# Connect 4 AI

Daniel Yates

Computer Science 2018

# Table of Contents

# 1 Analysis

## 1.1 Project Outline

The project I am going to create will be a version of the widely known board game "Connect 4". The original game of Connect 4 has two players (Red and Yellow) who each take it in turn to drop their respective piece into one of the columns of the 6 high 7 wide grid. The aim of the game, hence the name, is to connect four of your pieces in a row (either vertically, horizontally or diagonally). If however the game board fills up before either player gets four in a row, the game is declared a draw.

My game, however, will be different. The main objective will be to create an artificial intelligence (AI) which the user will be able to play against. The end game will have the option for player versus (vs) player (PVP), player vs AI (PVAI) and a bonus mode of AI vs AI (AIVAI). There will be other features, but these will be outlined in the objectives stage of the documentation.

## 1.2 Problem Outline

Based on research online, there are not an awful lot of Connect 4 games that provide any unique or different features to the base game. I am to change this by creating a game of Connect 4 with features not commonly seen such as different skill levels of AI, an AI vs AI mode, changeable board sizes and the ability to save the game. These are some of the main features that current games lack. A Connect 4 game I found online at the URL "http://connect4.gamesolver.org" had a few major drawbacks. The first was that at times the AI would become slow. When there were many options for it to take, it would lock up and think for maybe 8+ seconds which is completely unreasonable. I aim for my AI to choose its position in no longer than 2 seconds. Another major drawback is that this system has no difficulty changer. So you essentially have no choice but to play against a perfect playing AI, making it almost impossible to beat. For the majority of people, this ruins the experience as it goes from being an enjoying challenge to completely unreasonable. Hence for my project there will be a way to change the difficulty of the AI. A lot of other examples of Connect 4 programs I found online seemed to all be player vs player as a pose to having any sort of AI whatsoever. This to me was quite surprising, as I thought this would be a more common feature, but clearly this was not the case.

## 1.3 Users and Their Needs

This project really appeals to the masses. It could be used by primary school children, looking to play a game of Connect 4 as a learning exercise or even entertainment purposes, or the game could be used by professionals, looking to improve their skills by playing against an incredibly hard to beat artificial intelligence. This will be achievable by having the ability to change the difficulty from very easy difficulties to very hard difficulties using an intuitive menu. There will also be the ability to change the board dimensions in order to provide fun and potentially easier to understand or even more challenging alternatives for children/teenagers. I decided to create a questionnaire and give it to my Computer Science as well as Information

Technology classes with some questions about a Connect 4 game/system to see what features they thought would be good to have. The results were as follows.

| Question | Answer | |
|---|---|---|
| | **Yes** | **No** |
| **Game Modes** | | |
| Would a Player vs Player mode be of interest to you? | 100% (10) | 0% (0) |
| Would a Player vs Artificial Intelligence mode be of interest to you? | 100% (10) | 0% (0) |
| Would a mode where the computer plays itself be something you would be interested in? | 70% (7) | 30% (3) |
| Would the ability to play against another player/user across the same network be of interest to you? | 30% (3) | 70% (7) |
| Would more than 2 players (3 or more) playing on the same board be a feature you would like the game to have? | 20% (2) | 80% (8) |
| Would a "Connect n" where n is a positive integer be of interest to you? | 30% (3) | 70% (7) |
| **Game Features** | | |
| Would a save game feature allowing you to come back to a previously half-finished game be of use to you? | 90% (9) | 10% (1) |
| Would the ability to choose/change the colour of the piece that you would like to play as e.g. (Red, Yellow, Blue, etc.) be a useful feature to you? | 80% (8) | 20% (2) |
| **Game Settings** | | |
| Is the ability to change the size of the board something that you would like? | 70% (7) | 40% (3) |
| Would the ability to change the difficulty of the AI be a good feature to include? | 100% (10) | 0% (0) |
| **Miscellaneous** | | |
| Is the speed at which the computer makes its decision important to you? | 80% (8) | 20% (2) |

Overall, it is very apparent that most of my initial ideas are things that people would be interested to see implemented into the game. Overall, it seems clear to me that people are not interested in changing the fundamental game mechanics of Connect 4 such as having n players as a pose to 2 or having Connect n as a pose to Connect 4. This was interesting to see and also somewhat surprising to me. It is clear now that although people do want extra features such as changing the board size, these other more significant features are not something the majority of people would be comfortable with. I was also surprised to find out that playing against another player across the same network was not something the majority of people were interested in. For these reasons, I will not be including these features in my game.

# 1.4 Objectives

## 1.4.1 Game Modes

### 1.4.1.1 Player vs Player

There will be a mode in which two different human players play against one another on the same computer. Each player will be able to choose a piece colour. The board dimensions will have to be decided amongst both players and inputted. This feature will be implemented because 100% of the people who completed my questionnaire said that this was a feature they would like in the game.

### 1.4.1.2 Player vs Artificial Intelligence

There will be a mode in which one human player plays against one artificial intelligence (AI) player. The human player will be able to choose a piece colour and the AI player will use the letter "A" as its piece. The board dimensions will be decided by the human player. The difficulty for the AI will also be decided by the human player. The difficulty will either be "Easy", "Normal" or "Hard". The hardest difficulty for the AI should take no more than a maximum of two seconds. This is so that the game does not become slow or appear to have frozen or crashed for the user. This feature will be implemented because 100% of the people who completed my questionnaire said that this was a feature they would like in the game.

### 1.4.1.3 Artificial Intelligence vs Artificial Intelligence

There will be a mode in which two AI players play against one another. The first AI player will use the letter "A" as its piece and the second AI player will use the letter "I" as its piece. The board dimensions will be decided by the human user before the game begins. This feature will be implemented because the majority (70%) of the people who completed my questionnaire said that this was a feature they would like in the game.

## 1.4.2 Game Features

### 1.4.2.1 Main Menu and Other Menus

The game will have a main menu at the beginning, allowing the user to choose the game mode that they wish to play. Upon choosing, they will then be taken to the respective menus such as choosing board dimensions or game difficulty in order to customise the game that they are playing to their liking.

### 1.4.2.2 Ability to Save and Load a Game

There will be a way in which the user(s) can save the game. This will save the board state, the free columns and the game data to respective text files so that the game can be loaded at a later date. The user will be able to enter a value during the game to save and exit the game. The user will be able to load a game based on the data in these save files at the main menu of the game. This feature will be implemented because the mass majority (90%) of the people who completed my questionnaire said that this was a feature they would like in the game.

### 1.4.2.3 Ability to Choose/Change Piece Colour

After the user has chosen either the "Player vs Player" or "Player vs Artificial Intelligence" game mode, they will be able to choose their piece colour/letter. They will be able to enter the value of their piece/colour before the game starts and this is what their piece will be represented by throughout the game. This feature will be

implemented because the majority (80%) of the people who completed my questionnaire said that this was a feature they would like in the game.

### 1.4.3 Game Settings

#### 1.4.3.1 Change Artificial Intelligence Difficulty

After the user has chosen the "Player vs Artificial Intelligence" game mode, they will be able to change the difficulty of the AI. They will be able to choose either "Easy", "Normal" or "Hard". If, for example, I decide to use a game tree for the AI, then the greater the difficulty, the further into the game tree the AI will look. It does not surprise me that this is a feature that people would like due to the fact that depending on whether you are a low, medium or high skilled player it is good to be able to choose a level of AI that suits your ability. This feature will be implemented because 100% of the people who completed my questionnaire said that this was a feature they would like in the game.

#### 1.4.3.2 Change Board Size/Dimensions

After the user has chosen any of the three game modes, they will be given the ability to change the dimensions of the board. This will be done by them providing a width and height of the board, which will then be used by the program in order to dynamically generate the board. There will be a minimum width/height of 4 and a maximum width/height of 10 that the user can choose in order to keep the board size reasonable and the AI to not get too slow. This feature will be implemented because the majority (70%) of the people who completed my questionnaire said that this was a feature they would like in the game.

### 1.4.4 Miscellaneous

#### 1.4.4.1 Optimised and quick algorithms

All of the algorithms in the program will be as optimised and execute as quickly as possible. This is very important due to the fact that the user does not want to be left waiting, potentially even thinking that the program has crashed if something takes too long. This feature will be implemented because the majority (80%) of the people who completed my questionnaire said that this was a feature they would like in the game.

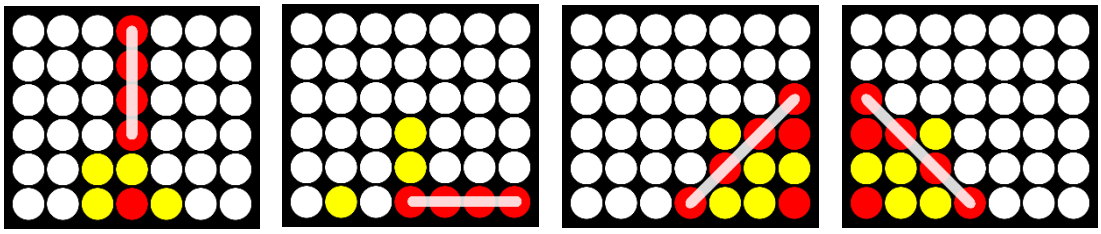
## 1.5 Research for The Solution

### 1.5.1 Connect 4

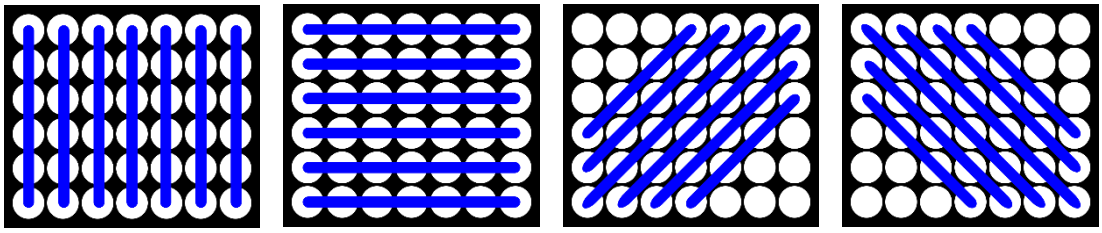
Connect 4, otherwise known as Four in a Row, is a two player abstract strategy board game. The aim, hence the name, is to connect 4 of your colour of piece in a row. These rows can be vertical, horizontal, or diagonal (either with a positive or negative gradient). For the standard board size of 7x6, there are 4,531,985,219,092 different board positions. This shall be discussed further in terms of optimisation for the artificial intelligence part of this documentation. Connect 4 is a solved game. It was solved by James Dow Allen in 1988 which was originally done using nine knowledge-based strategies. Since then, Connect 4 has been solved using brute force methods in 1995. Connect 4 is a perfect information game. A perfect information game is a game in which both players can see the entirety of what is going on in the game. A game like Poker is an example of a game that is not a perfect information game. This is because in Poker, each player has a hand of cards that each other player cannot necessarily see. This makes it difficult to create an AI to play a game like Poker. Connect 4 on the other hand is a perfect information game. This means that certain AI algorithms can be used in order to evaluate the board state, and choose positions based on heuristic values that best benefit the AI player or least benefit the opposing player. Connect 4 is also a zero-sum game. What this means is that the AI's gain or loss of score/utility is exactly balanced by that of the gain/loss of the opposing player. What this means is that one player's gain is another player's loss. This is perfect for the AI algorithms I will discuss later in the documentation.

### 1.5.2 Win Checking Algorithm

In Connect 4, there are 4 directions in which a player can win or have four pieces in a row. These directions are vertically (|), horizontally (_), diagonally positively (/) and diagonally negatively (\).



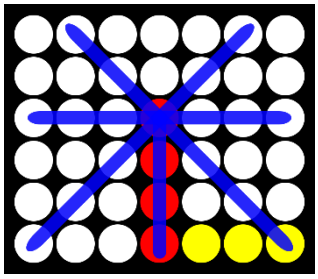
Above are representations to show examples of each of the four different win states.

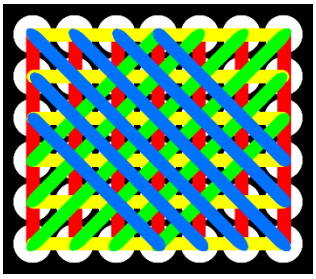


Above are all of the possible ways that a win could be generated.

My main goals for the win checking algorithm is that it needs to be as quick as possible so as not to leave players waiting or the program to lag/run slow. My initial idea was to check every single location on the board for a win. This would mean starting in the top left of the board (0, 0) and working across the rows for horizontal checks until the end of the board, then repeating the process vertically. Diagonal

checking would follow a similar approach, except starting at designated starting locations (for positive diagonals the first location would start at the first column and the fourth row). This, however, was a bad way of checking for wins. First of all, checking the entire board is just completely unnecessary, meaning lots of areas will be checked that simply have not been touched by players either at all or at all recently. Also, one feature for the game will be to have a dynamic board size, allowing players to make a grid of dimensions x by y where x and y are whole integers of their choosing. Another major issue and reason for having a well optimised win checking algorithm is simply because the artificial intelligence part of the program will have to check for so many wins that checking the entire board every time a single piece is placed in a single location is unfeasible.





For this reason, I devised a new, better, significantly more optimised win checking algorithm. Every time a piece is placed, that particular piece can cause a win: below itself (note: not vertically/above as this is impossible), horizontally to itself and either positively or negatively diagonally to itself. On the right (leftmost image) is a diagrammatical representation of this win checking algorithm. The blue lines show the directions in which the algorithm would check for a win. The other image on the right however (rightmost image) is a diagrammatical representation of the previous win checking algorithm (red for vertical checks, yellow for horizontal checks, green for diagonal positive checks and blue for diagonal negative checks. It is very apparent from the diagrams alone that the old win checking algorithm has significantly more to check, hence blatantly showing that it would take much longer to check for a win than with the newer, more optimised method. For this reason, the new algorithm shall be used.

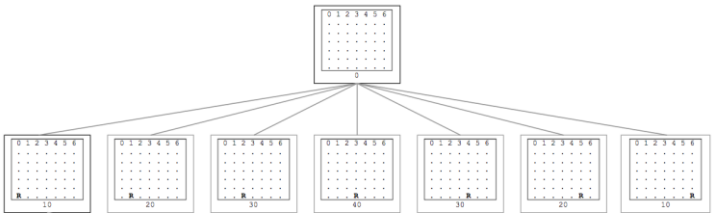
In terms of the logistics behind the algorithm, a consecutive pairing system shall be used. What this means is that in each direction, the algorithm is looking for 3 consecutive pairs of the same coloured piece, which in turn means there is a win. 3 pairs of the same colour piece in a row means that there are 4 of that piece in a row. If, say, it finds two pairs (3 in a row), and then afterwards it does not find a pair, the program will set the pairs counter back to a value of 0. The algorithm itself will be developed in such a way that it does not use specific or exact values of a 7x6 board, instead being dynamic in such a way that a win checking algorithm will work no matter what the dimensions of the board. This is essential for my project as the user will be able to define their own dimensions for the board.

### 1.5.3 Artificial Intelligence Algorithm

Artificial intelligence (AI) comes in a variety of different types. Based on research from a variety of different sources online, there appears to be 4 main "Types/Categories" of AI (Type 1, Type 2, Type 3 and Type 4). Currently, type 3 and 4 do not actually exist, they are simply conceptual, so I shall only discuss types 1 and 2. Type 1 AI are referred to as reactive machines. They are the most basic types of AI systems. They do not have the ability to create memories or use experiences from the past to make informed decisions. Deep Blue by IBM is an example of a type 1 AI, which beat professional Chess player Garry Kasparov in the 1990s at a game of

Chess. This type 1 AI differs to type 2 AIs which are otherwise known as machine learning AI. These AI can look into the past and make informed decisions about what to do in situations based on how they have previously done well or done badly. Autonomous or self-driving cars are an example of type 2 AI. It seems quite clear that it makes more sense to create a type 1 AI, than to make a machine learning AI, but I shall discuss the reasons why further.

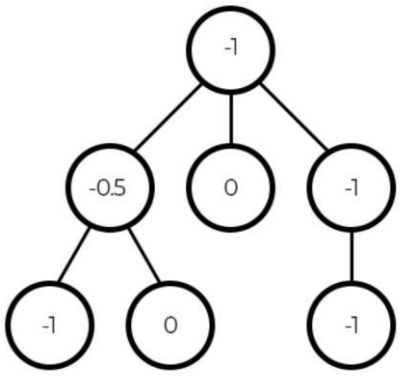
In terms of Connect 4, a zero-sum, perfect information, solved game, there are two main algorithms I could find online that correlate with the problem I am intending to solve. These algorithms are the Minimax algorithm and the Monte Carlo algorithm. These two algorithms are similar in a sense that they both evaluate game trees. However, they are distinctly different in their approach to evaluating the trees.



The Minimax algorithm gets its name from the idea behind the algorithm. That is, to maximise the minimum gain. Essentially, the algorithm traverses a tree. This tree has every single board position up to a certain depth. The diagram above shows a Connect 4 game tree up to a depth of 1. The tree starts at the root node which, in this case, is an empty board. The tree then spans off with leaves or children which are each of the (in this case 7) possible positions that a piece could be placed by the AI. If the depth was more than 1 (for example 2), then these leaves would become nodes in their own right and have their own children. In this case, the tree would then have another 7 children for each of the 7 original children of the original root node. These leaves would be the opposite piece to their parent node, as in a game this is how the pieces would be placed (one piece, then the other, etc.). The tree would be generated in this way up to the specified depth. Upon creation of the tree, each node is assigned a heuristic value. This is a value to represent whether the node causes a win, loss or nothing. Minimax assigns positive infinity (+∞) to a winning node, negative infinity (-∞) to a losing node or 0 to a node where nothing occurs (in the middle of +∞ and -∞). The Minimax algorithm is a recursive algorithm which evaluates this tree, returning the heuristic value of the best node that it finds.

The Monte Carlo algorithm is somewhat similar, but actually distinctly different to the Minimax algorithm. The main similarities are that it traverses a tree structure and bases its decisions on heuristics. But that is where the similarities end. The Monte Carlo algorithm is a randomised algorithm which bases all of its decisions on probability. In terms of Connect 4, it would take a number of starting positions and play a series of completely random games or simulations until either a win, loss or draw. Based on these games, the algorithm can use probabilistic as well as heuristic scoring techniques in order to evaluate each of the chosen positions and whether or not they are the best to choose in terms of the best possible chance of a win. The Monte Carlo algorithm tends to be better for games that are intractable to brute force search, such as Go. At every level of say a full game tree for Go, there would be hundreds of different choices, making it completely unrealistic to evaluate this tree with the sort of computers we have to this date. AlphaGo was created by Google DeepMind which used the Monte Carlo algorithm (as well as other neural

networking techniques and other algorithms) in order to beat a professional Go player in 2015. This was a massive breakthrough for artificial intelligence.




Both algorithms to me seem very interesting to me. The Minimax algorithm fits the project very well, yet the Monte Carlo algorithm uses interesting heuristic and probabilistic techniques in order to generate the best move for the AI to make. So, I have decided to create an algorithm that is a "hybrid" of the two. The main basis of the game tree and recursive evaluation will be similar to that of the Minimax algorithm, yet it will use clever heuristic scoring and heuristic evaluation functions in order to make the decision for the best possible move. For my algorithm, a node will be scored a -1 if this move causes a win for the AI player, a 1 if this move causes a win for the opposing player and 0 if this move either causes a draw or nothing happens. This diagram on the right shows how the tree is then evaluated. There are four leaf nodes in the tree (-1, 0, 0 and -1). The parent of leaf nodes finds the average value of its children in order to calculate a heuristic value for itself. In the case of this tree, -0.5 is the average of -1 and 0 ((-1 + 0) / 2) and -1 is the average of -1 (-1 / 1). Once the tree gets to the layer below the root node, the program will choose the node (out of -0.5, 0 and -1) which has the lowest possible value (in this case -1) as this is the node that probabilistically causes the best chance of a win for the AI. If you look at the tree you can see clearly that it makes sense to go for that move due to the fact that the opposing player would have no choice but to go for a move that leads to a win for the AI. The diagram is a very small and simplified version of a tree that would otherwise have hundreds of nodes in a real game of Connect 4. The diagram is simply to demonstrate how the algorithm works. So all in all I am not going to use either the Minimax algorithm or the Monte Carlo algorithm as they stand by converting their pseudocode to actual code, instead I am going to create a hybrid algorithm using parts from both.

## 1.6 Possible Programming Languages

### 1.6.1 VB.NET (Windows Forms Program)

My initial thought for this project was to create a windows forms program with a full UI for Connect 4. The reason for VB.NET is because it is the language that I have been taught in my College and will be using it during the examinations and the skeleton program part of the course. I am extremely comfortable with writing programs (both console and windows forms) in VB.NET due to the amount of time that I have spent using the language. One benefit of using VB.NET is that I can make a graphical user interface (GUI) for the Connect 4 game. This is because VB.NET (using Visual Studio) has easy to use development tools to create form controls and GUIs. The main disadvantage of VB.NET is that is mainly aimed at making generic software such as database driven windows forms applications. This means that trying to use VB.NET for making an artificial intelligence would be somewhat slower than an alternative programming language such as Python which is significantly more lightweight and efficient.

### 1.6.2 Python (Console Program)

My next and only other thought for a programming language was to create a console program using Python. Python is a programming language that I have used a small amount at GCSE level, but not at all up until this project. Not only do I feel like Python would be more beneficial to me to learn than a language like VB.NET but it also suits my project significantly more. Based on research I have done on the internet, Python is a perfect language that is used a lot for AI projects. This is because it is fast, lightweight and efficient. It would be harder to make a graphical user interface (GUI) using Python as I would have to programmatically create every single control, however I do not believe a GUI is necessary for my project. A problem is that the extra processing power that would be needed to generate UI elements could cause the AI itself to slow down which, based on my survey where people said the program needed to be quick at making decisions, seems like a bad thing to cause.

## 1.7 Chosen Solution

The algorithms that my program is going to use for win checking, the artificial intelligence, etc. all need to run as quickly and efficiently as possible. A language like VB.NET is more than capable of implementing these algorithms, but it would not run them anywhere near as quickly as Python due to Python being efficient and lightweight. So, in terms of the algorithms that will be making the program perform its functionality, Python is clearly superior to a language like VB.NET.

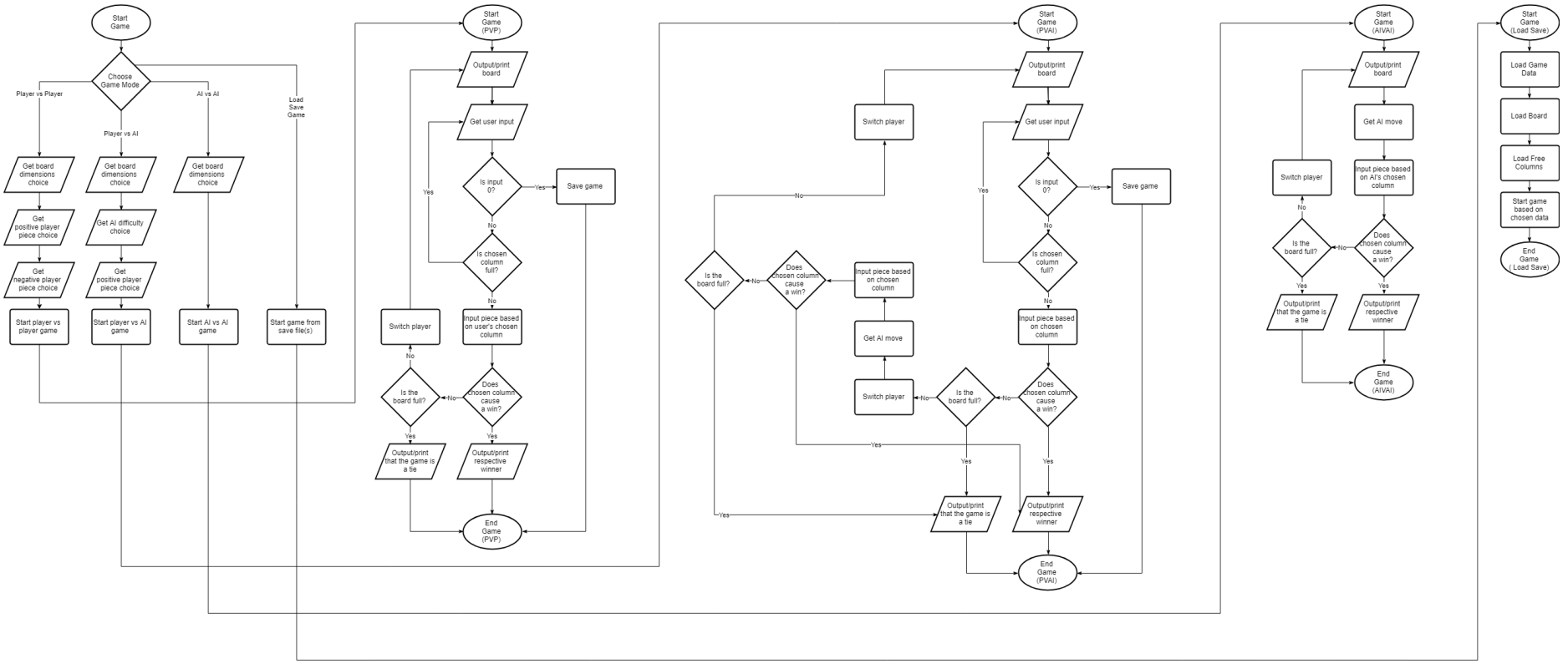
To produce a graphical user interface (GUI), it would make significantly more sense to use VB.NET due to the development tools which it provides. GUIs are very quick and easy to put together using an IDE like Visual Studio with VB.NET. I have had some experience putting together GUIs with Python before and whilst not impossible, they are tedious and complicated to put together. This is because, rather than a simple drag and drop process in the IDE like in Visual Studio, every form control has to be programmatically coded in. This tends to be very time consuming and often difficult to get everything correctly structured on the screen using coordinates. Although Python is at a disadvantage in this area, I do not believe a GUI is something that my program would greatly benefit from. This is due to the fact that, as stated earlier, the program needs to be quick and not use processing power displaying images or shapes on the screen.

Based on my research, I have decided I am going to use the programming language Python to make a console-based program. This is because of the features highlighted previously such as it being a fast, lightweight and efficient language, with simplistic syntax as well as being perfect for AI projects.

# 2 Design

## 2.1 Program Structure

### 2.1.1 Flowchart




This flowchart represents the flow that the system will follow and what the user will be greeted with. Simply put, it takes the user's initial input for the game mode that they would like, in turn followed by that mode's settings and then it performs the appropriate function/flow chart for that game mode.

### 2.1.2 Program Conventions

#### 2.1.2.1 Board Appearance

The board itself will be represented with an array. As the game/program is being made with a command line interface, the board will be represented with symbols/letters. For a 7 x 6 board it will look like the following:

```
1  2  3  4  5  6  7
-  -  -  -  -  -  -
-  -  -  -  -  -  -
-  -  -  -  -  -  -
-  -  -  -  -  -  -
-  -  -  -  -  -  -
-  -  -  -  -  -  -
```

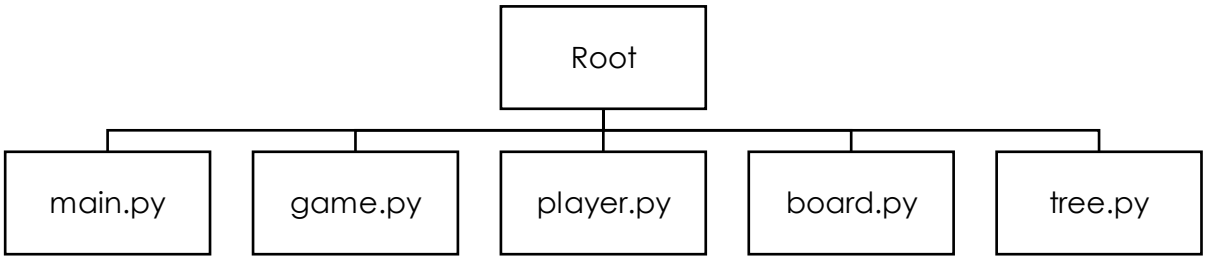
The column number (starting at 1 for the user rather than 0) will appear at the top of the column. Each of the empty cells/spaces/pieces in the board will be represented by a hyphen (-). If a user has a red piece (R) then when they input a piece the respective hyphen will be changed to their letter and stored in the board array.

#### 2.1.2.2 PEP8

PEP8 (Python Enhancement Proposal 8) is a programming convention or tool used for the programming language Python in order to maintain a consistent structure and naming conventions. Some main features of PEP8 are as follows:

- All function and variable names should be lower case or upper case, with each word in the function or variable name separated with an underscore (my_variable). Mixed case naming can also be used (myVariable) however I will be using the underscore naming convention to maintain consistency.
- Class names should start with a capital letter.
- Inline comments should be two spaces to the right of the code, with a single space before the comment text after the hash (#).
- There must be at most one blank line between lines of code.

#### 2.1.2.3 File/Module Structure




This is the structure that the files/modules for the program will follow. The "main.py" module will import the "game.py" module. The "game.py" module will import the "player.py", "board.py" and "tree.py" modules.

## 2.2 Algorithms

### 2.2.1 Win Checking

For the win checking algorithm for my game, I initially had two different ideas as stated in the analysis part of this documentation. Simply put, the first algorithm was going to check every single part of the board for four in a row, which in retrospect is completely unnecessary. For a win checking algorithm, I only need to check the areas below, horizontally to and diagonally to the piece that was most recently placed. This is due to the logistics of the game Connect 4 in that you cannot cause a win by placing a piece anywhere else on the board other than in these directions.

The function itself will take the board array as well as the x and y values (in terms of the array) that the piece was placed into. Based on this information, the algorithm will be able to work out if a win has been found or not. I intend to use a pairing approach for working out if there has been a win or not. For four pieces to be connected in a row, that is also the same as there being three consecutive pairs of the same value in a row. Note that they must be consecutive pairs, otherwise it makes no sense and the algorithm would not work at all.

#### 2.2.1.1 Vertical Win Checking (Down)

The first win checking function I will talk about is for wins vertically downwards. As a win cannot be caused above a piece immediately after it is placed, the algorithm only needs to compare itself to the three other pieces below itself to see if a win has been caused. The algorithm will also only check for a win downwards if the piece that has been placed is 4 or more spaces above the bottom of the grid. This is simply because a vertical win cannot have been caused if the piece is not four or more pieces above the bottom. The pseudocode for this algorithm is as follows:

```
win_check_vertically(board, row value of piece, column value of
piece):
      pairs = 0
      x = row value of piece
      y = column value of piece
      if x <= number of rows - 4:
            for i = 1 to 3:
                  if board[x][y] == board[x + i][y]:
                        pairs += 1
                        if pairs == 3:
                              return True
                  else:
                        pairs = 0
                        break
```

This function takes the board array as well as the row and column values of where the piece is/was placed in the board as parameters. First of all, a "pairs" variable is defined and set equal to 0 because to start with there are no pairs of pieces at all. It also defines "x" and "y" variables and sets the "x" variable equal to the row value that the piece was placed into and sets the "y" variable equal to the column value that the piece was placed into. Then, it checks if "x" is less than or equal to the number of rows minus four. This is to make sure that the piece is the fourth or higher piece vertically from the bottom of the board as there is no point checking to see if

there are three pairs of pieces when there are less than 3 pieces below the piece that has been placed anyway. If it is the fourth piece or higher above the bottom of the board then a for loop looping through the numbers 1, 2 and 3 is initiated. For each of the "i" values in the loop, an if statement checks to see if the original piece is equal to the piece 1 piece below itself, 2 pieces below itself, etc. Every time a match is found, the pairs variable is incremented by 1. Otherwise, if a pair is not found, the "pairs" variable is set to 0 and a break statement is used in order to break out of the loop. This is because if a pair has not been found then there is no chance of there being four in a row vertically downwards.

### 2.2.1.2 Horizontal Win Checking (Across)

The first win checking function I will talk about is for wins horizontally across. The algorithm will check everywhere in the horizontal axis for a win or for four pieces in a row. This algorithm will get the widdth of the board, and in turn check this entire row for whether or not a win has been generated. The pseudocode for this algorithm is as follows:

```
win_check_horizontally(board, row value of piece, column value of
piece):
      pairs = 0
      x = row value of piece
      y = column value of piece
      for i = 0 - y to len(board) - 1 - y:
            if board[x][y + i] != "-":
                  if board[x][y + i] == board[x][y + i + 1]:
                        pairs += 1
                        if pairs == 3:
                              return True
                  else:
                        pairs = 0
```

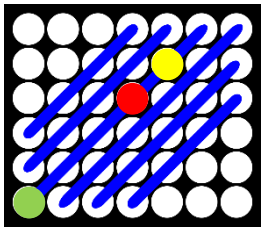
This function takes the board array as well as the row and column values of where the piece is/was placed in the board as parameters. First of all, a "pairs" variable is defined and set equal to 0 because to start with there are no pairs of pieces at all. It also defines "x" and "y" variables and sets the "x" variable equal to the row value that the piece was placed into and sets the "y" variable equal to the column value that the piece was placed into. Then, it initiates a loop. The loop is essentially just the full width of the board, created using the variables at hand. Let's say the piece was placed in the column 2 out of 7 columns (0, 1, **2**, 3, 4, 5, 6) then the "i" loop will be initiated from -2 (0 - 2) to 4 (7 (board length) - 1 - 2). This is necessary because these numbers will be added on in the next if statement(s). If the particular piece is not equal to an empty piece ("-"), then the code will check to see if the current piece is equal to the piece next to it. This is done by adding 1 to the "y" value of the board, as this is the across value. If it is then it will set the pairs value equal to itself plus 1, otherwise it will set the pairs value equal to 0. If there are three pairs found, the function will return True as this means that a win has been found horizontally.

### 2.2.1.3 Diagonal Win Checking (Positive)

The diagonal win checking algorithm is a lot more complex than the previous algorithm(s). This is just due it its nature and the fact that there is a lot more variables to take into account when checking for a win diagonally. Every time a piece is placed, the algorithm needs to work out which diagonal it is in. The diagram on the

right shows the different diagonal positions available for a 7x6 board. When a piece is placed, the algorithm does not check every single one of the blue lines, instead only the blue line that corresponds to where the player's piece has been placed. The pseudocode for this algorithm is as follows:

```
win_check_diagonally_positive(board, row value of piece, column
value of piece):
      pairs = 0
      x = row value of piece
      y = column value of piece
      x_max = board length (top to bottom) - 1
      y_max = board length (left to right) - 1


      if x + y <= x_max:
            start_x = x + y
            start_y = 0
      elif x + y > x_max:
            start_x = x_max
            start_y = (x + y) - x_max

      num_to_check = 0

      if start_y == 0:
            num_to_check = start_x
      elif start_y > 0:
            num_to_check = (start_x - start_y) + 1

      for i = 0 to num_to_check:
            if board[start_x - i][start_y + i] != "-":
                  if board[start_x - i][start_y + i] == board[start_x
- I - 1][start_y + i + 1]:
                        pairs += 1
                        if pairs == 3:
                              return True
                  else:
                        pairs = 0
```



This is the pseudocode for the algorithm. It works in a somewhat similar way to the other win checking algorithms, but the actual theory behind how and why it works is slightly harder to understand. The "x_max" and "y_max" values have 1 minused from them. "start_x" and "start_y" values are defined to start with. Essentially, if the total of the x and y values of the player's piece is less than or equal to that of the maximum height of the board then the starting x is made equal to the combination of the two x and y values and the starting y value is made equal to 0. Otherwise, the starting x value is equal to the maximum height of the board and the starting y value is made equal to the combination of the x and y values, minus the maximum height of the board. To enforce the fidelity of this algorithm, I will use an example illustrated with the diagram on the right. The red circle is where the player has placed their piece, and the green

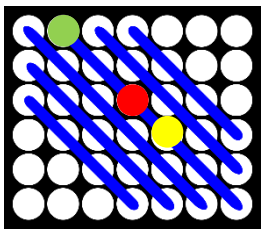
circle is the starting point of the diagonal in line with this piece. The x value of the red piece is 3 and the y value is 2. The combination of the two is 5, which is less than or equal to the value of the maximum height of the board (5 [height of board minus 1]). This in turn means that the starting x value for this win check is x + y = 5 and the starting y value is 0. This lines up with what the diagram shows. The pseudocode then sets the number to check (the total number in the row that there is to check [which is 5 in the case described]) equal to the starting x value if the starting y value is 0, otherwise it sets it to the starting x value minus the starting y value with 1 added on. The algorithm then starts a loop from 0 to the number of pieces to check. If the piece in question is not empty (equal to "-") then it will compare itself to the piece above it and to the right by minusing 1 from the x value and adding 1 to the y value. This is because, for example, if we had a piece at [2][3], the piece above and to the right of it would be at [1][4] (the red circle vs the yellow circle in the diagram). Again, the same way as the other algorithms, if 3 pairs of pieces are found then the algorithm will return true, otherwise it will set the pairs value back to 0 and keep looping.

#### 2.2.1.4 Diagonal Win Checking (Negative)

The negative diagonal win checking algorithm is extremely similar to that of the positive win checking algorithm (as discussed previously), however it is different in a sense that some of the signs (+ / -) have been changed due to the fidelity of how the pieces line up, starting points, etc. This differs from the positive algorithm in a sense that rather than starting in the bottom left segment of the board and working upwards, the algorithm starts in the top left segment of the board and works downwards. The pseudocode for this algorithm is as follows:

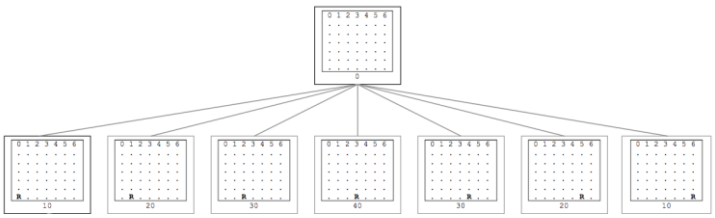
```
win_check_diagonally_negative(board, row value of piece, column
value of piece):
      pairs = 0
      x = row value of piece
      y = column value of piece
      x_max = board length (top to bottom)
      y_max = board length (left to right)


      if x >= y:
            start_x = x - y
            start_y = 0
      elif x + < y:
            start_x = 0
            start_y = y - x

      num_to_check = 0

      if start_y == 0:
            num_to_check = x_max - start_x
      elif start_y > 0:
            num_to_check = y_max - start_y

      for i = 0 to num_to_check:
            if board[start_x + i][start_y + i] != "-":
```

```
                if board[start_x + i][start_y + i] == board[start_x
+ i + 1][start_y + i + 1]:
                      pairs += 1
                      if pairs == 3:
                            return True
                else:
                      pairs = 0
```



The first thing this algorithm does differently is setting the start x and y values. If the x value is greater than or equal to the y value, then the starting x value is set to the x value minus the y value and the starting y value is set to 0. Otherwise, the starting x value is set to 0 and the starting y value is set to the y value minus the x value. In the example on the diagram to the right, the x value is 2 and the y value is 3 (the red piece). The x value is not greater than or equal to the y value, so the starting x value is 0, and the starting y value is the y minus the x (3 - 2) which is 1. This correlates with where the starting position should be (the green piece). The algorithm then works out the number of pieces there are to check by seeing if the starting y value is equal to 0 or greater than it. If it is greater than 0, like ours, the number to check is the width of the board minus 1 (y_max) minus the starting y value. This in our case is 6 minus 1 which is 5. This correlates with the number of pieces to check in this diagonal row. Again, although there are 6 pieces to check, when adding 1 onto the x/y values, I do not want it to overrun and so this means that the number to check must be 1 less than what there actually is. The rest of the algorithm is the same as the previous win checking algorithm, but the main difference being that the "i" is added on to the x value because we are going down rather than up and one is added onto both the x and y values because again, we are going diagonally down rather than diagonally up.

### 2.2.2 Tree Creation



For the artificial intelligence (AI) algorithm to work, a game tree is needed. This game tree will store every single game position up to a certain specified depth that the game can have. The diagram on the right shows an example of a game tree for connect 4 up to a depth of 1 (as the tree goes one move into the future). In order to create the tree, a recursive algorithm will be used. Although iteration could be used, this would be much less "elegant" and also would not allow for a specified depth. Instead, this would have to be a set depth based on the number of iterative loops placed within one another. The algorithm will create a "Node" object and create children for this "Node" based on the number of free columns (playable moves). Each of the children will themselves be "Node" objects, hence it will be a recursive tree creation algorithm. The pseudocode for this algorithm is as follows:

```
class Node:
      init(depth, player_num, player1_piece, player2_piece, board,
free_columns, heuristic = 0):
            depth = depth
            player_num = player_num
```

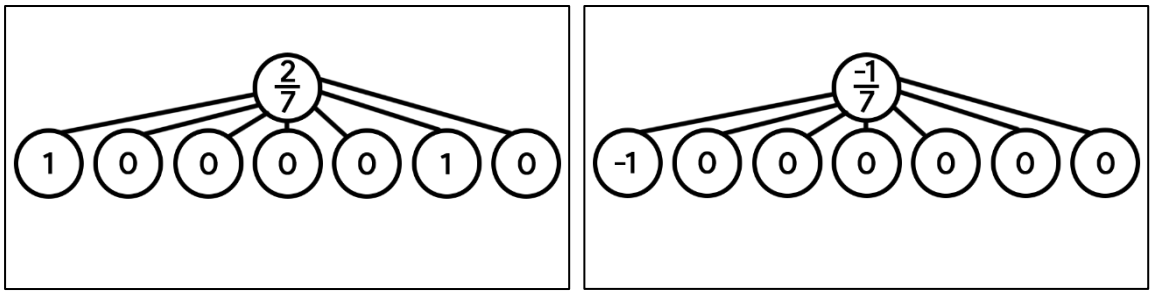
```
            player1_piece = player1_piece
            player2_piece = player2_piece
            board = board
            free_columns = free_columns
            heuristic = heuristic
            children = []
            create_children()

      create_children():
            if depth > 0:
                  for y in free_columns:
                        for x = board.length - 1 to 0, step -1:
                              if board[x][y - 1] == "-":
                                    board_copy = board[:]
                                    columns_copy = free_columns[:]
                                    board_copy[x][y - 1] =
player1_piece if player_num is 1 else player2_piece
                                    chosen_x = x
                                    chosen_y = y - 1
                                    if x == 0:
                                          delete
columns_copy[columns_copy.index_of(y)]
                                    break

                        children.append(Node(depth - 1, -player_num,
player1_piece, player2_piece, board_copy, columns_copy,
heuristic_value(chosen_x, chosen_y, board_copy)))

      heuristic_value(chosen_x, chosen_y, board_copy):
            win_check = check_for_win(board_copy, chosen_x, chosen_y)
            if win_check == True:
                  return 1 * player_num
            else:
                  return 0
```

The pseudocode for this algorithm is all a class. This is because each node is an object in its own right, with its own attributes and methods. The class starts off by creating a "Node" object with various attributes (depth, player_num, etc.). These are all passed into the class when the object is instantiated in the main code. An empty "children" array is created first as this will contain the other node objects added when the recursive algorithm is initiated. The "create_children" function (which is a method of the "Node" class) begins the process of creating the children for this node. The children for each node are all of the game states that can be generated after the currently passed board state. The "create_children" function first checks that it has not reached a depth of 0, as if it does then it needs to stop creating children. This is the base case for the recursive algorithm. If the depth is greater than 0 then a loop will be started for each of the free columns in the board. For each of the free columns a reverse loop going backwards from 5 up to 0 is started as this goes from the bottom of the board up to the top. For each of these x values it checks to see if this particular space is empty ("-") and if it is then it will fill this space with the player's respective piece (based on whether the "player_num" value is -1 or 1 where -1 is the AI's piece). The currently chosen x and y values are

stored in variables and if the x value is 0 (which means that this column is now full) then the respective columns for that piece is deleted from the "columns_copy" array. The loop then breaks at the end when a position for the piece has been found. The children array for this node object then has a "Node" object appended to itself with the changed board and column values and most importantly with a depth minus 1. This is essential as it ensures that the recursion does not go on forever. The reason this is recursive is because every time a new "Node" object is instantiated, the "create_children" function is performed straight away which appends "Node" objects to its "children" array which in turn runs the "create_children" function stragith away which appends "Node" objects, etc., etc. When "Node" objects are appended to the "children" array, they are rated with a heuristic value. This is done by using a function which takes the chosen x and chosen y values as well as the board state after these values have been inputted. The function "heuristic_value" then uses these values in order to use the win checking algorithm(s) (designed earlier) in order to rate each node. If a win is found, then the node is given a heuristic value of either 1 or -1 depending on which player the node is for. Otherwise, the node is given a heuristic value of 0 which means that the position caused no change to the board. This tree will be used by the AI algorithm(s) by traversing it and finding the average of all of the heuristic values, in turn working out which initial depth 1 node has the best probabilistic chance of causing a win.

### 2.2.3 Artificial Intelligence

Based on the analysis stage, the artificial intelligence (AI) algorithm for my game will evaluate all of the nodes within the tree, namely their heuristic value(s), and in turn work out which position is the most beneficial to choose based on probability. This will work by adding up the heuristic value(s) of the children of the node(s) and dividing this number by the total number of children. The two trees below are examples of trees that could be evaluated by this algorithm. The leaf nodes have heuristic values of either 1, 0 or minus 1. The AI algorithm will essentially find the average of said leaf nodes and make the heuristic value of the parent node equal to that of the average of its children. Let's say these two trees were connected to one another at a root node, the AI algorithm would choose the tree on the right due to the fact that it has the lowest heuristically evaluated value out of the two nodes. (-1/7 vs 2/7). The tree on the left would be a bad choice for the AI player because it leaves the AI susceptible to two positions where the opposing player can win. The tree on the right would be a good choice for the AI player because it leaves the AI with 1 position where it can win the game.




The algorithm will be created using a couple of functions. The first will do what has just been discussed which is to evaluate the entire tree recursively and in turn create an array of values for every possible choice. Let's say there are 5 free columns, the array will be of length 5, and for each free column there will be a probabilistic

heuristic value assigned to it. This array of values will be returned and then another function will evaluate the array for the best move/column to choose based on which value in the array is the lowest or the "most negative". If there are multiple of the same heuristic value, then an array with these values will be generated and a random function will be used to randomly choose from these columns. This is necessary as during the early game there are typically a lot of positions that are based completely on randomicity and hence this feature is necessary. The pseudocode for the main AI algorithm is as follows:

```
get_moves_array(tree, board, free_columns, depth, player_num):
      if depth == 0:
            return [0] * len(free_columns)

      arr_moves = [0] * len(tree.children)

      for i = 0 to len(tree.children):
            board_copy = board[:]
            if tree.children[i].heuristic == 1 * player_num:
                  arr_moves[i] = 1 * player_num
            elif tree.children[i].heuristic == 0:
                  arr_moves[i] = 0
            arr_best_moves = get_moves_array(tree.children[i],
board_copy, free_columns, depth - 1, player_num * -1)
            arr_moves[i] += (sum(arr_best_moves) / len(free_columns))
      return arr_moves
```

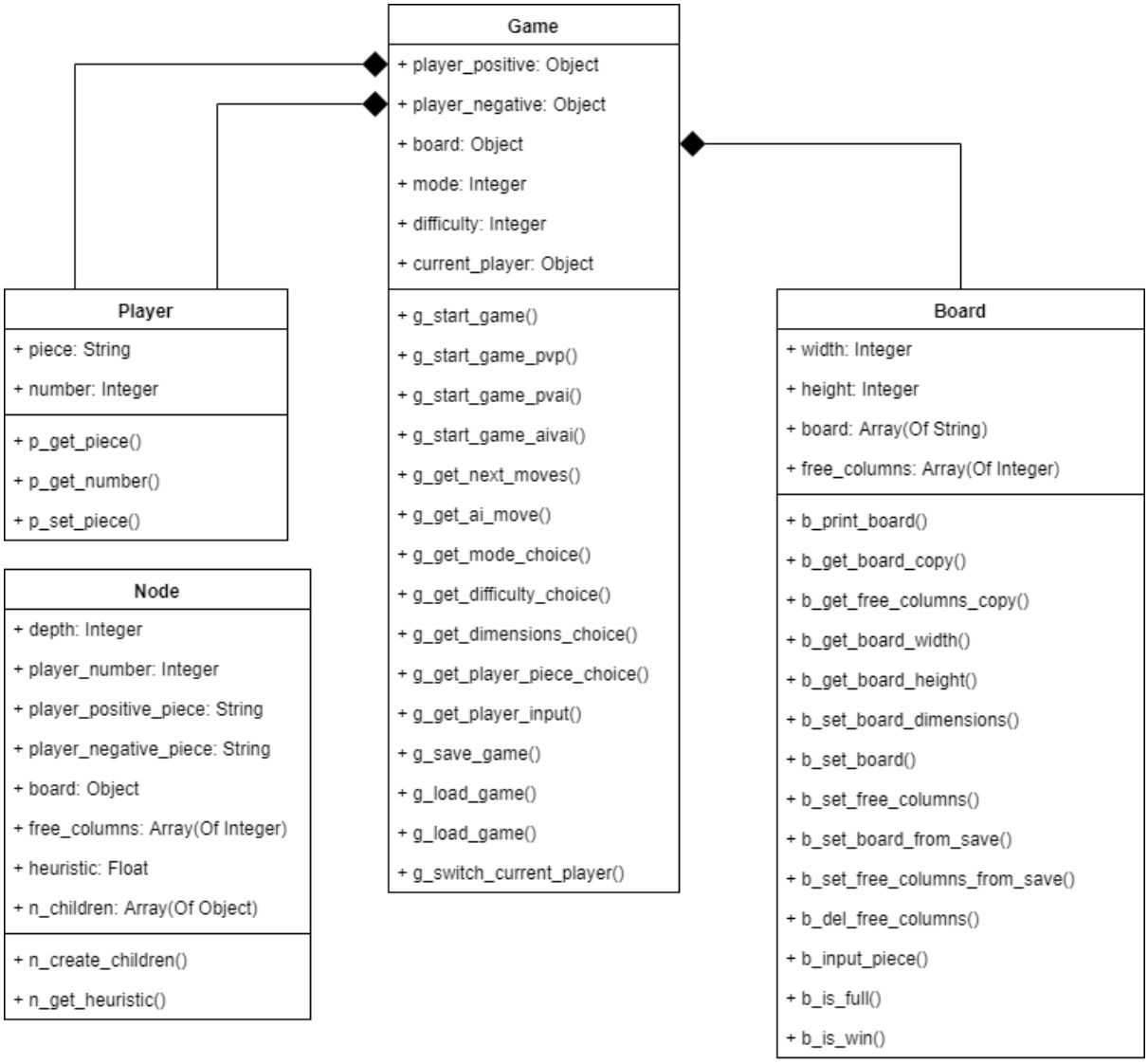
This function takes a tree (generated using the "Node" class), the board array, the free columns array, the depth that the algorithm is to go to (must be the same as the depth used when making the tree) as well as the player's number (either 1 or -1). If the depth is 0, an array full of 0's (neutral board heuristic) is made for the length of the number of free columns. This is to account for the scenario that a depth of 0 is passed into the array. Otherwise, the rest of the code will begin. An array called "arr_moves" is initially made based on the length of the tree's children array and it is filled with neutral heuristic values to start with. A loop is then started which loops through the tree's children and for each iteration a new copy of the board is created for when it is passed into the recursive algorithm. The code then uses some simple if statements to evaluate each child to see whether it is a 1, -1 or 0. This value is then appended onto the "arr_moves" array made earlier in the respective position of where that move is in the range of free columns. The recursion then begins, which does the same thing again until the depth of 0 is reached. Using the array(s) generated using this recursion, the "arr_moves" array has the average node value of each node appended onto it, again at the respective position(s) based on the current column that is being dealt with. The sum of all of the values is divided by the length of the array of the number of free columns. This is because the AI will be dynamic in a sense that it will always be able to work out the average no matter how many columns there are or how big/small the board is. At the end of all of the loop(s), the "arr_moves" array is returned. This contains the 7 averages, which (depending on how large the depth was set to) could be averages of averages of averages, etc. This will then be evaluated by the following pseudocode:

```
arr_next_moves = get_moves_array(my_node, board, free_columns,
depth, player_num):
```

```
best_heuristic = 1

for i = 0 to len(free_columns):
      if arr_next_moves[i] < best_heuristic:
            best_heuristic = arr_next_moves[i]

arr_best_heuristics = []

for i = 0 to len(arr_next_moves):
      if arr_next_moves[i] == best_heuristic:
            arr_best_heuristics.append(i)

ai_chosen_column =
free_columns[(random.choice(arr_best_heuristics))]
```

This piece of pseudocode first starts off by defining the array of next possible move heuristic averages by using the function created earlier. The "best_heuristic" is first defined as 1 as this is the worst possible case that the AI could choose deliberately because in the next part when it compares itself to the array values in the array "arr_next_moves" it needs to be as high as possible because it is looking for the lowest or "most negative" values in the array. A loop is started looping from 0 to the length of the "free_columns" array. It checks the "arr_next_moves" array to see if each value in the array is less than the current best heuristic value (which defaults to 1 as discussed earlier) and if it is then it sets "best_heuristic" variable to the heuristic average value at this point in the "arr_next_moves" array. An empty array is then created which will eventually store all of the instances of this particular heuristic. There is a good chance that the same heuristic value can be generated more than once. It is for this reason that the next part(s) of code occur. The array is filled with all of the columns where the "best_heuristic" value found previously is located. The program will then randomly choose from the free_columns where to place the piece. The randomicity will usually only happen at the very start and very end of games. This is because of the fact that during the early game when there are a lot of different options, there might not be any wins generated straight away. This means that unless you go unrealistically deep into the game tree, there is a level or randomness required when the AI chooses the column that it wants to place its piece into.

## 2.3 Class Diagram




This is the UML (Unified Modeling Language) diagram or class diagram for the program. There are four classes. The "Game" class uses composition and is composed of two "Player" classes as well as one "Board" class. The "Node" class is a class in its own right and is used in order to recursively create a game tree given a depth to search to. The classes use a very specific naming convention for the methods. Each method starts with the first letter of the class they are from (e.g. methods in the "Game" class start with the letter "g") followed by an underscore. Every word in a method is separated by an underscore. Examples of this are in every class, such as the "Board" class. Every method in the board class starts with "b_" and each word in the method name is separated by an underscore. Another convention used is that "getter" and "setter" methods follow the first underscore with the word "get" or "set" respectively. These naming conventions keep everything tidy, and easy for other programmers to understand and find the methods associated with specific classes.

# 3 Technical Solution

## 3.1 Preface

This is the technical solution. As designed in the design stage, the technical solution is made up of five files/modules. The program itself is made up of four classes and the "Game" class uses composition in order to compose itself out of two "Player" objects and one "Board" object.

## 3.2 Main (main.py)

```
from game import Game  # The "Game" class is imported from the "game.py" file/module


def main():  # Main subroutine that instantiates the game object and starts the game using the game object's "start game" subroutine
    # Game object instantiated
    g = Game()

    # Game started using the game object's "start game" subroutine
    g.g_start_game()


if __name__ == "__main__":  # Checks to see if this is the main file, and if it is then it runs the "main" subroutine
    main()
```

This is the "main" module. This is the module which is run when the user wishes to play/execute the game. It instantiates a "Game" object and starts the game using the game object's "start game" subroutine.

## 3.3 Game (game.py)

```
from player import Player  # The "Player" class is imported from the "player.py" file/module
from board import Board  # The "Board" class is imported from the "board.py" file/module
from tree import Node  # The "Node" class is imported from the "tree.py" file/module
from copy import deepcopy  # The "deepcopy" function is imported from the built in "copy.py" file/module
import random  # The built in "random.py" file/module is imported


class Game:  # Game class. Each game object will be composed of 2 players and a board object
    def __init__(self):  # Game object constructor
        self.player_positive = Player(1)  # Game object has a  positive player (positive player therefore the player number of 1 is passed into it)
        self.player_negative = Player(-1)  # Game object has a negative player object (negative player therefore the player number of -1 passed is into it)
        self.board = Board()  # Game object has a board object
        self.mode = None  # Game mode set to none. The mode will either be 1, 2 or 3 based on whether pvp (player versus (vs) player), pvai (player versus (vs) AI) or aivai (AI versus (vs) AI) is chosen)
        self.difficulty = None  # Game difficulty set to none. This will remain as none if the pvp mode is chosen, otherwise it will be set to either 2, 4 or 6 (easy, normal or hard).
        # The 2, 4 or 6 is the depth that the game tree will go to, the lower the number the easier the AI is to beat. The aivai mode will have a default difficulty of 6 (hard)
        self.current_player = self.player_positive  # The current player is set to the positive player object as the positive player is always the first to start

    def g_start_game(self):
```

```
        self.g_get_mode_choice()  # Get game mode. This sets the self.mode
attribute, which is then evaluated by the following if statement(s)
        self.g_get_dimensions_choice()  # Get board dimensions
        if self.mode == 1:  # Player versus Player
            self.g_get_player_piece_choice()  # Get positive player piece choice
(switches the current player at end of function to the negative player)
            self.g_get_player_piece_choice()  # Get negative player piece choice
(switches the current player at end of function back to the positive player)
            self.g_start_game_pvp()  # Begins the player versus player game mode
        elif self.mode == 2:  # Player vs AI (Artificial Intelligence)
            self.g_get_difficulty_choice()  # Get AI difficulty choice
            self.g_get_player_piece_choice()  # get player's choice for piece which
in turn sets the current player to the negative (ai) player
            self.g_switch_current_player()  # and so setting the current player
back to the positive (human) player
            self.g_start_game_pvai()  # Begins the player versus AI game mode
        elif self.mode == 3:  # AI vs AI
            self.g_start_game_aivai()  # Begins the AI versus AI game mode

    def g_start_game_pvp(self):  # This is the function for starting the player vs
player game mode. It is a function rather than a subroutine because it returns 0 if
a save game request is made
        while True:  # The game loop is started
            self.board.b_print_board()  # Board is printed using the board object's
"print board" subroutine
            print()  # Empty space printed below board
            print("It is {0}'s turn.".format(self.current_player.p_get_piece()))  #
The game prompts which player's turn it is. This is done by getting the current
player's piece using the player object's "get piece" function

            p_input_xy = self.g_get_player_input()  # A one dimensional array
containing the x and y values for the player's input is done by using the game's
"get player input" function (or a boolean value of 0 if the player chooses to save
the game)
            if p_input_xy != 0:  # This value is checked to see if it is equal to
0, if it is not then it performs the code below
                p_input_x = p_input_xy[0]  # A variable is set to the 0th value in
the xy position array (the x value)
                p_input_y = p_input_xy[1]  # A variable is set to the 1st value in
the xy position array (the y value)
                g_b_is_board_full = self.board.b_is_full()  # A variable is set to
True if the board is full using the board's "is board full" function, otherwise the
variable is assigned None
                if self.board.b_is_win(p_input_x, p_input_y) is True:
                    g_b_is_win = True
                    # Checks to see if either there is a winner using the board's
"win check" function (using the x and y values from before)
                    break
                elif g_b_is_board_full is True:
                    g_b_is_win = False
                    # It then checks to see if the board is full. If either of
these are true then the game loop is broken using the "break" statement.
                    # It is important that the board full check is done after the
win check, as you can have a win with the board full so the win check must be
prioritised
                    break
                self.g_switch_current_player()  # Or, if a winner is not found or
the board is not full, the current player is switched using the game's "switch
player" subroutine and the loop returns to the start
            else:  # If the user's input is equal to 0, then that means that the
user wishes to save the game, so the function ends by returning 0
                return 0
        self.board.b_print_board()  # If a winner is found, or the board is full,
the board is printed using the board's "print board" subroutine
        #  In order to decide what is outputted at the end of the game, a check is
made to see if the game was ended because of the game having a winner
        #  If the game has not had a winner, but the game loop has been broken,
then the game must have ended because the board is full, hence a tie is printed
```

```
        if g_b_is_win is True:
            print("Player {0} wins!".format(self.current_player.p_get_piece()))
        else:
            print("Game is a tie!")

    def g_start_game_pvai(self):  # This is the function for starting the player vs AI game mode. It is a function rather than a subroutine because it returns 0 if a save game request is made
        self.player_negative.p_set_piece("A")  # The AI's piece is set to the letter "A"
        while True:  # The game loop is started
            self.board.b_print_board()  # Board is printed using the board object's "print board" subroutine
            print()  # Empty space printed below board
            print("It is {0}'s turn.".format(self.player_positive.p_get_piece()))  # The game prompts the positive (human) player that it is their turn. This is done by getting the positive player's piece using the player object's "get piece" function

            p_input_xy = self.g_get_player_input()  # A one dimensional array containing the x and y values for the player's input is done by using the game's "get player input" function (or a boolean value of 0 if the player chooses to save the game)
            if p_input_xy != 0:  # This value is checked to see if it is equal to 0, if it is not then it performs the code within the if statement
                p_input_x = p_input_xy[0]  # A variable is set to the 0th value in the xy position array (the x value)
                p_input_y = p_input_xy[1]  # A variable is set to the 1st value in the xy position array (the y value)
                g_b_is_board_full = self.board.b_is_full()  # A variable is set to True if the board is full using the board's "is board full" function, otherwise the variable is assigned None
                if self.board.b_is_win(p_input_x, p_input_y) is True:
                    g_b_is_win = True
                    # Checks to see if either there is a winner using the board's "win check" function (using the x and y values from before)
                    break
                elif g_b_is_board_full is True:
                    g_b_is_win = False
                    # It then checks to see if the board is full. If either of these are true then the game loop is broken using the "break" statement.
                    # It is important that the board full check is done after the win check, as you can have a win with the board full so the win check must be prioritised
                    break
                self.g_switch_current_player()  # Or, if a winner is not found or the board is not full, the current player is switched using the game's "switch player" subroutine
                ai_input_x = self.g_get_ai_move()  # The x board value for the AI's position is assigned to a variable using the game object's "get AI move" function
                ai_input_y = self.board.b_input_piece(ai_input_x, self.current_player.p_get_piece())  # By passing the x value as well as the AI's piece into the board's "input piece" function, the y board value is assigned to a variable
                print("The AI has chosen column {0}.".format(ai_input_x + 1))  # For the user's benefit, if they missed the move or are not sure where the AI went, the column the AI has chosen is printed. 1 is added to the column as for the user the columns start with "1" whereas for the programming side it starts with 0
                print()  # Empty space printed below the AI's chosen column statement
                g_b_is_board_full = self.board.b_is_full()  # A variable is set to True if the board is full using the board's "is board full" function, otherwise the variable is assigned None
                if self.board.b_is_win(ai_input_y, ai_input_x) is True:
                    g_b_is_win = True
                    # Checks to see if either there is a winner using the board's "win check" function (using the x and y values from before)
                    break
```

```
                elif g_b_is_board_full is True:
                    g_b_is_win = False
                    # It then checks to see if the board is full. If either of
these are true then the game loop is broken using the "break" statement.
                    # It is important that the board full check is done after the
win check, as you can have a win with the board full so the win check must be
prioritised
                    break
                self.g_switch_current_player()  # Or, if a winner is not found or
the board is not full, the current player is switched using the game's "switch
player" subroutine
            else:  # If the user's input is equal to 0, then that means that the
user wishes to save the game, so the function ends by returning 0
                return 0
        self.board.b_print_board()  # If a winner is found, or the board is full,
the board is printed using the board's "print board" subroutine
        #  In order to decide what is outputted at the end of the game, a check is
made to see if the game was ended because of the game having a winner
        #  If the game has not had a winner, but the game loop has been broken,
then the game must have ended because the board is full, hence a tie is printed
        if g_b_is_win is True:
            print("Player {0} wins!".format(self.current_player.p_get_piece()))
        else:
            print("Game is a tie!")

    def g_start_game_aivai(self):  # This is the subroutine for starting the AI vs
AI game mode
        self.player_positive.p_set_piece("A")  # The positive AI's piece is set to
the letter "A"
        self.player_negative.p_set_piece("I")  # The negative AI's piece is set to
the letter "I"
        self.difficulty = 6  # The default difficulty for the AI is set to 6 (hard)
        while True:  # The game loop is started
            self.board.b_print_board()  # Board is printed using the board object's
"print board" subroutine
            print()  # Empty space printed below board
            ai_input_x = self.g_get_ai_move()  # The x board value for the AI's
position is assigned to a variable using the game object's "get AI move" function
            ai_input_y = self.board.b_input_piece(ai_input_x,
self.current_player.p_get_piece())  # By passing the x value as well as the AI's
piece into the board's "input piece" function, the y board value is assigned to a
variable
            g_b_is_board_full = self.board.b_is_full()  # A variable is set to True
if the board is full using the board's "is board full" function, otherwise the
variable is assigned None
            if self.board.b_is_win(ai_input_y, ai_input_x) is True:
                g_b_is_win = True
                # Checks to see if either there is a winner using the board's "win
check" function (using the x and y values from before)
                break
            elif g_b_is_board_full is True:
                g_b_is_win = False
                # It then checks to see if the board is full. If either of these
are true then the game loop is broken using the "break" statement.
                # It is important that the board full check is done after the win
check, as you can have a win with the board full so the win check must be
prioritised
                break
            self.g_switch_current_player()  # Or, if a winner is not found or the
board is not full, the current player is switched using the game's "switch player"
subroutine
        self.board.b_print_board()  # If a winner is found, or the board is full,
the board is printed using the board's "print board" subroutine
        #  In order to decide what is outputted at the end of the game, a check is
made to see if the game was ended because of the game having a winner
        #  If the game has not had a winner, but the game loop has been broken,
then the game must have ended because the board is full, hence a tie is printed
        if g_b_is_win is True:
```

```
            print("Player {0} wins!".format(self.current_player.p_get_piece()))
        else:
            print("Game is a tie!")

    def g_get_next_moves(self, node, cols, depth, player_number):  # This is a
recursive function that returns an array of the heuristically based values that the
AI can make next
        if depth == 0:
            return [0] * len(cols)  # If the depth equals 0 then an empty array
with heuristic values of 0 is created based on the length of the free columns array

        arr_moves = [0] * len(node.n_children)  # To start with, an empty array
with heuristic values of 0 is created and assigned to a variable based on the
length of the node object's children array

        for i in range(len(node.n_children)):  # For loop initiated based on the
length of the node object's children array
            arr_moves[i] = node.n_children[i].heuristic  # Each item in the
"arr_moves" variable/array is made equal to that of the node object's children
array's heuristic value. Each child is a node object in itself so it has a
heuristic attribute
            results = self.g_get_next_moves(node.n_children[i], cols, depth - 1,
player_number * -1)
            # This is where the recursion occurs with the results variable. It
calls itself until the depth equals 0 in order to reach the bottom of the game
tree. The "node" object passed each time is in itself the child of the node object.
            arr_moves[i] += (sum(results) / len(cols))  # Each of the values in the
"arr_moves" array then has the average value of the heuristic value that has been
returned added to itself.
        return arr_moves  # By the end of the function, this will return the
average value of the heuristics for each column in the board based on which column
will be the most beneficial and least detremental to the AI player

    def g_get_ai_move(self):  # This is a function that returns the best column for
the AI to choose based on the array generated using the game object's
"g_get_next_moves" function
        board_copy = deepcopy(self.board)  # A copy of the board object is created
so that it is not affected/changed when passed into the node object
        columns_copy = deepcopy(self.board.b_get_free_columns_copy())  # The same
is done for the columns array for the same reason
        g_node = Node(self.difficulty, self.current_player.p_get_number(),
self.player_positive.p_get_piece(), self.player_negative.p_get_piece(), board_copy,
columns_copy)
        # The node object is generated based on the difficulty the user has chosen,
as well as the player piece(s), the current player and the copy of the board and
free columns
        g_ai_move_next_moves = self.g_get_next_moves(g_node,
deepcopy(self.board.b_get_free_columns_copy()), self.difficulty,
self.current_player.p_get_number())
        # The array of the heuristic value for each of the free columns is then
generated by using the game object's "g_get_next_moves" function
        g_ai_move_best_value = 1  # The "best value" is initially set to 1. This is
the worst possible value that the heuristic function could return for the AI player
as -1 is the best. By setting it to 1, when compared to other heuristic values it
can change to (a) better one(s)

        for i in range(len(self.board.b_get_free_columns_copy())):
            if g_ai_move_next_moves[i] < g_ai_move_best_value:
                g_ai_move_best_value = g_ai_move_next_moves[i]
        # This for loop compares every heuristic value in the heuristic next moves
array ("g_ai_move_next_moves") to the current best heuristic value,
        # updating it if it is less than what is currently is (the AI is after the
lowest possible heuristic value)

        g_ai_move_best_moves = []
        for i in range(len(g_ai_move_next_moves)):
            if g_ai_move_next_moves[i] == g_ai_move_best_value:
                g_ai_move_best_moves.append(i)
```

```
        # This for loop assumes that there is a chance that there could be more
than one of the same best heuristic value in the "g_ai_move_next_moves" array.
        # It looks through the "g_ai_move_next_moves" array for any instances of
the "g_ai_move_best_value" heuristic value, and if it finds any then it
        # appends the respective column(s) to the "g_ai_move_best_moves" array.

        g_ai_col =
self.board.b_get_free_columns_copy()[(random.choice(g_ai_move_best_moves))]
        return g_ai_col  # A random function is then used to choose the best column
from any of the columns generated that have the same heuristic value.
        # this value is then returned and the function ends

    def g_get_mode_choice(self):  # This subroutine gets the game mode that the
user wishes to play
        print("_____")
        print("          Game Modes")
        print("_____")
        print("1. Player vs Player")
        print("2. Player vs AI")
        print("3. AI vs AI")
        print("Or")
        print("4. Load Save Game")
        print()
        # Prints out the game mode choices (1 = pvp, 2 = pvai, 3 = aivai or 4 =
load save game)
        while True:  # Starts a loop that will only be broken if they choose 1, 2,
3 or 4
            # The following is a "try except else" loop to ensure that they enter
an integer within the boundaries
            try:
                p_mode_input = int(input("Please choose a game mode (1, 2, 3 or 4):
"))  # It tries to get the user to input an integer
            except ValueError:
                print("Please enter a number.")  # If the user does not enter an
integer (which would otherwise generate an error) then they are prompted to enter a
number and returned to the "try" statement
            else:
                # If the user does enter an integer, then the following runs
                if 1 <= p_mode_input <= 3:  # If the integer that they inputted is
either 1, 2 or 3 then the mode is set to their input and the "try except else" loop
is broken out of
                    self.mode = p_mode_input
                    print()
                    break
                elif p_mode_input == 4:  # If the integer that they inputted is 4
then the game from the save file is loaded using the game's "load game" subroutine
and the "try except else" loop is broken out of
                    self.g_load_game()
                    break
                else:  # If the integer that they inputted is not 1, 2, 3 or 4 then
they are prompted to input one of them and the loop returns to the start
                    print("Please enter either 1, 2, 3 or 4.")

    def g_get_difficulty_choice(self):
        print("_____")
        print("         Difficulties")
        print("_____")
        print("1. Easy")
        print("2. Normal")
        print("3. Hard")
        print()
        # Prints out the difficulty choices (1 = easy, 2 = normal or 3 = hard)
        while True:  # Starts a loop that will only be broken if they choose 1, 2
or 3
            # The following is a "try except else" loop to ensure that they enter
an integer within the boundaries
            try:
                p_difficulty_input = int(input("Please choose a difficulty (1, 2 or
```

```
3): "))  # It tries to get the user to input an integer
            except ValueError:
                print("Please enter a number.")  # If the user does not enter an
integer (which would otherwise generate an error) then they are prompted to enter a
number and returned to the "try" statement
            else:
                # If the user does enter an integer, then the following runs
                if 1 <= p_difficulty_input <= 3:  # If the integer that they
inputted is either 1, 2 or 3 then the following code is run and ends with the "try
except else" loop being broken out of
                    if p_difficulty_input == 1:
                        self.difficulty = 2
                    elif p_difficulty_input == 2:
                        self.difficulty = 4
                    elif p_difficulty_input == 3:
                        self.difficulty = 6
                    print()
                    # This code sets the game's difficulty attribute to either 2
(for "Easy" (1)), 4 (for "Normal" (2)) or 6 (for "Hard" (3))
                    break
                else:  # If the integer that they inputted is not 1, 2 or 3 then
they are prompted to input one of them and the loop returns to the start
                    print("Please enter either 1, 2 or 3.")

    def g_get_dimensions_choice(self):
        print("_____")
        print("           Dimensions")
        print("_____")
        print()
        # Prints out just the title of this choice section
        while True:  # Starts a loop that will only be broken if they choose a
number (for the width) between or equal to 4 and 10
            # The following is a "try except else" loop to ensure that they enter
an integer within the boundaries
            try:
                p_dimensions_width_input = int(input("Please choose a board width
(between 4 and 10): "))  # It tries to get the user to input an integer
            except ValueError:
                print("Please enter a number.")  # If the user does not enter an
integer (which would otherwise generate an error) then they are prompted to enter a
number and returned to the "try" statement
            else:
                # If the user does enter an integer, then the following runs
                if 4 <= p_dimensions_width_input <= 10:  # If the integer that they
inputted is between or equal to 4 and 10 then the loop is broken out of
                    break
                else:  # If the integer that they inputted is not between or equal
to 4 and 10 then they are prompted to input an appropriate number and the loop
returns to the start
                    print("Please choose a number between 4 and 10.")
        while True:  # Starts a loop that will only be broken if they choose a
number (for the height) between or equal to 4 and 10
            try:
                p_dimensions_height_input = int(input("Please choose a board height
(between 4 and 10): "))  # It tries to get the user to input an integer
            except ValueError:
                print("Please enter a number.")  # If the user does not enter an
integer (which would otherwise generate an error) then they are prompted to enter a
number and returned to the "try" statement
            else:
                # If the user does enter an integer, then the following runs
                if 4 <= p_dimensions_height_input <= 10:  # If the integer that
they inputted is between or equal to 4 and 10 then the loop is broken out of
                    break
                else:
                    print("Please choose a number between 4 and 10.")  # If the
integer that they inputted is not between or equal to 4 and 10 then they are
prompted to input an appropriate number and the loop returns to the start
```

```
        # Once both loops have successfully acquired a width and height, the board
has its width and height set to these values using the board object's "set board
dimensions" subroutine
        self.board.b_set_board_dimensions(p_dimensions_width_input,
p_dimensions_height_input)

    def g_get_player_piece_choice(self):
        print("_____")
        print("           Colours")
        print("_____")
        print("Red (R)")
        print("Orange(O)")
        print("Yellow (Y)")
        print("Green (G)")
        print("Blue (B)")
        print()
        # Prints out the piece/colour choices (R = Red, O = Orange, Y = Yellow, G =
Green and B = Blue)
        while True:  # Starts a loop that will only be broken if they choose an
available letter
            # The following is a "try except else" loop to ensure that they enter a
letter within the array of available letters
            try:
                p_colour_input = str(input("Please choose a colour for player {0}
(R, O, Y, G or B): ".format(self.current_player.p_get_number())))  # It tries to
get the user to input a string
            except ValueError:
                print("Please enter a letter.")  # If the user does not enter a
string (which would otherwise generate an error) then they are prompted to enter a
string/"letter" and returned to the "try" statement
            else:
                # If the user does enter an string/character, then the following
runs
                if p_colour_input.upper() in ["R", "O", "Y", "G", "B"]:  # If the
uppercase version of the string/character that they inputted is within the array of
letters then the current player's piece is set to the piece that they chose and
then the loop is broken out of
                    self.current_player.p_set_piece(p_colour_input.upper())
                    print()
                    break
        self.g_switch_current_player()  # The current player is switched using the
game's "switch current player" subroutine

    def g_get_player_input(self):  # This function returns a one dimensional array
with the x and y board position of the piece after the player chooses the column
they would like to place their piece in
        b_width = self.board.b_get_board_width()  # The width of the board is
stored in a variable using the board object's "get board width" function. This
width will be used to get the user's column input
        p_input_xy = []  # An empty array is defined which will be used to store
the x and y board values based on the column that the user chooses
        while True:  # Starts a loop that will only be broken if they choose an
available column
            # The following is a "try except else" loop to ensure that they enter
an integer that is one of the free columns
            try:
                p_board_input = int(input("Please enter a column from 1 to {0} or
enter '0' to save and exit: ".format(b_width)))  # It tries to get the user to
input an integer
            except ValueError:
                print("Please enter a number.")  # If the user does not enter an
integer (which would otherwise generate an error) then they are prompted to enter a
number and returned to the "try" statement
            else:
                if 1 <= p_board_input <= b_width:  # It then checks to make sure
that the user has inputted a number that is within the range of the available
columns
                    p_row_chosen = self.board.b_input_piece(p_board_input - 1,
```

```
self.current_player.p_get_piece())  # To find out either the row/x value that the
user has chosen, a variable is set to the result of the board object's "input
piece" function. Keep in mind this function returns 0 if the column is full
                    if p_row_chosen is not False:  # If the row chosen variable is
not False (which means that the column the user has chosen is not full) then the
code within this if statement is run
                        p_input_xy.append(p_row_chosen)  # The 0th value in the xy
array is set to the x position that the user's input causes
                        p_input_xy.append(p_board_input - 1)  # The 1st value of
the xy array is set to the user's input minus 1 (because if they input 1 that is
actually the 0th column)
                        return p_input_xy  # The xy array is returned
                elif p_board_input == 0:  # If the user inputs 0 then the game
object's "save game" subroutine is run and the player input functions returns 0
                    self.g_save_game()
                    return 0
                else:  # If the user does not input a number that is in the range
of total columns that the board has then the user is prompted to enter a number
that is within the range
                    print("The number {0} is out of the
range.".format(p_board_input))

    def g_save_game(self):  # This subroutine saves the board array, free columns
array and game data to three separate text files
        # Save Board
        s_b_save_file = open("s_b_save.txt", "w")  # This opens (or creates if it
does not exist) a save file called "s_b_save.txt" in write mode
        for line in self.board.b_get_board_copy():  # This for loop loops through
every line in the board array by getting a copy of the board state using the board
object's "get board copy" function
            for line_item in line:  # This for loop loops through every
character/item in the board array on each of the lines
                s_b_save_file.write(line_item)  # Each of the items on each of the
lines it is written to the save file
            s_b_save_file.write("\n")  # After each line in the board array is
written to the save file, a new line is written to the save file so that the save
file eventually has the same structure as the actual board array itself
        s_b_save_file.close()  # The save file is then closed so that the changes
will take effect

        # Save Free Columns
        s_c_save_file = open("s_c_save.txt", "w")  # This opens (or creates if it
does not exist) a save file called "s_c_save.txt" in write mode
        for item in self.board.b_get_free_columns_copy():  # This for loop loops
through every item in the column array by getting a copy of the free columns array
using the board object's "get free columns copy" function
            s_c_save_file.write(str(item))  # Each of the items in the free columns
array is written to the save file in string form as you cannot write integers
            s_c_save_file.write("\n")  # Every free column number is written to a
new line
        s_c_save_file.close()  # The sale file is then closed so that the changes
will take effect

        # Save Game Data
        s_data__save_file = open("s_data_save.txt", "w")  # This opens (or creates
if it does not exist) a save file called "s_data_save.txt" in write mode
        # The following writes (on a new line) all of the data to do with the
current game that is to be saved. This data is the current player's number, the
        # positive player's piece, the negative player's piece, the board width,
the board height and the game difficulty. By saving all of this metadata for
        # the game, it means that these pieces of data can be set when the game is
loaded and in turn meaning that the game will be exactly the same as it was
        # before it was saved. In terms of the array that the "s_data_save.txt"
will be written to, array[0] = current player number, array[1] = positive player
        # piece, array[2] = negative player piece, array[3] = board width, array[4]
= board height and array[5] = game difficulty.
        s_data__save_file.write(str(self.current_player.p_get_number()) + "\n"  #
current player [0]
```

```
                                + self.player_positive.p_get_piece() + "\n"  #
positive player piece [1]
                                + self.player_negative.p_get_piece() + "\n"  #
negative player piece [2]
                                + str(self.board.b_get_board_width()) + "\n"  #
board width [3]
                                + str(self.board.b_get_board_height()) + "\n"  #
board height [4]
                                + str(self.difficulty))  # game difficulty [5]
        s_data__save_file.close()  # The sale file is then closed so that the
changes will take effect

        print("_____________________________________")
        print("          Game Saved!")
        print("_____________________________________")
        # The user is prompted that the game has been saved.

    def g_load_game(self):  # This subroutine loads the board array, free columns
array and game data from three separate text files

        # Load Game Data:
        l_data_save_file = open("s_data_save.txt", "r")  # This opens a save file
called "s_data_save.txt" in read mode
        l_data = []  # An empty array is created which will store the game data
        for line in l_data_save_file:
            l_data.append(line)  # Each line in the save file is appended to the
"l_data" array
        # Current player number [0]: the "strip()" subroutine is used in order to
remove any whitespace or non breaking spaces from the data
        # If the current player number is 1 then the current player is the positive
player, otherwise if the current player number is -1 then the current player is the
negative player
        if l_data[0].strip() == "1":
            self.current_player = self.player_positive
        elif l_data[0].strip() == "-1":
            self.current_player = self.player_negative
        # Positive player piece [1]: the "strip()" subroutine is used in order to
remove and whitespace or non breaking spaces from the data. The positive player's
piece is set to this piece using the "set piece" subroutine
        self.player_positive.p_set_piece(l_data[1].strip())
        # Negative player piece [2]: the "strip()" subroutine is used in order to
remove and whitespace or non breaking spaces from the data. The negative player's
piece is set to this piece using the "set piece" subroutine
        self.player_negative.p_set_piece(l_data[2].strip())
        # Board width [3] and board height [4]: these values are converted to
integers, and the board is set to these dimensions using the board object's "set
board dimensions" subroutine
        self.board.b_set_board_dimensions(int(l_data[3]), int(l_data[4]))
        # Game difficulty [5]: if the difficulty saved "None" then the difficulty
is set to None (the non string version), otherwise it is set to the integer value
of what was saved
        if l_data[5] == "None":
            self.difficulty = None
        else:
            self.difficulty = int(l_data[5])

        # Load Board
        l_b_save_file = open("s_b_save.txt", "r")  # This opens a save file called
"s_b_save.txt" in read mode
        l_board = []  # An emtpy array is created which will store the board state
array
        row = []  # An empty array is created which will store the current row that
the for loop is on
        for line in l_b_save_file:
            for i in range(0, self.board.b_get_board_width()):  # The board width
is now available for use based on the data acquired from loading the save data from
previously
                row.append(line[i])  # For each row in the board save file, each
```

```
value from the line is appended to the "row" array
            l_board.append(row)  # Each of these row arrays are then appended to
the "board" array
            row = []  # The "row" array is then set equal to nothing so that it can
be append to with the next row of data in the save file
        self.board.b_set_board_from_save(l_board)  # After the entire board array
has been filled from the text file, the board is set to the array using the board
object's "set board from save" subroutine

        # Load Free Columns
        l_c_save_file = open("s_c_save.txt", "r")  # This opens a save file called
"s_c_save.txt" in read mode
        l_free_columns = []  # An emtpy array is created which will store the free
columns array
        for line in l_c_save_file:
            l_free_columns.append(int(line.strip()))  # For each line in the free
columns save file, the integer value of the data is stripped and appended to the
free columns array
        self.board.b_set_free_columns_from_save(l_free_columns)  # After the entire
free columns array as been filled from the text file, the free columns array is set
to the array using the board object's "set columns from save" subroutine

        # Begin Loaded Game
        if self.difficulty is None:
            self.g_start_game_pvp()  # If the difficulty (acquired from the save
file) is None then this means that the game mode must have been the player vs
player game so the game object's "g_start_game_pvp" subroutine is run
        else:
            self.g_start_game_pvai()  # If the difficulty (acquired from the save
file) is not None then this means that the game mode must have been the player vs
AI game so the game object's "g_start_game_pvai" subroutine is run

    def g_switch_current_player(self):  # This subroutine changes the current
player object to the opposite to what it currently is
        self.current_player = self.player_positive if self.current_player ==
self.player_negative else self.player_negative
```

This is the "game" module. The game module contains only a "Game" class. This class contains all of the attributes and methods to do with a game. Each "Game" object uses composition in order to compose itself out of two "Player" objects and one "Board" object. This is because for any game of connect 4, two players and one board is required. The "Game" object's "start game" subroutine is an important method to take note of due to the fact that it starts the entire game.

## 3.4 Player (player.py)

```
class Player:  # Player class. Two player objects will be instantiated when
composing a game object
    def __init__(self, p_number):  # Player object constructor
        self.piece = None  # Each player object has a piece (a character such as
"R" or "Y") to represent the colour/piece that they are using
        self.number = p_number  # Each player object has a player number, this will
be either 1 (positive player) or -1 (negative player). The player number is used by
the AI/tree algorithm(s) for maximising and minimising

    def p_get_piece(self):  # This is the getter function used to return the player
object's piece, as a pose to accessing the attribute directly
        return self.piece

    def p_get_number(self):  # This is the getter function used to return the
player object's number, as a pose to accessing the attribute directly
        return self.number

    def p_set_piece(self, p_piece):  # This is the setter subroutine used to set
the player object's piece, as a pose to accessing the attribute directly
        self.piece = p_piece
```

This is the "player" module. The player module contains only a "Player" class. This module is used by the "Game" class twice in order to assign two "Player" objects to the "Game" class. The "Player" class contains all of the attributes and methods to do with a player.

## 3.5 Board (board.py)

```
class Board:  # Board class. One board object will be instantiated when composing a game object
    def __init__(self):  # Board object constructor
        self.width = None  # Each board object has a width. This along with the height will be used to construct/set/create the board array. Initially set to None as this will be determined by the user
        self.height = None  # Each board object has a height. This along with the width will be used to construct/set/create the board array. Initially set to None as this will be determined by the user
        self.board = []  # Each board object has the board array itself. This will be a two dimensional array. It is initially set to be empty as the array will be dynamically created based on the width and height attributes
        self.free_columns = []  # Each board object has a free columns array. This will be a one dimensional array containing the columns in the board that are not full e.g. (0, 1, 2, 4, etc.) It is initially set to be empty as the array will be dynamically created based on the width and height attributes

    # Print Functions/Subroutines

    def b_print_board(self):  # This function is used to print the board object's board array/attribute.
        for i in range(0, len(self.board[0])):  # It initially prints the column numbers (with 1 added to each column because the first for the computer is 0 but the first for the user is 1)
            print("{0}  ".format(i + 1), end="")
        print()
        for j in range(0, len(self.board)):  # This loop prints each row of the board object's board array/attribute, character by character joined by spaces between
            print("  ".join(self.board[j]))

    # Getter functions

    def b_get_board_copy(self):  # This function returns a copy of the board object's board array/attribute using Python's [:] which returns a copy of an array rather than a reference
        return self.board[:]

    def b_get_free_columns_copy(self):  # This function returns a copy of the board object's free columns array/attribute using Python's [:] which returns a copy of an array rather than a reference
        return self.free_columns[:]

    def b_get_board_width(self):  # This function returns the width attribute of the board object
        return self.width

    def b_get_board_height(self):  # This function returns the height attribute of the board object
        return self.height

    # Setter Subroutines

    def b_set_board_dimensions(self, b_width, b_height):  # This subroutine sets the board object's board attribute/array based on a width and height passed into it
        self.width = b_width  # Stores the width
        self.height = b_height  # Stores the height
        self.b_set_board()  # Uses the width and height to generate/set the board array using the board object's "set board" subroutine
```

```
        self.b_set_free_columns()  # Uses the width and height to generate/set the
free columns array using the board object's "set free columns" subroutine

    def b_set_board(self):  # This subroutine is called when the width and height
have been passed into the board object and fills the board array
        self.board = []  # The board object's board array/attribute is set first to
an empty array
        row = []  # A "row" variable is set also to an empty array
        for i in range(0, self.width):
            row.append("-")  # A "width" number of the character "-" are appended
to the "row" array
        for i in range(0, self.height):
            self.board.append(row[:])  # These rows are then appended a "height"
number of times to the board array in order to create the empty board array

    def b_set_free_columns(self):  # This subroutine initially sets the free
columns array based on the width of the board
        self.free_columns = []
        for i in range(0, self.width):
            self.free_columns.append(i)  # Based on the width, the value(s) of all
of the columns will be appended to the board object's free columns array/attribute

    def b_set_board_from_save(self, l_board):  # This subroutine sets the board
object's board array/attribute equal to that of a board array that is passed into
it. This is to be used by the load function/subroutine
        self.board = l_board

    def b_set_free_columns_from_save(self, l_free_columns):  # This subroutine sets
the board object's free columns array/attribute equal to that of a free columns
array that is passed into it. This is to be used by the load function/subroutine
        self.free_columns = l_free_columns

    # Miscellaneous Functions/Subroutines

    def b_del_free_columns(self, del_piece_column):  # This subroutine is for
deleting a specific column from the board object's free columns array/attribute.
This is done by passing the column that should be deleted as a parameter into the
subroutine
        del self.free_columns[self.free_columns.index(del_piece_column)]  # The
board object's free columns array/attribute has the value deleted at the index
where the column that is passed occurs

    def b_input_piece(self, input_piece_column, piece):  # This function performs
two actions. First of all it inputs/inserts a piece into the board at the right row
based on the column entered. It also, importantly, returns the row that the piece
is inputted in, or False/0 if the column is full
        for i in range(len(self.board) - 1, -1, -1):  # This is a loop from the
total height of the board downwards to 0 (or -1 in Python as it leaves out the last
iteration), with a negative 1 step, meaning it increments "backwards". It goes
bottom up e.g. 5, 4, 3, 2, 1, 0 (with a height of 6)
            if self.board[i][input_piece_column] == "-":  # If the current piece is
empty then following within the if statement occurs
                self.board[i][input_piece_column] = piece  # If the current piece
is empty then it will set it equal to the piece that has been passed as a
parameter, otherwise it will go to the next row (above)
                if i == 0:
                    self.b_del_free_columns(input_piece_column)  # If i is equal to
0 then this means that that is the last piece that can be placed in that column, so
the column is deleted from the board object's free columns array/attribute using
the "delete free columns" subroutine
                return i  # Finally, the row that the piece has been placed into is
returned and in turn the function ends
        print("Column {0} is full.".format(input_piece_column + 1))  # If a piece
cannot be placed into the column then the user is prompted that the column is full
        return False  # False/0 is returned if this is the case and in turn the
function ends

    def b_is_full(self):  # This function is for finding out if the board object's
```

```
board array/attribute is full of pieces and has no blank areas ("-")
        for i in range(0, self.height):
            for j in range(0, self.width):
                if self.board[i][j] == "-":  # Two for loops looping through the
height and the width respectively are used to check if each piece on each row is
equal to "-"
                    return 0  # If a piece is equal to "-" then the board is not
full, so False/0 is returned
        return 1  # Otherwise, the board is full, so True/1 is returned

    def b_is_win(self, b_board_x, b_board_y):  # This function works out whether
the pieces around a certain piece (usually the most recently placed piece) causes a
win (either vertically down, horizontally, diagonally positively or diagonally
negatively).
        # board_x = 0, 1, 2, 3, 4, 5 (Down Left)   |   board_y = 0, 1, 2, 3, 4, 5,
6 (Across Top)
        board_x_max = len(self.board) - 1  # board x values (0, 1, 2, 3, 4, 5) if
len(board) = 6
        board_y_max = len(self.board[0]) - 1  # board y values (0, 1, 2, 3, 4, 5,
6) if len(board[0]) = 7

        # Down (|)  This checks the currently placed piece against the three other
pieces below itself (four in total)
        pairs = 0  # Sets the pairs value equal to 0. There must be three pairs for
there to be four pieces in a row
        if b_board_x <= len(self.board) - 4:  # Checks to see if this piece is four
or more above the bottom of the board (as a win cannot be caused down if this is
not true)
            for i_down in range(1, 4):  # Loops only through four pieces (including
itself) as it cannot cause a win any further below itself than four
                if self.board[b_board_x][b_board_y] == self.board[b_board_x +
i_down][b_board_y]:  # Checks to see if the piece passed as a parameter is equal to
the piece below itself, then the one below that, etc.
                    pairs += 1  # Each time it finds a pair, the "pairs" variable
is incremented by 1
                    if pairs == 3:  # If the pairs variable is equal to three
(three pairs = four in a row) then the function returns True
                        return True
                else:  # If the piece is not equal to a piece below itself then the
pairs variable is set back to 0 and the loop is broken
                    pairs = 0
                    break

        # Across (-)
        pairs = 0  # Sets the pairs value equal to 0. There must be three pairs for
there to be four pieces in a row
        for i_across in range(0 - b_board_y, board_y_max - b_board_y):  # Starts a
loop from 0 to the maximum width of the board
            if self.board[b_board_x][b_board_y + i_across] != "-":  # Checks to see
if the current piece is not equal to "-" (e.g. there is a piece there so it is
worth checking)
                if self.board[b_board_x][b_board_y + i_across] ==
self.board[b_board_x][b_board_y + i_across + 1]:  # Checks to see if the current
piece is equal to the piece to the right of itself (by adding 1 to the board y
value)
                    pairs += 1
                    if pairs == 3:
                        return True  # If it is then the pairs value is incremented
by 1, and if the pairs variable reaches 3 then the function returns True
                else:
                    pairs = 0  # Otherwise if it is not then the pairs value is set
to 0 and the loop continues

        # Diagonal Positive (/)
        pairs = 0  # Sets the pairs value equal to 0. There must be three pairs for
there to be four pieces in a row
        start_x_pos = 0  # The starting x position for the positive check is
initially set to 0 as this will be worked out shortly
```

```
        start_y_pos = 0  # The starting y position for the positive check is
initially set to 0 as this will be worked out shortly
        if b_board_x + b_board_y <= board_x_max:  # If the sum of the board x and
board y values passed is less than or equal to the maximum height of the board
then:
            start_x_pos = b_board_x + b_board_y  # The starting x position is set
to the sum of the board x and board y values passed
            start_y_pos = 0  # And the starting y position is set to 0 if this is
also the case
        elif b_board_x + b_board_y > board_x_max:  # If the sum of the board x and
board y values passed is greater than the maximum height of the board then:
            start_x_pos = board_x_max  # The starting x position is set to the
maximum height of the board
            start_y_pos = (b_board_x + b_board_y) - board_x_max  # And the starting
y position is set ot the sum of the board x and board y values passed minus the
maximum height of the board
        num_to_check_pos = 0  # The number of pieces to check diagonally positive
is initially set to 0 as this will be worked out shortly
        if start_y_pos == 0:
            num_to_check_pos = start_x_pos  # If the starting y position calculated
is equal to 0 then the number to check is equal to the starting x position
        elif start_y_pos > 0:
            num_to_check_pos = (start_x_pos - start_y_pos) + 1  # Or if the
starting y position is greater than 0 then the number to check is equal to the
starting x position minus the starting y position with 1 added
        for i_diagonal_pos in range(0, num_to_check_pos):  # For loop starting
looping through the number of positions there are to check diagonally positive
            if self.board[start_x_pos - i_diagonal_pos][start_y_pos +
i_diagonal_pos] != "-":  # First checks that the piece is not empty before doing
any comparisons
                if self.board[start_x_pos - i_diagonal_pos][start_y_pos +
i_diagonal_pos] == self.board[start_x_pos - i_diagonal_pos - 1][start_y_pos +
i_diagonal_pos + 1]:
                    pairs += 1
                    if pairs == 3:
                        return True  # If a piece is equal to a piece up one space
and to the right one space to itself (diagonally positive) then the pairs variable
is incremented by 1, and if the pairs variable reaches 3 then the function returns
True
                else:
                    pairs = 0  # Otherwise if it is not then the pairs value is set
to 0 and the loop continues

        # Diagonal Negative (\)
        pairs = 0  # Sets the pairs value equal to 0. There must be three pairs for
there to be four pieces in a row
        start_x_neg = 0  # The starting x position for the positive check is
initially set to 0 as this will be worked out shortly
        start_y_neg = 0  # The starting y position for the positive check is
initially set to 0 as this will be worked out shortly
        if b_board_x >= b_board_y:  # If the board x value passed is greater than
or equal to the board y value passed then:
            start_x_neg = b_board_x - b_board_y  # The starting x position is set
to the board x value passed minus the board y value passed
            start_y_neg = 0  # And the starting y position is set to 0 if this is
also the case
        elif b_board_x < b_board_y:  # If the board x value passed is less than the
board y value passed then:
            start_x_neg = 0  # The starting x position is set equal to 0
            start_y_neg = b_board_y - b_board_x  # And the starting y position is
set to the board y value passed minus the board x value passed
        num_to_check_neg = 0  # The number of pieces to check diagonally positive
is initially set to 0 as this will be worked out shortly
        if start_y_neg == 0:
            num_to_check_neg = board_x_max - start_x_neg  # If the starting y
position calculated is equal to 0 then the number to check is equal to the maximum
height of the board minus the starting x position
        elif start_y_neg > 0:
```

```
            num_to_check_neg = board_y_max - start_y_neg  # Or if the starting y
position is greater than 0 then the number to check is equal to the board y value
passed minus the starting y position
        for i_diagonal_neg in range(0, num_to_check_neg): # For loop starting
looping through the number of positions there are to check diagonally negative
            if self.board[start_x_neg + i_diagonal_neg][start_y_neg +
i_diagonal_neg] != "-":  # First checks that the piece is not empty before doing
any comparisons
                if self.board[start_x_neg + i_diagonal_neg][start_y_neg +
i_diagonal_neg] == self.board[start_x_neg + i_diagonal_neg + 1][start_y_neg +
i_diagonal_neg + 1]:
                    pairs += 1
                    if pairs == 3:
                        return True  # If a piece is equal to a piece down one
space and to the right one space to itself (diagonally negative) then the pairs
variable is incremented by 1, and if the pairs variable reaches 3 then the function
returns True
                else:
                    pairs = 0  # Otherwise if it is not then the pairs value is set
to 0 and the loop continues
```

This is the "board" module. The board module contains only a "Board" class. This module is used by the "Game" class once in order to assign a "Board" object to the "Game" class. The "Board" class contains all of the attributes and methods to do with a board.

## 3.6 Tree (tree.py)

```
class Node:  # Node class. Multiple node objects will be contained within one
another recursively in order to create a tree. This tree will be a game tree for
the AI to use
    def __init__(self, g_depth, p_number, p_player_positive_piece,
p_player_negative_piece, board, b_free_columns_copy, n_heuristic=0):
        self.depth = g_depth  # The depth that the tree should go to. The greater
the tree depth, the greater the difficulty
        self.player_number = p_number  # The number of the player that wishes
        self.player_positive_piece = p_player_positive_piece  # The piece for the
positive player. This will be used when testing board states
        self.player_negative_piece = p_player_negative_piece  # The piece for the
negative player. This will be used when testing board states
        self.board = board  # The current board object
        self.free_columns = b_free_columns_copy  # The current free columns array
        self.heuristic = n_heuristic  # The heuristic value of this node
        self.n_children = []  # The array of children for this node
        self.n_create_children()  # Creates the children for the node as soon as
the node is created using the node object's "create children" subroutine

    def n_create_children(self):  # This subroutine creates the children for the
node object(s). The children of the node objects are node objects in themselves
        if self.depth > 0:  # Checks to make sure that the depth has not reached 0
            for board_y in self.free_columns:  # Loops through the free columns on
the board
                for board_x in range(len(self.board.board) - 1, -1, -1):  # Loops
"backwards" with a -1 step through the board height
                    if self.board.board[board_x][board_y] == "-":  # Checks to see
if this row has an empty piece/value. If it does then:
                        board_copy = self.board  # The board object is copied to a
variable
                        free_columns_copy = self.free_columns[:]  # The free
columns array is copied to a variable
                        board_copy.board[board_x][board_y] =
self.player_positive_piece if self.player_number is 1 else
self.player_negative_piece  # Sets the board equal to the current player's piece
based on whether the player number passed is a 1 or -1 where the empty space is
found
                        chosen_x = board_x  # The chosen x value is stored in a
variable
```

```
                        chosen_y = board_y  # The chosen y value is stored in a
variable
                        if board_x == 0:
                            del free_columns_copy[free_columns_copy.index(board_y)]
# If the board x value is equal to 0 then this means that the column is full so the
columns copy array has this column removed
                        break
                self.n_children.append(Node(self.depth - 1, -self.player_number,
self.player_positive_piece, self.player_negative_piece, board_copy,
free_columns_copy, self.n_get_heuristic(chosen_x, chosen_y)))
                # For each of the free columns, a child is appended to the children
array. This child is a node object in itself, but importantly, with the depth
decremented by 1, and the player number
                # set to the negative of what it currently is. The board that was
generated from this move as well as the free columns array is also passed into this
node object. Finally, the heuristic value
                # for the node is generated using the node object's "get heuristic"
function
                self.board.board[chosen_x][chosen_y] = "-"  # Finally, the board
piece that was changed is set back to an empty value so that the loop can continue
and try another piece

    def n_get_heuristic(self, x, y):  # This function returns a heuristic value
based on what the piece that has been placed causes. It takes the x and y positions
of where the piece was placed
        win_check = self.board.b_is_win(x, y)
        if win_check is True:
            return 1 * self.player_number  # If the piece caused a win (either for
the positive or negative player) then the heuristic value of 1 multiplied by the
player's number is returned
        else:
            return 0  # Or if the piece did not cause a win then the heuristic
value of 0 is returned
```

This is the "tree" module. The tree module contains only a "Node" class. This module is used by the AI methods in the "Game" class in order to evaluate a game tree for the best position for the AI to take.

# 4 Testing

## 4.1 Test Plan

| ID | Description | Type | Input | Expected Result | Actual Result |
|---|---|---|---|---|---|
| | | | **Menu Navigation Testing** | | |
| 1 | Test that the game mode selection menu appears. | N/A | Run the program. | The game mode selection menu should appear. | |
| 2 | Test that option "1" (PVP) works. | Normal | Enter "1" into the game mode selection menu. | It should be accepted, and the dimensions selection menu should appear. | |
| 3 | Test that option "2" (PVAI) works. | Normal | Enter "2" into the game mode selection menu. | It should be accepted, and the dimensions selection menu should appear. | |
| 4 | Test that option "3" (AIVAI) works. | Normal | Enter "3" into the game mode selection menu. | It should be accepted, and the dimensions selection menu should appear. | |
| 5 | Test that option "4" (load game) works (with an existing save file). | Normal | Enter "4" into the game mode selection menu whilst there are existing save files. | It should be accepted, and the game should start based on the data from the save file(s). | |
| 6 | Test that option "4" (load game) works (without an existing save file). | Normal | Enter "4" into the game mode selection menu whilst there are not existing save files. | The input should be accepted but the game should prompt that the save file(s) have not been found and ask for another input. | |
| 7 | Test entering an erroneous number for the game mode choice. | Erroneous | Enter "7" into the game mode selection menu. | The input should be rejected, and the user should be prompted to enter either 1, 2, 3 or 4. | |
| 8 | Test a normal "width" when prompted for a dimensions input. | Normal | Enter "7" for the width. | The input should be accepted. | |

| | | | | | |
|---|---|---|---|---|---|
| 9 | Test an extreme “width” when prompted for a dimensions input. | Extreme | Enter “10” for the width. | The input should be accepted. | |
| 10 | Test an erroneous “width” when prompted for a dimensions input. | Erroneous | Enter “11” for the width. | The input should be rejected, and another input for the width should be asked for. | |
| 11 | Test a normal “height” when prompted for a dimensions input. | Normal | Enter “7” for the height. | The input should be accepted. | |
| 12 | Test an extreme “height” when prompted for a dimensions input. | Extreme | Enter “10” for the height. | The input should be accepted. | |
| 13 | Test an erroneous “height” when prompted for a dimensions input. | Erroneous | Enter “11” for the height. | The input should be rejected, and another input for the height should be asked for. | |
| 14 | Test a normal “colour” when prompted for a player colour input. | Normal | Enter “R” for the player colour. | The input should be accepted. | |
| 15 | Test an erroneous “colour” when prompted for a player colour input. | Erroneous | Enter “7” for the player colour. | The input should be rejected, and another input for the player colour should be asked for. | |
| 16 | Test a normal “difficulty” when prompted for a difficulty input. | Normal | Enter “2” for the difficulty. | The input should be accepted. | |
| 17 | Test an extreme “difficulty” when prompted for a difficulty input. | Extreme | Enter “3” for the difficulty. | The input should be accepted. | |
| 18 | Test an erroneous “difficulty” when prompted for a difficulty input. | Erroneous | Enter “7” for the difficulty. | The input should be rejected, and another input for the difficulty should be asked for. | |
| **Game Testing** | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| 19 | Test that the board appears when a game is started. | N/A | Start a PVP game, set the board width to 7 and height to 6 (player colours are insignificant). | The board should appear, and the width and height should be based on the dimensions input to start with. | |
| 20 | Test that, after choosing the PVP game mode, both players can play the game and the game alternates correctly between the two players. | N/A | Start a PVP game, (the board width, height and player colours are insignificant) and play ten total moves (five per player). | Ensure that each player can input pieces into the game board correctly, and that after each accepted input the current player alternates. | |
| 21 | Test that, after choosing the PVAI game mode, the human player can play the game correctly, alternating between the two players, and that the AI player also plays along correctly. | N/A | Start a PVAI game, (the board width, height, player colour and difficulty are insignificant) and play ten total moves (five for the human plyer). | Ensure that the human player can input pieces into the game board correctly, and that after each accepted input from the human, it alternates to the AI player. Ensure that the AI player inputs pieces correctly and fairly. | |
| 22 | Test a normal input for a column input for a PVP game with width 7 and height 6. | Normal | Start a PVP game, set the board width to 7 and height to 6 (player colours are insignificant), and enter the number "4". | The input should be accepted, and the current player's piece should appear in the fourth column. | |
| 23 | Test an extreme input for a column input for a PVP game with width 7 and height 6. | Extreme | Start a PVP game, set the board width to 7 and height to 6 (player colours are insignificant), and enter the number "7". | The input should be accepted, and the current player's piece should appear in the seventh column. | |
| 24 | Test an erroneous input for a column input for a PVP game with width 7 and height 6. | Erroneous | Start a PVP game, set the board width to 7 and height to 6 (player colours are insignificant), and enter the number "8". | The input should be rejected, and another input for the column should be asked for. | |
| 25 | Test that the "save" option works during a game. | N/A | Start a PVP game, set the board width to 7 and height to 6 and set the player 1 piece to "R" and | The input should be accepted, and the board array, free columns array and | |

| | | | | | |
|---|---|---|---|---|---|
| | | | the player -1 piece to "Y". Enter a piece into column 1 for player "R" and a piece into column 4 for player "Y". Then, enter the number "0". | the game data should be saved to three text files ("s_b_save.txt", "s_c_save.txt" and "s_data_save.txt"). | |
| 26 | Test the vertical win checking algorithm by creating a "stack" or "tower" of four pieces to see if it recognises it as a win. | N/A | Start a PVP game (the board width, height and player colours are insignificant) and create a "stack" or "tower" in column "4" for four turns in a row for player 1. | After the fourth input for player 1, the game should recognise that there is four in a row and end the game with a win statement. | |
| 27 | Test the horizontal win checking algorithm by creating a "row" of four pieces to see if it recognises it as a win. | N/A | Start a PVP game, set the board width to 7 and height to 6 (player colours are insignificant), and create a "row" across columns "2", "3", "4" and "5" for four turns in a row for player 1. | After the fourth input for player 1, the game should recognise that there is four in a row and end the game with a win statement. | |
| 28 | Test the diagonal positive win checking algorithm by creating a positive diagonal "row" of four pieces to see if it recognises it as a win. | N/A | Start a PVP game, set the board width to 7 and height to 6 (player colours are insignificant), and create a positive diagonal row starting in column "2". The piece should be in column "2" in the bottom row, then column "3" in the second from bottom row, and so forth. | After the fourth diagonal piece is inputted for player 1, the game should recognise that there is four in a row and end the game with a win statement. | |
| 29 | Test the diagonal negative win checking algorithm by creating a negative diagonal "row" of four pieces to see if it recognises it as a win. | N/A | Start a PVP game, set the board width to 7 and height to 6 (player colours are insignificant), and create a negative diagonal row starting in column "5". The piece should be in column "5" in the bottom row, then column "4" in the second from bottom row, and so forth. | After the fourth diagonal piece is inputted for player 1, the game should recognise that there is four in a row and end the game with a win statement. | |

| | | | | | |
|---|---|---|---|---|---|
| 30 | Test choosing player pieces works when inputted into the board. | N/A | Start a PVP game (the board width and height are insignificant) and set the player 1 piece to "R" and the player -1 piece to "O" and input a piece into column "1" for player 1 and a piece into column "2" for player -1. | After the two inputs, there should be an "R" in column "1" and an "O" in column "2". | |
| 31 | Test that full columns do not allow the input of a piece. | N/A | Start a PVP game, set the board width to 7 and height to 6 (player colours are insignificant), and make a "stack" or "tower" of seven pieces in column "2". Try to enter an eighth piece into column "2". | The game should prompt the player that the column is full and ask for another input. | |
| 32 | Test that if the board becomes full then the game ends and is made a tie. | N/A | Start a PVP game (the board width and height are insignificant) and fill the entire board with pieces without making a win. | Once every column is full, the game should end, and it should be outputted/printed that the game is a tie. | |
| | | | **Artificial Intelligence (AI) Testing** | | |
| 33 | Test the "easy" (depth 2) AI works by creating a vertical "column" of pieces to see if it blocks the human player from getting four in a row (vertically). | N/A | Start a PVAI game with "easy" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and enter column "2" for three turns in a row. | After the third input, the AI should place its piece in column 2 in order to block the human player from winning. | |
| 34 | Test the "normal" (depth 4) AI works by creating a vertical "column" of pieces to see if it blocks the human player from getting four in a row (vertically). | N/A | Start a PVAI game with "normal" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and enter column "2" for two turns in a row. | After the second input, the AI should place its piece in column 2 in order to block the human player from winning. | |

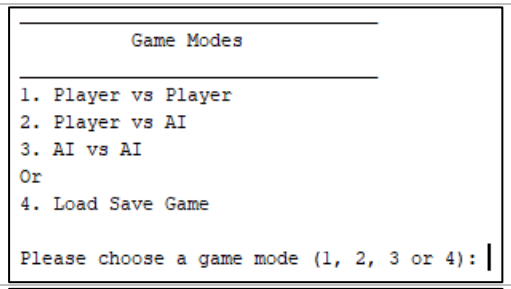
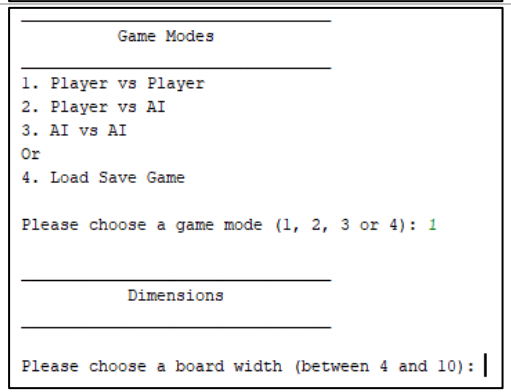
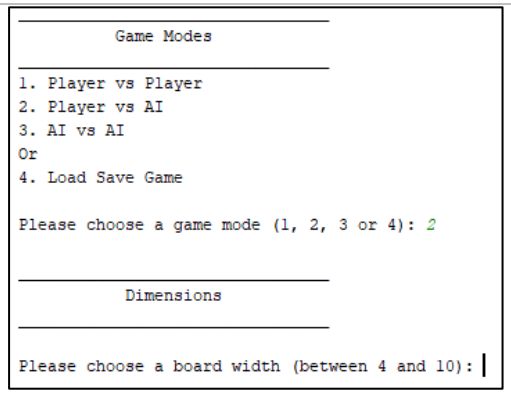
| | | | | | |
|---|---|---|---|---|---|
| 35 | Test the "hard" (depth 6) AI works by creating a vertical "column" of pieces to see if it blocks the human player from getting four in a row (vertically). | N/A | Start a PVAI game with "hard" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and enter column "2" for two turns in a row. | After the second input, the AI should place its piece in column 2 in order to block the human player from winning. | |
| 36 | Test the "easy" (depth 2) AI works by creating a horizontal "row" of pieces to see if it blocks the human player from getting four in a row (horizontally). | N/A | Start a PVAI game with "easy" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and enter columns "1", "2" and "3". | After the third input of "3", the AI should place its piece in the fourth column in order to block the human player from winning. | |
| 37 | Test the "normal" (depth 4) AI works by creating a horizontal "row" of pieces to see if it blocks the human player from getting four in a row (horizontally). | N/A | Start a PVAI game with "normal" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and enter columns "1" and "2". | After the second input of "2", the AI should place its piece in either column "3" or column "4" in order to prevent any chance of the human player having a winning row from that initial starting position. | |
| 38 | Test the "hard" (depth 6) AI works by creating a horizontal "row" of pieces to see if it blocks the human player from getting four in a row (horizontally). | N/A | Start a PVAI game with "hard" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and enter column "1". | After the first input of "1", the AI should place its piece in either column "3" or column "4" in order to prevent any chance of the human player having a winning row from that initial starting position. | |
| 39 | Test the "easy" (depth 2) AI works by creating a diagonal positive "row" of pieces to see if it blocks the human player from getting four in a row (diagonally positive). | N/A | Start a PVAI game with "easy" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and create a diagonal positive row starting from column "1". | After there are three pieces in a row (diagonally positive) and the next position the human player can make makes a win, the AI should place its piece in that | |

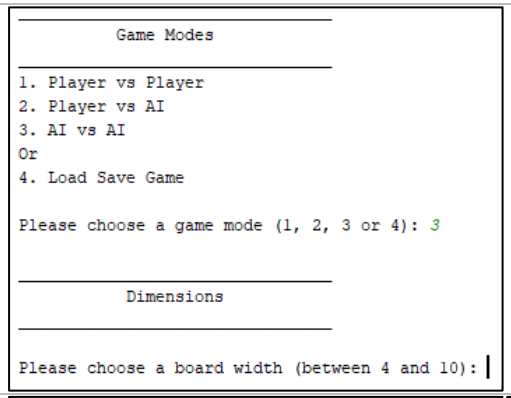
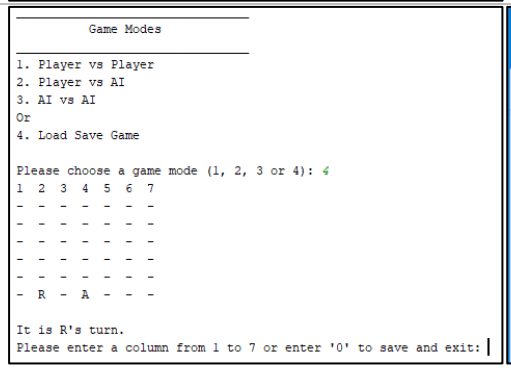
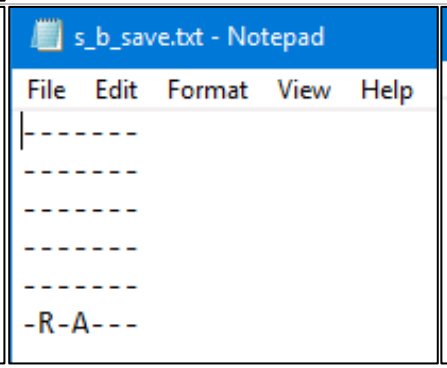
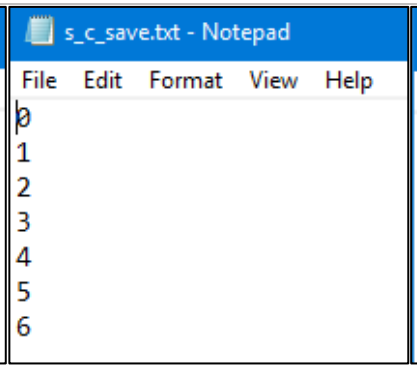
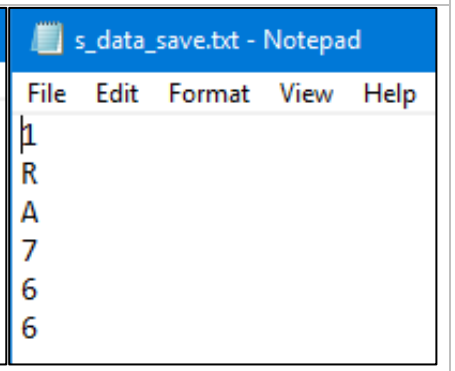
| | | | | | |
|---|---|---|---|---|---|
| | | | | column to prevent a win for the human player. | |
| 40 | Test the "normal" (depth 4) AI works by creating a diagonal positive "row" of pieces to see if it blocks the human player from getting four in a row (diagonally positive). | N/A | Start a PVAI game with "normal" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and create a diagonal positive row starting from column "1". | It may not be obvious at first what the AI is doing as due to probability and randomicity there is no way of predicting what the AI will do. However, it should make an effort to prevent a win for the human player after the human player has two or three pieces in a row. | |
| 41 | Test the "hard" (depth 6) AI works by creating a diagonal positive "row" of pieces to see if it blocks the human player from getting four in a row (diagonally positive). | N/A | Start a PVAI game with "hard" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and create a diagonal positive row starting from column "1". | It may not be obvious at first what the AI is doing as due to probability and randomicity there is no way of predicting what the AI will do. However, it should make an effort (a much stronger and harder effort than "normal") to prevent a win for the human player after the human player has two or three pieces in a row. | |
| 42 | Test the "easy" (depth 2) AI works by creating a diagonal negative "row" of pieces to see if it blocks the human player from getting four in a row (diagonally negative). | N/A | Start a PVAI game with "easy" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and create a diagonal positive row starting from column "4". | After there are three pieces in a row (diagonally negative) and the next position the human player can make makes a win, the AI should place its piece in that column to prevent a win for the human player. | |
| 43 | Test the "normal" (depth 4) AI works by creating a | N/A | Start a PVAI game with "normal" difficulty, set the board width to | It may not be obvious at first what the AI is doing as due to | |

| | | | | | |
|---|---|---|---|---|---|
| | diagonal negative "row" of pieces to see if it blocks the human player from getting four in a row (diagonally negative). | | 7 and height to 6 (player colour is insignificant) and create a diagonal positive row starting from column "4". | probability and randomicity there is no way of predicting what the AI will do. However, it should make an effort to prevent a win for the human player after the human player has two or three pieces in a row. | |
| 44 | Test the "hard" (depth 6) AI works by creating a diagonal negative "row" of pieces to see if it blocks the human player from getting four in a row (diagonally negative). | N/A | Start a PVAI game with "hard" difficulty, set the board width to 7 and height to 6 (player colour is insignificant) and create a diagonal positive row starting from column "4". | It may not be obvious at first what the AI is doing as due to probability and randomicity there is no way of predicting what the AI will do. However, it should make an effort (a much stronger and harder effort than "normal") to prevent a win for the human player after the human player has two or three pieces in a row. | |
| 45 | Test the "AIVAI" game mode by letting it play out a game. | N/A | Start an AIVAI game, set the board width to 7 and height to 6, and let the game play out. | There is practically no way of predicting what the AI will do due to the probability and randomicity of how it chooses the best position(s). However, the AI should play almost a full game against one another and could likely result in a tie. | |

## 4.2 Test Plan Usage

| ID | Actual Result |
|---|---|

| | | |
|---|---|---|
| 1 |  | After running the program, the game mode selection menu appears as expected. |
| 2 |  | After entering "1" on the game mode selection menu, the input is accepted, and the dimensions selection menu appears. |
| 3 |  | After entering "2" on the game mode selection menu, the input is accepted, and the dimensions selection menu appears. |

| | | |
|---|---|---|
| 4 |  | After entering "3" on the game mode selection menu, the input is accepted, and the dimensions selection menu appears. |
| 5 |     | After entering "4" on the game mode selection menu with these save files, the game started correctly. The board has the correct pieces displayed and the free columns array is also correct. The data from the data file (current player = 1 ("R"), positive player piece = "R", negative player piece = "A", board width = 7, board height = 6 and game difficulty = 6) is all displayed correctly and the game has started correctly using this data. |

6

```
_____________________________
          Game Modes
_____________________________
1. Player vs Player
2. Player vs AI
3. AI vs AI
Or
4. Load Save Game

Please choose a game mode (1, 2, 3 or 4): 4
Traceback (most recent call last):
  File "C:/Users/dyate/OneDrive - King George V College/A2 - Computer Science - Daniel Yates/Year 13/C4 AI Main (PyCharm)/main.py", line 13, in <module>
    main()
  File "C:/Users/dyate/OneDrive - King George V College/A2 - Computer Science - Daniel Yates/Year 13/C4 AI Main (PyCharm)/main.py", line 9, in main
    g.g_start_game()
  File "C:\Users\dyate\OneDrive - King George V College\A2 - Computer Science - Daniel Yates\Year 13\C4 AI Main (PyCharm)\game.py", line 19, in g_start_game
    self.g_get_mode_choice()  # Get game mode. This sets the self.mode attribute, which is then evaluated by the following if statement(s)
  File "C:\Users\dyate\OneDrive - King George V College\A2 - Computer Science - Daniel Yates\Year 13\C4 AI Main (PyCharm)\game.py", line 203, in g_get_mode_choice
    self.g_load_game()
  File "C:\Users\dyate\OneDrive - King George V College\A2 - Computer Science - Daniel Yates\Year 13\C4 AI Main (PyCharm)\game.py", line 356, in g_load_game
    l_data_save_file = open("s_data_save.txt", "r")  # This opens a save file called "s_data_save.txt" in read mode
FileNotFoundError: [Errno 2] No such file or directory: 's_data_save.txt'

Process finished with exit code 1
```

After entering "4" on the game mode selection menu with no save files, the error message "FileNotFound" occurred as it could not find the save file(s).

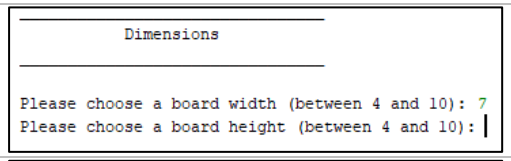
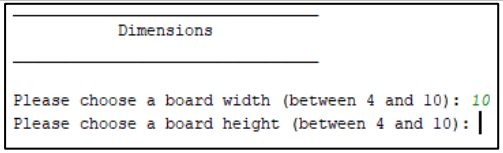
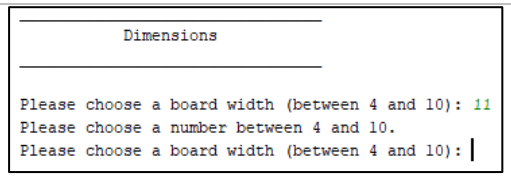
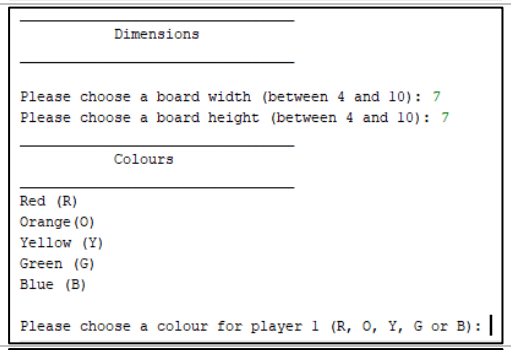
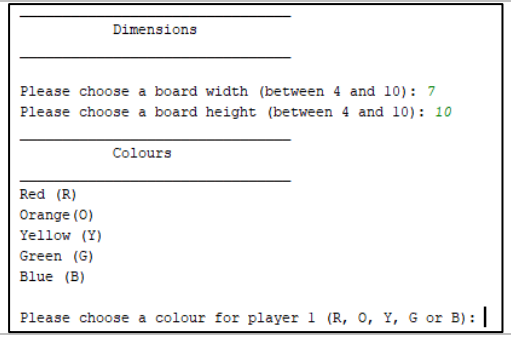
```
import os.path
```

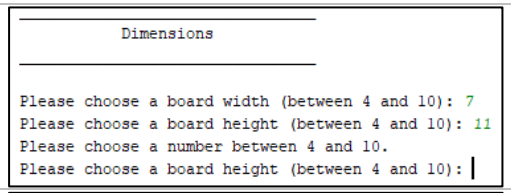
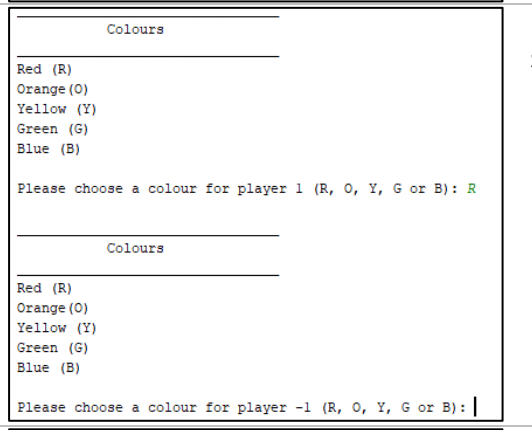
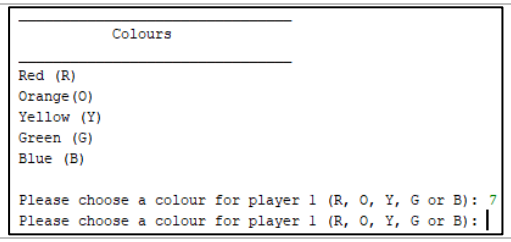
```
357     def g_load_game(self):  # This subroutine loads the board array, free columns array and game data from three separate text files
358         if os.path.isfile("s_data_save.txt") is False or os.path.isfile("s_b_save.txt") is False or os.path.isfile("s_c_save.txt") is False:
359             return False
```

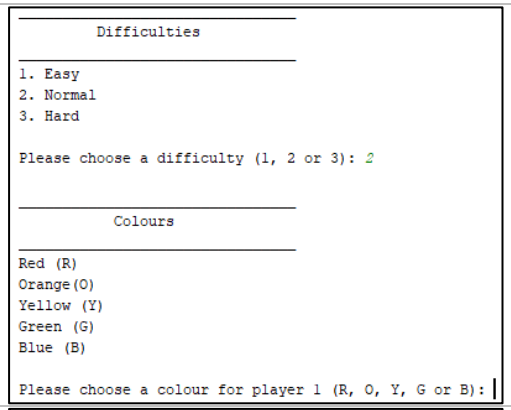
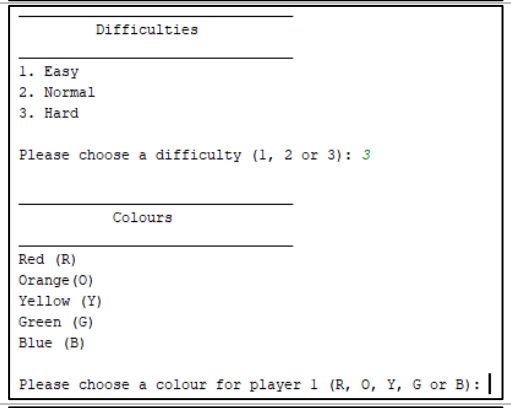
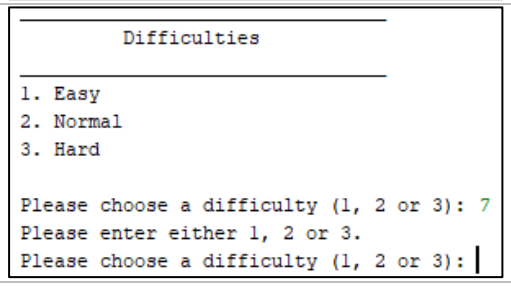
```
203         elif p_mode_input == 4:  # If the integer that they inputted is 4 then the game from the save
204             if self.g_load_game() is False:
205                 print("Sorry, you do not have a saved game to load from. Please try another option.")
206                 self.g_get_mode_choice()
207             else:
208                 break
```

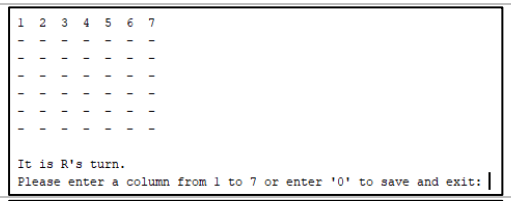
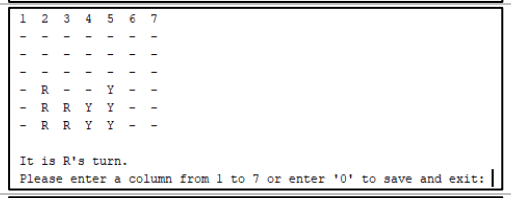
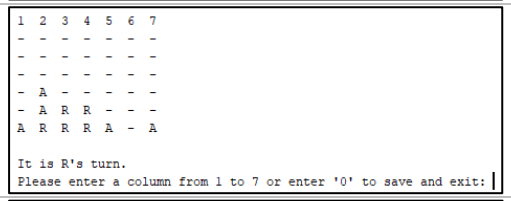
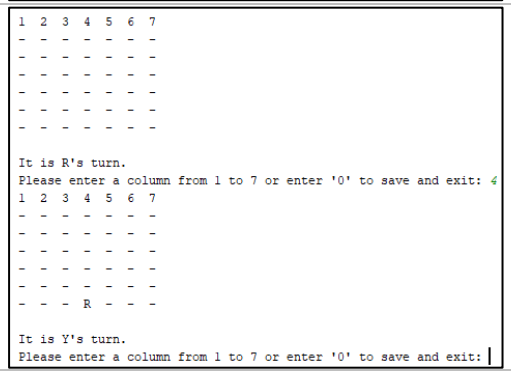
The above changes were made to my code in order to solve this problem. First of all, the "path" module is imported from the built in "os.py" module in Python. This is used in the second screenshot to check if any of the files do not exist. If any of them do not exist, then the game object's "g_load_game" function (it was a subroutine but now it returns a value so it is a

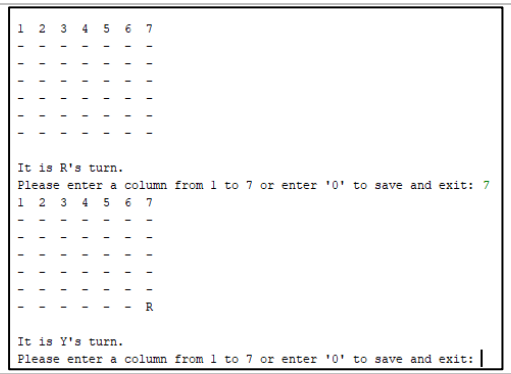
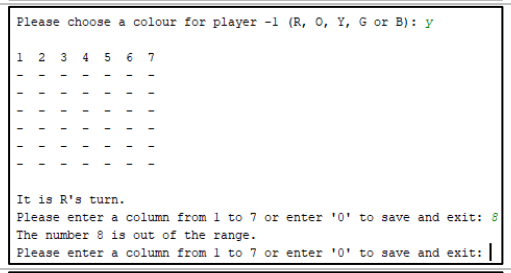
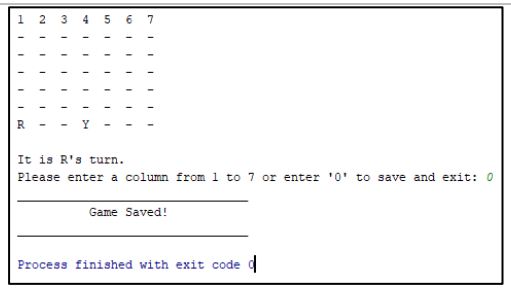
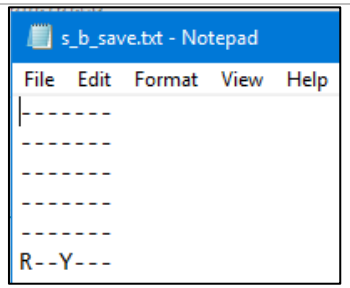
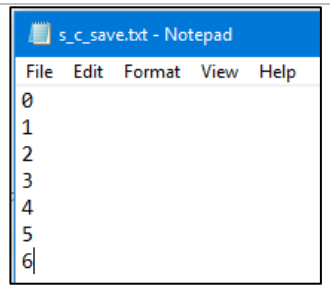
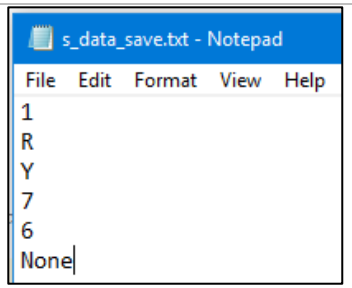
| | |
|---|---|
| | function) returns "False". This is then used in the last screenshot. What previously just performed the "g_load_game" subroutine, now checks to see if the function returns the value False. If it does return False, then the user is prompted to choose another option and the game object's "g_get_mode_choice" subroutine is run again to get another user input. Otherwise, the input is fine, and the "try except else" loop is broken out of as the input is accepted because all three save files must have been found.<br>`Game Modes`<br>`1. Player vs Player`<br>`2. Player vs AI`<br>`3. AI vs AI`<br>`Or`<br>`4. Load Save Game`<br>`Please choose a game mode (1, 2, 3 or 4): 4`<br>`Sorry, you do not have a saved game to load from. Please try another option.`<br>`Game Modes`<br>`1. Player vs Player`<br>`2. Player vs AI`<br>`3. AI vs AI`<br>`Or`<br>`4. Load Save Game`<br>`Please choose a game mode (1, 2, 3 or 4):`<br>This is now the result of choosing to load from a save game when no save files exist. The user is prompted that there is no save game to load from and then the game modes menu appears again for them to choose another option from. |
| 7 | `Game Modes`<br>`1. Player vs Player`<br>`2. Player vs AI`<br>`3. AI vs AI`<br>`Or`<br>`4. Load Save Game`<br>`Please choose a game mode (1, 2, 3 or 4): 7`<br>`Please enter either 1, 2, 3 or 4.`<br>`Please choose a game mode (1, 2, 3 or 4):` Entering a game mode choice of "7" prompts the user to "enter either 1, 2, 3 or 4" as "7" is an invalid game mode choice. |

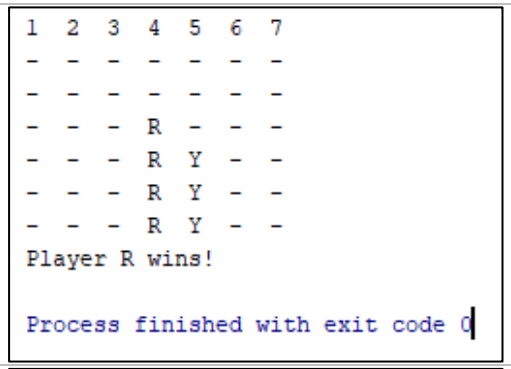
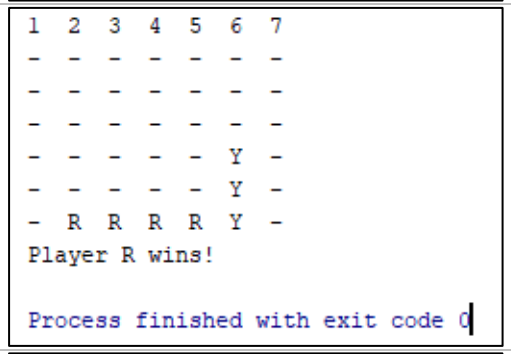
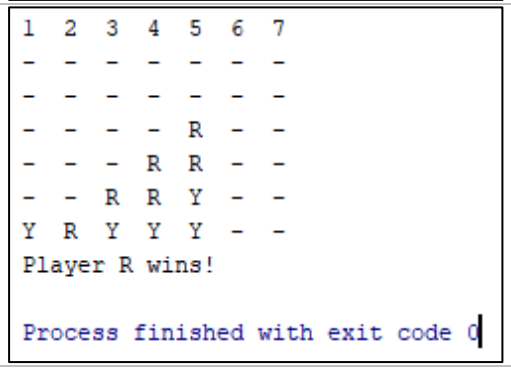
| | | |
|---|---|---|
| 8 |  | Entering a board width of "7" is accepted with no errors. |
| 9 |  | Entering a board width of "10" is accepted with no errors. |
| 10 |  | Entering a board width of "11" prompts the user to enter a width that is between four and ten as "11" is an invalid board width. |
| 11 |  | Entering a board height of "7" is accepted with no errors. |
| 12 |  | Entering a board height of "10" is accepted with no errors. |

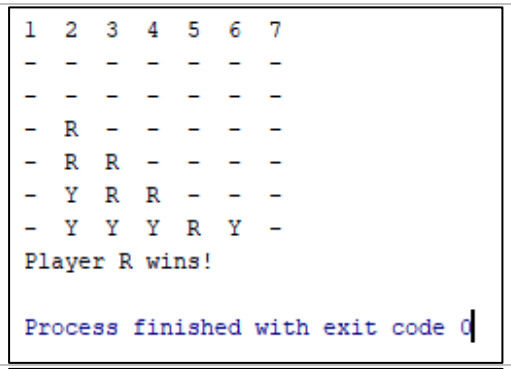
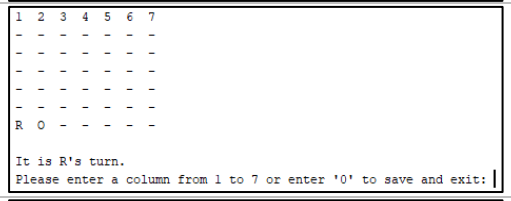
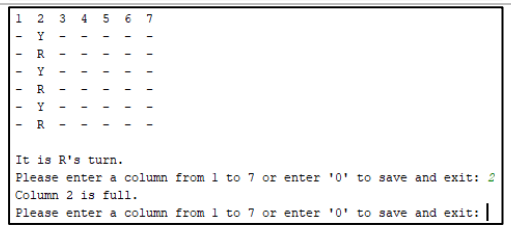
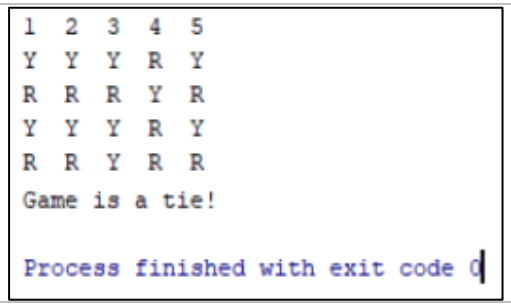
| | | |
|---|---|---|
| 13 |  | Entering a board height of "11" prompts the user to enter a height that is between four and ten as "11" is an invalid board height. |
| 14 |  | Entering a player colour of "R" is accepted with no errors as it goes on to ask for the second player's colour (for PVP). |
| 15 |  | Entering a player colour of "7" prompts the user to enter a player colour that is equal to either "R, O, Y, G or B" as "7" is an invalid player colour. |

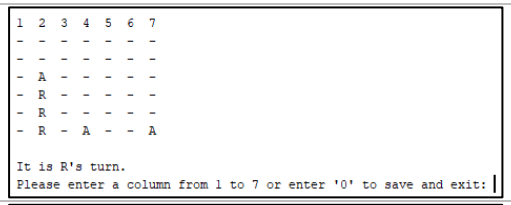
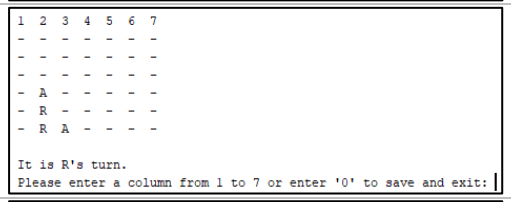
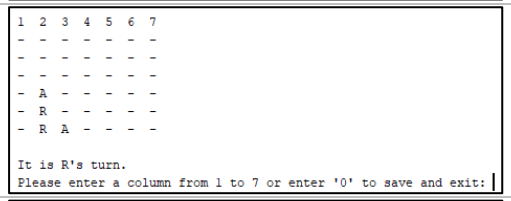
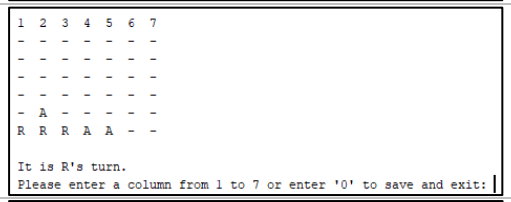
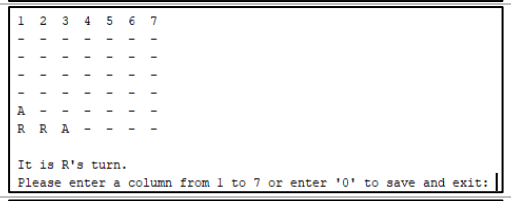
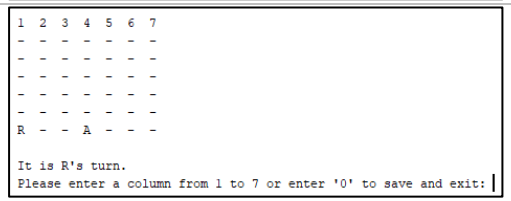
| | | |
|---|---|---|
| 16 |  | Entering an AI difficulty of "2" is accepted as it goes on to ask for the player colour choice. |
| 17 |  | Entering an AI difficulty of "3" is accepted as it goes on to ask for the player colour choice. |
| 18 |  | Entering an AI difficulty of "7" prompts the user to enter a difficulty that is either "1, 2 or 3" as "7" is an invalid AI difficulty. |

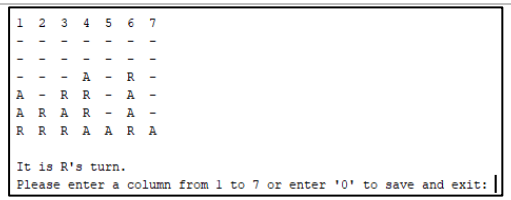
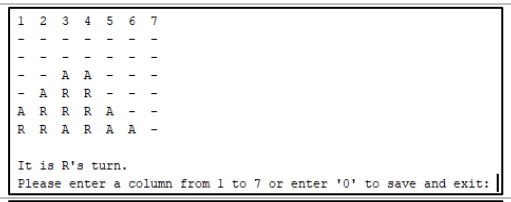
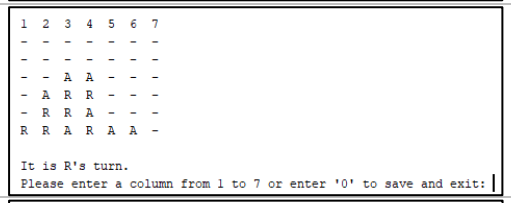
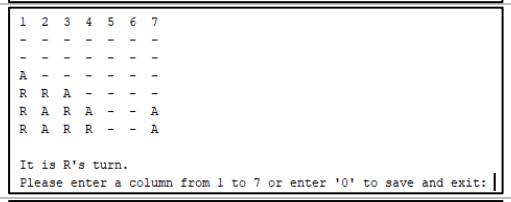
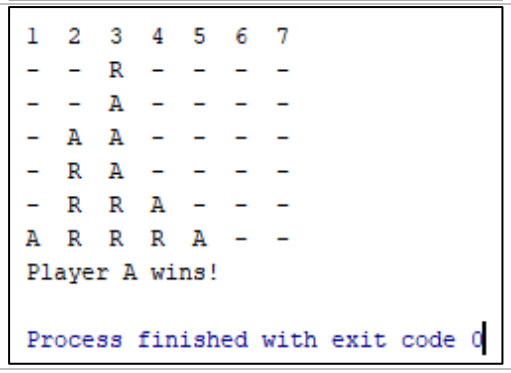
| | | |
|---|---|---|
| 19 |  | After starting the game with the specified settings, the board appears with the correct width (7) and height (6) that was inputted in the initial menu(s). The game then asks for the first player's input. |
| 20 |  | After ten moves, there are five "R" pieces and five "Y" pieces, as expected. These were placed into a variety of columns and the pieces stacked correctly on top of one another. The game also correctly alternated between the two players when asking for each respective player's column input. |
| 21 |  | After ten moves, there are five "R" pieces and five "A" pieces, as expected. These were placed into a variety of columns and the pieces stacked correctly on top of one another. The game also correctly alternated between the human player and the AI player. The human is able to input their piece, and after that happens the AI player inputs their piece, and then it alternates back to the human player's turn. |
| 22 |  | Entering column "4" on a board with width 7 and height 6 is accepted with no errors. |

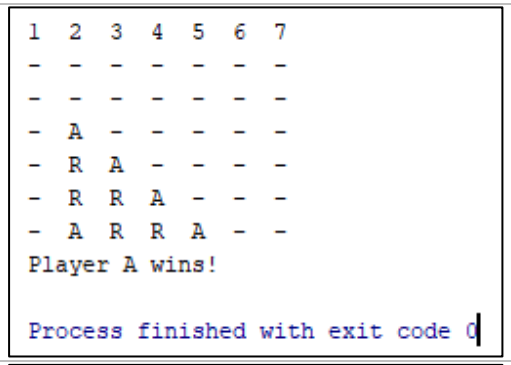
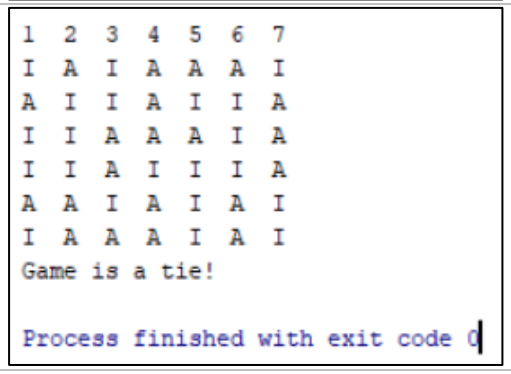
| | | |
|---|---|---|
| 23 |  | Entering column "7" on a board with width 7 and height 6 is accepted with no errors. |
| 24 |  | Entering column "8" on a board with width 7 and height 6 prompts the user that the number "8" is out of range and prompts them to enter another column. |
| 25 |     | After entering the data specified in the test plan and then entering "0", the input is accepted with no errors and the game prompts that it has been saved. In the same folder as the python files themselves, these three text files were then generated. The board has been saved correctly as well as the free columns array (no columns are full so they all appear in the text/save file). The data text file is also structured correctly and contains the data that is expected. Row by row, these pieces of data are: "1" is the current player which is correct as when saved it was "R"'s turn, "R" which is the positive player's piece, "Y" which is the negative player's piece, "7" which is the board width, "6" which is the board height and "None" which is the game difficulty. With "None" set, this implies that it is a PVP game as PVP games do not have difficulties. |

| | | |
|---|---|---|
| 26 |  | Entering four pieces in a row vertically for the positive player ("R") is recognised as a win by the game and so it outputs the winner and ends the game. |
| 27 |  | Entering four pieces in a row horizontally for the positive player ("R") is recognised as a win by the game and so it outputs the winner and ends the game. |
| 28 |  | Entering four pieces in a row diagonally positively for the positive player ("R") is recognised as a win by the game and so it outputs the winner and ends the game. |

| | | |
|---|---|---|
| 29 |  | Entering four pieces in a row diagonally negatively for the positive player ("R") is recognised as a win by the game and so it outputs the winner and ends the game. |
| 30 |  | After choosing "R" for player 1 (the positive player) and "O" for player -1 (the negative player), the correct pieces entered into the columns 1 and 2 as expected. |
| 31 |  | Entering "2" for the column choice when the column is full prompts the user that the column is full and asks them for another input. |
| 32 |  | After entering pieces into all columns without making a win (on a board with width 5 and height 4), the game pronounces that the board is full, outputs that the game is a tie, and exits the game. |

| | | |
|---|---|---|
| 33 |  | After inputting three pieces vertically in column "2", the "easy" AI places its piece on top of the "column" of pieces to prevent the human player from winning. |
| 34 |  | After inputting two pieces vertically in column "2", the "normal" AI places its piece on top of the "column" of pieces to prevent the human player from getting close to getting a win. |
| 35 |  | After inputting two pieces vertically in column "2", the "hard" AI places its piece on top of the "column" of pieces to prevent the human player from getting close to getting a win. |
| 36 |  | After inputting three pieces horizontally in columns "1", "2" and "3", the "easy" AI places its piece in column "4" in order to prevent the human player from winning. |
| 37 |  | After inputting two pieces horizontally in columns "1" and "2", the "normal" AI places its piece in column "3" (or it could have chosen column "4") in order to prevent the human player from having any chance of getting a winning row horizontally from that initial starting position. |
| 38 |  | After inputting one piece in column "1", the "hard" AI places its piece straight away into column "4" (or it could have chosen column "3") in order to prevent the human player from having any chance of getting a winning row horizontally from that initial starting position. |

| | | |
|---|---|---|
| 39 |  | After creating a row of three pieces diagonally positively and making a column of pieces in the fourth column ready to place the fourth diagonally positive piece, the "easy" AI places its piece in the fourth column in order to stop the human player from winning. |
| 40 |  | After starting from the first column, the "normal" AI makes a strong effort to defend itself as well as make its own winning positions. It played much more of a difficult game than the "easy" AI and, of course, blocked my diagonally positive win attempt when I had three in a row. |
| 41 |  | After starting from the first column, the "hard" AI makes an even stronger effort to defend itself as well as make its own winning positions. It played much more of a difficult game than the "normal" AI and, of course, blocked my diagonally positive win attempt when I had three in a row. |
| 42 |  | After creating a row of three pieces diagonally negatively and making a column of pieces in the first column ready to place the fourth diagonally negative piece, the "easy" AI places its piece in the first column in order to stop the human player from winning. |
| 43 |  | Any time I tried to create a diagonally negative row of pieces starting from the fourth column on the "normal" AI difficulty, the AI would perform different unpredictable moves in order to essentially win itself every single time. This difficulty, as expected and as it should be, is much harder than the "easy" difficulty. This test works due to the fact that it is preventing there from being a diagonally negative win by winning itself. |

| | | |
|---|---|---|
| 44 |  | Any time I tried to create a diagonally negative row of pieces starting from the fourth column on the "hard" AI difficulty, the AI would perform different unpredictable moves in order to essentially win itself every single time. This difficulty, as expected and as it should be, is much harder than the "normal" difficulty. This test works due to the fact that it is preventing there from being a diagonally negative win by winning itself. |
| 45 |  | The AIVAI (artificial intelligence vs artificial intelligence) game resulted in a tie. This game could have ended sooner (with one of the AI players winning) but in this instance there was a tie. As the difficulty for the AIVAI mode is set to 6 ("hard"), the AI played a very strong game against one another. |

# 5 Evaluation

## 5.1 Evaluation of Objectives

### 5.1.1 Game Modes

#### 5.1.1.1 Player vs Player

This objective has been fully met. There is a mode in which two human players can play against each other. They are each able to choose a piece/player colour as well as between them decide on the board dimensions. When the game begins, it correctly alternates between the two players, allowing them to each input pieces into the board. When a win is found, the game recognises this correctly and outputs the respective winner of the game.

#### 5.1.1.2 Player vs Artificial Intelligence

This objective has been fully met. There is a mode in which one human player can play against an artificial intelligence (AI) player. The human player can choose their piece/player colour (the AI player uses the letter/piece "A"), decide on the board dimensions and choose an AI difficulty. There are three difficulties for the player to choose from ("easy", "normal" and "hard"). Each difficulty has a sufficiently increased level of difficulty, each being harder for the human player to win against.

#### 5.1.1.3 Artificial Intelligence vs Artificial Intelligence

This objective has been fully met. There is a mode in which two AI players play against one another. The positive player uses the letter/piece "A" and the negative player uses the letter/piece "I". The board dimensions are decided upon by the user before the game begins. The AI uses a difficulty of 6 ("hard") and it plays a strong game against itself, either ending with a close win for one of the players or more often a tie.

### 5.1.2 Game Features

#### 5.1.2.1 Main Menu and Other Menus

This objective has been fully met. In order to begin a game there are menus that the user can easily navigate through. The initial main menu is the game mode selection menu. This menu is used in order to choose the game mode that the user would like to play. After choosing a game mode, the other respective menus associated with that game mode for customising the settings for that mode are shown and allow the user to customise the game to their liking.

#### 5.1.2.2 Ability to Save Game

This objective has been fully met. At any point during either the "PVP" or "PVAI" game mode(s), the user can enter "0" in order to save the game and exit. This means that a user can stop a game part way through in order to come back to it at a later date. The board state, free columns and game data are all saved to text files in the root of the main Python file(s). With these files available, the user can then load a saved game and continue playing whenever they wish to do so. Before my testing, if there was no save files available and the user tried to load a game then the game would produce an error. This was quickly fixed by checking first to see if the files exist before trying to load data from them. If they do not exist, then the user is prompted accordingly and asked to input another game mode choice.

#### 5.1.2.3 Ability to Choose/Change Piece Colour

This objective has been fully met. There is a way in which the user can change the colour/letter of their piece that is inputted into the board. This is done before either the "PVP" or "PVAI" game mode begins, allowing them to play with one of the predefined letters/colours/pieces.

### 5.1.3 Game Settings

#### 5.1.3.1 Change Artificial Intelligence Difficulty

This objective has been fully met. There is a way in which the user can change the difficulty of the artificial intelligence (AI) before the "PVAI" game mode begins. The three difficulties are a good and board range and allow for a variety of players to play the game at their own level of experience. Each level of difficulty is a good increase in "hardness" from its predecessor. The difficulty changing has been achieved simply by evaluating the game tree at a further depth the harder the difficulty is set to ("easy" goes to a depth of 2, "normal" goes to a depth of 4 and "hard" goes to a depth of 6).

#### 5.1.3.2 Change Board Size/Dimensions

This requirement has been fully met. There is a way in which the user can change the dimensions of the board. This is done before any of the game modes begin and allows them to input both a width and height for the board, between a reasonable range of 4 and 10.

### 5.1.4 Miscellaneous

#### 5.1.4.1 Optimised and quick algorithms

This requirement has been fully met. Overall it was very important to the program as well as the user base that the game played quickly and was not slow. It was important that all of the algorithms created were as efficient as possible in order to make the game run with maximum efficiency and speed. The maximum waiting time for the game is on the "PVAI" game mode when the difficulty is set to "hard" (a tree depth of 6). When the AI chooses its position, it takes around 1.5 seconds in order to make its decision. This is very reasonable considering how deep into the tree it is looking, and does not keep the human player waiting too long.

## 5.2 Extensions to the Project

### 5.2.1 Multiple Save Files

One extension that could be made to the project would be the ability to have multiple save files with custom names inputted by the user. In its current state, the game can only read and write one saved game, storing the board state, free columns state and board data to three separate text files respectively. There may be multiple users using the game/system, and so they may all want their own save files. Similarly, one user may just want to have multiple save files to continue at any time of their choosing. To implement a feature like this would not be too difficult and would require some changes to the saving and loading functions/subroutines. The change would require the user to input a name for their save both when saving the game and when trying to load their saved game.

### 5.2.2 Playing Across a Network

Another extension that could be made to the project would be the ability to play the game across a network with another player. In its current state, the game supports local play only, and for two human players to play against one another they have to be playing on the same computer. The solution to this would be for some sort of network communication, allowing two players connected either to the same network or the internet to play against/with one another. To implement a feature like this would be somewhat difficult as I would have to research networking in Python, which is something I have never tried before.

### 5.2.3 Raspberry Pi LED Matrix

A final extension that could be made to the project would be to output the game board to a Raspberry Pi LED matrix. In its current state, the game simply outputs a command line interface, with the board and pieces represented with ASCII characters. By connecting the game to a Raspberry Pi and outputting it onto an LED matrix, the board will have a much more visual representation, with each player's piece represented with a coloured "dot" or "pixel" on the LED matrix. To implement a feature like this would not be too difficult but would require some research into how to output to certain pixels on an LED matrix.

## 5.3 Conclusion

To conclude, the game meets all of the objectives from the initial analysis section of the documentation and I believe I have been very successful in achieving my goals/objectives for this project. The artificial intelligence aspect of the game took significantly longer to develop than I had anticipated and required a lot of research as well as logical thinking in order to fully develop it, bug free. The algorithms I have created are not only effective and work but are also very optimised and efficient at what they do. This means the game does not just work, but it works fastidiously.

## 5.4 Sources Used

Trevor Payne's YouTube Channel: used for Python tutorials as well as guidance with Python AI: https://www.youtube.com/user/TPayneExperience

Reddit: a variety of posts used in order to find a baseline for where to start with making a Connect 4 AI: https://www.reddit.com

Stack Overflow: a variety of posts used in order to find a baseline for where to start with making a Connect 4 AI: https://stackoverflow.com