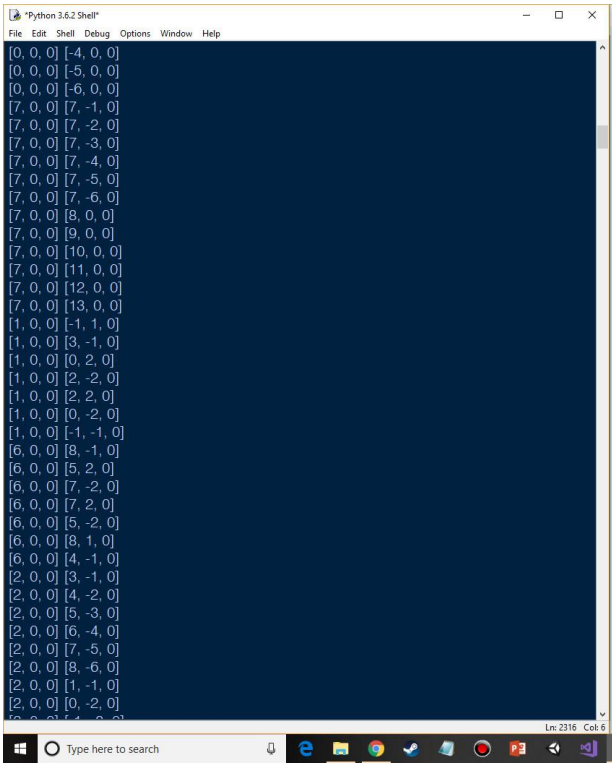


Image 4

All Moves printed. Below are the moves shown for White Knight at B1:

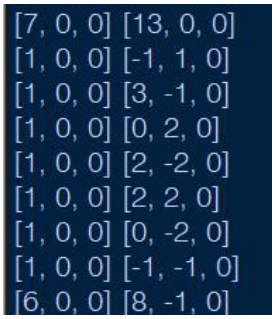


Image 5

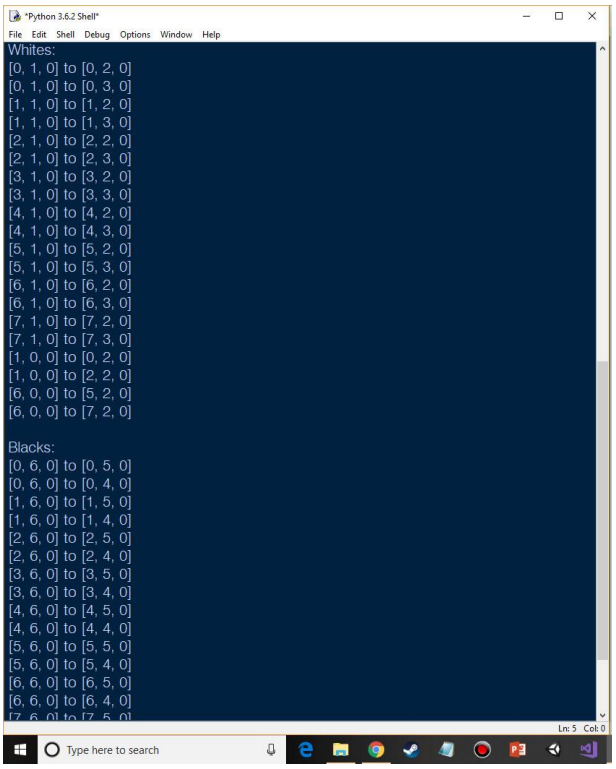


Image 6

Move lists have been refined so as not to include any illegal moves.Shown below are the new moves for the White Knight at B1:

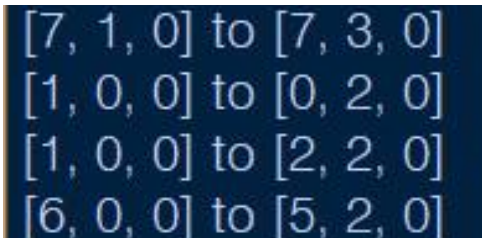


Image 7

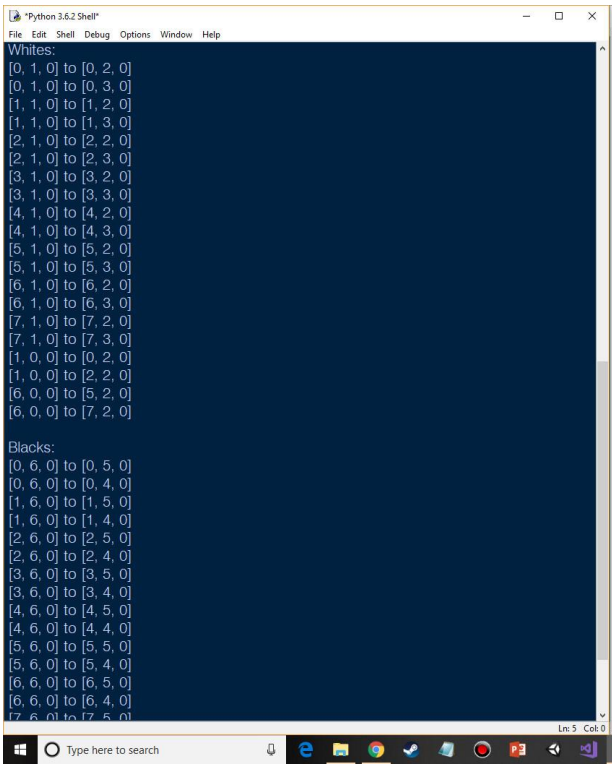


Image 8

Moves are separated into two lists, one for Black Pieces, and one for White Pieces

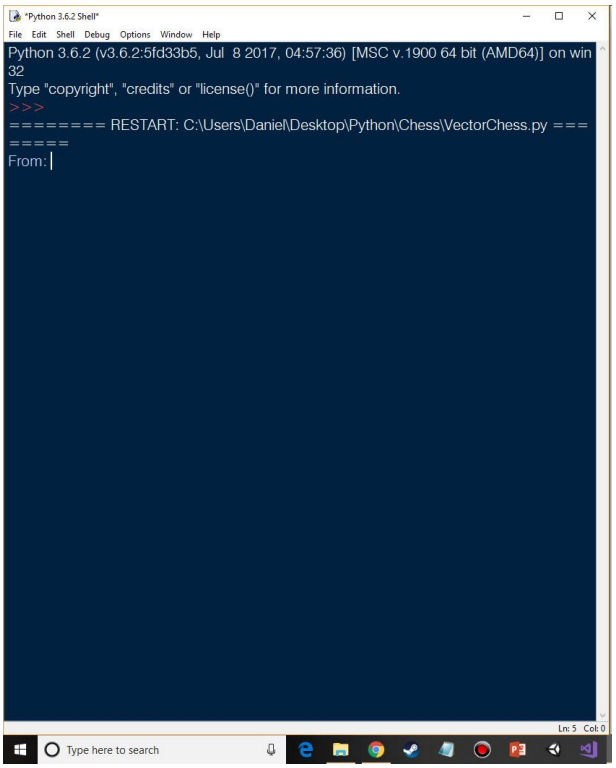


Image 9

Shows first Input point

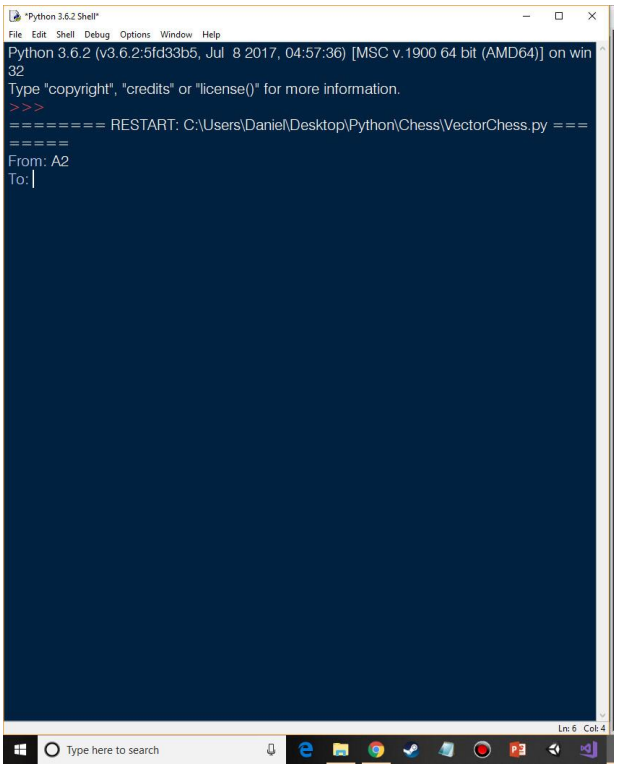


Image 10

After first Input has been filled, second input point appears

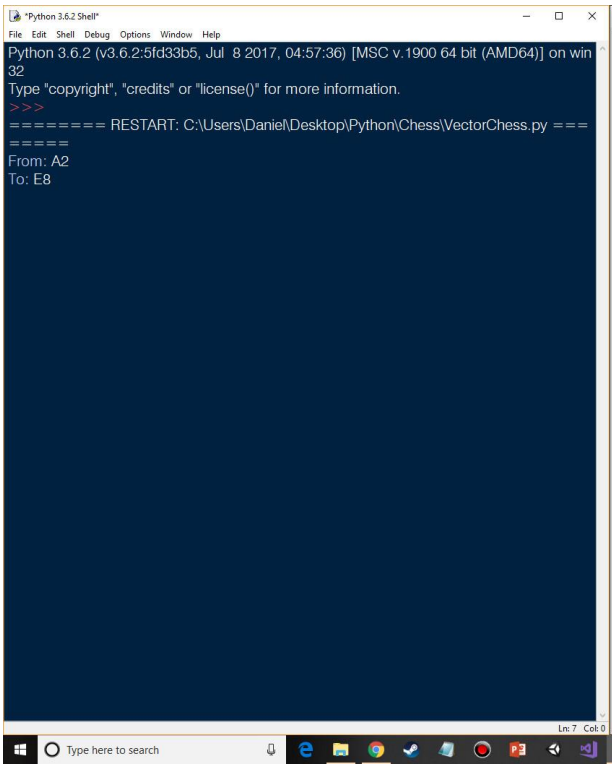


Image 11

Input for illegal move is made

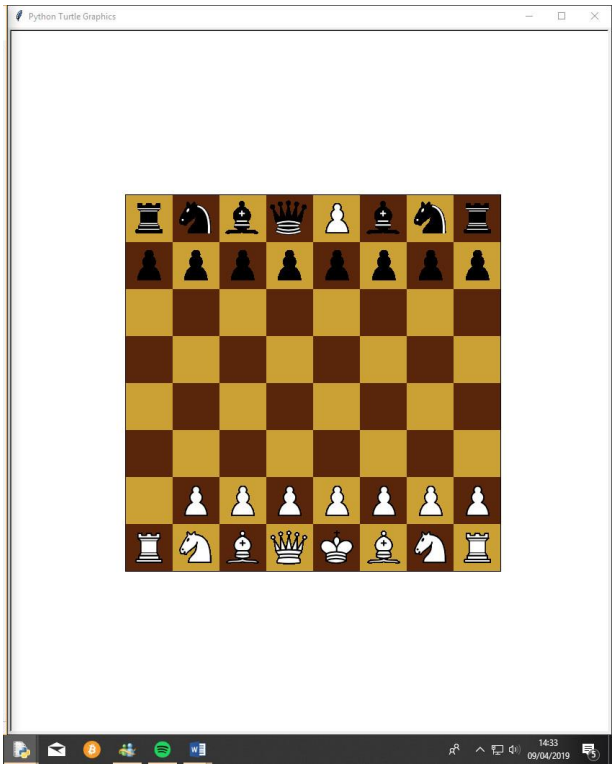


Image 12

Illegal move is made, no error message appears

Pawn should not be able to move from A2 to E8 in one move, therefore a message should have displayed. Because it didn't, it has failed the test

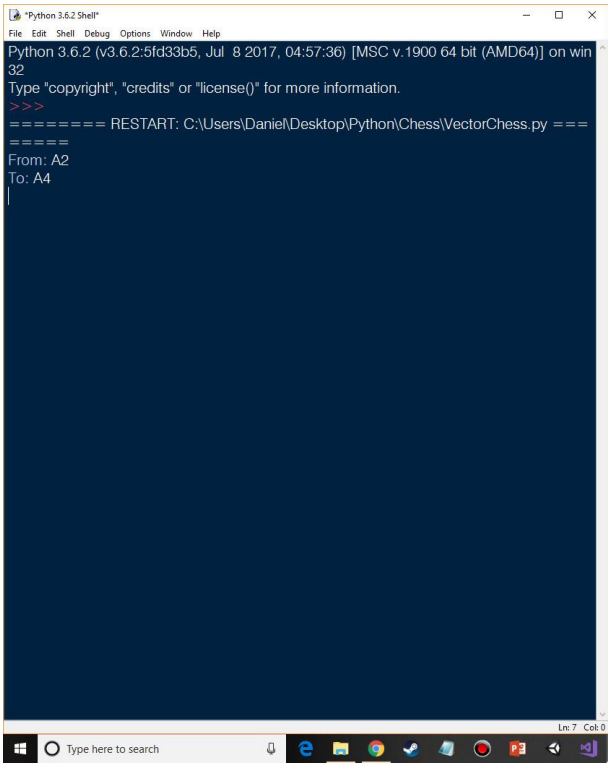


Image 13

Move input is made

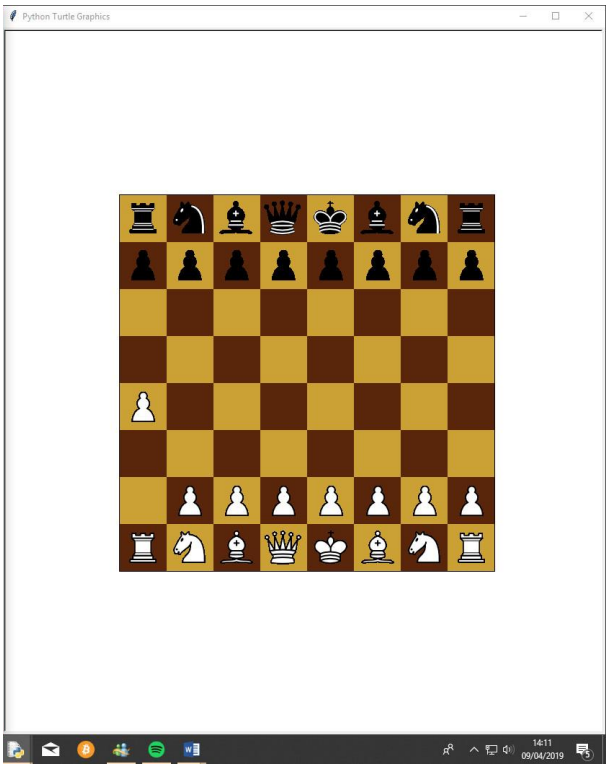


Image 14

GUI has updated to display new configuration

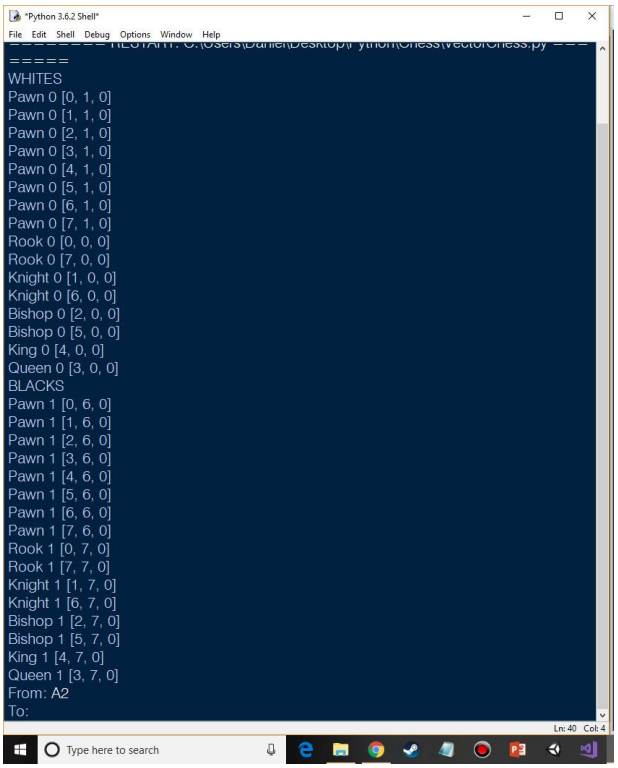


Image 15

First Pawn entry is at Vector [0, 1, 0] before move

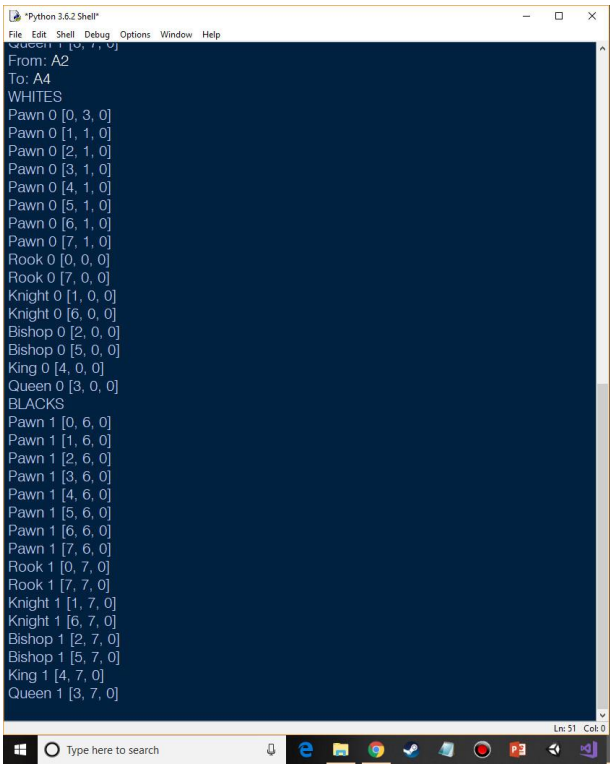


Image 16

First pawn is now at Vector [0, 3, 0] after move

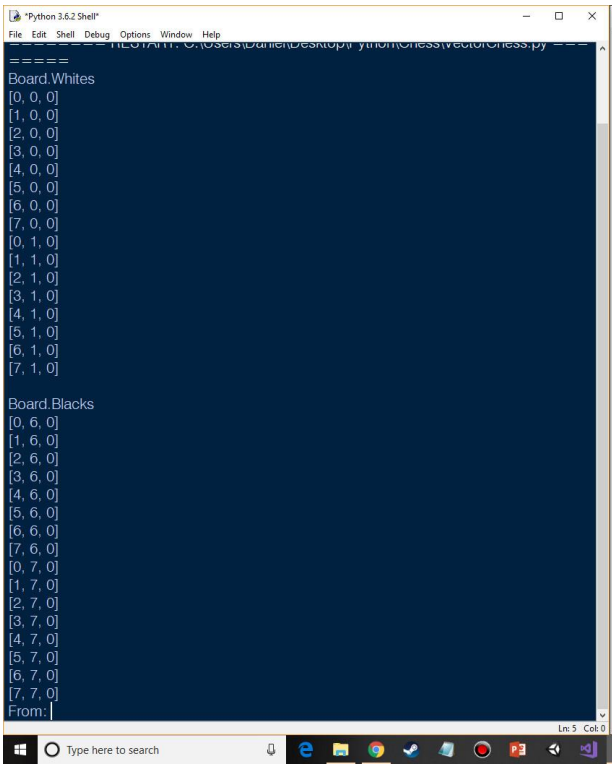


Image 17

Pawn starts at [0, 1, 0] (Line 9 of Board.Whites) before move

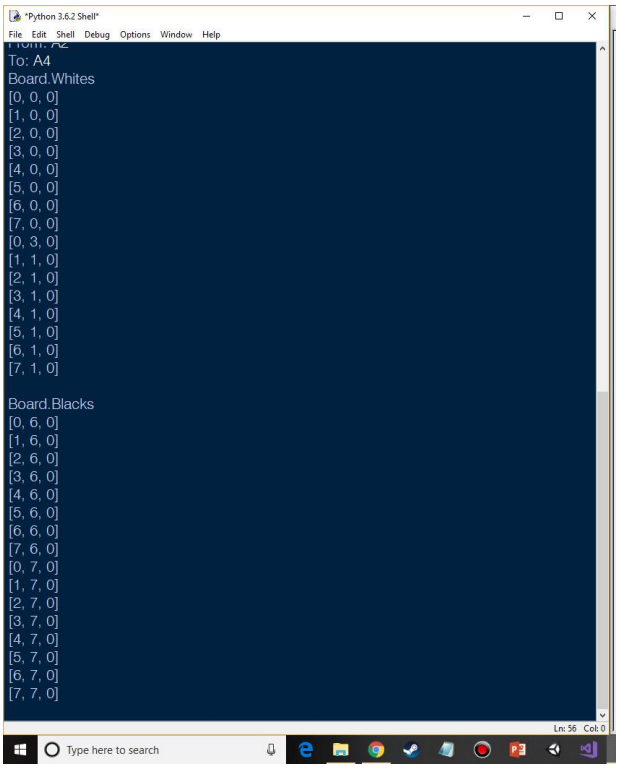


Image 18

Pawn ends up at [0, 3, 0] (Line 9 of Board.Whites) after move

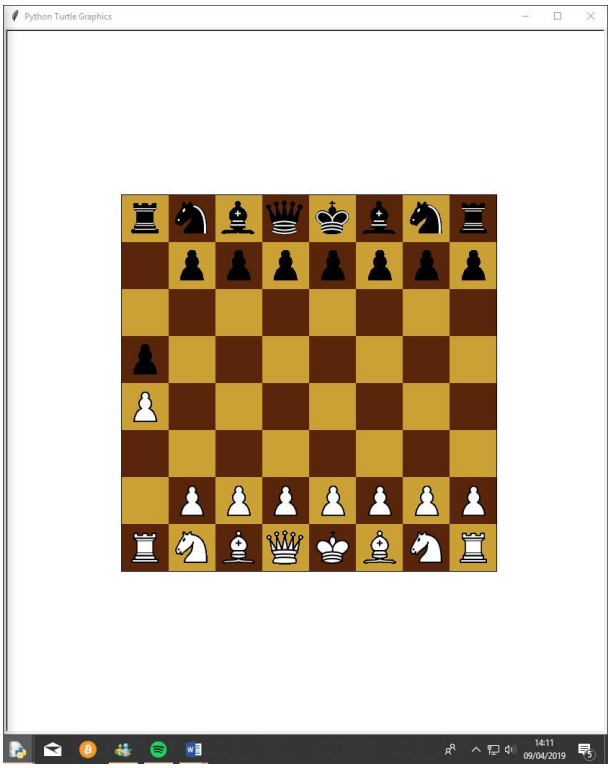


Image 19

When MiniMax is at Depth of 1, Algorithm works correctly

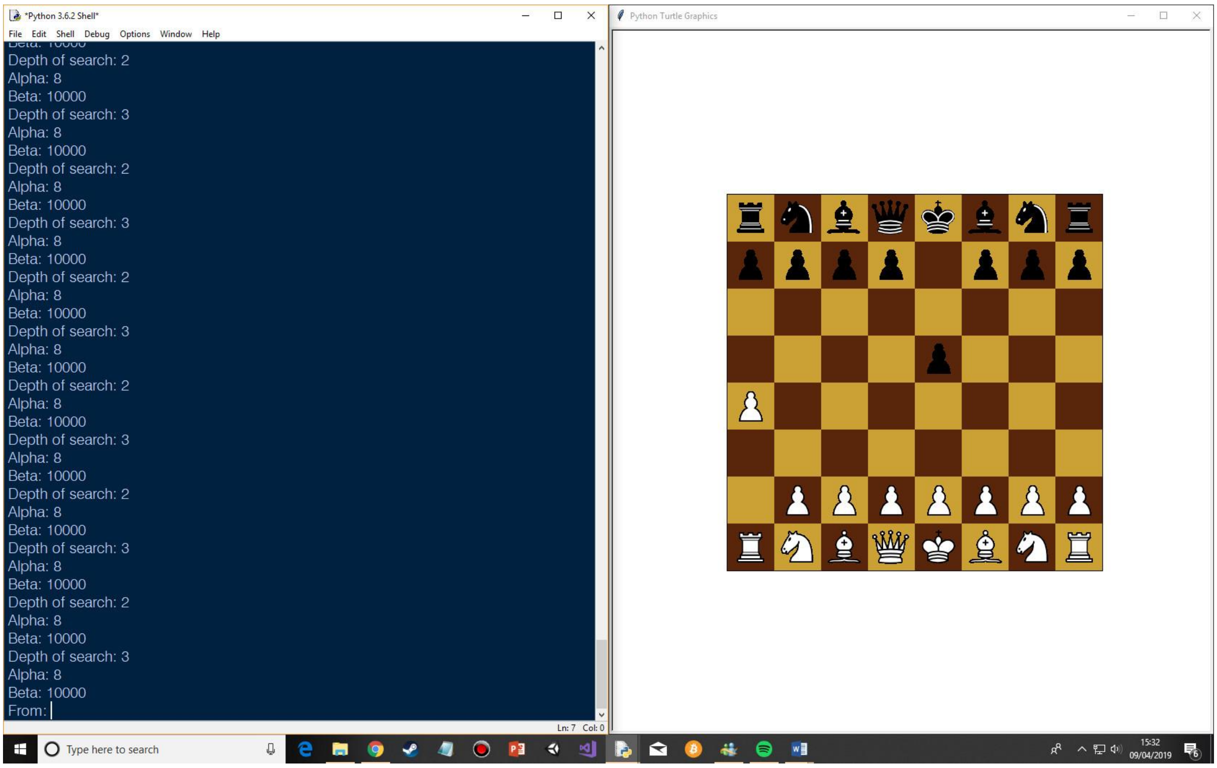


Image 20

When MiniMax is at Depth of 2, Algorithm still works correctly

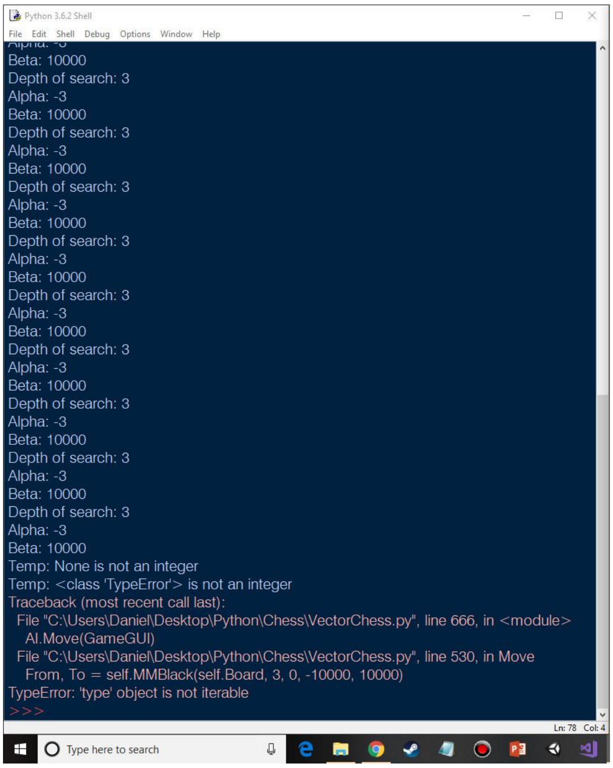


Image 21

When MiniMax is at Depth of 3, Algorithm returns a TypeError

Note: Through extensive debugging, the origin of this error remains unknown. If time allows, I will go back and try to remove the error, however given that MiniMax works to a depth of 2, I classify this test as having succeeded, as I was testing to make sure MiniMax is being applied, however there is a bug in the recursion which causes it to break after a certain depth

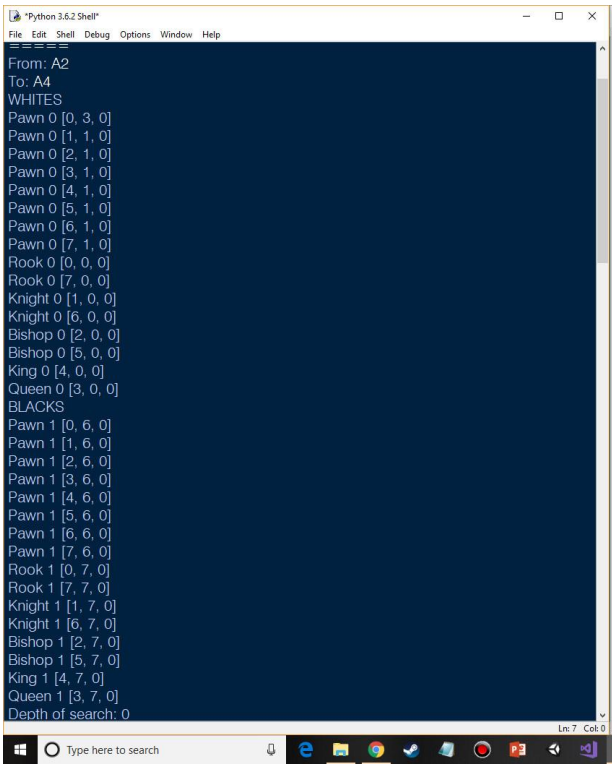


Image 22

The First Value under the 'BLACKS' column shows a pawn at Vector [0, 6, 0] before Black's Move

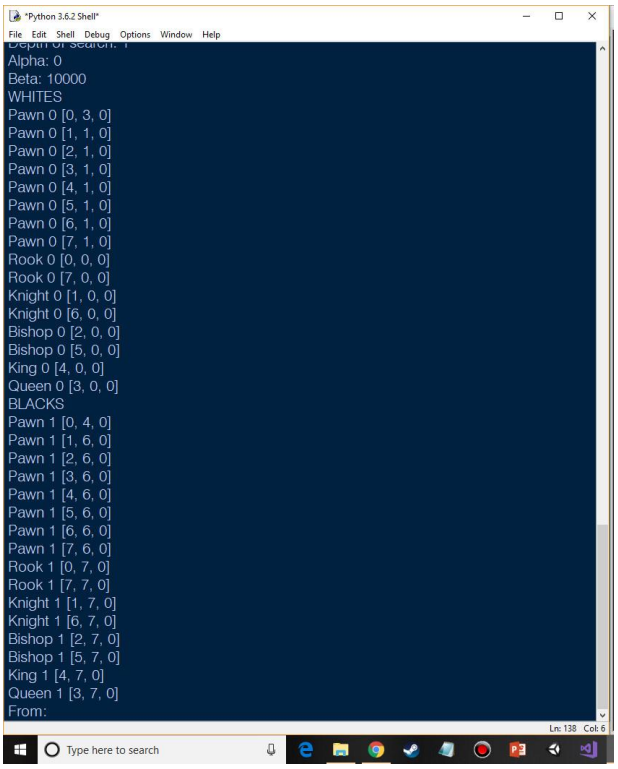


Image 23

The Value has now changed to Vector [0, 4, 0]. This shows that values are being updated correctly

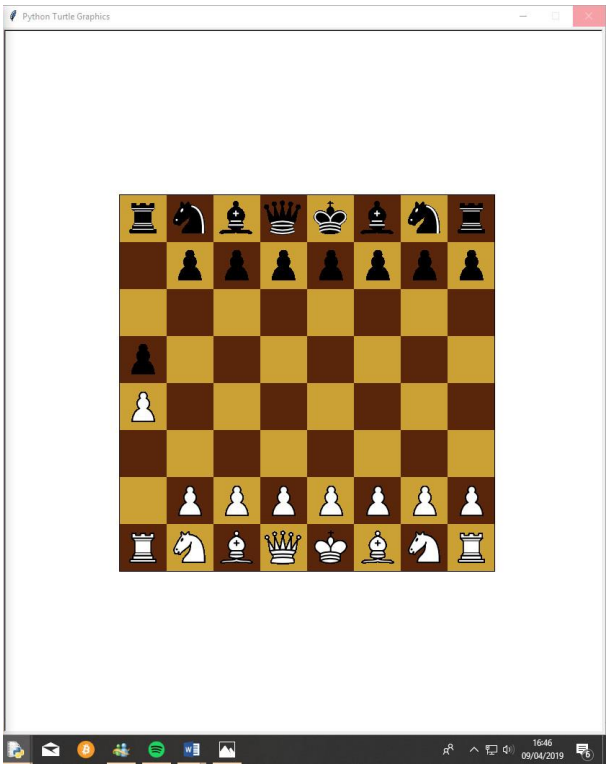


Image 24

The GUI has also updated to show the same information as in Image 23

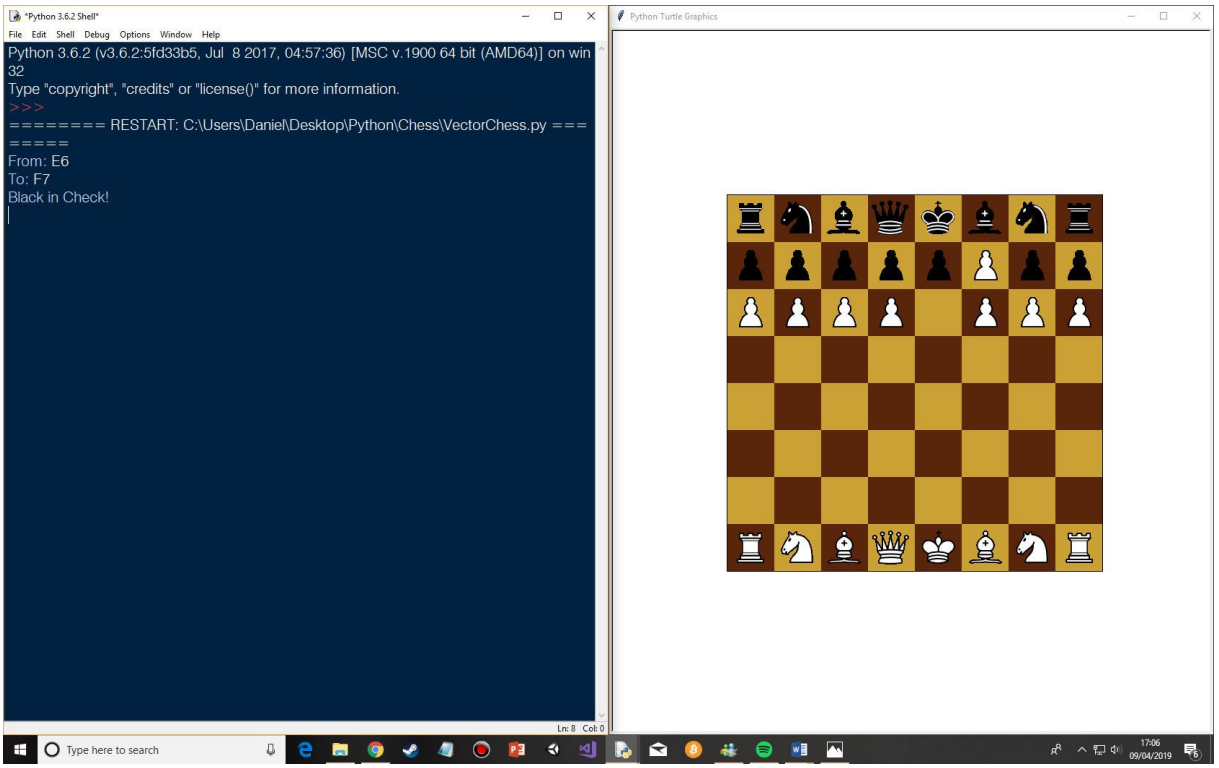


Image 25

I edited the game start values to get to a Check scenario quickly. As you can see, White put black into Check, and then a message was displayed to clarify this

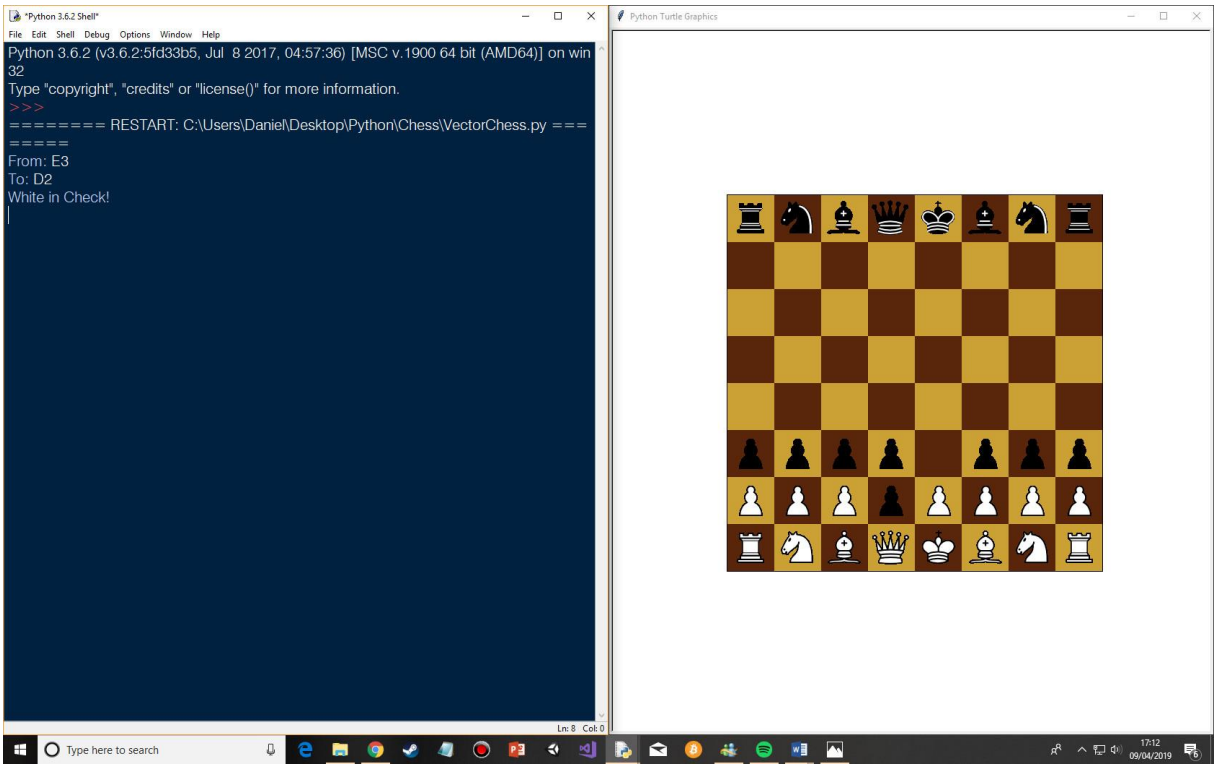


Image 26

As you can see, when a scenario puts White in Check, a message is also displayed

Evaluation

Overall, my application performs as required, with only a few minor issues. When the user makes their move, the program will allow them to move anywhere, however the logic behind the program will not update these values unless the move is legal, it is only the graphics which update. This could easily be remedied by adding a *'return'* statement to the *'Board.MakeMove'* function. If I implemented a return statement that would allow the function to return *True* or *False* depending on whether it actually made the move or not, then I could extend my *'For loop'* in the main game process to encapsulate the line calling *'Game.MakeMove(StartPos, EndPos)'* (*Line 721 of 'VectorChess.py'*). This would stop the game continuing until a move entered was considered fully valid by the verification algorithms, thus stopping the *'GUI'* class from drawing the move until it had been verified.

```
while Valid == False:
    StartPos, EndPos, Valid = GetPlayerMove()
    Valid = Game.MakeMove(StartPos, EndPos)
```

(Proposed fix for the error)

The error that occurred in the MiniMax algorithm that wouldn't allow me to surpass a depth of 2, was caused by hastily releasing the application without stress testing it first. The fault lies within a chunk of code that has been indented one too many times (*Lines 508-512 of 'VectorChess.py'*). After remedying the indent error, the code now works smoothly to however high a depth you would like.

As far as the endgame for the application goes, the deadline I had to meet cut me off short before I could implement different endgame scenarios, so it falls to the user to figure out when the game is over. It's not ideal, however given that the core logic of the game is in place, it is no harder than calculating it on a physical chessboard.

Post-Evaluation Solution Changes:

I revamped the entire main game algorithm (*Lines 711-727 of 'VectorChess' library*) to include a Checkmate detection and Stalemate detection system. Instead of what was there, I now have this:

```
Game = Board()
GameGUI = GUI()
AI = ArtificialAgent(Game)
i = 0
while i == 0:
    Game.GetWMoves()
    Game.RemoveWhiteCheck()
    Valid = False
    while Valid == False:
        StartPos, EndPos, Valid = GetPlayerMove()
        Valid = Game.MakeMove(StartPos, EndPos)
    Game.GetBMoves()
```

```
        Game.RemoveBlackCheck()
        ##Checkmate occurs when no moves can be made by the party in Check##
        ##A##
        if len(Game.BMoves) == 0:
            if Game.IsBlackCheck():
                print("Checkmate!")
                print("White Wins!")
                break
            ##Stalemate occurs when no moves can be made by current player and
    they are not in Check##
            ##B##
            print("Stalemate!")
            print("Draw!")
            break
        if Game.IsBlackCheck():
            print("Black in Check!")
        GameGUI.MakeMove(StartPos, EndPos)
        AI.Move(GameGUI)
        Game.GetWMoves()
        Game.RemoveWhiteCheck()
        ##A##
        if len(Game.WMoves) == 0:
            if Game.IsWhiteCheck():
                print("Checkmate!")
                print("Black Wins!")
                break
            ##B##
            print("Stalemate!")
            print("Draw!")
            break
        if Game.IsWhiteCheck():
            print("White in Check!")

    input("Press enter to close game.")
```

And I have implemented the proposed fixes stated in the evaluation. I believe that it will now pass the tests I set out to pass at the beginning

Post-Evaluation Testing:

Test	Pass/Fail	Reference to Image
Display error message if user enters illegal move	Pass	Image 27
Agent applies MiniMax to found moves	Pass	Images 28-30
Display Checkmate when relevant	Pass	Image 31
Display Stalemate when relevant	Pass	Image 32
Display Winner	Pass	Images 31 & 32

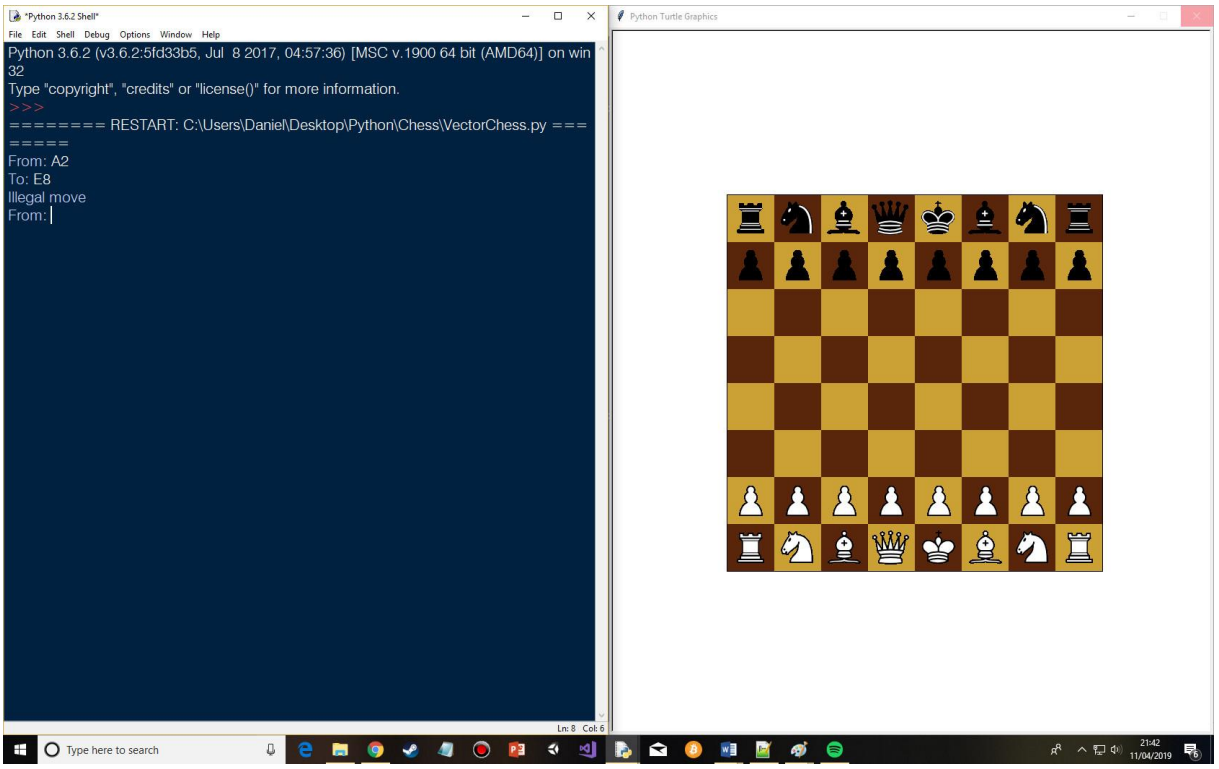


Image 27

As you can see, the program will now display a message when an illegal move is entered, will make the user enter a new move, and won't move the piece on the visual board

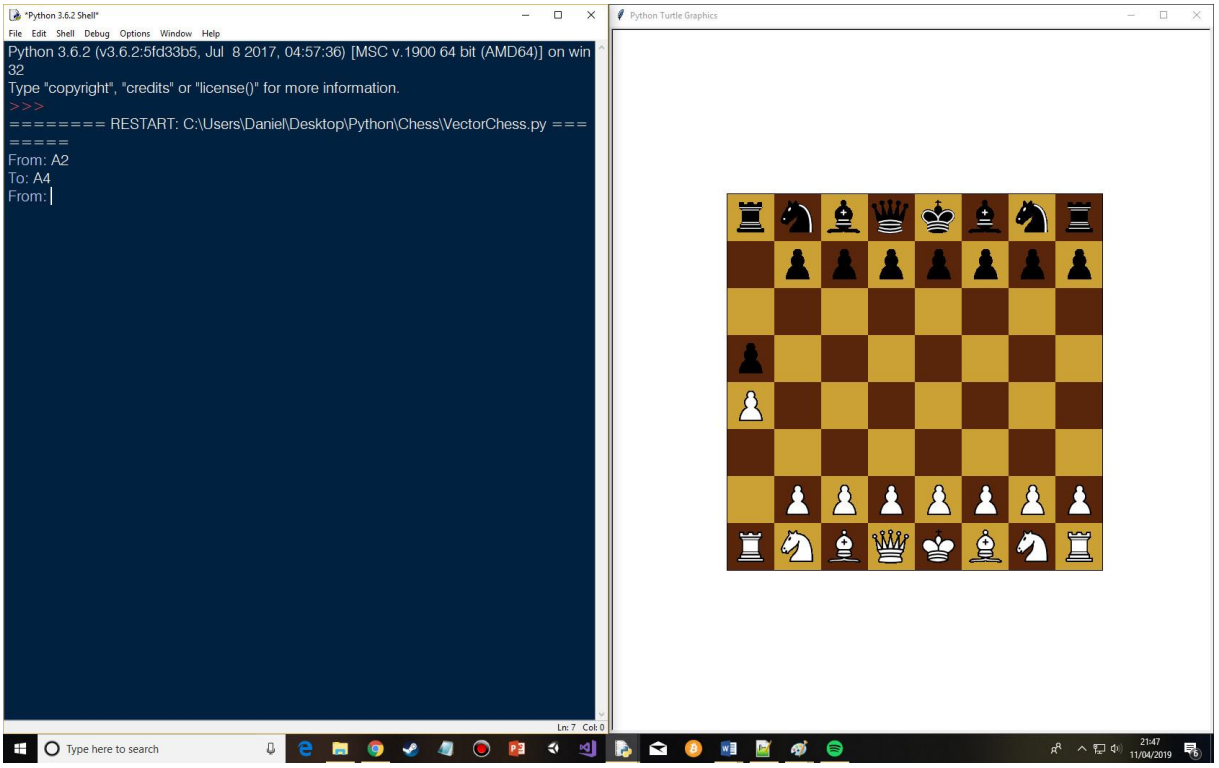


Image 28

This shows the computer's move when searching at a depth of 1

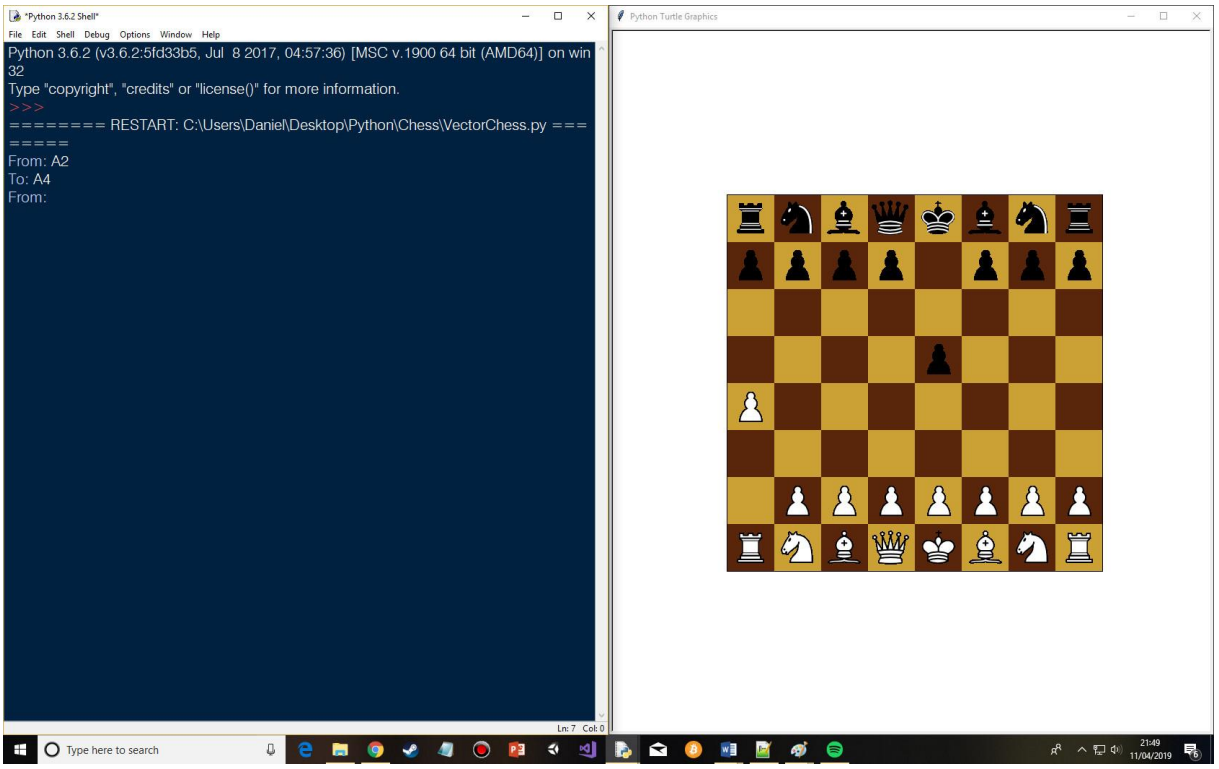


Image 29

This shows the computer's move when searching at a depth of 2

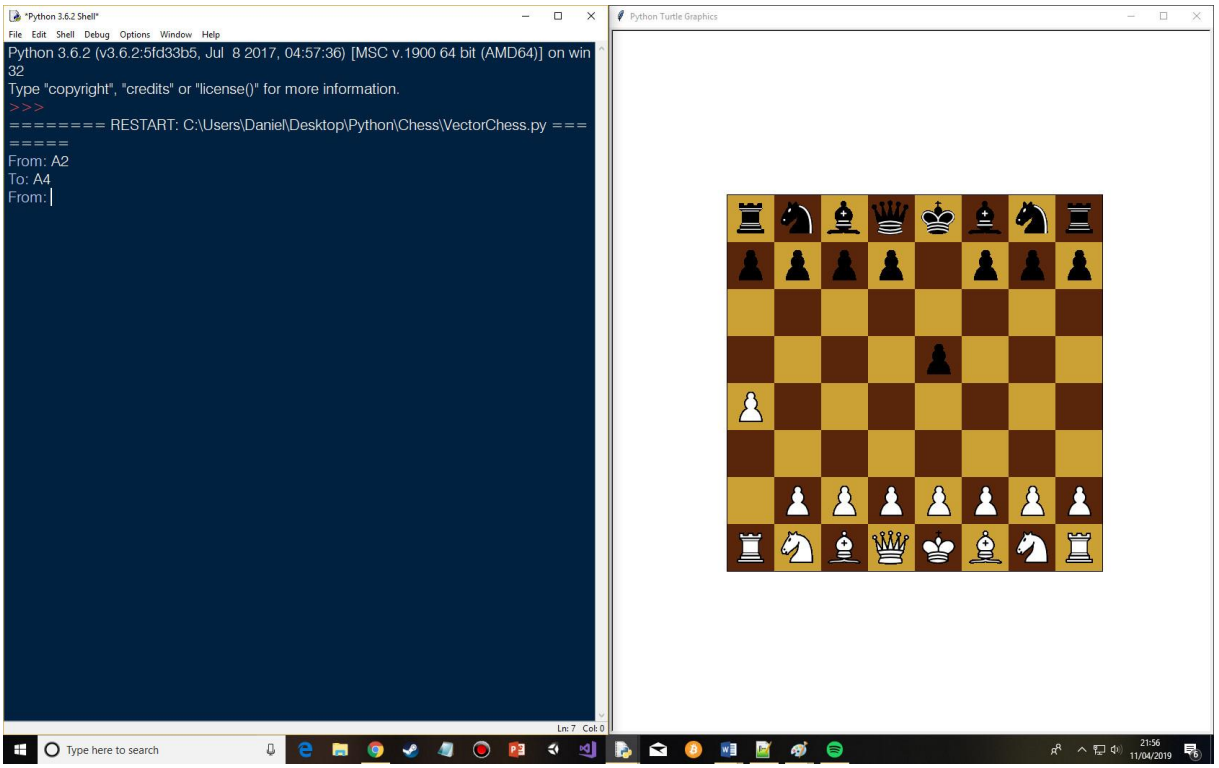


Image 30

This shows the computer's move when searching at a depth of 3

The move is the same as at depth 2, however this time it hasn't crashed when processing because I fixed the indented block

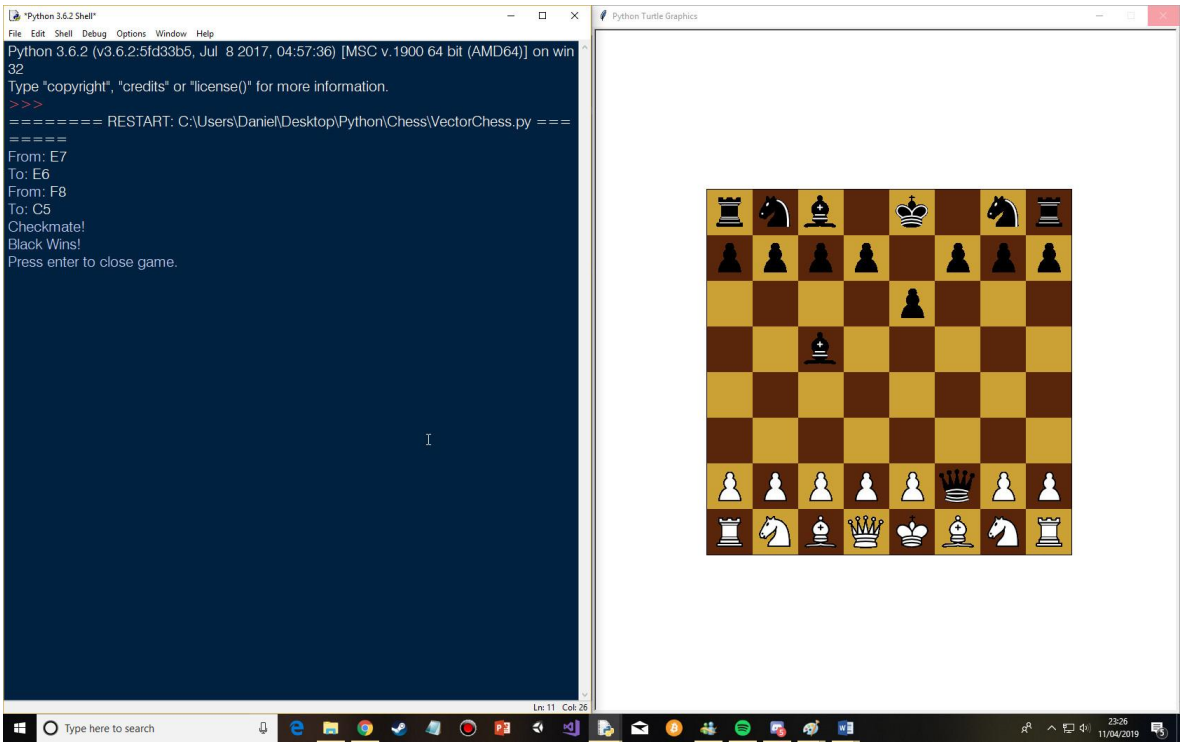


Image 31

As you can see, when forced into Checkmate, the game acknowledges Checkmate and displays it on screen who has won

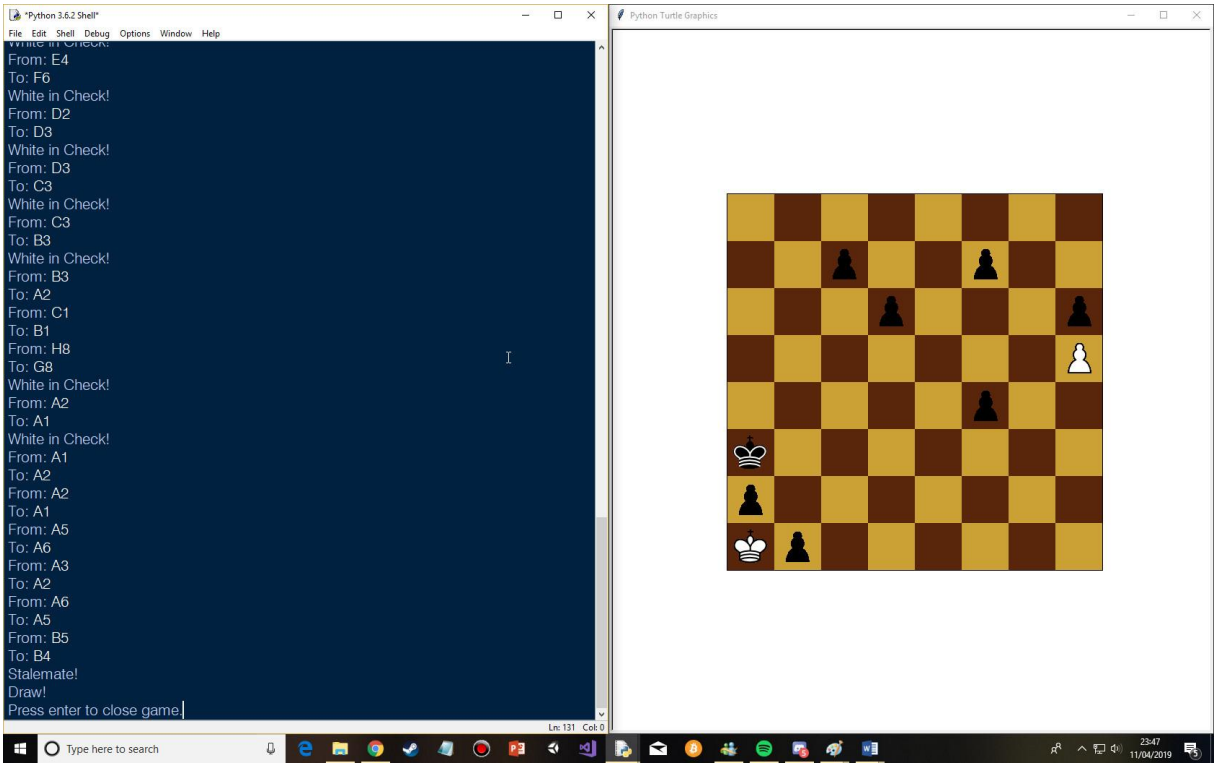


Image 32

The game was forced into a Stalemate scenario, and the correct message was displayed for it, along with displaying that there was no winner, but a draw instead

After re-testing the solution, I released an executable version to a small control group for feedback. The responses I got were:

"You should add the letters and numbers to the edges of the board so it's easier to figure out what each individual tile's grid position is"
I think this is a good idea, and one that I slightly overlooked in development. Whilst not essential, having the files and ranks displayed on screen could make for much quicker and more enjoyable games as the user spends less time counting tiles and more time actually playing

"The graphical window opens up over the console window. If you haven't used the application before, you might not realise you need the console window and then it makes it hard to figure out what to do"
This lends to one of my extension activities that would be to make the game have graphical input rather than typing into a console window. To remedy this now, I could always print a help screen on the console before the board starts drawing, then people can read the rules whilst the board appears and it would make it easier to understand how the game accepts user input

Extension:

Were I to undertake this problem again, I'd probably look more into Genetic Algorithms to help shorten the MiniMax algorithm's runtime. As it stands, the algorithm has to search through an exponentially increasing game tree, and whilst it does it effectively, it doesn't do it efficiently. The more I looked into the problem during the design stage and the programming stage, the more I found that MiniMax is actually highly inefficient for programming a Chess Engine as it can't scan the large game tree very quickly. I still believe it is a good entry point for any Artificial Agent programming, however Algorithms such as the *Monte Carlo Tree Search* are much better for Chess. I also believe that I overcomplicated my code. Using many classes to represent components of the game, whilst making it easier to compartmentalise and read through, actually don't lend themselves to the problem of Chess. The constant references to abstract datatypes appears to slow the processing speed and increase the memory usage, which are two massive pitfalls in the application. Bitboards were originally used because of their low data consumption, and given that a program using bitboards would have many more simple comparisons rather than complex ones, I feel they would be much better at increasing the efficiency of my code