*Othello AI*

Computer Science NEA

**Name:** *Samuel Venable*

**Candidate Number:** *2284*

**Centre Name:** Barton Peveril College

**Centre Number:** 58231

# 0 Contents

[**0 Contents**](#_tfp75dsgii83) **1**

[**1 Analysis**](#_5h5fv5y73l9a) **3**

[1.1 Statement of Problem](#_90scz7te70m5) 3

[1.2 Background](#_mthbiti33lwz) 3

[1.3 End Users](#_u9n7ihho91fp) 4

[1.3.1 Primary End User](#_ak4hjba0x894) 4

[1.3.2 Expert User](#_ny5gsbfw7cbg) 4

[1.4 Initial Research](#_af0tsbwg2h2g) 4

[1.4.1 Existing, similar programs](#_ufe2qk5b2hf5) 4

[1.4.2 Potential algorithms](#_fxyy70vqpcf0) 6

[1.4.3 Initial interviews](#_s89bw8qyv9hq) 7

[1.4.4 Key components](#_5d2txrxb5ph8) 10

[1.5 Further Research](#_yuwv01j2684v) 11

[1.5.1 Prototype](#_cecewnm58hrh) 11

[1.5.2 Second interview](#_y281ar2715i7) 13

[1.6 Objectives](#_o9yqsy3au79w) 14

[1.7 Modelling](#_guy6lh921wcb) 15

[**2 Design**](#_ijzy4fnc6zq4) **16**

[2.1 Call-Hierarchy Flowchart:](#_w512n04l3se2) 16

[2.2 MinimaxResult / MinimaxCall Explanation:](#_skx05baxxkm) 19

[2.2.1 Improvements](#_94bbv3m6xypu) 20

[2.3 Other Algorithms](#_3kd91yv3ov8b) 20

[2.3.1 Amateur AI Algorithm](#_3678yhly97do) 20

[2.3.2 Valid Move Function](#_e1as2dfpxjrc) 21

[2.4 Depth Selection](#_3vsr5maurgo5) 21

[2.5 Data structures](#_4q3i2u4vk5wk) 22

[2.6 Technologies](#_xiw2w9rl6wtx) 22

[2.7 Human-computer interaction](#_u5h4yqo5g9rn) 23

[**3 Testing**](#_r7pb6436jqxa) **24**

[**4 Evaluation**](#_7mlo95b14if2) **27**

[4.1 Overall Effectiveness of the System](#_9ac6vsfre8oc) 27

[4.2 Evaluation of Objectives](#_5hbp8pjhqnd7) 29

[4.3 End User Feedback](#_dms1z2udz6y4) 30

[4.3.1 Conclusions on Feedback](#_icwwxhzg915v) 31

[4.4 System Improvements](#_wkqmdcpcpbip) 31

[**5 Technical Solution**](#_hhjx9edou2pb) **32**

[Contents page for code:](#_3xugit6t3pzx) 32

[5.1 Prototype Code](#_5h2h31efaw4m) 60

# 1 Analysis

## 1.1 Statement of Problem

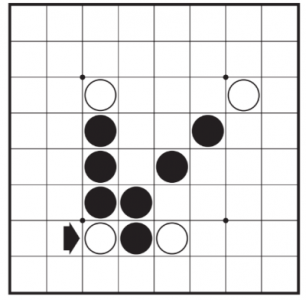
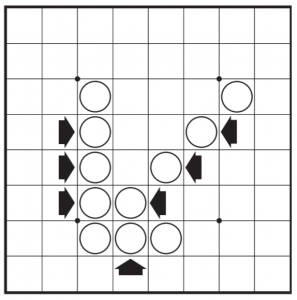
My friend Neev finds it difficult to get opponents to practise playing Othello with, and this stops him improving as fast as his friends, some of whom pay for coaches which really helps them get more chance to practise. He wants to improve without the cost of paying for a trainer.

## 1.2 Background

The game Othello was released in 1975 and begins as pictured below, with one team playing with white counters and the other black.



The objective is to have the majority of your colour counters/discs on the board at the end of the game. From the starting position, black takes their turn first and (unless one player has nowhere to place their counter), the players take it in turns to play until there are no moves left to play for either player (usually when the board is full). A turn consists of placing a counter on a square that outflanks one or more of the opponents discs, and these disks are then flipped. Outflanking means to place a disc on the board so that your opponent's pieces along a row, column, or diagonal are bordered at each end by a disc of your colour. This is shown below.

Disk placed by arrow Tiles flipped

## 1.3 End Users

### 1.3.1 Primary End User

My end user is Neev, my friend (an amateur Othello player) who asked me for an Othello game to practise his skills against. I will be communicating with him throughout my project to make sure I fulfil his requirements for the game so he can learn from playing against the AI.

### 1.3.2 Expert User

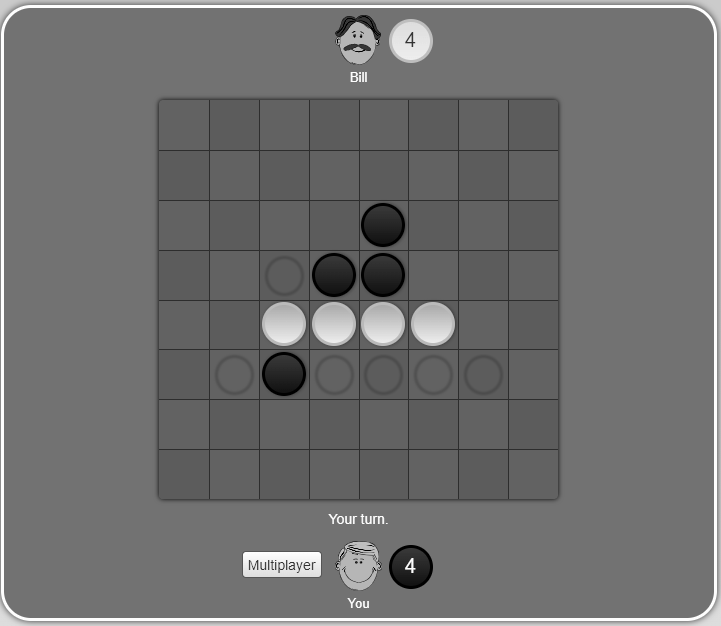
I will also have an expert user Matt, a Computer Science teacher at my college, as he has experience with algorithms such as minimax and can give some advice and helpful feedback on my methods and objectives. I will interview him as well as my end user to get advice and feedback on my algorithms.

## 1.4 Initial Research

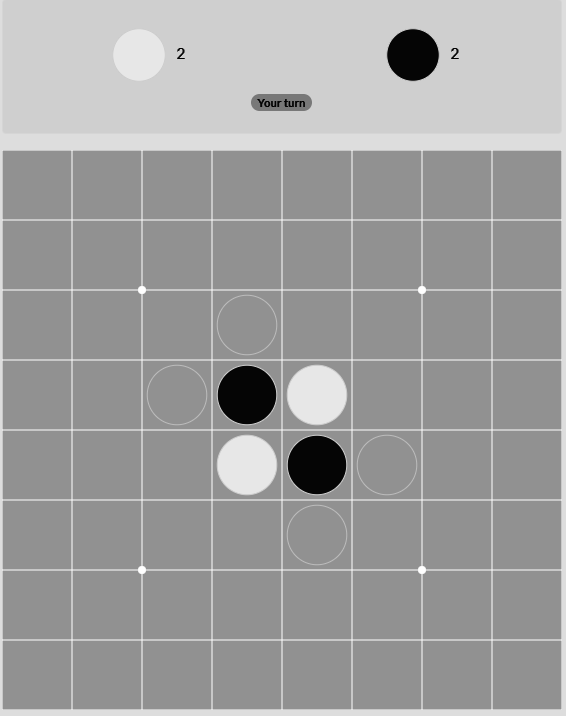
### 1.4.1 Existing, similar programs

There have been many othello-playing AIs already made, such as Saio, Ntest and Cyrano, however they’re not well documented online so it’s not easy to make too many comparisons between these engines and my own. Owen Shevlin, a student at Cardiff University, wrote a paper on creating an AI to play Othello¹, and he used the Minimax algorithm alongside a Monte Carlo Tree Search, so mine will be different in that I’ll be using a simpler minimax algorithm to focus on minimising bad moves rather than choosing the very best one.

In terms of user experience, there are a few online Othello games that I’ve looked at and they’re usually quite similar. eOthello², cardgames’ reversi³ and hewgill’s othello⁴ all share a real representation of an othello board that’s clearly laid out with black and white counters, and show the user all the possible places where they can go on each particular move. This, although a helpful addition, is not something I’ll be pursuing, as my version will be tailored to Neev and other real life Othello players who must learn to identify possible moves naturally.

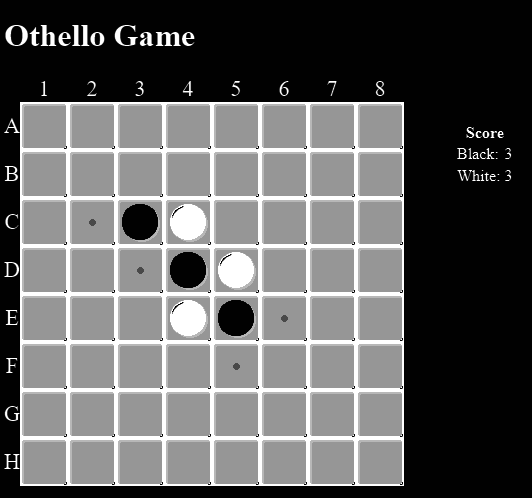


^Cardgames.io Reversi



^eOthello

Other versions allow you to click on the board to place a piece, which although a helpful feature, is difficult to achieve. I will attempt to find an alternative, intuitive interaction method for my program. One thing almost all these programs lack is showing clearly which counter has been placed by the computer and which counters have been turned by that move, as you can see below - the AI placed at C3 and the game showed a short animation of the counter at D4 flipping, but now that has happened there’s no way of telling where the last counter was placed and this makes it hard to follow how the game has progressed:



^Hewgill’s Othello

I think it would be a helpful addition to my project to ensure this is clearer, as in real life you’d watch the counters being turned over and this avoids confusion about how the board has changed.

¹Shevlin, Owen. “Creating an AI to play Othello - Final Report.” *Project Allocation & Tracking System (PATS)*, 5 May 2017, https://pats.cs.cf.ac.uk/@archive\_file?p=708&n=final&f=1-Final\_Report.pdf&SIG=48a08fc35e0169234716d776bec6f534c2e97babbba9edc889d4c99291178a38. Accessed 8 August 2022.

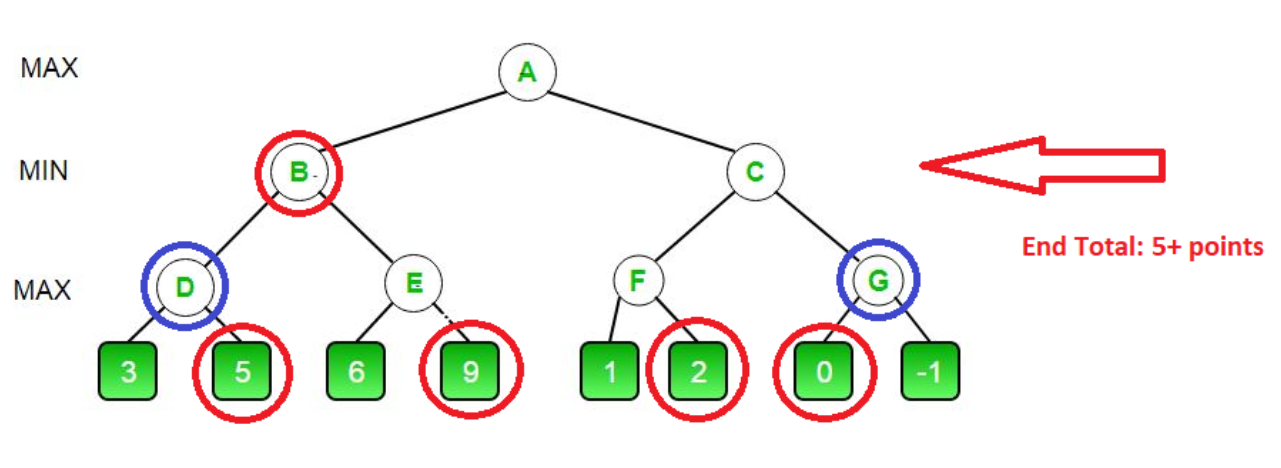
²eOthello. *eOthello: Play Othello online*, https://www.eothello.com/. Accessed 2 September 2022.

³“Reversi | Play it online!” *CardGames.io*, https://cardgames.io/reversi/. Accessed 2 September 2022.

⁴“Othello Game.” *hewgill.com*, https://hewgill.com/othello/. Accessed 2 September 2022.

### 1.4.2 Potential algorithms

The main algorithm of my project will be the minimax algorithm, which will be used to determine where the professional AI moves. The basic idea of it is to search through every possible move at a point in the game, every move from that point and so on down to the depth that it’s searching. When it gets to the bottom of this move tree, it evaluates the board state at that point, backpropagates back up to the first move and chooses the one which minimises the potential for the opponent to be in a good position in some number of moves time. This should provide a good standard of play, as it should at least avoid the worst moves. Here’s a scaled down example below.



In this example, A is the current board state. B and C are possible board states after playing one of the two possible moves from this position. D, E, F and G are subsequent possible states depending on where the player goes, and the numbers below them represent the value of the board state after the AI has taken its turn again. Here we can see a comparison is made from the bottom up - each of the two possible moves are compared and the higher one chosen. At the level above (which is chosen by the player) we assume the player chooses the path with most gain for them and therefore least gain for the AI (so the smallest value). Then back to the AI’s go, it chooses the largest number assigned to B or C, which is the 5 assigned to move B, so this is the move that will be chosen.

I’m also considering using alpha-beta pruning along with this algorithm to improve its efficiency and therefore its strength as an opponent, but I’ll ask my expert user Matt about the possibility of this before moving forward, as it may be too much to include. The advantage of using Alpha-Beta pruning is that it may allow me to set the depth of the minimax’s search higher, as the idea of this variation of minimax is to ‘prune’ branches of the move tree which when evaluated, are already worse than a previously examined branch and can therefore be left without further searching, saving computational complexity and therefore allowing for a deeper search which should provide a better move choice.

### 1.4.3 Initial interviews

This is the transcript of my Interview with Neev, my end user:

1. **What are your most important requirements for this program?**

*The most important thing for me is for it to be simple to interface with, especially with placing counters, so it’s not easy to get column and row mixed up and so I can re-enter my go if I place somewhere that I shouldn’t be able to. I’d also like more than one standard of AI to play against, so that as I get better I can have a better opponent to practise against.*

1. **How long are you happy to wait for the AI to choose its turn (assuming it will make a better move the longer you wait?**

*I do care about how good the AI is, but I think if it takes much more than 5 seconds it would slow down the game too much for me which would interfere with my concentration and experience of playing - as I could play against another person and not have to wait much longer for a go and this is one of the benefits of having a program to practise against.*

1. **Do you need there to be a player vs player mode in this game so you can play with friends?**

*No, I have a real board at home if I want to play against someone else, so the only thing I need this program to do is let me play against the AI.*

This is the transcript of my first interview with Matt Arnold, my expert user:

1. **Firstly, are these realistic goals to be able to code in the time we have?**

*Yes*

1. **Are the questions I’ve put in my statement of problem (at the bottom of this email) enough, or would you recommend any others to investigate?**

*I think you've created a good set of questions. In particular I like the idea of comparing the performance between different amounts of 'lookahead' on the minimax AI. It would be good in your conclusions to see a graph/chart showing this relationship. Does adding more lookahead improve the performance at a linear rate, or do the gains diminish?*

1. ***Do you have any experience with Monte-Carlo tree searching(MCTS) or Alpha-beta pruning (or another method), and could you suggest which might be best for me to try and make?***

*From my understanding, alpha-beta pruning with minimax would be more reliable but does require a well-defined move evaluation algorithm (so that different possible moves can be compared). I can imagine that, for Othello, the difference between the number of white and black pieces would produce a good evaluation function. You may want to experiment with using different functions for this though. For example, having rows or sections of pieces near the edge of the board is better than sections of pieces in the middle that could be trapped by the other player.*

*MCTS could also be good, but I think it might be more computationally expensive to be successful because the randomised games may need to run until completion.*

1. **Is this an appropriate difficulty to get top marks, and is there anything I could add to improve the score it’s likely to get?**

*This is good already*

1. **What should I focus my initial research on?**

*How the algorithms work, how to represent the game tree abstractly (ideally using a Tree data structure), how to define success of the AI and the evaluation function.*

1. **How should I determine an AI’s successfulness - would it be too much to get them to play against each other or should I just get friends and family to play games against it to discover how effective each is?**

*The advantage of playing against humans is that it is closer to the purpose of these algorithms - to win against humans. However, it will be quite difficult to control for human ability if multiple people are playing against the algorithms. In addition, this will take a long time to get a lot of data.*

*There may be a way to have the AIs play against each other. This will be much quicker and easier to gather data for, but that data may be less useful. If you did this option, you'd want to have some kind of baseline AI that all the others are compared against. A good example might be one which picks random moves every turn.*

**Conclusions from this:**

This has given me some really helpful information and pointers about where I can take my project - I certainly agree that a graph/conclusion about how far a minimax “lookahead” affects its performance would be an interesting thing to find out. However, I worry that this will be very difficult data to collect, as I’d need a very large sample size to ensure that the results aren’t down to the skill of the users, so this might be out of the scope of this project. Regardless of this, I will investigate how deep the minimax algorithm can be set to go without running for so long that it interfere’s with the player’s experience (5 seconds as Neev estimates above), to ensure that it is as powerful as it can be.

I think a minimax algorithm on it’s own would be the best option for me, as with a good evaluation function it should still yield a good result, and MCTS especially would be difficult to implement here as (as Matt mentions) the random games would have to run until their end and this would likely take too long and ruin the experience for the player.

As Matt points out in response to question 5, working out how to evaluate the score of a move and compare the options to one another will be one of the most difficult but important parts of my project, so I plan to research this thoroughly and teach myself the basics of the game so this might show in the AI’s playing strategy.

In terms of determining the AI’s success, I believe the best route forward is to collect as much data from human players as I can, as this will give me the most accurate data, and although it’s not so easy to collect, I’ll ask friends with limited knowledge of the game to play, allowing me to work out how the AI will fare against this level of players as this is what it’s aimed at.

### 1.4.4 Key components

Having researched other examples of Othello games with AI opponents, there are some components shared by most (if not all ) versions of the game, that I will also have to implement as detailed here.

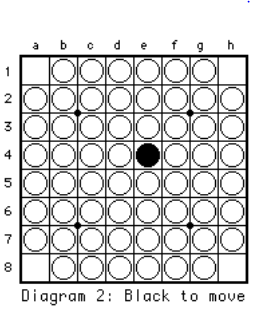
To represent the playing of an AI versus a human, I will need to create a digital version of the board in which a player can input their turn easily (by entering the row and column where they’d like to play), and the game will display information such as:

* Score
* Time Played
* Moves Made
* Which player’s turn it is
* Whether a turn can be taken by either player or if it must be skipped due to no placement opportunities (note that a player cannot voluntarily skip taking a turn)

For an AI to tackle the playing of Othello, it will need the following:

* The ability to recognise which spaces of the board are possible squares to place a counter on its go.
* A heuristic function to value and compare each of the possible placements dependent on the benefit to the AI of the board state after its turn is over, the difficulty/success of the opponent's subsequent turn, and whether it is likely to lead to a winning position/game for it.

A good thing to note for the use of a heuristic function is that a strategy of just choosing the move which will turn over the most discs is often very easily countered by a good player, as seen below, black has just one disc, but if played out, black will win 40-24 due to getting all the remaining moves. This (although a dramatic example) shows that it is likely for the best algorithms to need to implement a more complex algorithm if they are to be successful.



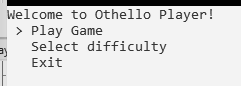
## 1.5 Further Research

### 1.5.1 Prototype

Code at the end of this analysis. See appendix Page 60

I decided to make my prototype a basic two-player version of the game, where you play against another person (or yourself). This required me to implement a large chunk of the final program - menu system, move selectors, disc turnover checkers, valid move placement checkers - and showed me that it’s definitely possible to complete in the time I have, while giving me a good foundation to work on making the AI’s to play against.

First of all I made a menu screen with an arrow selector controlled by the arrow keys, layed out as seen below:

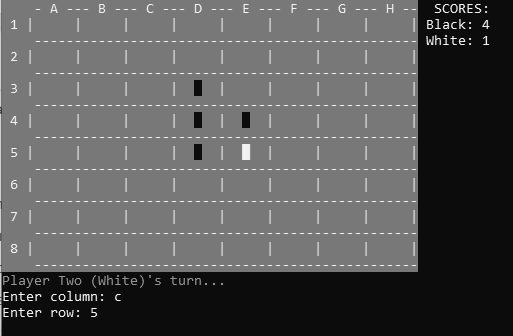


Select difficulty has no purpose yet but will be used to select the AI type once they’re made.

Selecting ‘Play Game’ takes you to a view of the board and allows the Player One(who is always black) to enter their first move:

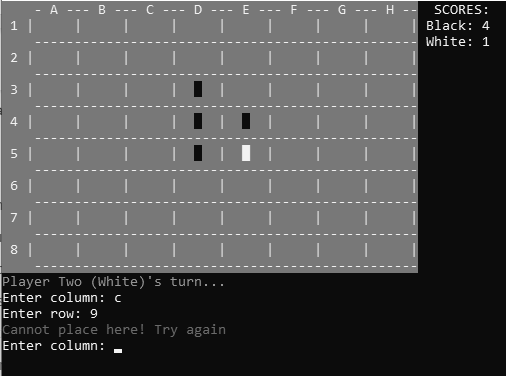


It will then (if a valid move) place the counter, flip the correct associated counters, and allow the second player to place.

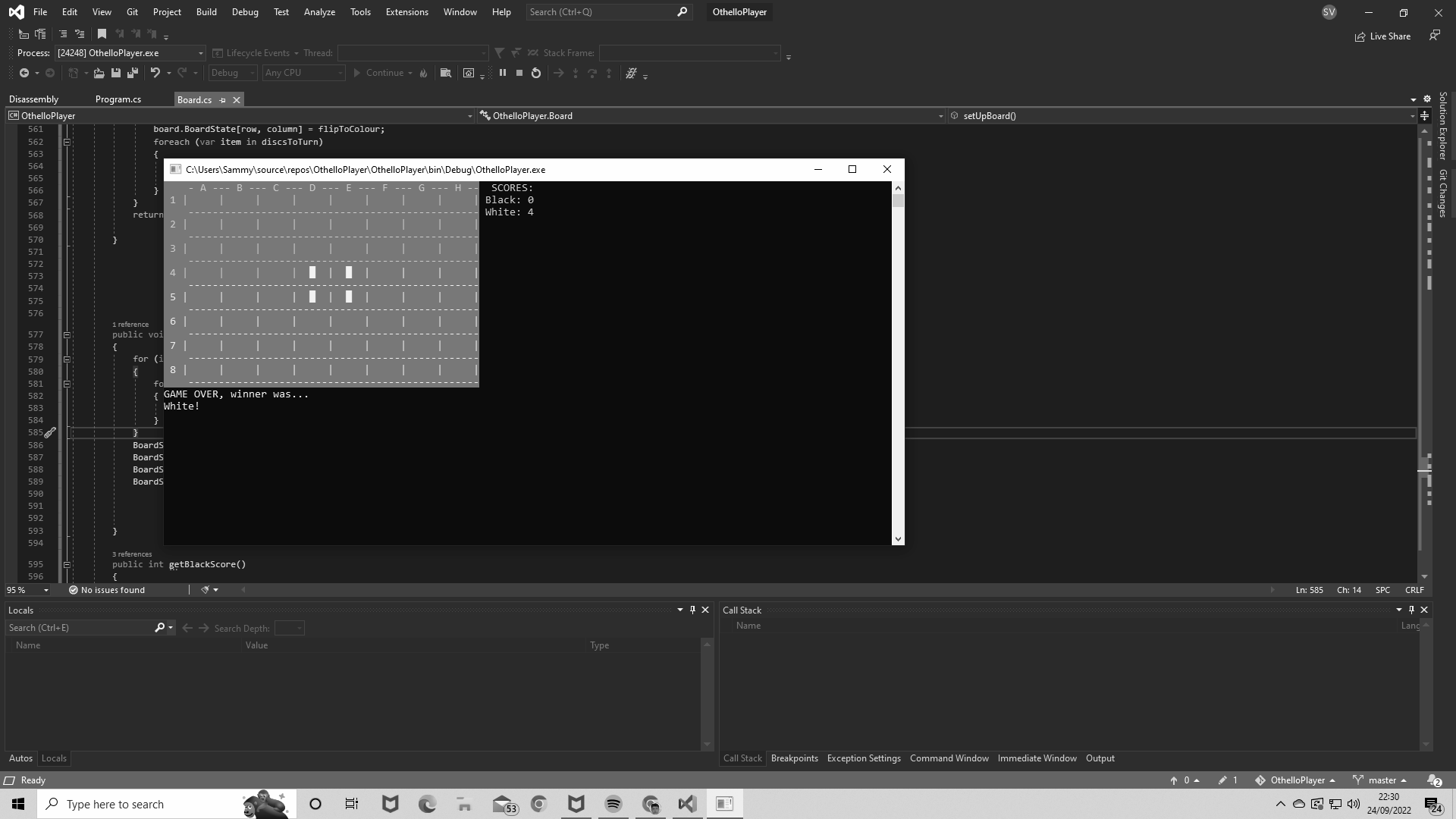


If a player tries to place a counter in an invalid position (ie it won’t flip any counters or there is already a counter there), it will display a message and ask the player to enter a different place.

### 



At the end of the game (once there are no more moves to play), the program will count the counters of each colour and declare the winner:



So far I’ve found no bugs and it works well, so I’m confident to move onto making the whole program and focusing on the AI algorithms. The code from this prototype will largely be carried over and used for the full project, as apart from the turn taking mechanics, it will be useful with the AI incorporated.

**Code for prototype: Page 60**

### 1.5.2 Second interview

This is the transcript from my second interview with Matt Arnold, my expert user.

**1. What's good in there**

*It appears that the game has been created and seems intuitive enough to use. Clear distinction between the black and white pieces. I like that there's letters for columns and numbers for rows (rather than numbers for both) because it makes it easier to type in a cell reference.*

**2. What should I change for the final project (obviously noting that it'll be slightly different given there will be an opposing AI rather than a second player) and what extra features would you like to see?**

*As the prototype is mainly based on the game itself and the UI, I can only comment on those parts. I have a concern that, after the AI chooses its play, the interface might make it confusing to see where they played. This is because the played piece and the resulting flips happen instantaneously. It would be useful to make it so that it's clearer what's happening on the AI's turn (and perhaps also the player's turn). A few possibilities include: highlighting the cell that has just been played to, introducing a delay after playing a piece before each other tile gets flipped. This second option would give a much closer experience to playing the physical game.*

**Conclusions from this:**

The comment on the letters and numbers on the comments was helpful, as I previously did have numbers on both but decided to change it for ease of use. I also agree that when the AI is introduced, I need to make it clear where it’s gone and also which pieces it has turned over. To remedy this, I plan to add a pause and a message saying ‘AI is thinking… ’ for a few seconds (possibly while the minimax program runs as far as it can in the time) before changing the board and the score. I will also set the pieces which are newly placed/flipped to be yellow for a couple of seconds and output text saying where the AI went, so the user knows what happened.

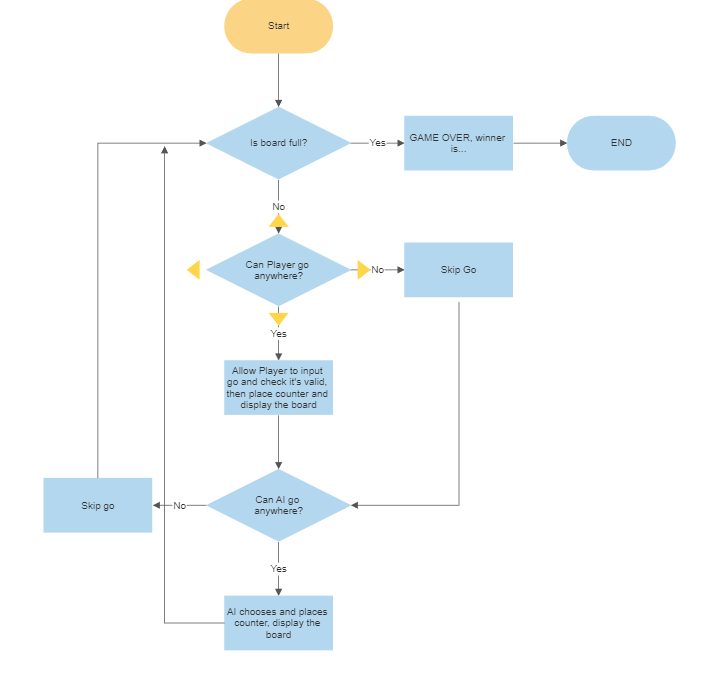
## 1.6 Objectives

1. Develop two AI’s which can choose moves and play against a user
   1. 2 difficulties of AI opponents to play against: amateur and professional.
   2. The Amateur AI will choose the move which turns the most counters on its go.
   3. The Professional AI will be determined by a **minimax algorithm** with the depth set to the highest it can go without taking more than 5 seconds to evaluate on any given go.
   4. Evaluation function which values each possible move of the AI each turn, and is utilised in the minimax algorithm.
2. User interface in the console
   1. Board which displays where each counter is.
   2. Allow the user to enter their move by inputting a letter and a number corresponding to a row and column on the board.
   3. Inform the user if their input wasn’t in the correct format and let them try again.
   4. Inform the user if their input was out of bounds (not 1-8) and let them try again.
   5. Inform the user if the move they chose is illegal (if there’s already a counter there or doesn’t flip any counters) and let them take it again if so.
   6. Display the new board state after each turn.
   7. Counters represented by the character ‘█’ in black and white colours.
   8. Current player’s turn is displayed.
3. Menu option for selection of difficulty
4. Check if the game has ended after every move.
5. Check if the player has a possible place to put a counter on their turn or whether they must skip their go, and send a message if the latter.
6. Be able to place a counter and turn all the counters which are flanked by that colour.
7. Display each player’s score on the console throughout the game and at the end with the winner declaration (or draw).
8. It is clear where the AI has placed on it’s last go.
   1. Counters placed and flipped by the AI will turn yellow for a short time after the turn is taken.
   2. AI’s last-turn coordinate will be shown under the board throughout the player’s next turn
9. User can exit the game from the menu

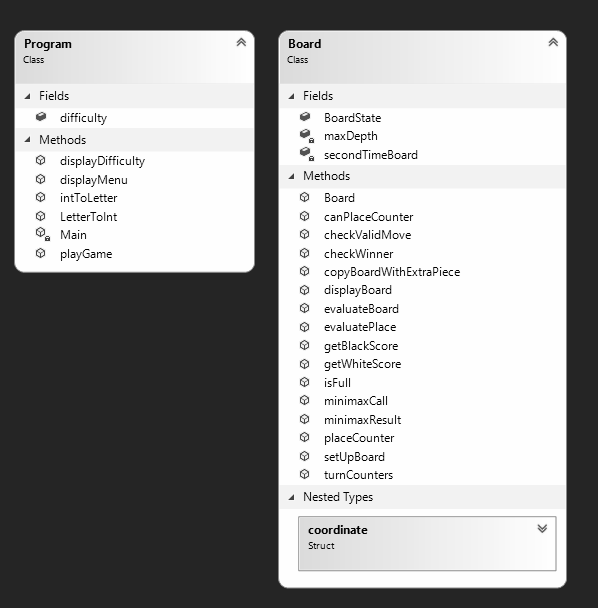
## 1.7 Modelling

**Flowchart of game play from user perspective:**

Here you can see a flowchart showing the gameplay how the user will use it. Every turn it checks if the board is full, and if so, it checks if the player can go anywhere, and skips their turn or lets them enter their counter placement depending on this. The AI then has the same check and it goes back to the start, checking if the board is full.



Class diagram showing which methods belong to which class:



I created the Board class to manage all the methods related to accessing / editing the board. It helps encapsulate these methods so that my Main class is clearer and more simply laid out.

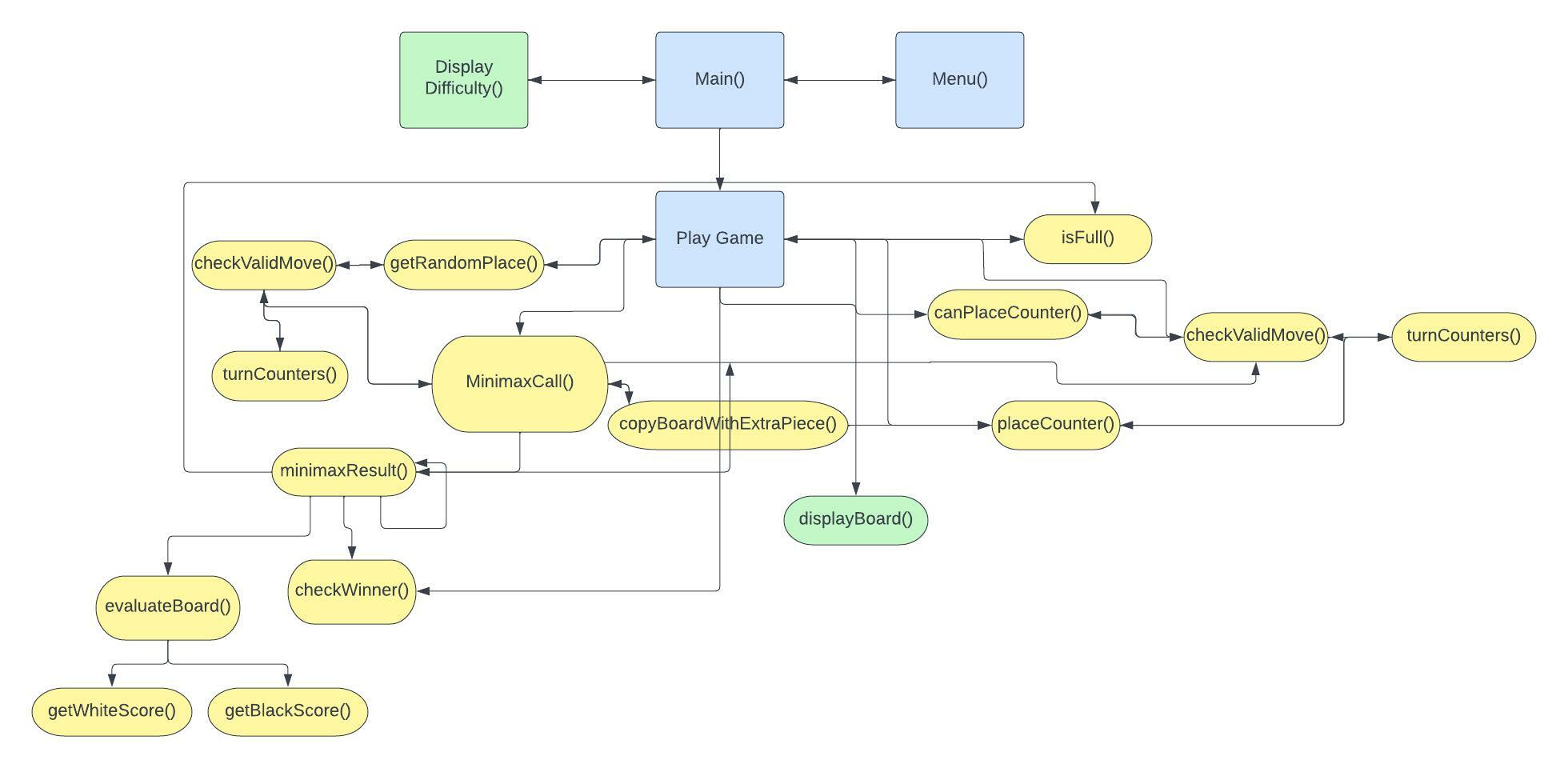
# 

# 2 Design

## 2.1 Call-Hierarchy Flowchart:

This is a flowchart detailing which methods are called from where. Blue boxes represent main subroutines, green boxes represent subroutines, and yellow boxes show functions.

The layout shown here allowed me to split apart functions as much as possible, to make it easier for me to set up, organise and debug.



## 2.2 MinimaxResult / MinimaxCall Explanation:

The Minimax Algorithm is called every time the AI needs to choose a place to play. MinimaxCall is called and given the board state, and from there calls MinimaxResult. It returns the move which minimises the strength of the player’s subsequent goes, using the heuristic function (evaluateBoard) to determine how good a board state is. It’s described in detail below:

1. MinimaxCall is called
2. Loop through every valid move for the AI:
   1. Call minimaxResult and pass through the new board state where the move has been done.
   2. The result of this is checked against the previous lowest score returned by minimaxResult (to find the move which picks the move for the AI which subsequently minimises the possible score that the player could get.)
   3. If the lowest score so far, set equal to lowestScore variable and set the current best spot variable (holds a coordinate) equal to the spot which returned that score.
3. Return the best spot variable to be played in
4. Inside MinimaxResult:
   1. Best Score variable set to + or - 100,000 if player’s turn and AI’s turn respectively.
   2. If we’re not at the max depth then check if the board is full
      1. If full then check who won and set best score to +or- 1000 points depending on who won, as a winning state is what the AI is aiming for. If a draw, set to -100 points as we don’t want the AI to play for a draw.
      2. Else: for every possible move from that board state, call MinimaxResult again with a new board having placed the extra piece, and adding one to the current depth and flipping the PlayerTurn boolean. Assign the score returned by this call to the bestScore variable if it’s bigger/smaller (if AI’s turn or Player’s turn respectively) than it was previously.
      3. Then return bestScore after doing this for all valid moves.
   3. If it is at max depth then return the value from evaluateBoard(Board board, bool PlayerTurn), which is calculated by finding the difference between black and white’s scores, +or- depending on whose turn it is, and adding appropriate values for number of edges and corners held by each.

### 2.2.1 Improvements

The minimax algorithm, although working, minimises the ‘score’ that the opponent could achieve on subsequent goes, without taking into account how the AI will play on those goes, as it simulates every possible move that the AI could play from a board state. This means it can reject making a move because of a bad ‘score’ that the AI could end up with, should it play badly in the following moves. This is an unnecessary consideration, as the AI will only choose one move for each board state, meaning that taking all the possible moves it could make into consideration is useless, and skews the results so that it doesn’t effectively minimise the opponent’s score.

Another improvement I would make were I to do this project again would be to the heuristic function that the minimax algorithm uses. I have varied the extra points assigned to edge and corner spaces to consider how they affect the play of the AI to settle on their current values, but I think I could improve how the AI values a board state by randomly sampling games from that point to completion and setting the score to the percentage of wins to losses that occurred subsequent to it. This would avoid any arbitrary assignments of value to corners and edges where clearly the game isn’t so simple.

## 2.3 Other Algorithms

### 2.3.1 Amateur AI Algorithm

This is a simple algorithm which governs the choice of the amateur AI on each move. It selects the move which turns the most opposing counters as possible, using the steps detailed below:

* Variable bestPlace holds coordinate of the chosen move
* topScore set to 0
* Loops through every space on the board:
  + If it’s a **valid move** (if it’s an unoccupied square and it turns counters) THEN:
  + If the number of counters it turns is greater than topScore variable:
  + bestPlace is set to that coordinate
  + topScore is set to the number of counters that move would turn
* bestPlace is then returned.

### 2.3.2 Valid Move Function

The valid move function checks is given a coordinate and the board state and returns whether it is a valid move. How it works (including the turnCounters function) is shown below:

* If the row or column is out of bounds of the board (e.g < 0), return false
* If the spot chosen is already occupied by a counter, return false
* If the function **turnCounters** (called on that move) does not return 0, return true

**turnCounters** takes a move, a player (or AI) and a board state.

* It checks in each direction whether there is a counter of the opposite colour next to it. If so, it continues while the next square along is still of the opposing colour, until it reaches either the given player’s colour (in which case it is a legal move and the flanked counters are added to a list to be returned by the function) or it reaches the edge of the board, which means it does not turn counters in that direction.
* If the move turns no counters, the list returned by the function is empty, signalling that it is not valid. Otherwise a list is returned containing all the counters that would be flipped by the given move.

## 2.4 Depth Selection

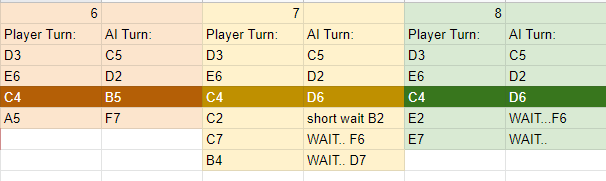
There was a tradeoff to be made between the skill of the professional AI using a minimax algorithm and the time it takes to choose a move. I had previously asked Neev what the limit would be with regard to how long he’d wait for a move (knowing that the longer this takes the better the chosen move should be ) and he said that any more than 5 seconds would interfere with his concentration and the experience of playing against the AI.

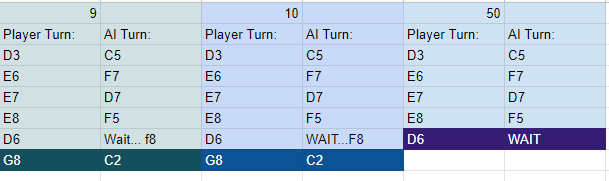
While investigating how differently the AI plays based on the depth of the minimax function, I discovered that if the depth was too high, it would sometimes “think” for longer than 5 seconds, so to ensure that the AI doesn’t use more than this time, I tried a game with the depth set to different depths to determine how long each might take, and played through until it ran into a move which took the algorithm more than 5 seconds to decide upon.

Below you can see how the AI’s play similarly in the opening few moves and how the last three (depth 9, 10 and 50 ) play exactly the same over the moves tested while the others differ before this, suggesting that once the depth gets past a certain level, the AI still makes the same moves. However, testing this further wasn’t possible as the program would pause for so long between goes that it became infeasible to continue following them through.

What this testing did provide was a better idea of how high the depth could be set while keeping the “thinking time” below 5 seconds. Where “WAIT” is written, it refers to a pause of 5 seconds or more, so clearly the depth cannot be set above 6, or very quickly it runs into too many move choices to evaluate. After trying the AI for more prolonged periods at depth 6 and subsequently 5, I discovered that as the game progressed, there would often be times where the wait would exceed 5 seconds. Therefore I have kept the depth at 4, with many games played to test that this holds for my aim.







These tables show the differences between the play of each depth level, and the highlighted rows show where they differ from the others in the groups I’ve put them in.

## 2.5 Data structures

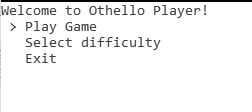
The board is stored in a 2-dimensional array of characters (chars). I chose this as it’s the best data structure to hold a board state, as an Othello board is a 2-dimensional grid, so a data structure that matches that perfectly is an obvious choice.

## 2.6 Technologies

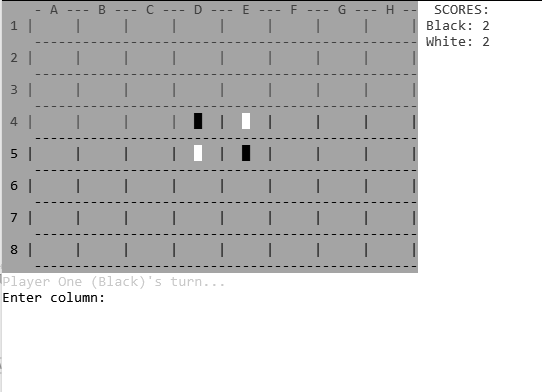
C# was the best option for me, as I have had experience with java (a similar syntax) and c# before, and it does what I need it to as a high-level language, and I have a good knowledge of using Visual Studio with it and linking that to Github so I can access code from home and at college.

## 2.7 Human-computer interaction

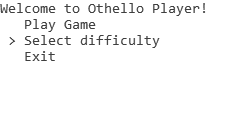
I kept the game’s interface as basic and simple as possible, as I wanted to be able to focus on improving the game and I wanted users to be able to get straight into a game without scrolling through too many menu screens. As a result, there is one menu which opens at the start of the program, showing this:

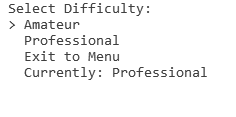


Using the arrow keys and enter button, you can select either Play Game, which will take you straight to the board and ask you to input your first placement:

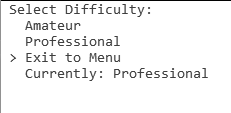


Or if you’d like to change the difficulty (by default it’s set to professional), select this option and you’ll be provided with the two difficulty options of amateur and professional, and a display of which one it is currently.





Once chosen, you would select Exit to Menu, where you can select Play Game to play with your chosen difficulty.



# 3 Testing

**Testing Video: https://youtu.be/rXU2yhmfIWY**

| Test No. | Description/Purpose | Objective Related | Test Data | Expected Outcome | Pass/Fail | Evidence |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | User should be able to input row and column | 2b: Allow the user to enter their move by inputting a letter and a number  corresponding to a row and column on the board. | d', '3' (when prompted for column letter then row no.) | Move appears on screen where intended | Pass | 00:11 |
| 2 | Should reject input if column is not within correct bounds (1-8) | 2c: Inform the user if their input wasn’t in the correct format or out of  bounds and let them try again. | i', '4'' (when prompted for column letter then row no.) | Move not allowed and asks for inputs again | Pass | 00:23 |
| 3 | Should reject input if row is not within correct bounds (1-8) | 2c: Inform the user if their input wasn’t in the correct format or out of  bounds and let them try again. | a', '11' (when prompted for column letter then row no.) | Move not allowed and asks for inputs again | Pass | 00:32 |
| 4 | Should reject input if column input is not a char | 2d: Inform the user if their input was out of bounds (not A-H) and let  them try again. | 'i', '4' (when prompted for column letter then row no.) | Move not allowed and asks for inputs again | Pass | 00:18 |
| 5 | Should reject input if row input is not an integer | 2d: Inform the user if their input was out of bounds (not 1-8) and let  them try again. | 'a', 'b' (when prompted for column letter then row no.) | Move not allowed and asks for inputs again | Pass | 00:32 |
| 6 | Player should not be able to place somewhere that does not turn any counters | 2e: Inform the user if the move they chose is illegal (if there’s already a  counter there or doesn’t flip any counters) and let them take it again if  so. | 'g', '2' (when prompted for column letter then row no.)  When it wouldn't turn any counters | Move not allowed and asks for inputs again | Pass | 00:39 |
| 7 | Player should not be able to place somewhere that is already taken | 2e: Inform the user if the move they chose is illegal (if there’s already a  counter there or doesn’t flip any counters) and let them take it again if  so. | 'd, '3' (when prompted for column letter then row no.)  when there is a counter in (d, 3) | Move not allowed and asks for inputs again | Pass | 00:41 |
| 8 | Menu Selections Work | 4: 2 difficulties of AI opponents to play against: amateur and  professional. Chosen by settings menu. | Menu used | Changes difficulty correctly and begins game | Pass | 00:00 |
| 9 | Amateur AI plays in correct place | 4a: The Amateur AI will choose the move which turns the most counters  on its go. | Piece placed by player | AI plays where it flips the most counters possible | Pass | 01:04 |
| 10 | Program checks every turn whether the game has ended | 5: Check if the game has ended after every move. | Board is full | Game ends with win message displayed | Pass | 01:14 |
| 11 | Player's (and AI's) turn is skipped if they cannot go | 6: Check if the player has a possible place to put a counter on their turn  or whether they must skip their go, and send a message if the latter. | Board such that player cannot make a move | Turn skipped with message showing shortly | Pass | 01:41 |
| 12 | Professional AI does not take more than 5 seconds to pick a move | 4b: The Professional AI will have a minimax algorithm with the ‘sight’  set to the highest it can go without taking more than 5 seconds to  evaluate on any given go. | Whenever it it the AI's go | AI places a counter in less than 5 seconds | Pass | 00:14 |
| 13 | All counters flipped/placed flash yellow shortly | 10a: Counters placed and flipped by the AI will turn yellow for a short  time after the turn is taken. | Whenever it it the AI's go | Counters flipped/placed are yellow for a short time  before changing back to their original colour | Pass | 00:14 |
| 14 | AI's previous turn is displayed under board correctly | 10b: AI’s last-turn coordinate will be shown under the board throughout  the player’ next turn | After AI's go | A message under the board contains the  AI's last placement coordinate | Pass | 00:17 |
| 15 | Scores correctly displayed at all times during the game | 9: Display each player’s score on the console throughout the game  and at the end with the winner declaration. | Game in progress | Scores displayed at side of board | Pass | 00:17 |
| 16 | Counters are flipped if flanked vertically | 8: Be able to place a counter and turn the counters which are flanked  by that colour. | Place flanking white pieces vertically | Correct pieces are flipped after placement | Pass | 00:44 |
| 17 | Counters are flipped if flanked horizontally | 8: Be able to place a counter and turn the counters which are flanked  by that colour. | Place flanking white pieces horizontally | Correct pieces are flipped after placement | Pass | 00:51 |
| 18 | Correct counters are flipped if flanked diagonally | 8: Be able to place a counter and turn the counters which are flanked  by that colour. | Place flanking white pieces diagonally | Correct pieces are flipped after placement | Pass | 00:58 |
| 19 | Program displays correct win message if AI wins | 9: Display each player’s score on the console throughout the game  and at the end with the winner declaration (or draw). | End of game and white has more counters | Correct end win message displayed | Pass | 01:16 |
| 20 | Program displays correct win message if Player wins | 9: Display each player’s score on the console throughout the game  and at the end with the winner declaration (or draw). | End of game and black has more counters | Correct end win message displayed | Pass | 01:25 |
| 21 | Program displays correct message if game is a draw | 9: Display each player’s score on the console throughout the game  and at the end with the winner declaration (or draw). | End of game and black and white have equal counter  counts | Correct end win message displayed | Pass | 01:37 |
| 22 | The user can exit the program from the menu | 11: User can exit the game from the menu | Exit option selected | Console closes | Pass | 01:45 |
| 23 | The program should display the board throughout the game | 2a: Board which displays where each counter is. | Game in progress | Board is shown | Pass | 00:13 |
| 24 | Each piece should be should be shown on the board after a placement | 2f: Display the new board state correctly after each go. | Turn has just taken place | Board is shown correctly | Pass | 00:16 |
| 25 | Full Game plays through successfully | 1.Othello game AI that developing players can train on. | Game played | Game plays successfully | Pass | 01.50 |

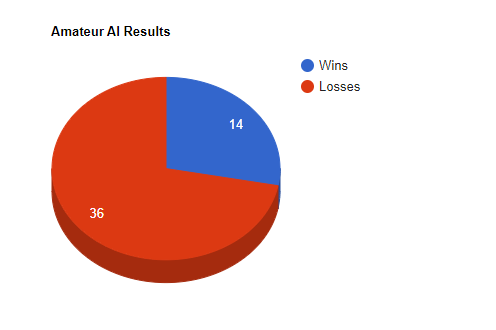
# 4 Evaluation

## 4.1 Overall Effectiveness of the System

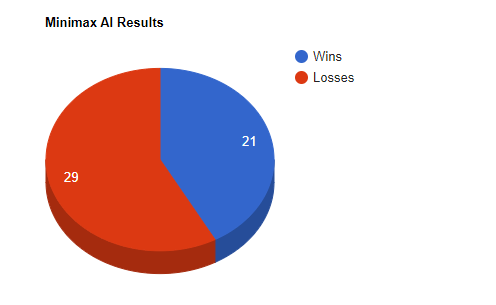
The system does what I set out to create, and does so efficiently, but I would have liked to make a higher standard of AI player using either a different algorithm or an improved version of minimax, as sometimes it makes moves that even an amateur human would never make, that disadvantages itself and sometimes imbalances the game by giving away a corner or suchlike. This sometimes makes the game too easy for the player and results in it being suboptimal practice for Neev or another user. Usually though, the AI is a good opponent and makes for good practice for an improving player and allows them to practice when there isn’t someone else around. The system is intuitive to interact with, allowing the player to focus on the game without unnecessary distractions, with only the column and row of the counter needed to be input by the player.

The amateur AI allows beginner players to work on their game against a simple but ruthless opponent: it will never miss an opportunity to turn as many counters as possible on a single go, but it’s easily outwitted. My experience of Othello playing with family and friends is that people generally look for the move which simply turns the most counters, so practicing against a very good version of this is fantastic practice for a player starting out, especially if their opponents are at the same level they are, as it teaches how to play against this strategy.

In an aim to quantify how well my AI’s perform against amateur human players, I asked 10 friends and family members to each play 5 games against each AI, so I could estimate how well they play real people. A larger sample size might have yielded more accurate results, but I believe these results reflect the abilities of the algorithms reasonably well:



The Amatuer AI won only 28% of the time, using a strategy of choosing the move which turned most counters. This (although a common strategy among beginners) really fails during the endgame section, as it misses key opportunities to capture corner spots or set up for an even bigger flip next turn, where even a basic human player can pick this up and use it to their advantage. I do think it’s still a useful addition to the program, as when starting out, users need to understand what makes a good player, and playing a very basic one is a good way of starting to understand that.



My minimax algorithm did not perform as well as I had expected. It won 42% of the time in this sample, which although not a bad outcome, I had hoped that the ‘foresight’ that the minimax algorithm provides would have had a greater effect on the AI’s play. Having played a lot of games against it myself, I noticed some interesting things about the way it plays. I would have expected it to value taking corners much more than it did (despite my explicitly assigned value given to them in the evaluateBoard function) - sometimes ignoring an opportunity to take a corner for several turns in a row. It did seem to limit my options which was good, as I rarely got the chance to take a big swathe of its pieces during the game, but as the game got near the end I found much more chance to take large numbers of pieces, and I suspect that an increase in the depth (if made feasible through alpha-beta pruning or another method) might prevent this drop in it’s ability to minimise my strongest moves nearer the end as it might ‘plan’ better for this.

## 4.2 Evaluation of Objectives

Objectives have been copied below for ease of reference:

1. **Develop an AI that can choose moves and play against a user**

This has definitely been achieved, with a working game that’s intuitive to use and fast to play against.

* 1. **2 difficulties of AI opponents to play against: amateur and professional.**
  2. **The Amateur AI will choose the move which turns the most counters on its go.**
  3. **The Professional AI will be determined by a minimax algorithm with the depth set to the highest it can go without taking more than 5 seconds to evaluate on any given go.**

Both opponents can be selected and played against, with distinct differences in their play styles as there should be, and no more than a 5 second delay to make a move.

* 1. **Evaluation function which values each possible move of the AI each turn, and is utilised in the minimax algorithm.**

This has been created and works, but more work could have been done to work out how to better evaluate the board, as mentioned in the Design section.

1. **User interface in the console**
   1. **Board which displays where each counter is.**
   2. **Allow the user to enter their move by inputting a letter and a number corresponding to a row and column on the board.**
   3. **Inform the user if their input wasn’t in the correct format and let them try again.**
   4. **Inform the user if their input was out of bounds (not 1-8) and let them try again.**
   5. **Inform the user if the move they chose is illegal (if there’s already a counter there or doesn’t flip any counters) and let them take it again if so.**
   6. **Display the new board state after each go.**
   7. **Counters represented by the character ‘█’ in black and white colours.**
   8. **Current player’s turn is displayed.**

The user interface has been effectively implemented, however if I were to do it again I would improve it by adding more specific error messages with input validation, where it would tell the user exactly what was wrong with the input rather than just that it was wrong, before asking for another input. This would help with any issues a user might have with inputting their go.

1. **Menu option for selection of difficulty**

Clearly shown with display of current difficulty

1. **Check if the game has ended after every move.**
2. **Check if the player has a possible place to put a counter on their turn or whether they must skip their go, and send a message if the latter.**

4&5 both successfully added as shown in testing.

1. **Be able to place a counter and turn all the counters which are flanked by that colour.**

Completed successfully and shown in testing.

1. **Display each player’s score on the console throughout the game**

Tested and completed

1. **Display the winner at the end of the game along with scores**

Done as shown in testing

1. **It is clear where the AI has placed on it’s last go.**
   1. **Counters placed and flipped by the AI will turn yellow for a short time after the turn is taken.**
   2. **AI’s last-turn coordinate will be shown under the board throughout the player’s next turn**

This was a good objective to include as it was often hard to be sure where the AI had placed, so this ensured more clarity for the user when implemented

1. **User can exit the game from the menu**

Done, but it might have been more beneficial to allow users to leave during the game, as there’s not much reason to leave from the menu when the program has only just started. Either way they can leave by closing the window so this isn’t essential.

## 4.3 End User Feedback

Here’s the transcript for my final feedback interview with Neev:

1. **What did you like about the final game?**

I really enjoyed the simple board layout and input system, especially the letters and numbers representing rows and columns, as it allows me to focus on my play without being distracted by how to input my turn or work out where the opponent went, as it’s very clear with the yellow counters showing the AI’s last go. I also enjoyed that the AI wasn’t perfect and sometimes made mistakes like a human would, as this allowed me to practice taking advantage of their mistakes as best I could.

1. **What further improvements could I have made?**

Well although it doesn’t affect my actual play, I would’ve liked to have some animation of placing pieces to create a better feel aesthetically. It feels very inhuman and some animation could also make it visually clearer which piece has been placed and which pieces have been flipped by that piece.

The main improvement I could think of would be a more consistent AI which plays more like a human with occasional mistakes ( which as I said are beneficial in a way ) but ones which weren’t so obviously harmful to it’s own game like ignoring an opportunity to take a corner, to better replicate an average player.

Something I found while trying to practice and work out how to play better is that during the beginning and mid-game, it’s difficult to know how well you’re playing, as there’s no clear metric - sometimes it’s better to have more counters of your colour, but sometimes it’s beneficial to have less so you have more moves available. To rectify this, I’d suggest adding a ‘confidence rating’ for the AI, which is changed every move and reflects how well the AI ‘thinks’ it’s doing and how likely it is to win.

### 4.3.1 Conclusions on Feedback

These are fair comments, and I’m encouraged that the AI’s imperfections during play can be beneficial to a user. As for an average player, taking advantage of someone’s mistakes is an important part of the game. However, Neev does agree with me that the AI could be improved to be a more consistent opponent, and I’d take that into consideration were I to do the project again. I do understand his desire for some animation on the board, but under these circumstances and having made the game in console, I think it would’ve been outside the scope of this project and my time was better spent in other ways working on the project as it is. His comment about a confidence rating is a clever suggestion - the aim of the program is to help players improve their Othello strategy and skills, so telling the player how well they’re doing at any given point would allow them to modify their gameplay appropriately to maximise their gains in game.

## 4.4 System Improvements

As mentioned throughout my evaluation, the main improvement I would make to the system would be to try making the AI in different ways. I considered Alpha-Beta pruning earlier in the development of the game, and I would revisit that idea along with Monte-Carlo Tree Searching and perhaps other methods like random sampling (taking each move and playing randomised games from that state, taking the one with the best win percentage as best) which I could rank alongside each other to create more levels of AI for players to practise against as they progress, while speeding up the time it takes to evaluate a move by making the AI less resource-intensive.

These other AI algorithms could help tie in with a feature that Neev suggested. Having a ‘Confidence Rating’ given by the AI throughout the game would really help users understand which are good strategies and which are not. By using random sampling for example, I would attempt to create this rating by sampling many randomised games from the current state of the board and taking the win/loss percentage as the AI’s confidence level at any time. Of course this would need lots of testing and a more specific design so it was too much to add to this project but I would look into it if I were to do it again.

Another improvement could be to create the program using a game engine that would allow me to introduce animations to make even clearer transitions between moves showing how the board changes in real time as it would in real life. It could also better show how the opponent’s move flips counters; I could make it so that the placed counter is put on the board first, and subsequently the counters it turns are clearly flipped for the user to see. This might also create a better user experience through better graphics and more immersive feel to the game, to better simulate real life.

# 5 Technical Solution

## Contents page for code:

Main methods underlined, classes in **bold**

| **Code Section** | **Page Number** |
| --- | --- |
| **Main** | 33 |
| DisplayDifficulty | 34 |
| playGame | 35 |
| displayMenu | 39 |
| **Board Class** | 40 |
| getBestPlace | 41 |
| minimaxCall | 41 |
| copyBoardWithExtraPiece | 42 |
| minimaxResult | 42 |
| EvaluateBoard | 44 |
| canPlaceCounter | 47 |
| checkValidMove | 46 |
| isFull | 46 |
| CheckWinner | 47 |
| TurnCounters | 48 |
| getBlackScore | 57 |
| setupBoard | 57 |
| getWhiteScore | 57 |
| displayBoard | 58 |

| using System; using System.Collections.Generic; using System.Linq; using System.Text; using System.Threading.Tasks;  namespace OthelloPlayer {  class Program  {   public static string difficulty = "Professional";    static void Main(string[] args)  {  //set up board with starting discs   Board gameBoard = new Board();   //returns a menu option which sends you down one of three switch statements.  int menuOption = displayMenu();   while (menuOption != 1)  {  switch (menuOption)  {  case 2:  Console.Clear();  displayDifficulty();  Console.Clear();  menuOption = displayMenu();  break;  case 3:  Console.Clear();  Console.WriteLine("Thanks for Playing!");  break;  }   }  Console.Clear();  gameBoard.displayBoard();  playGame(ref gameBoard);   }   public static void displayDifficulty()  {  //sets up menu for selecting difficulty  Console.WriteLine(" Select Difficulty: ");  Console.WriteLine(" > Amateur");  Console.WriteLine(" Professional");  Console.WriteLine(" Exit to Menu");  Console.WriteLine(" Currently: " + difficulty);    ConsoleKeyInfo choice = Console.ReadKey(true);  int option = 1;   while (!(choice.Key == ConsoleKey.Enter && option == 3))  {   if (choice.Key == ConsoleKey.Enter)  {  switch (option)  {  case 1:  difficulty = "Amateur";  Console.Clear();  displayDifficulty();  return;  break;  case 2:  difficulty = "Professional";  Console.Clear();  displayDifficulty();  return;  break;    }  }   //allows for visual movement between options  if (choice.Key == ConsoleKey.UpArrow && option > 1)  {  Console.CursorLeft = 0;  Console.CursorTop = option;  Console.Write(" ");  Console.CursorTop = option - 1;  Console.CursorLeft = 0;  option--;  Console.Write(" > ");  }  if (choice.Key == ConsoleKey.DownArrow && option < 3)  {  Console.CursorLeft = 0;  Console.CursorTop = option;  Console.Write(" ");  Console.CursorTop = option + 1;  Console.CursorLeft = 0;  option++;  Console.Write(" > ");  }   choice = Console.ReadKey(true);  }      }   //used to get column letters into numbers  public static int LetterToInt(char letter)  {  string strLetter = letter.ToString().ToUpper();  char UpperCaseChar = strLetter.ToCharArray()[0];   return UpperCaseChar - (int)'A';  }   //changes a number to corresponding letter in ASCII values  public static char intToLetter(int num)  {  num += (int)'A';  char UpperCaseChar = (char)num;   return UpperCaseChar;  }   //runs the game  public static void playGame(ref Board board)  {  bool PlayerTurn = true;  int row = -1;  int column = -1;  while (!Board.isFull(ref board))  {  //Carries out the players turn  if (PlayerTurn)  {  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Player One (Black)'s turn...");  Console.ForegroundColor = ConsoleColor.White;   //checks if player can play at all  if (Board.canPlaceCounter(ref board, PlayerTurn))  {  int tries = 0;  //checks if move entered/set is valid and continues to run while not  while (!Board.checkValidMove(ref board, true, row, column))  {  tries++;  //stops error message running the first time it asks  if (tries > 1)  {  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Please enter a correct column and row...");  Console.ForegroundColor = ConsoleColor.White;  }   //move inputs  Console.Write("Enter column: ");  try  {  column = LetterToInt(char.Parse(Console.ReadLine().Substring(0, 1)));  }  catch (Exception e)  {  column = -1;  }  Console.Write("Enter row: ");  try  {  row = int.Parse(Console.ReadLine()) - 1;  }  catch (Exception e)  {  row = -1;  }    }   //Places counter in selected spot and displays board  Board.turnCounters(ref board, row, column, true, PlayerTurn);  Console.Clear();  board.displayBoard();  }  else  {  //if there is no valid move:  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Player 1 can't go anywhere! Go skipped...");  Console.ForegroundColor = ConsoleColor.White;  System.Threading.Thread.Sleep(2000);   }  PlayerTurn = false;  }   else  {   //Carries out the AI's turn  if (Board.canPlaceCounter(ref board, PlayerTurn))  {   Board.coordinate spotPick = new Board.coordinate(0, 0);  Console.ForegroundColor = ConsoleColor.Green;  Console.WriteLine("AI (White)'s turn, thinking...");  System.Threading.Thread.Sleep(1500);  Console.ForegroundColor = ConsoleColor.White;     if (difficulty == "Amateur")  {  //gets move which turns most counters  spotPick = Board.getBestPlace(ref board, false);  }   if (difficulty == "Professional")  {  //runs minimax on current board state  spotPick = Board.minimaxCall(board);   }   //places counter in selected place and displays board  Board.turnCounters(ref board, spotPick.row, spotPick.column, true, PlayerTurn);  Console.Clear();  board.displayBoard();  Console.WriteLine("AI placed at (" + intToLetter(spotPick.column) + ", " + (spotPick.row + 1) + "), the counters flipped turn yellow shortly.");    }   else  {  //if there is no valid move:  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Player 2 can't go anywhere! Go skipped...");  Console.ForegroundColor = ConsoleColor.White;   }  PlayerTurn = true;  }  }  //When game ends  Console.WriteLine("GAME OVER, winner was...");  System.Threading.Thread.Sleep(3000);  Console.ForegroundColor = ConsoleColor.Yellow;  //Winner declared  Console.WriteLine(Board.checkWinner(ref board) + "!");  Console.WriteLine("Black Score: " + board.getBlackScore() + ", White Score: " + board.getWhiteScore());   Console.ReadKey();  }   public static int displayMenu()  {  //Starting menu displayed  Console.WriteLine("Welcome to Othello Player!");  Console.WriteLine(" > Play Game");  Console.WriteLine(" Select difficulty");  Console.WriteLine(" Exit");   ConsoleKeyInfo choice = Console.ReadKey(true);  int option = 1;  while (choice.Key != ConsoleKey.Enter)  {   if (choice.Key == ConsoleKey.UpArrow && option > 1)  {  Console.CursorLeft = 0;  Console.CursorTop = option;  Console.Write(" ");  Console.CursorTop = option - 1;  Console.CursorLeft = 0;  option--;  Console.Write(" >");  }  if (choice.Key == ConsoleKey.DownArrow && option < 3)  {  Console.CursorLeft = 0;  Console.CursorTop = option;  Console.Write(" ");  Console.CursorTop = option + 1;  Console.CursorLeft = 0;  option++;  Console.Write(" >");  }   choice = Console.ReadKey(true);   }  return option;    }  } } |
| --- |

**Board class:**

| using System; using System.Collections.Generic; using System.Linq; using System.Text; using System.Threading.Tasks;  namespace OthelloPlayer {  class Board  {  public struct coordinate  {  //Create structure to hold coordinates of a counter  public int column;  public int row;   //Create constructor to set values  public coordinate(int setRow, int setColumn)  {  column = setColumn;  row = setRow;  }  }   //Create 2d array to hold board state  public string[,] BoardState = new string[8, 8];   //Allows for making a board without setting it up  public Board(bool setup = true)  {  if (setup)  {  setUpBoard();  }  }   //Method to find best spot to play based on amateur AI ( spot which turns most counters )  public static coordinate getBestPlace(ref Board board, bool playerTurn)  {  Board.coordinate bestPlace = new coordinate(0, 0);  int topScore = 0;  //Loops through every square  for (int i = 0; i < 7; i++)  {  for (int a = 0; a < 7; a++)  {  //if valid move  if (Board.checkValidMove(ref board, false, i, a))  {  //if it turns more counters than the previous highest  if (turnCounters(ref board, i, a, false, false).Count() > topScore)  {  //set as new best place and set topScore to the new highest number of counters turned  bestPlace = new coordinate(i, a);  topScore = turnCounters(ref board, i, a, false, false).Count();  }  }  }  }  return bestPlace;  }   //Depth for minimax algorithm to run  private static int maxDepth = 4;   //Minimax call  public static coordinate minimaxCall(Board board)  {  //base scores set  int tempScore = 0;  int lowestScore = 100000;  coordinate bestSpot = new coordinate(0, 0);  //for every move on the board  for (int i = 0; i < 8; i++)  {  for (int a = 0; a < 8; a++)  {  //if it's valid  if (checkValidMove(ref board, false, i, a))  {  Board newBoard = Board.copyBoardWithExtraPiece(board, i, a, false);  //call minimax on this move, player's turn  tempScore = Board.minimaxResult(newBoard, true, 0);  //minimise the player's score  if (tempScore <= lowestScore)  {  lowestScore = tempScore;  //sets bestSpot to best coordinate for minimising player's score  bestSpot = new coordinate(i, a);  }  }  }  }  return bestSpot;  }   public static Board copyBoardWithExtraPiece(Board board, int row, int column, bool PlayerTurn)  {  //returns a copy of the board once another piece has been places at (row, column)  Board boardCopy = new Board(setup: false);  for (int i = 0; i < 8; i++)  {  for (int a = 0; a < 8; a++)  {  boardCopy.BoardState[i, a] = board.BoardState[i, a];  }  }  Board.turnCounters(ref boardCopy, row, column, true, PlayerTurn);  return boardCopy;  }   //Called from minimax call on the move AFTER each possible move from the board state given and returns a value representing how good that spot is (for the player)  public static int minimaxResult(Board board, bool playerTurn, int currentDepth)  {   int bestScore = 0;   if (playerTurn) bestScore = 100000;  else bestScore = -100000;   if (currentDepth != maxDepth)  {  if (Board.isFull(ref board))  {  //if someone has wins give appropriate points:  if(playerTurn)  {  if (Board.checkWinner(ref board) == "Black")  {  return 1000;  }  if (Board.checkWinner(ref board) == "Draw")  {  return -10;  }  else return -1000;  }  else  {  if (Board.checkWinner(ref board) == "Black")  {  return -1000;  }  if (Board.checkWinner(ref board) == "Draw")  {  return -10;  }  else return 1000;  }  }   int nextGoScore = 0;  Board finalBoard = new Board();  //for each square on the board  for (int i = 0; i < 8; i++)  {  for (int a = 0; a < 8; a++)  {  //if it's a valid move  if (checkValidMove(ref board, false, i, a))  {  Board newBoard = new Board();  newBoard = copyBoardWithExtraPiece(board, i, a, playerTurn); //copies board and adds new counter  //calls minimax again on this move, with depth increased and on the other player's turn  nextGoScore = minimaxResult(newBoard, !playerTurn, currentDepth+1);  if (playerTurn)  {  //If the AI's score next go is worse (for the AI) than the previous score, replace it.  if (nextGoScore < bestScore)  {  bestScore = nextGoScore;  }  }  if (!playerTurn)  {  //If the player's score next go is better than the best previous, replace it.  if (nextGoScore > bestScore)  {  bestScore = nextGoScore;  }  }  }  }  }  return bestScore;  }  else  {  //if it's at it's max depth, evaluate the board state at this point for the player who's go it is.  int evaluate = Board.evaluateBoard(board, playerTurn);  return evaluate;  }  }  public static int evaluateBoard(Board board, bool playerTurn)  {  int score = 0;  if (playerTurn)  {  //base score given from difference between the number of counters on the board (player - opponent's)  score = board.getBlackScore() - board.getWhiteScore();  for(int i = 2; i < 6; i++)  {  //+2 for edges  if(board.BoardState[0, i] == "B") score+= 2;  if (board.BoardState[i, 0] == "B") score+= 2;  if (board.BoardState[7, i] == "B") score+= 2;  if (board.BoardState[i, 7] == "B") score+= 2;  //take off points for opponent holding edges  if (board.BoardState[0, i] == "W") score -= 2;  if (board.BoardState[i, 0] == "W") score -= 2;  if (board.BoardState[7, i] == "W") score -= 2;  if (board.BoardState[i, 7] == "W") score -= 2;  }   //+20 for corners  if (board.BoardState[0, 0] == "B") score += 20;  if (board.BoardState[7, 0] == "B") score += 20;  if (board.BoardState[7, 7] == "B") score += 20;  if (board.BoardState[0, 7] == "B") score += 20;  //take off points for opponent holding corners  if (board.BoardState[0, 0] == "W") score -= 20;  if (board.BoardState[7, 0] == "W") score -= 20;  if (board.BoardState[7, 7] == "W") score -= 20;  if (board.BoardState[0, 7] == "W") score -= 20;   }   else  {  //repeated in case of other player's go  score = board.getWhiteScore() - board.getBlackScore();  for (int i = 2; i < 6; i++)  {  //+2 for edges  if (board.BoardState[0, i] == "W") score+=2;  if (board.BoardState[i, 0] == "W") score+=2;  if (board.BoardState[7, i] == "W") score+=2;  if (board.BoardState[i, 7] == "W") score+=2;   if (board.BoardState[0, i] == "B") score -= 2;  if (board.BoardState[i, 0] == "B") score -= 2;  if (board.BoardState[7, i] == "B") score -= 2;  if (board.BoardState[i, 7] == "B") score -= 2;   }   //+3 for corners  //must check for R too while the board displays last move  if (board.BoardState[0, 0] == "W") score += 20;  if (board.BoardState[7, 0] == "W") score += 20;  if (board.BoardState[7, 7] == "W") score += 20;  if (board.BoardState[0, 7] == "W") score += 20;   if (board.BoardState[0, 0] == "B") score -= 20;  if (board.BoardState[7, 0] == "B") score -= 20;  if (board.BoardState[7, 7] == "B") score -= 20;  if (board.BoardState[0, 7] == "B") score -= 20;   }  //returns the 'goodness' of the board state  return score;   }   public static bool checkValidMove(ref Board board, bool PlayerTurn, int row, int column)  {  //checks if out of bounds of the board  if (row > 7 || row < 0 || column > 7 || column < 0)  {  return false;  }  //checks if counter already there  if (board.BoardState[row, column] != " ")  {  return false;  }  //checks if it turns any counters  if (turnCounters(ref board, row, column, false, PlayerTurn).Count > 0)  {  return true;  }  return false;   }   public static bool isFull(ref Board board)  {  //loop through values in board:  for (int i = 0; i < 8; i++)  {  for (int a = 0; a < 8; a++)  {  //if there is a valid move for neither player, return true as the board is full/game is ended  if (Board.checkValidMove(ref board, true, i, a))  {  return false;  }  if (Board.checkValidMove(ref board, false, i, a))  {  return false;  }  }  }  return true;  }   public static bool canPlaceCounter(ref Board board, bool PlayerTurn)  {  //checks if a player can make a move  for (int i = 0; i < 8; i++)  {  for (int a = 0; a < 8; a++)  {  if (Board.checkValidMove(ref board, PlayerTurn, i, a))  {  return true;  }  }  }  return false;  }   //check who won the game  public static string checkWinner(ref Board board)  {  int onePoints = 0;  int twoPoints = 0;  for (int i = 0; i < 8; i++)  {  for (int a = 0; a < 8; a++)  {  if (board.BoardState[i, a] == "B")  {  onePoints++;  }  if (board.BoardState[i, a] == "W")  {  twoPoints++;  }   }  }  if (onePoints > twoPoints)  {  return "Black";  }  if (onePoints < twoPoints)  {  return "White";  }  else return "Draw";  }   //checks which counters (if any) a move turns and changes the board accordingly if 'flip' is true  public static List<coordinate> turnCounters(ref Board board, int row, int column, bool flip, bool PlayerTurn)  {  //create lists to turn discs permanantly and temporarily to hold while it checks for a valid flank  List<coordinate> discsToTurn = new List<coordinate>();  List<coordinate> tempDiscsToTurn = new List<coordinate>();   //checks that there is not a counter already there  if (!(board.BoardState[row, column] == " ")) return discsToTurn;   //variables holding 'B' or 'W' representing the counters how they are held in the board state array  string myColour;  string notMyColour;  if (PlayerTurn)  {  myColour = "B";  notMyColour = "W";  }  else  {  myColour = "W";  notMyColour = "B";  }   //check horizontal right:  if ((column + 2) < 7)  {  if (board.BoardState[row, column + 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row, (column + 1)));  for (int i = column + 2; i < 8; i++)  {  if (board.BoardState[row, i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row, i));  //add them to temporary list  }  if (board.BoardState[row, i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (i < 8)  {  if (board.BoardState[row, i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  tempDiscsToTurn = new List<coordinate>();  }  }  //checked horizontal right   //check horizontal left  if ((column - 2) > -1)  {  if (board.BoardState[row, column - 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row, (column - 1)));  for (int i = column - 2; i > -1; i--)  {  if (board.BoardState[row, i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row, i));  //add them to temporary list  }  if (board.BoardState[row, i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = -1;  }  if (i > -1)  {  if (board.BoardState[row, i] == " ")  {  i = -1;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  tempDiscsToTurn = new List<coordinate>();  }  }  //checked horizontal left.  //check up:  if ((row - 2) > -1)  {  if (board.BoardState[row - 1, column] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - 1, column));  for (int i = row - 2; i > -1; i--)  {  if (board.BoardState[i, column] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(i, column));  //add them to temporary list  }  if (board.BoardState[i, column] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = -1;  }  if (i > -1)  {  if (board.BoardState[i, column] == " ")  {  i = -1;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  tempDiscsToTurn = new List<coordinate>();  }  }  //checked up.  //check down:  if ((row + 2) < 8)  {  if (board.BoardState[row + 1, column] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + 1, column));  for (int i = row + 2; i < 8; i++)  {  if (board.BoardState[i, column] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(i, column));  //add them to temporary list  }  if (board.BoardState[i, column] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (i < 8)  {  if (board.BoardState[i, column] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  tempDiscsToTurn = new List<coordinate>();  }  }  //Checked down.  //Check SE diagonal:  if (((row + 2) < 8) && ((column + 2) < 8))  {  if (board.BoardState[row + 1, column + 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + 1, column + 1));  for (int i = 2; row + i < 8 && column + i < 8; i++)  {  if (board.BoardState[row + i, column + i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + i, column + i));  //add them to temporary list  }  if (board.BoardState[row + i, column + i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (i < 8)  {  if (board.BoardState[row + i, column + i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  tempDiscsToTurn = new List<coordinate>();  }  //Checked SE diagonal  //Check NW diagonal:  if (((row - 2) > -1) && ((column - 2) > -1))  {  if (board.BoardState[row - 1, column - 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - 1, column - 1));  for (int i = 2; row - i > -1 && column - i > -1; i++)  {  if (board.BoardState[row - i, column - i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - i, column - i));  //add them to temporary list  }  if (board.BoardState[row - i, column - i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (row - i > -1)  {  if (board.BoardState[row - i, column - i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  tempDiscsToTurn = new List<coordinate>();  }  }  //Checked NW diagonal.  //Check NE diagonal:  if (((row - 2) > -1) && ((column + 2) < 8))  {  if (board.BoardState[row - 1, column + 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - 1, column + 1));  for (int i = 2; row - i > -1 && column + i < 8; i++)  {  if (board.BoardState[row - i, column + i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - i, column + i));  //add them to temporary list  }  if (board.BoardState[row - i, column + i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (row - i > -1)  {  if (board.BoardState[row - i, column + i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  tempDiscsToTurn = new List<coordinate>();  }  }  //Checked NE diagonal  //Check SW diagonal  if (((column - 2) > -1) && ((row + 2) < 8))  {  if (board.BoardState[row + 1, column - 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + 1, column - 1));  for (int i = 2; column - i > -1 && row + i < 8; i++)  {  if (board.BoardState[row + i, column - i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + i, column - i));  //add them to temporary list  }  if (board.BoardState[row + i, column - i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (row + i < 8)  {  if (board.BoardState[row + i, column - i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  tempDiscsToTurn = new List<coordinate>();  }  }  //Checked SW diagonal  //CHECKED ALL DIRECTIONS    //flip counters if flip is true  if (flip)  {    string flipToColour;  if (PlayerTurn)  {  flipToColour = "B";  }  //flip to R yellow counters which have just been flipped  else flipToColour = "R";  board.BoardState[row, column] = flipToColour;  foreach (var item in discsToTurn)  {  //R for recently placed/flipped  board.BoardState[item.row, item.column] = flipToColour;  }  }  return discsToTurn;   }        public void setUpBoard()  {  //sets up board at start of the game  for (int i = 0; i < 8; i++)  {  for (int a = 0; a < 8; a++)  {  BoardState[i, a] = " ";  }  }  BoardState[3, 3] = "W";  BoardState[4, 4] = "W";  BoardState[4, 3] = "B";  BoardState[3, 4] = "B";   }   public int getBlackScore()  {  int black = 0;  foreach (var item in BoardState)  {  if (item == "B")  {  black++;  }  }  return black;   }  public int getWhiteScore()  {  int white = 0;  foreach (var item in BoardState)  {  if (item == "W" || item == "R")  {  white++;  }  }  return white;   }   //represents whether board is 'flashing' yellow or not  private bool secondTimeBoard = false;   public void displayBoard()  {  //prints board and scores in console  Console.BackgroundColor = ConsoleColor.DarkGreen;  Console.Write(" - A --- B --- C --- D --- E --- F --- G --- H --");  Console.BackgroundColor = ConsoleColor.Black;  Console.Write(" SCORES:");  Console.WriteLine();  Console.BackgroundColor = ConsoleColor.DarkGreen;   List<coordinate> recentlyFlipped = new List<coordinate>();   for (int i = 1; i < 9; i++)  {  Console.Write(" " + i + " |");  for (int a = 0; a < 8; a++)  {  if (BoardState[i - 1, a] == "B")  {  Console.ForegroundColor = ConsoleColor.Black;  Console.Write(" █");  Console.ForegroundColor = ConsoleColor.White;   }  else if (BoardState[i - 1, a] == "W")  {  Console.ForegroundColor = ConsoleColor.White;  Console.Write(" █");   }  else if (BoardState[i - 1, a] == "R")  {  Console.ForegroundColor = ConsoleColor.DarkYellow;  Console.Write(" █");  Console.ForegroundColor = ConsoleColor.White;  recentlyFlipped.Add(new coordinate(i - 1, a));   }  else  {  Console.Write(" ");  }   Console.Write(" |");    }  if (i == 1)  {  Console.BackgroundColor = ConsoleColor.Black;  Console.Write(" Black: " + getBlackScore());  Console.BackgroundColor = ConsoleColor.DarkGreen;  Console.WriteLine();  Console.Write(" ------------------------------------------------");  Console.BackgroundColor = ConsoleColor.Black;  Console.WriteLine(" White: " + getWhiteScore());  Console.BackgroundColor = ConsoleColor.DarkGreen;   }  else  {  Console.WriteLine();  Console.WriteLine(" ------------------------------------------------");  }  }   Console.BackgroundColor = ConsoleColor.Black;   if (recentlyFlipped.Count != 0)  {  //if it's the first display, this allows the recently flipped counters to show orange for 2 seconds  if (secondTimeBoard == false)  {  System.Threading.Thread.Sleep(2000);  foreach (var item in recentlyFlipped)   {  BoardState[item.row, item.column] = "W";  }  recentlyFlipped.Clear();  secondTimeBoard = true;  }  }   //prints normal board with no yellows, after highlighting  if (secondTimeBoard == true)  {  secondTimeBoard = false;   Console.Clear();  displayBoard();  }  }  } }  5.1 Prototype Code **Program Class:**  using System; using System.Collections.Generic; using System.Linq; using System.Text; using System.Threading.Tasks;  namespace OthelloPlayer {  class Program  {     static void Main(string[] args)  {  Console.WriteLine("Welcome to Othello Player!");  //set up board with starting discs   Board gameBoard = new Board();   //returns a menu option which sends you down one of three switch statements.  switch (displayMenu())  {  case 1:  Console.Clear();  gameBoard.displayBoard();  playGame(ref gameBoard);  break;  case 2:  Console.Clear();  Console.WriteLine(" Difficulties:");  Console.ReadKey();  break;  case 3:  Console.Clear();  Console.WriteLine("Thanks for Playing!");  break;   }    }   public static int LetterToInt(char letter)  {  string strLetter = letter.ToString().ToUpper();  char UpperCaseChar = strLetter.ToCharArray()[0];   return UpperCaseChar - (int) 'A';  }   public static void playGame(ref Board board)  {  bool PlayerTurn = true;  int row;  int column;  while (!Board.isFull(ref board))  {  //Carries out the players turn  if (PlayerTurn)  {  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Player One (Black)'s turn...");  Console.ForegroundColor = ConsoleColor.White;     if (Board.canPlaceCounter(ref board, PlayerTurn))  {  do  {  Console.WriteLine("Where would you like to place your counter?");  Console.Write("Enter column: ");  column = LetterToInt(char.Parse(Console.ReadLine()));  Console.Write("Enter row: ");  row = int.Parse(Console.ReadLine()) - 1;  if (!board.checkValidMove(ref board, PlayerTurn, row, column))  {  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Cannot place here! Try again");  Console.ForegroundColor = ConsoleColor.White;  }     } while (!board.checkValidMove(ref board, PlayerTurn, row, column));  board.placeCounter(ref board, PlayerTurn, row, column);  Console.Clear();  board.displayBoard();  }  else  {   Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Player 1 can't go anywhere! Go skipped...");  Console.ForegroundColor = ConsoleColor.White;    }  PlayerTurn = false;  }   else  {  //Carries out the Player two's turn  if (Board.canPlaceCounter(ref board, PlayerTurn)) {    Console.ForegroundColor = ConsoleColor.Green;  Console.WriteLine("Player Two (White)'s turn...");  Console.ForegroundColor = ConsoleColor.White;   do  {   Console.Write("Enter column: ");  column = LetterToInt(char.Parse(Console.ReadLine()));  Console.Write("Enter row: ");  row = int.Parse(Console.ReadLine()) - 1;   if (!board.checkValidMove(ref board, PlayerTurn, row, column))  {  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Cannot place here! Try again");  Console.ForegroundColor = ConsoleColor.White;  }  } while (!board.checkValidMove(ref board, PlayerTurn, row, column));  board.placeCounter(ref board, PlayerTurn, row, column);  Console.Clear();  board.displayBoard();    }   else  {  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine("Player 2 can't go anywhere! Go skipped...");  Console.ForegroundColor = ConsoleColor.White;    }  PlayerTurn = true;   }     }  Console.WriteLine("GAME OVER, winner was...");  System.Threading.Thread.Sleep(3000);  Console.ForegroundColor = ConsoleColor.Yellow;  Console.WriteLine(Board.checkWinner(ref board));   Console.ReadKey();   }   public static int displayMenu()  {  Console.WriteLine(" > Play Game");  Console.WriteLine(" Select difficulty");  Console.WriteLine(" Exit");   ConsoleKeyInfo choice = Console.ReadKey(true);  int option = 1;  while (choice.Key != ConsoleKey.Enter)  {   if (choice.Key == ConsoleKey.UpArrow && option > 1)  {  Console.CursorLeft = 0;  Console.CursorTop = option;  Console.Write(" ");  Console.CursorTop = option - 1;  Console.CursorLeft = 0;  option--;  Console.Write(" >");  }  if (choice.Key == ConsoleKey.DownArrow && option < 3)  {  Console.CursorLeft = 0;  Console.CursorTop = option;  Console.Write(" ");  Console.CursorTop = option + 1;  Console.CursorLeft = 0;  option++;  Console.Write(" >");  }   choice = Console.ReadKey(true);   }  return option;    }  } }  **Board class:**  using System; using System.Collections.Generic; using System.Linq; using System.Text; using System.Threading.Tasks;  namespace OthelloPlayer {  class Board  {  public struct coordinate  {  public int column;  public int row;   public coordinate(int setRow, int setColumn)  {  column = setColumn;  row = setRow;  }  }   public static string[,] BoardState = new string[8, 8];   public Board()  {  setUpBoard();   }   public void placeCounter(ref Board board, bool PlayerTurn, int row, int column)  {    turnCounters(ref board, row, column, true, PlayerTurn);   }   public bool checkValidMove(ref Board board, bool PlayerTurn, int row, int column)  {  if(row > 7 || row < 0 || column > 7 || column < 0)  {  return false;  }   if (BoardState[row, column] != " ")  {  return false;  }    if (turnCounters(ref board, row, column, false, PlayerTurn).Count != 0)  {  return true;  }  return false;    }   public static bool isFull(ref Board board)  {  //loop through values in board:  for (int i = 0; i < 7; i++)  {  for (int a = 0; a < 7; a++)  {   if (turnCounters(ref board, i, a, false, true).Count != 0)  {  return false;  }  if (turnCounters(ref board, i, a, false, false).Count != 0)  {  return false;  }  }  }  return true;  }   public static bool canPlaceCounter(ref Board board, bool PlayerTurn)  {  for (int i = 0; i < 7; i++)  {  for (int a = 0; a < 7; a++)  {  if (board.checkValidMove(ref board, PlayerTurn, i, a))  {  return true;  }  }  }  return false;  }   public static string checkWinner(ref Board board)  {  int onePoints = 0;  int twoPoints = 0;  for (int i = 0; i < 7; i++)  {  for (int a = 0; a < 7; a++)  {  if(BoardState[i, a] == "B")  {  onePoints++;  }  if (BoardState[i, a] == "W")  {  twoPoints++;  }    }  }  if(onePoints > twoPoints)  {  return "Player One! (BLACK) Congratulations!";  }  if (onePoints < twoPoints)  {  return "Player Two! (WHITE) Congratulations!";  }  else return "Nobody, it was a draw!";  }   //trying to make it work for AI turn too, added bool PlayerTurn  public static List<coordinate> turnCounters(ref Board board, int row, int column, bool flip, bool PlayerTurn)  {  List<coordinate> discsToTurn = new List<coordinate>();  List<coordinate> tempDiscsToTurn = new List<coordinate>();   string myColour;  string notMyColour;  if (PlayerTurn)  {  myColour = "B";  notMyColour = "W";  }  else  {  myColour = "W";  notMyColour = "B";  }   //check horizontal right:  if((column + 2) < 7)  {  if (BoardState[row, column + 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row, (column + 1)));  for (int i = column + 2; i < 7; i++)  {  if (BoardState[row, i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row, i));  //add them to temporary list  }  if(BoardState[row, i] == myColour)  {  foreach(var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (i < 8)  {  if (BoardState[row, i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  }  //checked horizontal right   //check horizontal left  if ((column - 2) > -1)  {  if (BoardState[row, column - 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row, (column - 1)));  for (int i = column - 2; i > -1; i--)  {  if (BoardState[row, i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row, i));  //add them to temporary list  }  if (BoardState[row, i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = -1;  }  if (i > -1)  {  if (BoardState[row, i] == " ")  {  i = -1;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  }  //checked horizontal left.  //check up:  if ((row - 2) > -1)  {  if (BoardState[row - 1, column] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - 1, column));  for (int i = row - 2; i > -1; i--)  {  if (BoardState[i, column] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(i, column));  //add them to temporary list  }  if (BoardState[i, column] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = -1;  }  if (i > -1)  {  if (BoardState[i, column] == " ")  {  i = -1;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  }  //checked up.  //check down:  if ((row + 2) < 8)  {  if (BoardState[row + 1, column] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + 1, column));  for (int i = row + 2; i < 8; i++)  {  if (BoardState[i, column] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(i, column));  //add them to temporary list  }  if (BoardState[i, column] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (i < 8)  {  if (BoardState[i, column] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  }  //Checked down.  //Check SE diagonal:  if (((row + 2) < 8) && ((column + 2) < 8))  {  if (BoardState[row + 1, column + 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + 1, column + 1));  for (int i = 2; row + i < 8 && column + i < 8; i++)  {  if (BoardState[row + i, column + i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + i, column + i));  //add them to temporary list  }  if (BoardState[row + i, column + i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (i < 8)  {  if (BoardState[row + i, column + i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  }  //Checked SE diagonal  //Check NW diagonal:  if (((row - 2) > -1) && ((column - 2) > -1))  {  if (BoardState[row - 1, column - 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - 1, column - 1));  for (int i = 2; row - i > -1 && column - i > -1; i++)  {  if (BoardState[row - i, column - i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - i, column - i));  //add them to temporary list  }  if (BoardState[row - i, column - i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (row - i > -1)  {  if (BoardState[row - i, column - i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  }  //Checked NW diagonal.  //Check NE diagonal:  if (((row - 2) > -1) && ((column + 2) < 8))  {  if (BoardState[row - 1, column + 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - 1, column + 1));  for (int i = 2; row - i > -1 && column + i < 8; i++)  {  if (BoardState[row - i, column + i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row - i, column + i));  //add them to temporary list  }  if (BoardState[row - i, column + i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (row - i > -1)  {  if (BoardState[row - i, column + i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  }  //Checked NE diagonal  //Check SW diagonal  if (((column - 2) > -1) && ((row + 2) < 8))  {  if (BoardState[row + 1, column - 1] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + 1, column - 1));  for (int i = 2; column - i > -1 && row + i < 8; i++)  {  if (BoardState[row + i, column - i] == notMyColour)  {  tempDiscsToTurn.Add(new coordinate(row + i, column - i));  //add them to temporary list  }  if (BoardState[row + i, column - i] == myColour)  {  foreach (var item in tempDiscsToTurn)  {  discsToTurn.Add(item);  }  tempDiscsToTurn = new List<coordinate>();  i = 8;  }  if (row + i < 8)  {  if (BoardState[row + i, column - i] == " ")  {  i = 8;  tempDiscsToTurn = new List<coordinate>();  //removes coordinates from the list as not a valid flank  }  }  }  }  }  //Checked SW diagonal  //CHECKED ALL DIRECTIONS    //flip counters  if (flip)  {  BoardState[row, column] = myColour;  foreach (var item in discsToTurn)  {  BoardState[item.row, item.column] = myColour;  }  }  return discsToTurn;   }          public void setUpBoard()  {  for (int i = 0; i < 8; i++)  {  for (int a = 0; a < 8; a++)  {  BoardState[i, a] = " ";  }  }  BoardState[3, 3] = "W";  BoardState[4, 4] = "W";  BoardState[4, 3] = "B";  BoardState[3, 4] = "B";    }   public int getBlackScore()  {  int black = 0;  foreach(var item in BoardState)  {  if(item == "B")  {  black++;  }  }  return black;   }  public int getWhiteScore()  {  int white = 0;  foreach (var item in BoardState)  {  if (item == "W")  {  white++;  }  }  return white;   }   public void displayBoard()  {   Console.BackgroundColor = ConsoleColor.DarkGreen;  Console.Write(" - A --- B --- C --- D --- E --- F --- G --- H --");  Console.BackgroundColor = ConsoleColor.Black;  Console.Write(" SCORES:");  Console.WriteLine();  Console.BackgroundColor = ConsoleColor.DarkGreen;  for (int i = 1; i < 9; i++)  {  Console.Write(" " + i + " |");  for(int a = 0; a < 8; a++)  {  if(BoardState[i-1, a] == "B")  {  Console.ForegroundColor = ConsoleColor.Black;  Console.Write(" █");  Console.ForegroundColor = ConsoleColor.White;   }  else if (BoardState[i - 1, a] == "W")  {  Console.ForegroundColor = ConsoleColor.White;  Console.Write(" █");   }  else  {  Console.Write(" ");  }   Console.Write(" |");    }  if(i == 1)  {  Console.BackgroundColor = ConsoleColor.Black;  Console.Write(" Black: " + getBlackScore());  Console.BackgroundColor = ConsoleColor.DarkGreen;  Console.WriteLine();  Console.Write(" ------------------------------------------------");  Console.BackgroundColor = ConsoleColor.Black;  Console.WriteLine(" White: " + getWhiteScore());  Console.BackgroundColor = ConsoleColor.DarkGreen;   }  else {  Console.WriteLine();  Console.WriteLine(" ------------------------------------------------");  }      }   Console.BackgroundColor = ConsoleColor.Black;    }      } } |
| --- |