NEA Project Report

Connect 4 Game with Minimax Implementation

Name: Ollie Easterbrook
Candidate Number: 3025
Centre Number: 61679
Qualification Code: 601/4569/9 (QAN Code), 7517
Exam Year: June 2022

Contents:

ANALYSIS

Background and Problem Definition

The Game of Connect 4 is known to all ages, and I find that no matter who plays, they can always find some enjoyment in a difficult solution being found, or a tricky set up paying off dramatically. Recently, a teacher at my school, Mr Huxley, also became interested in the game of Connect4 and whether it could be played optimally; as a skilled player of the game, he wondered if he would be able to 'beat' it, using any kind of mathematical guesswork, or simply intuition. He would also like to challenge others on the go, or simply play alone, without having to lug around the clunky board with him everywhere, hence why a *playable* digital version would be preferred.

As Mr Huxley also teaches the lower years Computer Science at Tonbridge School, by giving the students an example of precisely how powerful algorithms can be, perhaps it could also bring in more students to his Computer Science Club he runs on Mondays. This is why he has also asked me to create a 'simulation' mode, where the user can enter a number of rounds of the game to be played, then select for the algorithm to play itself, or perhaps another opponent, with the scores being totalled and displayed as the game goes on.

The game of Connect 4 is played with two players, each of whom have 21 counters of their own colour. These counters can be dropped into any of seven columns into a grid of six rows, with the goal being to create a line of four in a row without any interruption from the opponent's counters. This implies that from every game state, there are *at most* seven possible moves that can be made by a player, only limited by the potential of a column being entirely full of counters. Whilst creating an algorithm to find the optimal move out of seven in a grid may be interesting, it does not account for the other player's turn, where, for example, it may now be possible to win the game outright. Therefore, we would need to create an algorithm that maximises the benefits of current player's potential moves, whilst also minimising that of the opposing player's. Hence why I decided that Minimax would be the best possible solution for this problem.

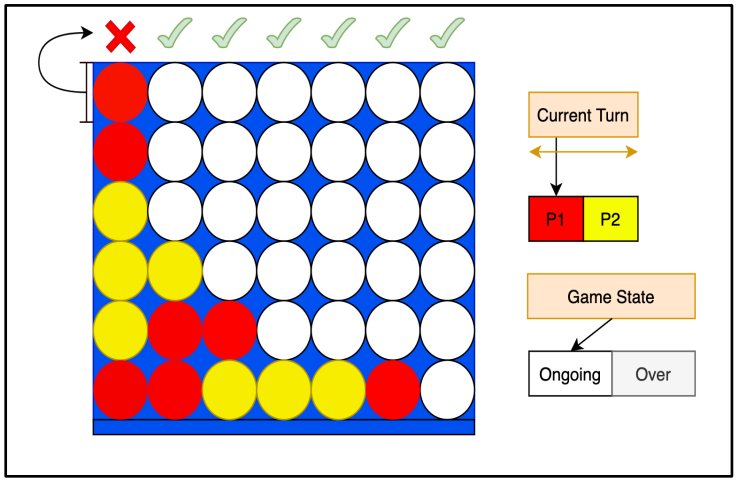
My project is designed to allow others to enjoy the game of Connect 4 wherever they are, as well as observe the possibility of an 'optimal' game through a self-designed Minimax algorithm; hopefully appealing to those interested in Maths, Computer Science, or games as a whole and bringing new blood to Mr Huxley's Computer Science Club.

I am going to interview Mr Huxley to gain his views on the current state of my ideas and any improvements he would like to see, as well as develop objectives[1] that must be met throughout my development, else my program would be insufficient for his use.

Description of the Current System

Within Connect 4, whenever it is your turn, you may place one counter in any of seven rows, given that the row is not full. Once this counter is placed, it becomes the opponent's turn to make a move, under which you cannot play. This system alternates till one of three outcomes occurs: your victory, your opponent's victory, or a draw, after which the game is over. A draw can only occur once *every* column in the grid is full, where no more legal moves are possible, whilst otherwise, the game is ongoing until four in a row of either player's counter are connected vertically, horizontally, or diagonally.

[1] Found on Page 8



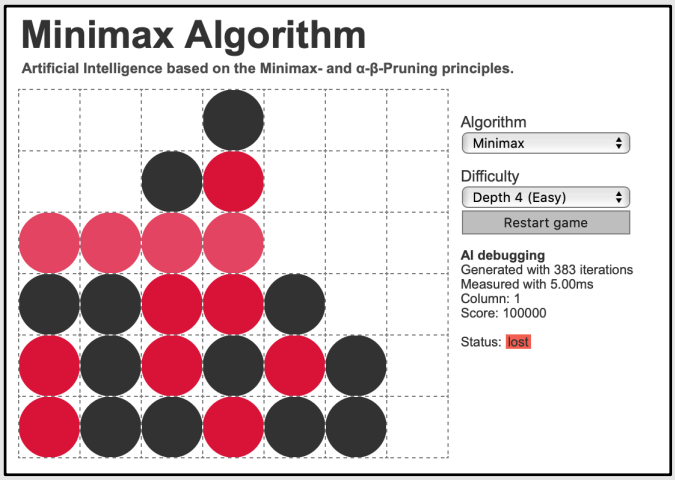
In this diagram, we can see how the board should look, where all pieces 'obey gravity' by travelling to the bottom of the empty columns, and the full columns being unplayable in. The game state is constantly Ongoing until it becomes Over once a result is met, after which it cannot be undone. The turn changes per move, hence its oscillations.

Analysis of Similar Systems

Whilst I hope to keep this application my own, there are other options online that could benefit the method with which we approach development by allowing me to identify 'objectives' which must be tackled to create a successful project:

System 1:[2]

Connect Four JavaScript AI based on the Minimax and Alpha Beta Pruning Algorithm



Overview:
A very verbose, compact implementation of minimax, with the ability to decide between Minimax with or without Alpha Beta Pruning, as well as the depth of Minimax. Small animations such as the glowing victory line or 'AI thinking…' screen make it feel polished and well created; whilst the simplistic, yet detailed information displayed under AI Debugging allow for easy understanding by the player.

Positive:

- UI Elements such as animation produce an interactive feeling for the player.
- User-friendly UI.
- The ability to choose depth allows for the user to better grasp the concept of Minimax as a whole.

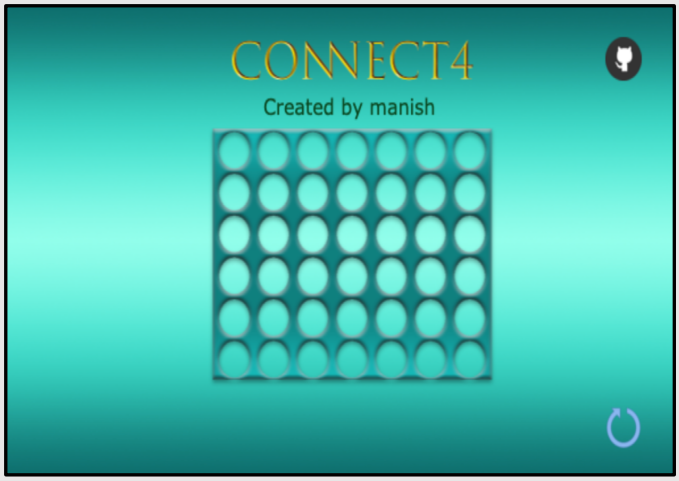
Negative:

- Despite being a refined Minimax, Alpha Beta Pruning doesn't seem to have much of an effect on the game of Connect 4, barely noticeable for a lot more extra effort.
- No ability to compare Minimax to any other systems except for the player.

[2] https://www.gimu.org/connect-four-js/plain/minimax/index.html

System 2:[3]

Connect 4 by iammanish17



Overview:
A browser run version of Connect Four, with some very outdated visuals. Whilst the colours are welcome, the lack of much information or customisability of the Minimax causes this to come off as slightly dull in comparison to System 1. Again, the small animations make the project feel more alive and fun to play against. If more information were given, and the UI made slightly more verbose, the project would be quite decent.

Positive:

- Minimax algorithm makes the game challenging, yet fun to play against.

Negative:

- No customisability.
- No information displayed to the user, which only slightly subtracts from game
- Browser ran doesn't add to the experience, especially since the window doesn't fit the screen.
- Too distracting of a colour scheme.

Summary Observations:

Our project must have:

- A simplistic UI with minimal colouring
- Customisability of the minimax algorithm depth
- Simulations between minimax and other opponents
- The ability to run on the current system without a browser implementation

Our project may also have:

- A verbose display of the Minimax's calculations to the user
- Implementation of the Alpha-Beta Pruning algorithm
- Animations between each turn

[3] https://iammanish17.github.io/Connect4/

Problems with the Current System

Inability to play alone:
Not many people have the time to stop what they're doing and play an intensive game, meaning that even if a player is found, the talented Mr Huxley may simply find no challenge and therefore not enjoy the game he tries to play so often. Therefore, this problem is in tandem with the difficulty of finding anybody challenging to play against, both of which require quite some luck and time to locate.

Lack of knowledge of the 'perfect' game:
Despite being a strong mathematician, some problems are nigh impossible for humans to calculate. Mr Huxley has no way of 'evaluating' the state of the board and using it to his advantage to predict moves, he simply plays by experience, which can lead to careless mistakes, or a miss of a victory. This distinctly lowers his chances of winning against certain types of opponents, whom he is curious on how to beat. Moreover, the ability to calculate and play the perfect move could inspire a growth in popularity of his school club, which at the moment, doesn't seem to understand the application of subjects such as game theory in computing.

Physical space taken up by the board:
Desk space, or table space is required for the board to be placed on and played with, with the board itself also having to either be stored in a safe location or taken with Mr Huxley wherever he travels. As the board is rigid, and the counters plentiful, it takes up quite a lot of space in which more important things could be stored, such as equipment for his job at the school, or as part of the computer science club.

Identification of the User, Requirements to be met and Limitations of the Project

User

The primary user of my program will be Mr Huxley, with distribution to other students at Tonbridge School, all of whom can run the program and access all of its features without the need for a passcode, or login. However, rather than making the code completely inaccessible to the user, I intend for everybody to be able to alter key variables such as the depth of the Minimax Algorithm, as all this does is project new behaviour for the user to observe. Of course, as Minimax algorithm code isn't hard to locate online, I will also be willing to show the code to any of the students who ask, as well as run through it with them.

Project Requirements

To be sure that I fully grasp what my user's requirements for the project are, I conducted an interview with him after school; asking him questions about his current situation and what improvement he would like to see. As a teacher, I believed his advice to be far more crucial than any of the surrounding pupils, and therefore did not ask them any additional questions.

- *What would you say fascinates you the most about Connect Four?*
  "I would say that the possibility of an optimal solution, because after all, if it did in fact exist, then surely the best ever player could simply just be a good mathematician… if, of course, maths has any part in the actual development of an optimal move…"

Mr Huxley has indicated his intrigue towards finding the optimal game, as well as the maths behind it. This extends into his curiosity of whether an 'optimal player' could exist.

- *And with your current level of skill, do you find it difficult to find games against others?*
  "Oh, all the time! But it isn't just finding people around my skill that would also want to play, it's a struggle to set the board up and get out the pieces and everything in time; I don't get many breaks, you know. That being said, even if somebody was to play me, there usually isn't much challenge."

Mr Huxley struggles to find a decent opponent due to all of the inefficiencies with a physical connect four board, such as setting it up or having the time to even start a game. The last comment implied that he also wanted to find a challenge within the game, as his skill level makes this difficult.

- *Could you state any issues that you're currently experiencing with the game as a whole?*
  "Hmm… Well, some students lost a few of the counters but didn't tell me, meaning that we can't exactly draw the game anymore… two weeks ago, the board also broke, meaning that I had to tape it together. It still falls apart sometimes when I put it away, because of how much room it takes up under the desk; I'm storing other things under there too, mostly for work."

Space is taken up by the board, yet far more importantly, the loss of counters and damaged board makes the physical game difficult to play. Mr Huxley finds that he is limiting himself from using it more often than not due to these physical limitations.

- *Are there any requirements you have for how the game looks? How it plays?*
  "For looks, I'd prefer something more simplistic than complex. If the original game doesn't have flashing lights and buzzers, then there's no need for that in this version… plus, that could easily distract me and others from what's actually happening. Seeing as there will some form of calculations for these moves, and the whole 'simulation' section has a high chance of getting dull quick… maybe create different opponents and make the simulation faster than the usual game."

According to Mr Huxley, animations and colours are not particularly necessary, my project should focus far more on the solution than the UI, however, it should be sure to **easily communicate the game itself**, hence why he wants to know what is 'actually happening'. He has also shown that he wishes to see **differing levels of difficulty in opponents**, as not only does this interest him, yet it may also attract others fascinated with things such as mathematics or computer science, both of which could easily attract to his Computer Science Club.

- *As for the simulation section, what would you like to see from the project?*
  "I thought about this for a while, and I've noticed that surely Minimax played solely against itself would become either a draw, or the same result every time…"

  That is true. However, Minimax could be used at varying depths to produce a varying 'difficulty' to these opponents.
  "Oh. Then yes, I would love to see a simulation of Minimax against itself with varying maximum depths, perhaps even a different kind of opponent to show its strength… Although under these time limits, you may want to just make the opponent random. Either way, a varying difficulty sounds fascinating."

The simulation has been requested to be against **varying opponents**, with the best possible implementation being against **varying levels of minimax** according to Mr Huxley. Although, to show its strength, perhaps playing

against a **random opponent** would help. Mr Huxley expressed interest in the 'strength' of each of these depths, so therefore allowing the user to **play against selectable depths** may be beneficial.

- *Why would you benefit from having this project made digitally?*
  "I mean, the storage problem is definitely a good thing to resolve. I'd also like to be able to send this to friends, maybe some people that I've played before to see if they can beat the challenge… and seeing as the 'optimal solution' is so interesting to me, I can easily take a look through the code if you explain it to me, maybe I'll benefit from the reasoning…"

Bypassing all physical limitations of the board would be useful for Mr Huxley. The ability to **freely use and send the game to others** means that Mr Huxley no longer must spend his own money on repairing the board. Furthermore, he has stated that as he is interested in the mathematical solution, a digital version would allow him to not only improve his game, but better understand connect four as he looks through the game.

From my interview and Mr Huxley's responses, I have identified a few key points that I need to address for my project to be a success:

1. The UI must be able to communicate an active game to the user, not particularly through animations.
2. Information, such as the level of the Minimax algorithm must be displayed to the user.
3. The simulation must be against varying opponents.
4. The Minimax depth of search must be selectable for both regular play and simulation games, as it adds difficulty to the game.
5. A random opponent must be selectable for a basis of the simulation.
6. The game must be freely runnable from the user's device without the use of wifi or internet connection, whilst also being easily sent to others.

Project Limitations

There are three limitations that will influence the development of my project, and how well executed I can make this piece of software:

Coding Skill
I will have to decide between and learn one of either Pygame or Tkinter, as these are two of the most well documented Python UI libraries, hopefully making them easier to learn in a short span of time. I also have never created an algorithm using complex data structures in Python, despite knowing Object Oriented Programming, the implementation of which may make things such as the evaluation function primitive and fall short of those found in the Systems.

Time
The project must be completed by Christmas, as after which I will quickly become involved with revision for my upcoming A Levels. Mr Huxley has suggested that I complete it before this time, as otherwise not only will it become a struggle to complete, but it will also come out at a lower quality, as will my eventual results.

Software
No matter which design library I use, animations and design will be forced into either a very simplistic format or come off as clunky. Animations take up a lot of time, and require some skill to code, and as specified above, I am struggling with both at once already; as a result, even if the

design were to slowly become more colourful and vibrant, I would struggle to create an appealing looking project for Mr Huxley.

Proposed Solution

Having observed the previous two solutions, I have decided to create a bespoke piece of software using Python and Tkinter, the former a language that I am familiar with, whilst the latter a library I will have to learn for the development. I will also be using the Integrated Development Environment "Visual Studio Code".

I have coded in Python for three years now and have found it an incredibly useful language to pick up and learn. The language is heavily object oriented as, apart from control flow, everything in Python is an object. This will allow me to easily define important objects such as the Connect Four Class itself, or the UI, which, when paired with the event driven library of Tkinter, can then be referenced, with class methods being called upon 'events' such as button presses or clicks. Although extra time will have to be put aside to learn the documentation of Tkinter, by using a familiar language in Python, hopefully this time will be made up for in the coding sections without Tkinter. As well as this, Python is a good option for running on other OSs. As this program should ideally run locally on a laptop, Python is the optimal language for this.

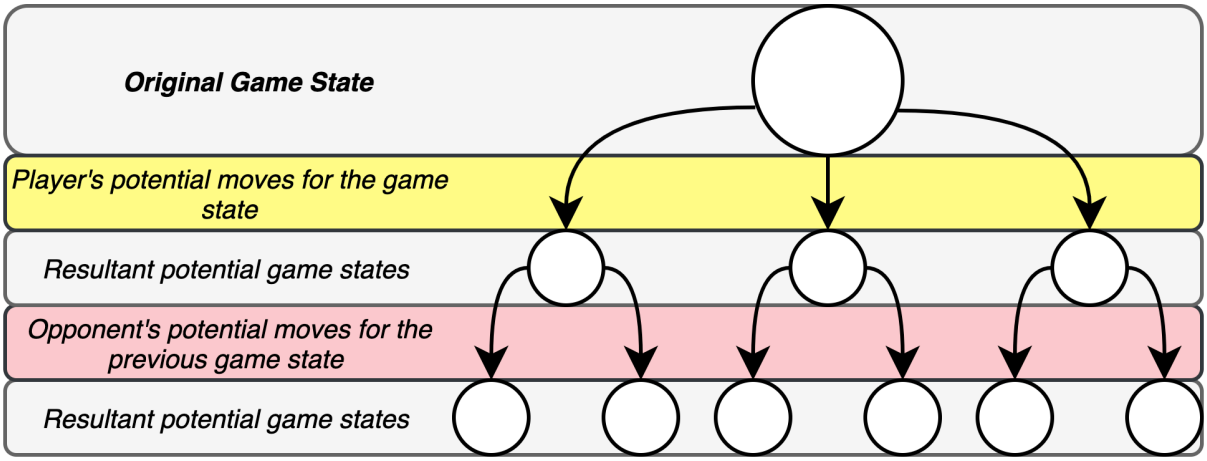
The Visual Studio Code IDE was chosen for its friendly design for newer programmers, such as its built in indentation, colour coding, or the easily accessible extension library, which installs whichever product may interest you on their store. Debugging is simple in Visual Studio Code, with the ability to add in breakpoints to the margin a helpful addition when debugging code, stopping at the highlighted line whenever it is run; this becomes less helpful the larger your project becomes, however, as most lines are read more than once, meaning that there is a chance that there are too many loops for your breakpoints to show you any meaningful information.

Python, Tkinter and Visual Studio Code are incredibly popular online, and as a result, have many easily accessible forums and official pages which may allow me to solve any issues I may be having with the project; Visual Studio has pages written by Microsoft themselves, whilst Tkinter and Python issues are best solved and located on Github.

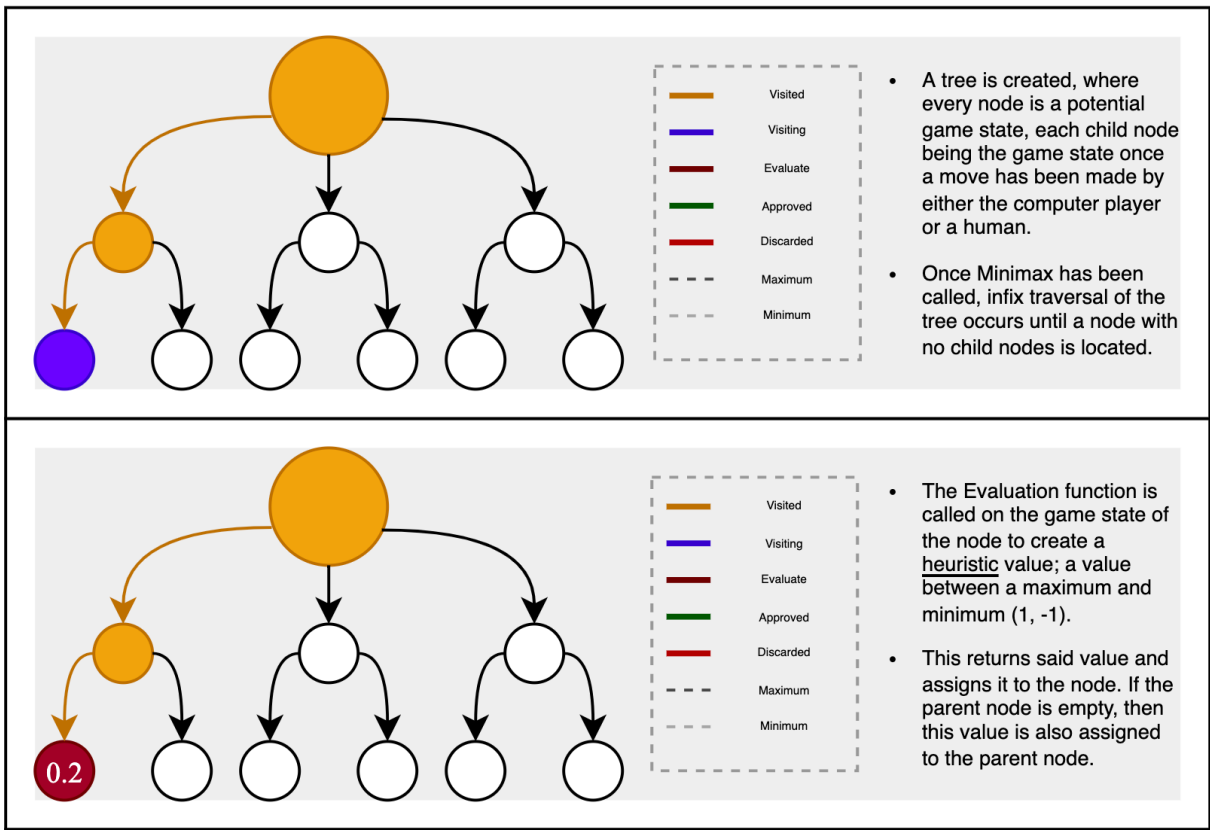
Whilst I could very easily use Pygame as a basis for my UI creation, it is mainly designed for *games*, rather than UI. It provides event handling, graphics sound binding, as well as other characteristics that can be useful for creating games, whilst Tkinter is far more simplified and cannot perform these functions. Due to my aforementioned limitations of time and skill, I found it best to proceed with Tkinter.

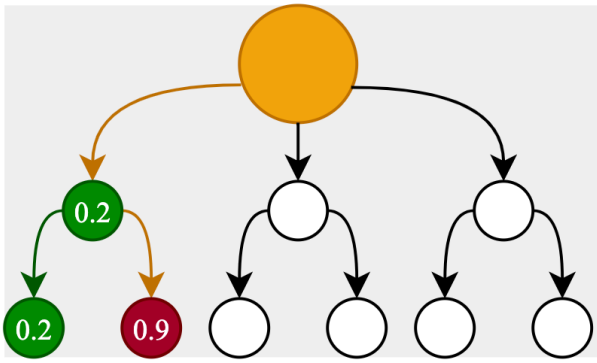
Benefits	Disadvantages
Free IDE, language, and library	Graphics will be challenging to animate
Easier to debug	Library required for event handling
Large number of online resources	Tkinter must be learnt from scratch
Experienced with language	
Easier to format	

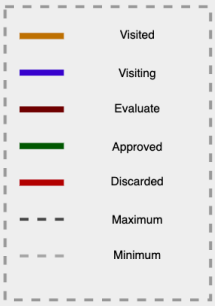
As to why I have selected Minimax, I would need to describe how Minimax functions. Minimax acts similarly to the simplified diagram below:



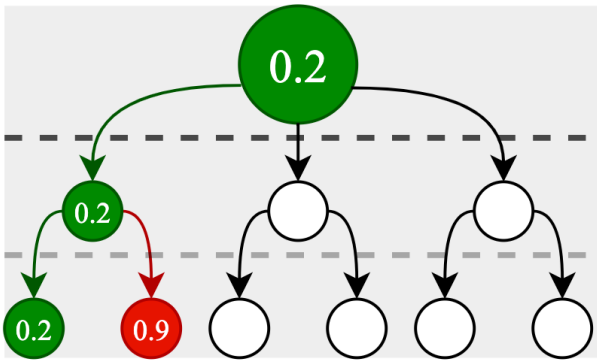
Given a select 'game state', in this example, our grid with however many counters in it, Minimax creates a tree of all potential moves for the player, followed by all potential moves by the opponent in this state, travelling down the tree to a set 'depth', before evaluating the 'score' of the game state, positive implying a positive position for the player, whilst negative implies a negative. I have highlighted each arrow in the diagram either red, or yellow, to indicate a minimum or maximum call respectively, as Minimax attempts to calculate the minimum possible gain of the opponent, yet the maximum possible gain of the player, comparing each state before returning the moves made to reach this position. As Connect Four has limited states (a maximum of *seven*), this would allow the Minimax to create less states overall and therefore calculate a move more quickly than for a game of chess, hence why I decided it to be the best possible algorithm to use would be Minimax. A more detailed diagram of Minimax can be seen below:

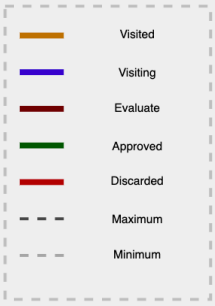




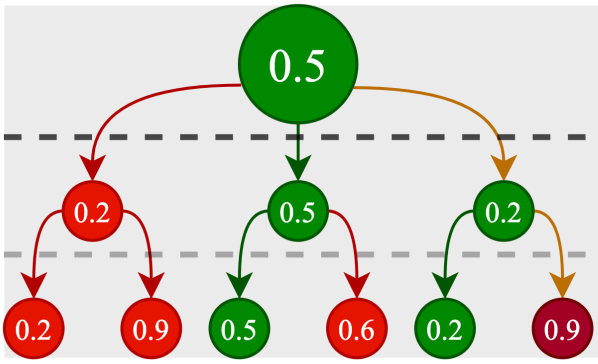


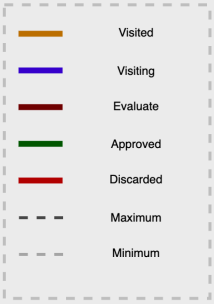
- The infix traversal continues to check the other child nodes formed from other possible moves. These child nodes have the same evaluation function called on them to create other heuristic values.
- If the parent node has an assigned value, an expression is checked.





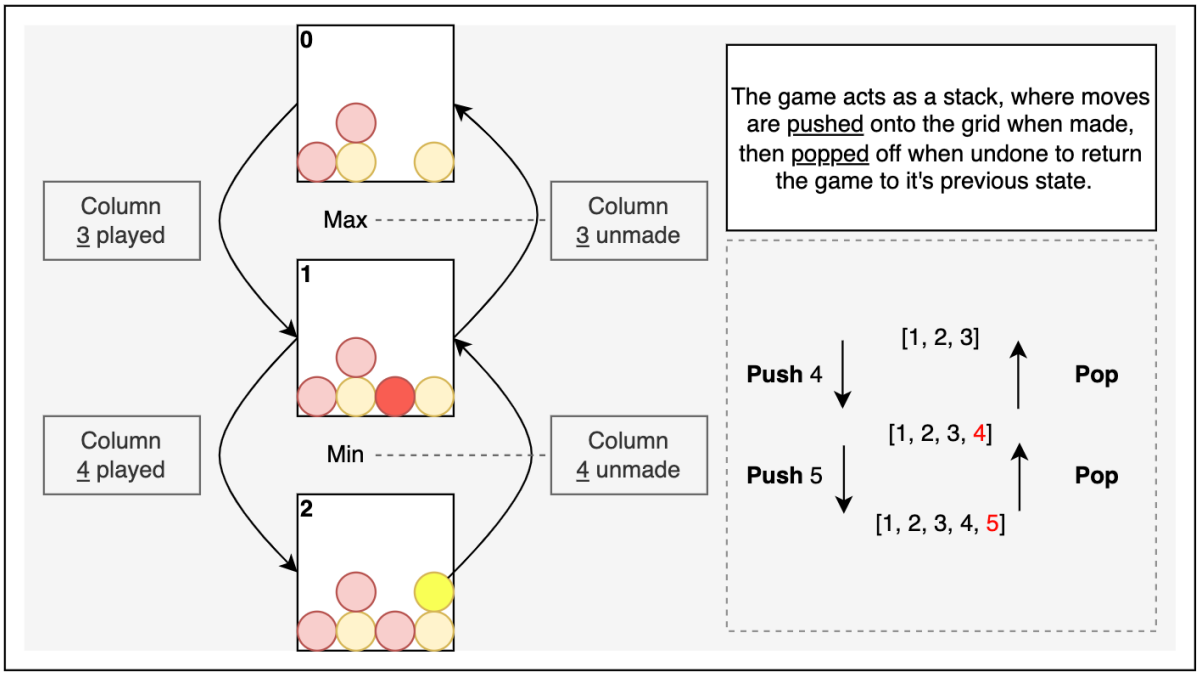
- As this row of game states is considered the *opponent*'s turn, the minimum value is assigned to the parent node, whilst the others are discarded. The move made to gain this heuristic value is stored.
- The comparison between child and parent nodes occur every time the heuristic is calculated for a node.





- The next row of game states is considered to be the player's move, hence why the maximum value is found out of all potential game states, before being assigned to the parent node.
- As a result, once all branches have been searched, each node will contain the move required to reach it's optimal child node.

To get to these different 'states' of the board, I must play a particular move to the Connect 4 grid, yet whenever I am traversing back upwards through this *tree* structure, I must return the score evaluated, whilst also undoing this action. Therefore, this 'making' and 'unmaking' of moves can be viewed as a stack data structure, with moves being 'pushed' and 'popped' to the grid.

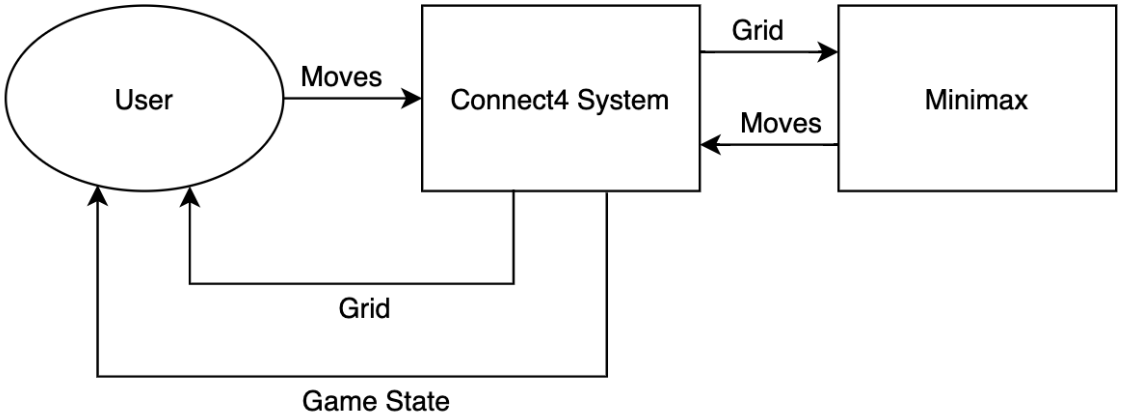


Objectives

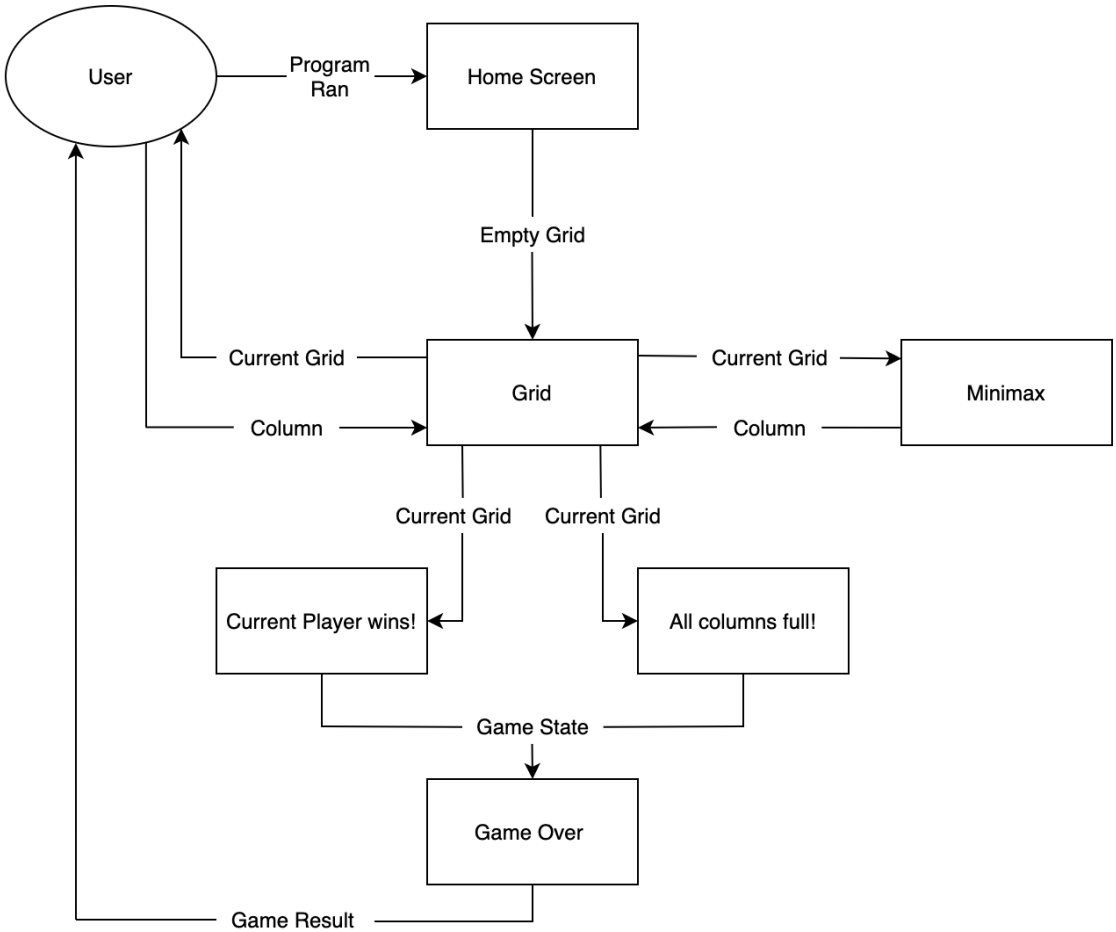
1. When loaded, the screen must display a game of Connect Four with buttons to play a new game or simulate a game.
2. When the Connect Four grid is clicked on, the program must check if it is the player's turn, as well as a valid move, before placing a counter at the correct row in the column where the player clicked and updating the grid.
3. The game should update and check the current turn, using it to manage whether the user can make a move or not.
4. The board must be able to check whether the game has been won and display the victor, or a draw with a visual indicator
5. When the "New Game" Button is pressed, the program must allow the user to select which player they wish to be, store this value, then clear the grid and start a new game with those values.
6. When the "Simulation" Button is pressed, the program must allow the user to select an opponent, as well as the number of rounds of the game to be played, store these values, then clear the grid start a new game with those values.
7. The screen should display a win count of each of the algorithms, as well as their depths/characteristics as the simulation continues.
8. The program should be able to run a minimax algorithm on the current game state to predict a move, optimal in the context of an evaluation heuristic.
9. The program should be able to make the move on the current grid when the minimax algorithm is called.
10. The depth of the Minimax should be specifiable by the user and the program should be able to alter it accordingly.

Project Flow Diagrams

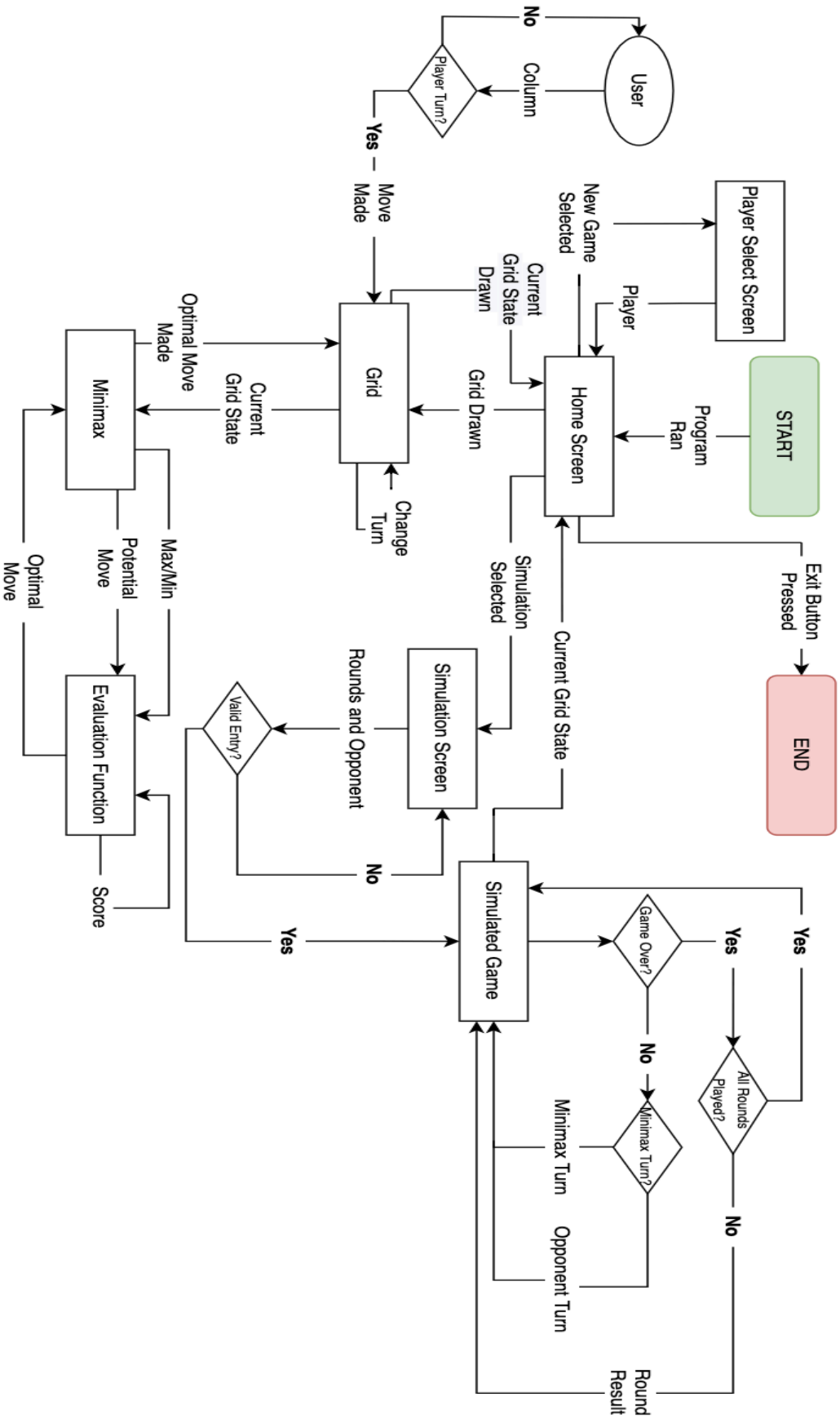
Level 0 Data Flow Diagram:



Level 1



Level 2


User
Column
No
Player Turn?
Yes
Move Made
New Game Selected
Player Select Screen
Player
Current Grid State Drawn
Optimal Move Made
Grid
Minimax
Current Grid State
Grid Drawn
Home Screen
Program Ran
START
Change Turn
Exit Button Pressed
END
Optimal Move
Potential Move
Max/Min
Simulation Selected
Current Grid State
Rounds and Opponent
Simulation Screen
Valid Entry?
Evaluation Function
Score
No
Yes
Simulated Game
Game Over?
Yes
Yes
No
All Rounds Played?
Minimax Turn?
Minimax Turn
Opponent Turn
No
Round Result

Data Sources and Destination

For the current state of my project to work, it will require these inputs of data:

1. State of Current Grid
2. Column of Grid selected
3. Button Selected
4. (If Simulation selected) Rounds of Simulation and Opponent
5. (If New Game selected) Player

And output:

1. Updated State of Grid
2. Minimax Moves
3. (If Simulation selected) Score of Simulation

Data Volumes

Space complexity
I am trying to minimise the amount of data being used in my project to allow it to run smoother on lower quality computers; Mr Huxley asked for the project to be 'portable', implying that it should take up minimal disk space and be easily processed by a laptop.

Stored Data	Limitations	Bytes Required
Grid (Array size 42)	3 Possible States of 1 Integer, 7 Columns, 6 Rows	(7*6)/8 = 5.25 Bytes
Rounds	Max = 50 → 2 Characters	2/8 = 0.25 Bytes
Total per Column (Array size 7)	7 Possible States of 1 Integer, 7 Columns	(7*1)/8 = 0.875 Bytes
Player	Max = 2 → 1 Character	1/8 = 0.125 Bytes

Over time, extra objects such as my UI, or my Minimax Algorithm may take up extra space, therefore I have minimised the data required for a command line GUI game of Connect Four and hope to use all its properties for my actual game. It will take up around 10 Bytes, implying a low space complexity for the program.

Time complexity
Despite no direct instructions from Mr Huxley, I believe that any kind of game in which you must wait a long time for your opponent to move becomes dull quickly, with the optimal 'challenging' game giving you more than enough time to think while also updating the opponent quickly, much like a puzzle. Therefore, I hope to minimise time complexity to create an enjoyable experience for the project; I expect my Minimax to have high complexity and therefore hope to achieve a low complexity in other areas.

Object	Methods	Time Complexity
Connect Four Game	Make/Undo Move – Updates the Grid to a value on a row specified by Total Per Column and updates Total Per Column Create Grid/Total – Creates an array dependent on the number of Columns and Rows in the grid. Check for Win – Checks every possible connect four, only searching the array in areas where it is still possible as to save time. Valid Move – Returns False if the top row of a column is occupied	Make/Undo Move: Relatively uncomplex, as reading a value from a list at any index is time order 1. Create Grid/Total: Time Complexity Order N^2 and N respectively, as the number of columns doubles, so do the number of items in the array. Check for Win: Time Complexity Order less than N, as almost item in the list is checked, yet not all. Valid Move: Order 1
Minimax	Minimax – Recursive Function which travels down nodes of 'potential' game states made by other moves to depth *n* Evaluation - Scans every item in the grid for each node, creating a score relative to 0 depending on how beneficial the game state is	Minimax: Minimax will have to call other functions such as make move and undo move, as well as calling Evaluation function and recursively calling itself, giving it a time complexity of N^2 Evaluation Heuristic: Order N

DESIGN

Project Overview

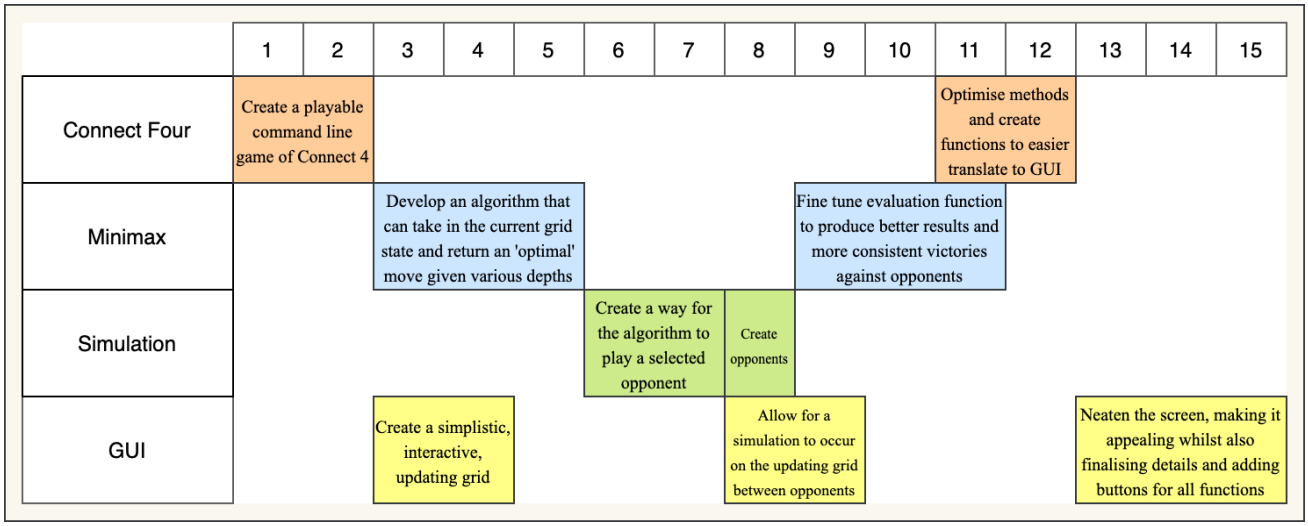
I intend to create a playable game of Connect Four, displayed in a graphical format on a screen. A minimax algorithm should be able to successfully calculate moves and make them on the grid, with the user having the choice on whether to play against it alone, or have it play in a simulation against another algorithm. Throughout the process, all key actions made by the program that are necessary for the user's understanding should be described in a text window of some format.

The software should first be able to create a dynamic game of connect four comprised of data structures and variables, then be able to display it in a graphical format to the user. It should then be able to respond to inputs from the user on the graphical interface, either calling a minimax algorithm, creating windows for a new game or simulation.

Critical Path Design

I have a time frame of around 15 weeks to produce and finish my project for Mr Huxley, hence I have created a basic timeline of how much time I want allocated to each section of my project, what I want to have finalised at the end of each period, as well as identifying the four key elements of my project:

1. Connect Four Game
2. Minimax Algorithm
3. Simulations between Opponents
4. Graphical User Interface



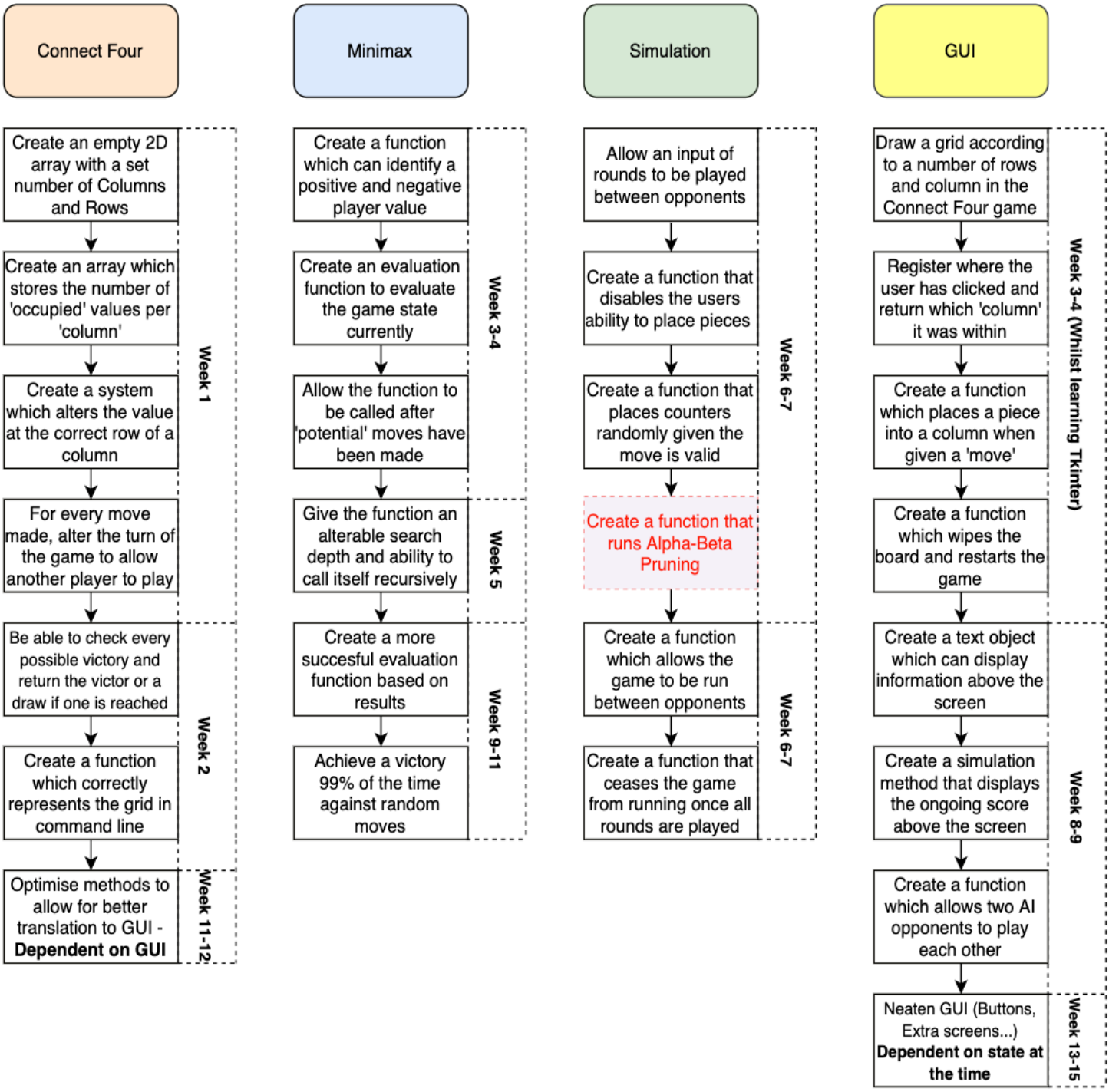
There are sections of the project that I still wish to include, such as:

- Animations
- Alpha-Beta Pruning
- A display of the internal workings of the Minimax Algorithm

Yet under the time restrictions previously mentioned, these are no longer part of the Critical Path, as they add no *crucial* detail to the project itself, they are simply there to make the

experience more enjoyable, not functional. I will still be attempting to work on these in any periods or breaks in which I am able to, such as the relatively calm Week 10, or 13 through to 15. Also, while not mentioned, testing will also be part of the critical path, occurring nearing the very end of Week 15, as I hopefully will have sufficiently validated the project by then.

The path and goals I hope to achieve at each section in depth are explained by the diagram below:



The only section that I am unsure about being the critical path is developing the Alpha-Beta Pruning algorithm, as I believe that it will not add enough to the project to be worth the time, hence why it is highlighted red. However, I will still attempt to achieve it under the time frame.

System Design

Before creating the actual project, I must consider how sections of my project will interact, link, and look. Once I have planned these details, I can begin work on each of them according to the plan.

The Home Screen should have four main components, the **Connect Four grid**, the **text display**, the option to play a **new game** and the option to start a **simulation**.

The grid will display the current game and handle all gameplay, where if the player were to click on the grid, then if it were their turn, then a piece would be played in the selected column, at the correct height in accordance with the Connect Four class. Once the player has clicked, the Minimax Algorithm will then evaluate the current game state with all possible moves to a selected depth to find the move with maximum gain for itself, while minimal gain for the opponent. The player can continue to play the game until an outcome of draw, or victory, after which the grid becomes unplayable, and the grid remains in its state. If a 'simulation' is being played, however, the grid cannot be clicked on by the user, and the game plays out at a regular speed until all rounds of the game have been played.

The text display will output a welcome message once the program has been run, remaining that way until an outcome has been reached in the game, after which it will display the result in the text display. During a simulation, the text will display a 'running total' of the game and the opponents, updating until the game has been completed, after which the final score is registered and displayed, while the board becomes unplayable.

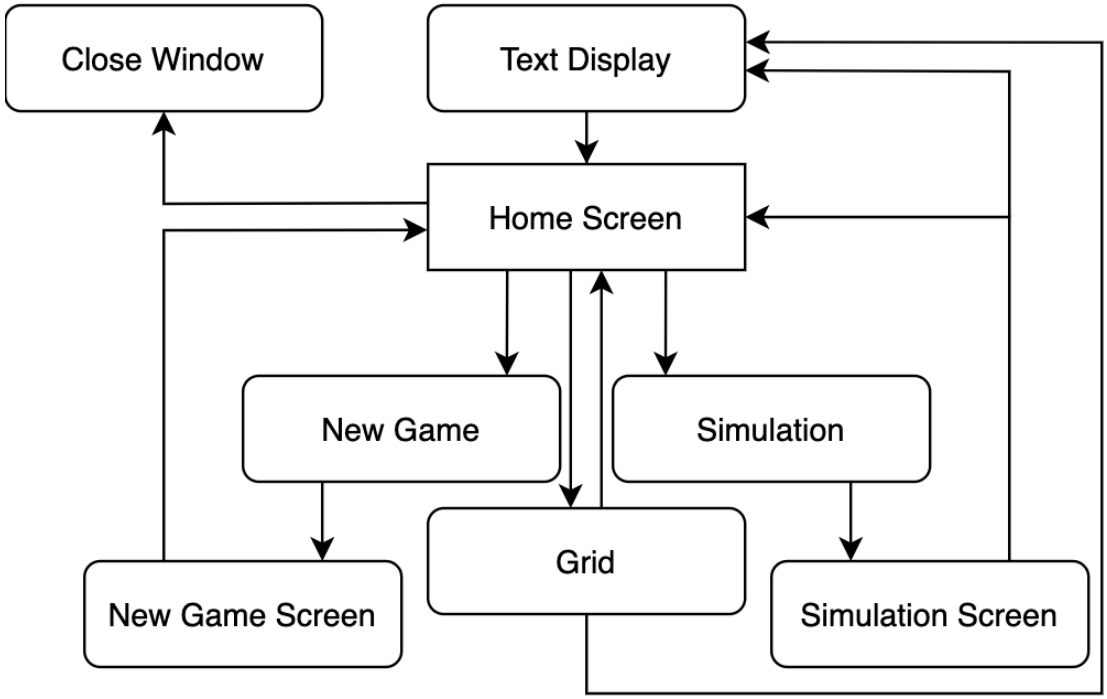
If the New Game button is pressed, a new window opens which allows the user to select which player they wish to be, Player 1 or Player 2, using radio buttons, before hitting a button to confirm their choice. The window will then close, the grid will be wiped and all previous data of the game reset to its original state before the game becomes playable again, with the choice being returned to the Home Screen.

If the Simulation button is pressed, a new window opens which allows the user to select an opponent using radio buttons, as well as enter in a value of rounds to be played into a text box, from 1-50. If any value outside of this range is entered, a default value of 10 is selected in its place. Another button confirms their choice, after which the window is closed, all previous data of the game reset to its original state, and the game begins to play the simulation, during which the player cannot manually change the state of the board.

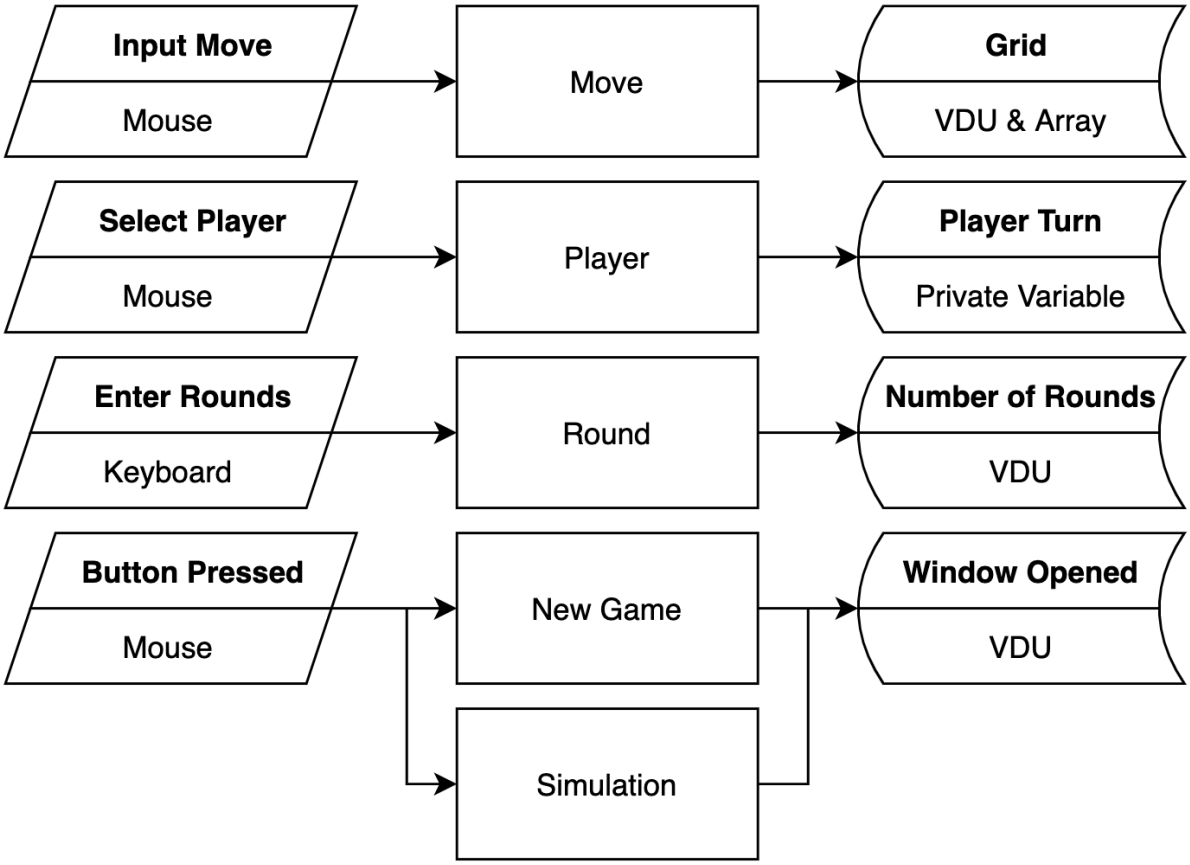
System Security

The system doesn't intend to have any sort of login system, as there wouldn't be much point since Mr Huxley intends to share the program to others interested; no data needs to be stored to begin with. Therefore, no encryption is required, nor do any security processes need to be put in place for this program; no files on anybody's computers should be accessed by the project.

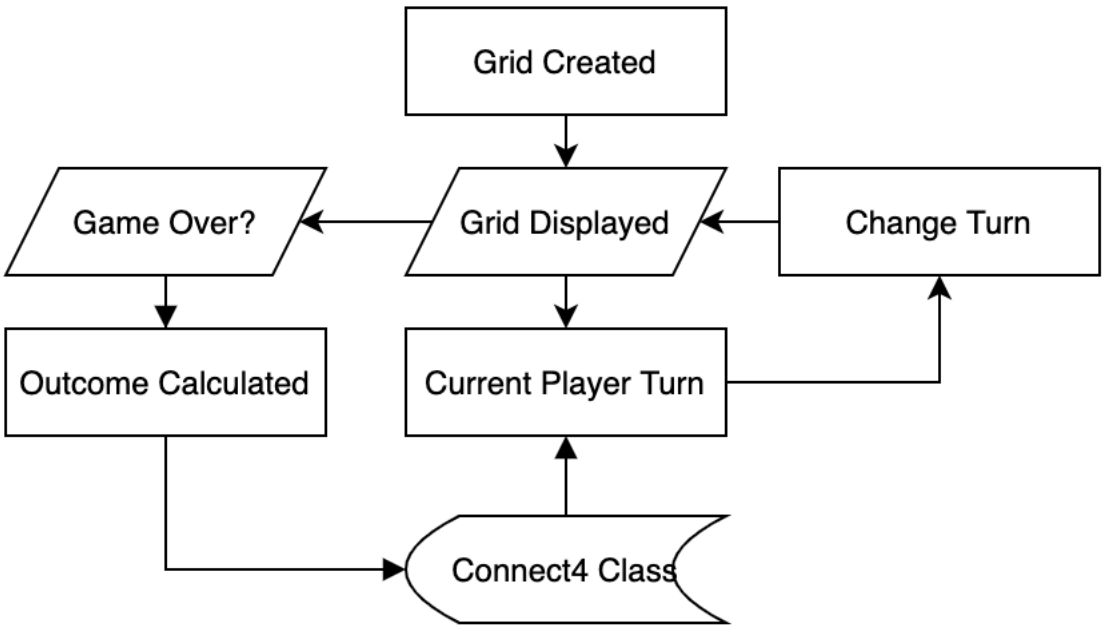
Modular System Design



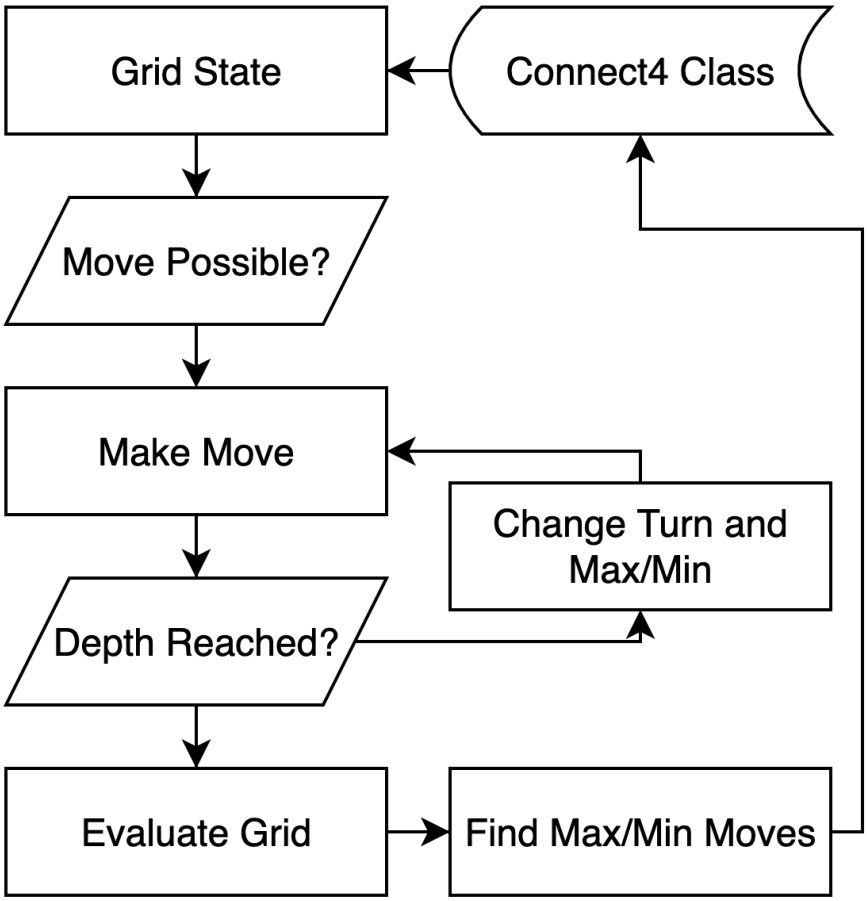
General System Flow Chart



System Flow Chart for Playing the Board

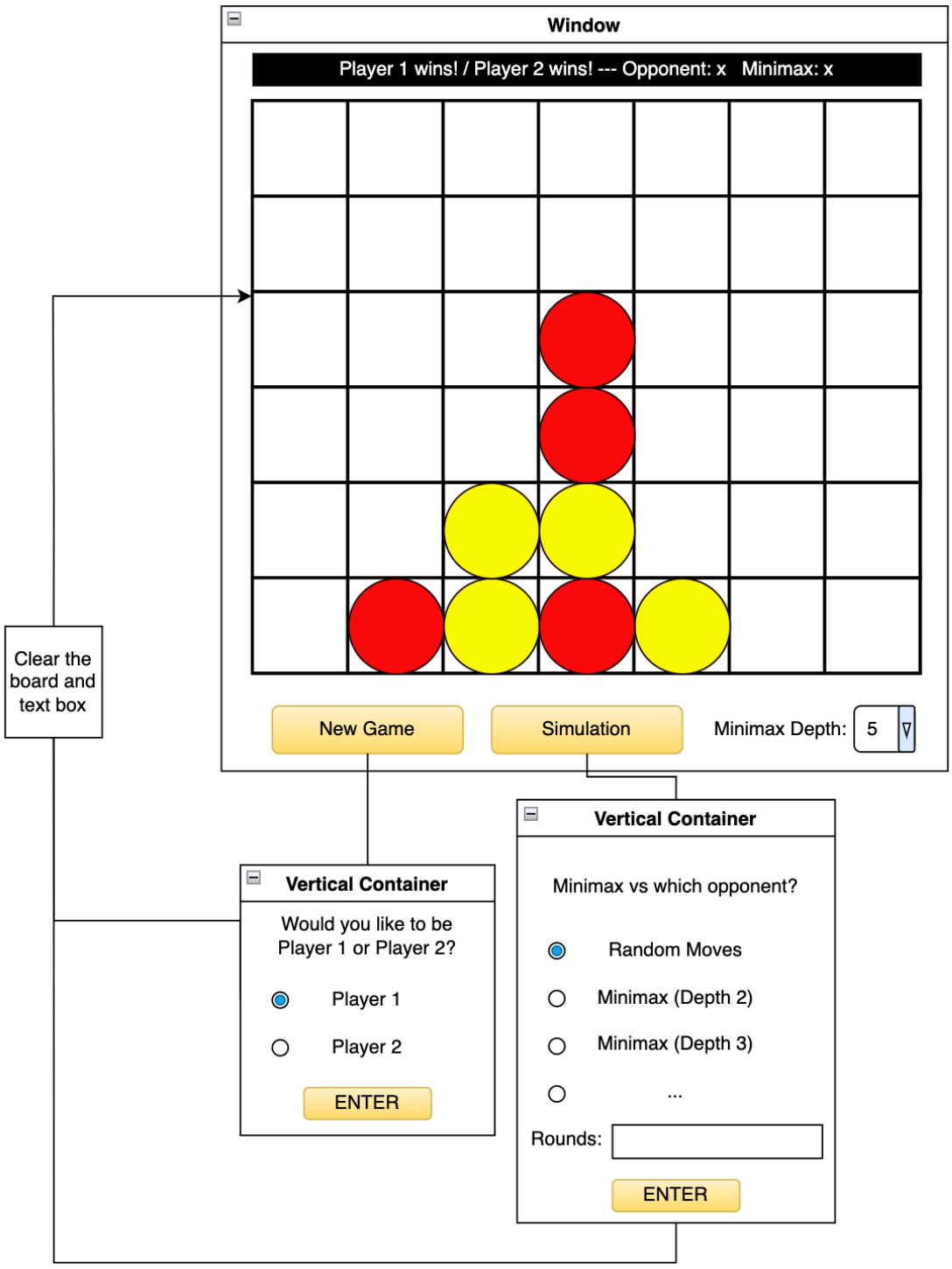


System Flow Chart for Minimax Algorithm



Interface Design

From my look into similar systems, I have decided to make the Interface appear like the diagram below, which displays the smaller windows created by the buttons, the buttons, and an example game of Connect Four on the system, which may not be too dissimilar from what Mr Huxley would see in the final product.



The primary focus of black and white colours, as well as a very simplistic, uncoloured grid was based off my previous look into similar systems[4], as I found the more minimalist view far more appealing than a distracting colour scheme; this not only benefits the game, yet also allows the user to observe and appreciate Minimax moves made without anything taking them away from the action. The two 'vertical containers' will be smaller screens, in the New Game button's example, being extremely small, as to not distract the user. Rather than implement animations within the text box such as scrolling, I have instead decided to let it update naturally, as whilst animations would be useful, they would become frustrating to wait for after a while; as well as cost extra time to learn how to program. As for adjusting the depths at which the minimax algorithm operates, I found that only being able to alter this for each game may become clunky, and it would be far more interesting to be able to visibly see the difference at whatever point you wish to. This basic UI should help the user grasp exactly what is happening, whilst compacting all key information within one text box rather than splitting it will make understanding of how an 'optimal game' is calculated, all of which should adhere to Mr Huxley's guidelines.

Algorithms

Checkwin:

This algorithm will check whether the game has reached an outcome, such as a victory, or a draw. If either has occurred, the algorithm will update the result property of the game to whichever value corresponds to the game result, before returning True. Otherwise, the algorithm will return False.

```
Grid as 2D Array
Tot as Array
FOR all rows in Grid
      FOR the first four columns in Grid
            IF item at (row, column) equal to (row, column+1) equal
            to (row, column+2) equal to (row, column+3) THEN
                  RETURN True
                  IF item at row, column == 1 THEN
                        Result = 1
                  ELSE
                        Result = 2
                  ENDIF
            ENDIF
      ENDFOR
ENDFOR
FOR all columns in Grid
      FOR the first three rows in Grid
            IF item at (row, column) equal to (row+1, column) equal
            to (row+2, column) equal to (row+3, column) THEN
                  RETURN TRUE
                  IF item at row, column == 1 THEN
                        Result = 1
                  ELSE
                        Result = 2
                  ENDIF
            ENDIF
```

[4] Can be found on Page 4

```
      ENDFOR
ENDFOR
FOR the last four rows in Grid
      FOR the first four columns in Grid
            IF item at (row, column) equal to (row-1, column+1) equal
            to (row-2, column+2) equal to (row-3, column+3) THEN
                  RETURN TRUE
                  IF item at row, column == 1 THEN
                        Result = 1
                  ELSE
                        Result = 2
                  ENDIF
            ENDIF
      ENDFOR
ENDFOR
FOR the last four rows in Grid
      FOR the last four columns in Grid
            IF item at (row, column) equal to (row-1, column-1) equal
            to (row-2, column-2) equal to (row-3, column-3) THEN
                  RETURN TRUE
                  IF item at row, column == 1 THEN
                        Result = 1
                  ELSE
                        Result = 2
                  ENDIF
            ENDIF
      ENDFOR
ENDFOR
IF any columns in Tot != number of rows in Grid
      RETURN FALSE
ENDIF
Result = 3
RETURN TRUE
```

```
55  def checkwin(self):
56      for row in self.grid:
57          for col in range(Connect4.COLS-3):  #Horizontal
58              if len(set(row[col:col+4])) == 1 and row[col] != 0:
59                  self.result = Result.P1WIN if self.turn == 1 else Result.P2WIN
60                  return True
61      for row in zip(*self.grid): #Vertical
62          for col in range(3):
63              if len(set(row[col:col+4])) == 1 and row[col] != 0:
64                  self.result = Result.P1WIN if self.turn == 1 else Result.P2WIN
65                  return True
66      for row in range(3,Connect4.ROWS):  #Diagonal
67          for col in range(0,4):
68              if self.grid[row][col] == self.turn and self.grid[row][col] == self.grid[row-1][col+1] and self.grid[row
69                  self.result = Result.P1WIN if self.turn == 1 else Result.P2WIN
70                  return True
71      for row in range(3,Connect4.ROWS):
72          for col in range(3, Connect4.COLS):
73              if self.grid[row][col] == self.turn and self.grid[row][col] == self.grid[row-1][col-1] and self.grid[row
74                  self.result = Result.P1WIN if self.turn == 1 else Result.P2WIN
75                  return True
76      for i in range(Connect4.COLS):
77          if self.tot[i] != Connect4.ROWS:
78              self.result = Result.NONE
79              return False
80      self.result = Result.DRAW
81      return True
```

These two run-off lines follow the pattern shown in the rest of the line.

Minimax:

This algorithm is intended to take the current game state, a current depth, a maximum depth, and a maximising player. Until the maximum depth has been reached, the algorithm makes all potential moves on the grid, alternating the turns as a normal game would. Once the maximum depth is reached, the game is evaluated and the evaluation heuristic is returned and compared. All evaluation functions are called from the maximising player's point of view, therefore the code also checks the current turn to decide whether it is looking for the *optimal* (max), or *inoptimal* (mini) move.

```
Depth, MaxDepth, MaximisingPlayer, BestMove as Integer
Game as Game Object
IF Game outcome reached THEN
      IF outcome equals MaximisingPlayer wins THEN
            RETURN +inf
      ELIF outcome equals Draw
            RETURN 0
      ELSE
            RETURN -inf
      ENDIF
ELIF Depth equals MaxDepth
      RETURN heuristic value of game grid from MaximisingPlayer POV
ELSE
      IF Player equals MaximisingPlayer THEN
            BestScore = -inf
      ELSE
            BestScore = +inf
      ENDIF
      FOR every valid move for the current grid
            Make move on grid
            Score = Call Minimax(Depth += 1, MaxDepth = MaxDepth,
            MaximisingPlayer = MaximisingPlayer)
            Undo move on grid
            IF MaximisingPlayer equals current turn THEN
                  IF Score greater than BestScore THEN
                        BestScore = Score
                        IF Depth equals 0
                              BestMove = move
                        ENDIF
                  ENDIF
            ELSE
                  IF Score less than BestScore THEN
                        BestScore = Score
                        IF Depth equals 0
                              BestMove = move
                        ENDIF
                  ENDIF
            ENDIF
      ENDFOR
ENDIF
```

```
RETURN BestMove
```

```
13      def minimax(self, depth, max_depth, maximising_player):
14          if self.game.game_over():
15              if self.game.result != Result.DRAW:
16                  res = 1 if self.game.result == Result.P1WIN else 2
17                  return 500 if res == maximising_player else -500
18              else:
19                  return 0
20
21          elif depth == max_depth:
22              return self.evaluation(self.game.grid, maximising_player)
23
24          else:
25              best_score = -500 if maximising_player == self.game.turn else 500
26              for i in range(self.game.COLS):
27                  if self.game.valid_move(i):
28                      self.game.make_move(i)
29                      score = self.minimax(depth+1, max_depth, maximising_player)
30                      self.game.undo_move(i)
31                      if maximising_player == self.game.turn:
32                          if score > best_score:
33                              best_score = score
34                              if depth == 0:
35                                  self.best_move = i
36                      else:
37                          if score < best_score:
38                              best_score = score
39                              if depth == 0:
40                                  self.best_move = i
41
42              return best_score
```

Canvas_click:

The Algorithm is designed to register where the player has clicked on the grid, if it is their turn, and if so, register their selected column. From there, it updates the grid from the game object and checks if the game has been won, before updating the grid and making a minimax move and updating the grid again.

```
Game as Game Object
X, Y as Event
Canvas as Canvas
Player, Turn, Move as Integer
IF Game outcome not reached THEN
      IF X in Canvas and Y in Canvas THEN
            Move = Column clicked
            IF Move valid THEN
                  Make Move on grid of Game
                  IF Call Checkwin(grid = grid) equals True THEN
                        Display result text
                  ENDIF
                  Draw grid
```

```
                Change turn of Game
            IF turn of Game not equal to Player
                Make Minimax move
            ELSE
                Display column full message
            ENDIF
        ENDIF
ENDIF
```

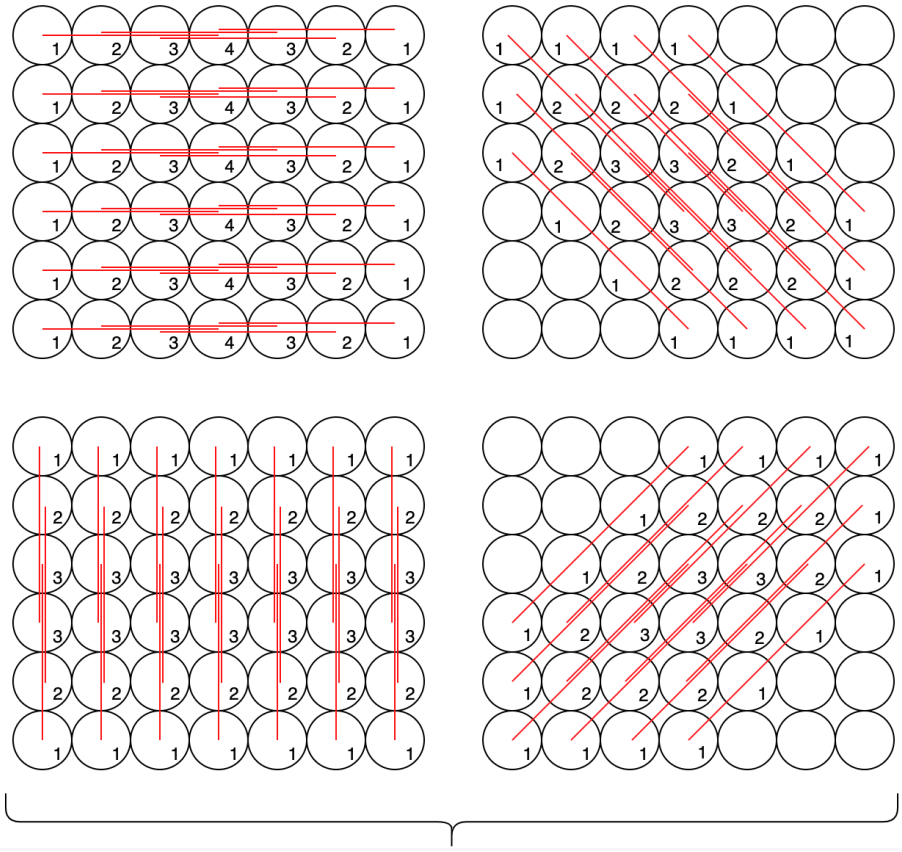
```python
def canvas_click(self, event):
    if not self.game.game_over() and self.game.turn == self.player and self.mode == True:
        if 0 < event.x < App.CANVAS_WIDTH and 0 < event.y < App.CANVAS_HEIGHT:
            move = self.find_column(event.x)
            if self.game.valid_move(move):
                self.game.make_move(move)
                self.draw()
                self.root.update_idletasks()
                if self.game.game_over():
                    self.final_result()
                else:
                    self.root.text.delete('1.0', '100.0')
                    self.root.text.insert('1.0', 'Minimax thinking...')
                    self.root.text.tag_add("tag_name", "1.0", "end")
                    self.root.update_idletasks()
                    self.root.after(500, self.min_max_move(self.max_depth.get()))
                    self.root.text.delete('1.0', '100.0')
                    self.draw()
                    if self.game.game_over():
                        self.final_result()
            else:
                self.root.text.delete('1.0', '100.0')
                self.root.text.insert('1.0', 'That column is full.')
                self.root.text.tag_add("tag_name", "1.0", "end")
```

Evaluation:

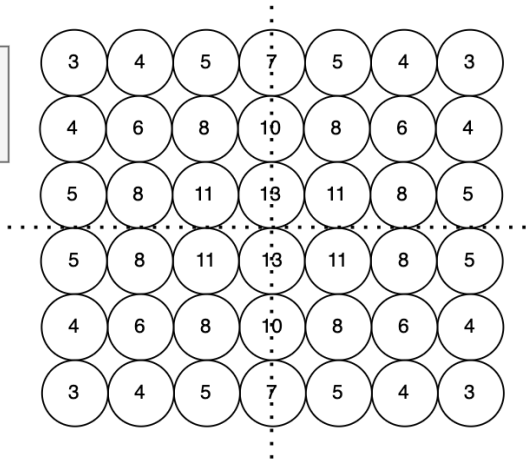
This algorithm takes the current board state fed to it and performs a calculation based on beneficial and negative states of the board, returning a value relative to whether the state is overall positive or negative to the player. As this was originally created by me, I have added an explanation below.

I have considered what may be considered an 'optimal' game state, and a 'negative' game state, and found that other than having three or two counters in a row, this is difficult to calculate. Instead of this, I tackled a more abstract reasoning: '**If the player were to place a piece here, how many of the opponent's possible victories would he cut off?**', which is the same as '**If the player were to play here, how many victories could they achieve from this point?**'. However, to begin this, I would need to find the total number of connect four's that each position on the board was involved with:

By considering every possible connect 4 that can be made from each point on the grid



We can calculate the optimal moves to minimise the possible victories of our opponent.



```
Game as Game Object
Maximising_player as Integer
Position_scores =
[[3,4,5,7,5,4,3],[4,6,8,10,8,6,4],[5,8,11,13,11,8,5],
[5,8,11,13,11,8,5],[4,6,8,10,8,6,4],[3,4,5,7,5,4,3]]
Player = Opponent = 0
FOR every row in grid of Game
      FOR every column in grid of Game
            IF item at row and column in grid of game equals
Maximising_player THEN
                  Player += row, column in Position_scores
            ENDIF
            IF item at row and column in grid of game doesn’t equal
Maximising_player or empty THEN
                  Opponent += row, column in Position_scores
            ENDIF
      ENDFOR
ENDFOR
RETURN Player - Opponent
```

```python
    def evaluation(self, grid, maximising_player):
        positions = [
        [ 3, 4, 5, 7, 5, 4, 3],
        [ 4, 6, 8,10, 8, 6, 4],
        [ 5, 8,11,13,11, 8, 5],
        [ 5, 8,11,13,11, 8, 5],
        [ 4, 6, 8,10, 8, 6, 4],
        [ 3, 4, 5, 7, 5, 4, 3]
        ]

        player_score = opponent_score = 0

        positions = [score for row in positions for score in row]
        grid = [piece for row in grid for piece in row]

        for i in range(len(positions)):
            if grid[i] == maximising_player:
                player_score += positions[i]
            if grid[i] != 0 and grid[i] != maximising_player:
                opponent_score += positions[i]

        return player_score - opponent_score
```

Class Definitions

Rather than create a 'player' base class under which I can create players and opponents, I created a *Connect4* Object, which contains the grid, as well as other properties required to play a full game of Connect Four, such as the current turn or the outcome of the game. From there, I added subroutines which are required to make moves and complete the game against another opponent, or validation checks which can be used by any other object in the class for ease of access. I did this as there was no need to create different validation checks for different opponents and creating many different opponent objects would be far more inefficient than simply allowing the user to parse arguments into a singular object.

The *Result* Object is an Enumeration, whose values are solely used to indicate an outcome of the game, which is assigned to the Connect4 object upon a result being achieved. The value 0 is assigned until this occurs, with no subroutines in the Enumeration itself.

As the inputs and outputs of my Minimax would be theoretically a grid, then a move being returned, I believed it would be easy to create a *Minimax* object under which all necessary information could be parsed. It contains the current game state and a randomly assigned 'best move', which can then be overwritten via calculation with certain moves and the evaluation function storing it in a property Best Move. Minimax requires three subroutines, one of which calls itself recursively to continue searching down the tree of possible moves to a maximum depth, another evaluates the grid parsed into it, which can be done after theoretical 'moves' have been made and unmade on it, and the final generates a random valid move to use for initial storage of a 'best move'.

To run the project in a GUI and potentially add animations, I created an *App* object, which creates a window stored at property of root. The property of player is used to draw the correct colour per turn, as done with counters in regular Connect Four, meanwhile the properties of rounds is used doubly for storing the number of rounds of a simulation, yet also for differentiating whether a simulation is occurring.

Class Name	Properties	Description
Connect4	Grid (2D Array)	Stores a 2D Array of 0's to signify empty spaces of a grid, gradually updating itself as moves are played by inserting the value of the current turn into the array according to the value of Tot at the same index.
	Tot (Array)	Stores an array of the total pieces in each column of the grid.
	Turn (Int)	Stores an integer value of the current turn.
	Result (Int)	Stores an integer from the Result class and updates to suggest different states of the game, such as Ongoing, Player 1 Win, Player 2 Win or Draw.
Result	NONE (Enum)	Integer 0, to be referenced when the game is still ongoing.
	P1WIN (Enum)	Integer 1, to be referenced when the game is won by Player 1.
	P2WIN (Enum)	Integer 2, to be referenced when the game is won by Player 2.
	DRAW (Enum)	Integer 3, to be referenced when the game is a draw.
Minimax	Game (Object)	Stores the game object from which properties and functions of Connect4 can be referenced or called.
	Best Move (Int)	Starts as an integer random move yet updates itself depending on the result of the evaluation function and whether the object is minimising or maximising.
App	Player (Int)	Stores the integer current player according to the Connect4 class, as well as later storing the selection from player select.
	Rounds (Int)	Begins as None, yet later stores the integer number of rounds entered by the user, defaulting to 10 if an invalid entry is detected
	Root	The TopLevel widget from which the main window of the application is displayed (Home Screen), and the mainloop() is called.
	Game (Object)	Stores the current Connect4 game, with all of its properties and functions.
Piece	R (Int)	Stores the integer radius of a piece to be referenced.

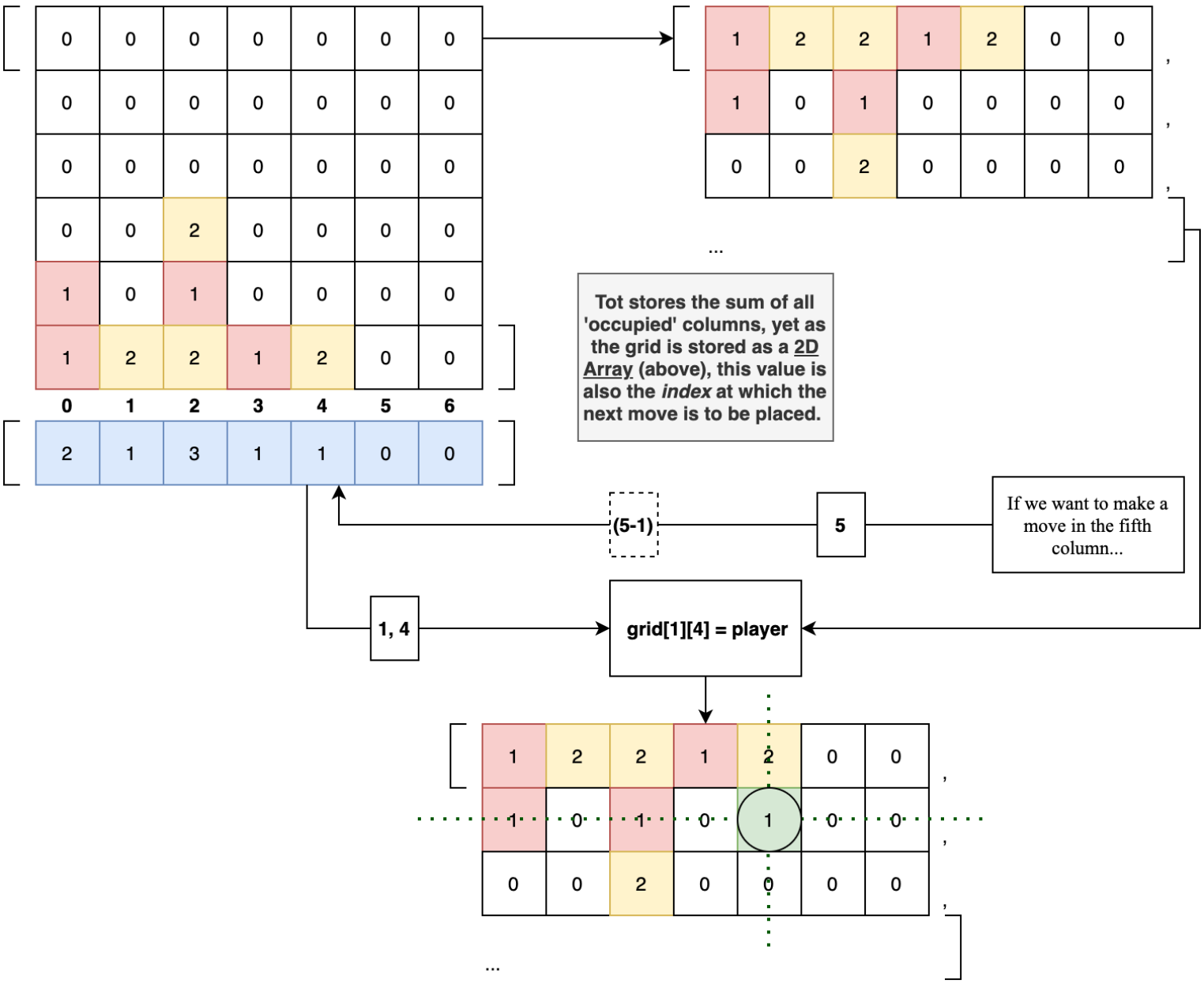
Class Name	Subroutine	Description
Connect4	create_grid	Creates an 2D Array of 6 arrays of 7 zeroes to represent the grid, before being returning it
	return_grid	Returns the current grid.
	create_tot	Creates an array of 7 zeroes to represent the number of pieces in each column.
	valid_move	Returns False only if the selected column's index of tot is equal to the number of rows.
	change_turn	Changes the value of the turn.
	checkwin	Returns True and alters the value of result to display the outcome of the game only if an outcome is detected.
	make_move	Alters a value of the grid in the selected column at the row specified by the value at the column index of tot from zero to the current turn and increments the value in tot by +1.
	undo_move	Increments the value in tot by -1, then alters a value of the grid in the selected column at the row specified by the value at the column index of tot from the current turn to zero.
	game_over	Returns True if result isn't equal to the NONE property of the Enumeration.
	grid_clear	Calls create_tot and create_grid and assigns them to the grid and tot of the current board.
Minimax	minimax	Checks whether the game is over or not, or if maximum search depth has been achieved, evaluating dependent on the result of each possible outcome, with beneficial outcomes for the minimax being positive, whilst negative outcomes are negative. If neither of these outcomes have been achieved, then the function finds every possible move from the inputted game state, before recursively calling itself on each of the game states created, with the minimising and maximising players reversed until it reaches the aforementioned base case. The function then compares the score of the evaluation function dependent on the 'player' and 'opponent' of the Minimax class as to calculate the most beneficial move for the player, whilst the least beneficial move for the opponent, returning the move which allows them to reach this state.[5]
	evaluation	Returns a positive or negative value dependent on the grid state, with positive being a beneficial state for the player, and negative being the opposite.

[5] Diagram on Page 10-11

	random_move	Returns a random valid move for a grid inputted into it, used for creating an initial 'best move' within the minimax function.
App	draw	Clears the canvas which the draw_grid and draw_pieces subroutine draws on, before calling both functions
	new_game	Sets the object game property to a new Connect4 game.
	delay	Runs a function after a break of 100 milliseconds whilst isolating it from the rest of the main loop, meaning that the function must be completed before any other lines of code are run.
	final_result	Updates the text display to show the result of the game.
	s_final_result	Updates the text and running totals of a simulation score for both opponent and minimax, before returning them.
	canvas_click	When the screen is clicked by the user, the function checks whether it is the user's turn, before finding the column clicked and checking if the result is a valid move. If it is, then the move is made, drawn, and the turn is changed. If it isn't, then the text is updated to inform the user that the column is full.
	simulation	Decides which opponent the player has selected, before playing a game between the two at a faster speed than usual, checking whether either side has one per move.
	random_move	Returns a random number out of the available moves in the current grid.
	min_max_move	Parses a player, opponent, and game into a minimax object, before calling minimax function on it, making the resultant move.
	draw_grid	Draws lines along the canvas according to the number of rows and columns in the grid
	draw_pieces	For every value in the grid that is occupied (not equal to 0), call the circle function
	find_column	Returns the x co-ordinate parsed into the function floor divided by the width of the canvas floor divided by the number of columns in the grid.
	circle	Draws a circle with radius r and a colour according to the current player at a given x and y co-ordinate.
	return_to_game	Clears and redraws the grid, setting the player property of the game, as well as either calling a simulation or normal game depending on the property 'mode'.
	player_select_screen	Creates a new window with radiobuttons allowing the user to select which player they

		wish to be, with another button which allows them to close the window and return the selections to the Home Screen.
	s_player_select_screen	Creates a new window with radiobuttons allowing the user to select which opponent they wish to see, a textbox allowing the user to enter in the number of rounds they want to see played, and another button which allows them to close the window and return the selections to the Home Screen.
Piece	R	Stores the radius of a piece to be referenced.

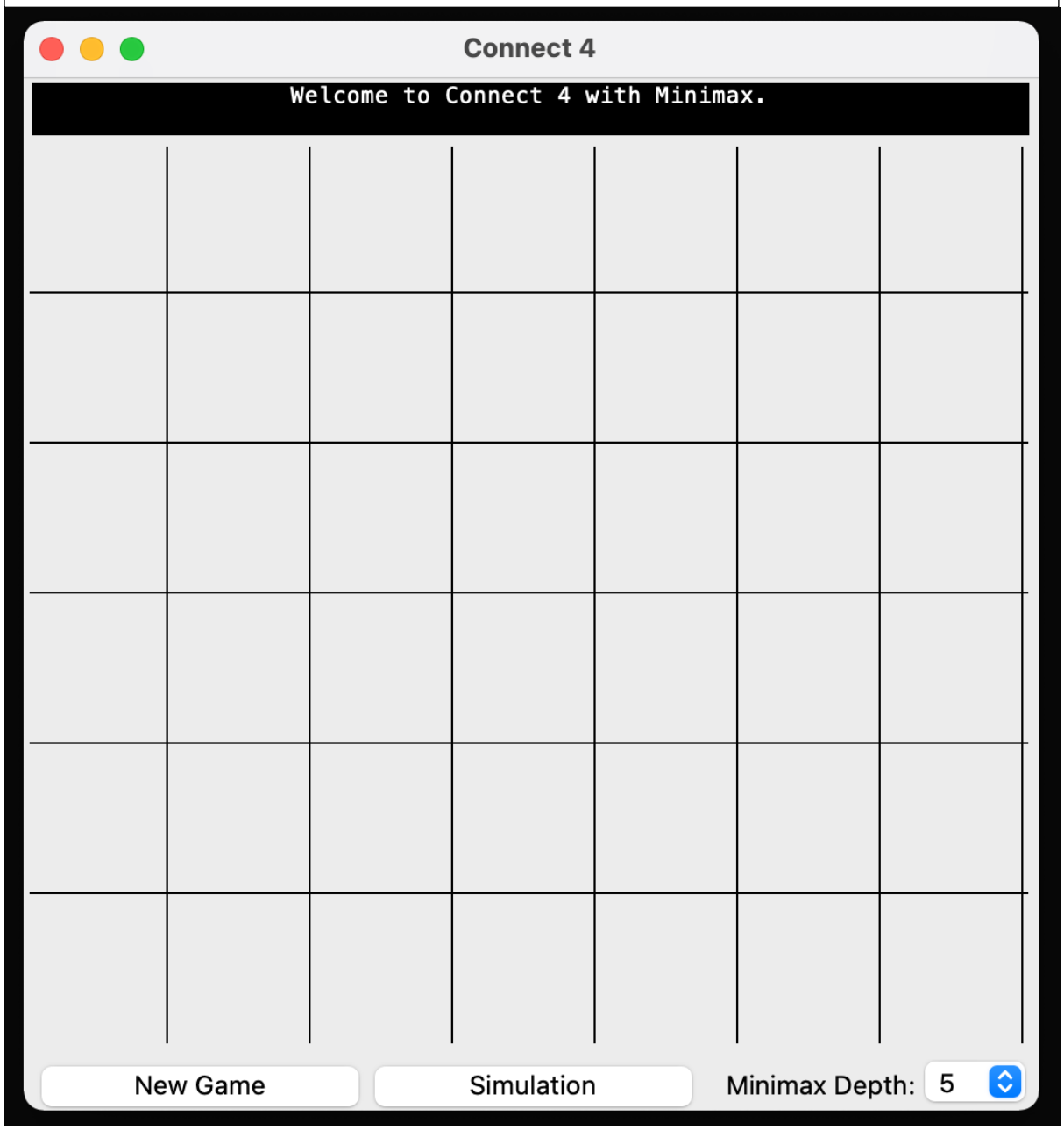
I find that I may not have established exactly how 'Tot' is expected to work, and have created a helpful diagram below to explain:



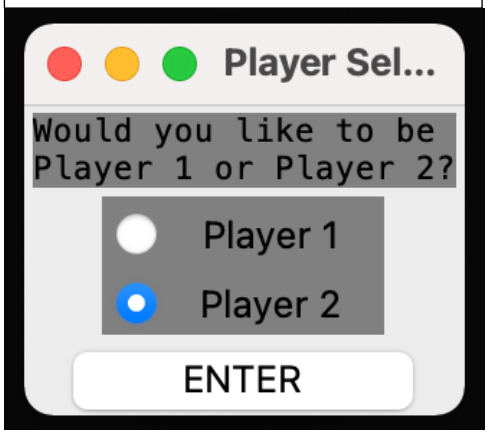
TECHNICAL SOLUTION

Design

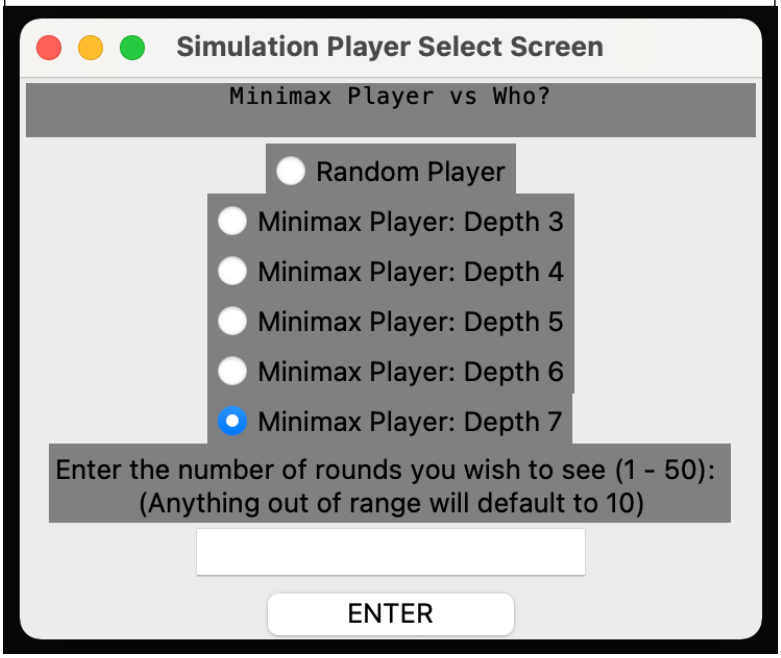
The initial state of the Home Screen window once the program is ran



The secondary window created once New Button is selected



The secondary window created once Simulation is selected



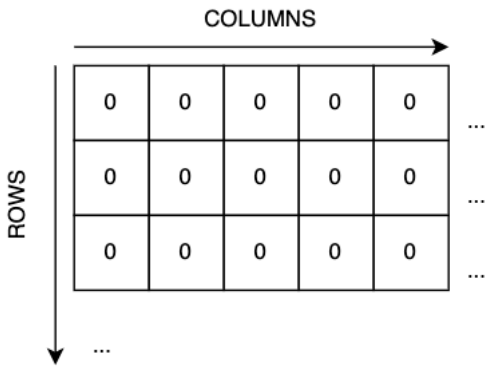
Implementing the Game Class

Creating the grid

Initially, I approached the game class by attempting to make a very easily *alterable* program, hence why for my grid I implemented class variables of Rows and Columns. I decided to represent the grid as a 2D array as such:

```
def create_grid(self):
        grid = []
        for i in range(Connect4.ROWS):
            grid.append([])
            for j in range(Connect4.COLS):
                grid[i].append(0)
        return grid
```

This meant that for every *row* specified, a new list would be added to the array, then filled with with a zero (representing an empty space) for every column. As a result, we create a 2D array as such.



As the concept of 'tot' has already been explained on the page above, you can see that I decided to make moves play to the *top* of this grid, rather than the bottom, as to allow for easier referencing.

Checking for an Outcome

To check for an outcome within the grid, I found that there are technically only three possibilities. Either a Victory, Draw, or an Ongoing Game. To represent this, I implemented an Enumeration to represent the possibilities.

```
class Result(Enum):
    NONE = 0
    P1WIN = 1
    P2WIN = 2
    DRAW = 3
```

Whilst the Victory is split into P1Win and P2Win, these are only decided by turn once it has been decided that a Victory has occurred. To check whether the game had reached an outcome:

1. We see whether any four adjacent positions in the grid are of the same type and not equal to zero (empty). I decided to check Horizontally, Vertically, then on either diagonal.

    1a. Horizontally and Vertically, I used set() and a set length of indexes, as if and only if all of the values within that index range were the same and not equal to zero would a victory be true. The correct result was then set using the enumeration and the current turn.

    1b. For the diagonals, I noticed that for each direction (left or right), there reaches a certain row and column at which no diagonals can be formed in this direction, therefore I altered the loops for checking each value to only check for the values in which diagonals are possible, improving efficiency.

2. We see whether the grid is completely full, in which case all values within the 'tot' list must equal the total number of rows.

```
def checkwin(self):
        for row in self.grid:
            for col in range(Connect4.COLS-3):  #Horizontal
                if len(set(row[col:col+4])) == 1 and row[col]
!= 0:
                    self.result = Result.P1WIN if self.turn ==
1 else Result.P2WIN
                    return True
        for row in zip(*self.grid):
            for col in range(3):
                if len(set(row[col:col+4])) == 1 and row[col]
!= 0:
                    self.result = Result.P1WIN if self.turn ==
1 else Result.P2WIN
                    return True
        for row in range(3,Connect4.ROWS):
            for col in range(0,4):
```

```
                if self.grid[row][col] == self.turn and
self.grid[row][col] == self.grid[row-1][col+1] and
self.grid[row-1][col+1] == self.grid[row-2][col+2] and
self.grid[row-2][col+2] == self.grid[row-3][col+3]:
                    self.result = Result.P1WIN if self.turn ==
1 else Result.P2WIN
                    return True
        for row in range(3,Connect4.ROWS):
            for col in range(3, Connect4.COLS):
                if self.grid[row][col] == self.turn and
self.grid[row][col] == self.grid[row-1][col-1] and
self.grid[row-1][col-1] == self.grid[row-2][col-2] and
self.grid[row-2][col-2] == self.grid[row-3][col-3]:
                    self.result = Result.P1WIN if self.turn ==
1 else Result.P2WIN
                    return True
        for i in range(Connect4.ROWS):
            if self.tot[i] != Connect4.ROWS:
                self.result = Result.NONE
                return False
        self.result = Result.DRAW
        return True
```

Whilst self.result stores the game's outcome, the function will always return True or False depending on whether the game is over or not. This allowed me to easily check and update whether the game should still be 'playable'.

Restarting the Game

Whilst making this game class, I was aware that I would require the ability to create a new game entirely, therefore I implemented a function that would reinstate most of the class properties to their defaults, which could easily be done using functions previously used to create them.

```
    def grid_clear(self):
        self.grid = self.create_grid()
        self.tot = self.create_tot()
        self.result = Result.NONE
        self.turn = Connect4.P1
```

Connect4.py

```
from enum import Enum

class Result(Enum):
    """ An Enumerator referenced by the Connect4 class to
differentiate between different 'states' of the game.
    """
    NONE = 0
    P1WIN = 1
    P2WIN = 2
```

```
    DRAW = 3

class Connect4:
    """ A playable game of Connect4 with functions allowing it
to be played with validation checks to find valid inputs and
outcomes to the game. It updates properties of the object to
reflect the current state of the game.
    """
    P1 = 1
    P2 = 2
    COLS = 0
    ROWS = 0

    def __init__(self, COLS, ROWS):
        """ Initialises the game object, creating a grid, a
total occupied spaces per column, the current turn and the
result of the game.

        Args:
            COLS (int): The number of columns desired in the
grid
            ROWS (int): The number of rows desired in the grid
        """
        Connect4.COLS = COLS
        Connect4.ROWS = ROWS
        self.grid = self.create_grid()
        self.tot = self.create_tot()
        self.turn = Connect4.P1
        self.result = Result.NONE

    def create_grid(self):
        """ Returns a 2D array with ROWS number of rows and
COLS number of columns, filled with zeroes, which represent
empty spaces in the grid.

        Returns:
            2D array: An array containing ROWS arrays, each
filled with COLS zeroes.
        """
        grid = []
        for i in range(Connect4.ROWS):
            grid.append([])
            for j in range(Connect4.COLS):
                grid[i].append(0)
        return grid

    def return_grid(self):
        """ Returns the grid.
```

```
        Returns:
            2D array: Represents the grid
        """
        return self.grid

    def create_tot(self):
        """ Returns an array of zeroes, COLS items long,
representing the number of counters in each column of the
grid.

        Returns:
            array: An array containing COLS zeroes
        """
        tot = []
        for i in range(Connect4.COLS):
            tot.append(0)
        return tot

    def valid_move(self, move):
        """ Returns False if the column specified by move is
full or outside of the grid.

        Args:
            move (int): The column in which a move is trying
to be made

        Returns:
            bool: Whether or not the move is valid
        """
        if self.tot[move] == Connect4.ROWS or move >
Connect4.COLS-1 or move < 0:
            return False
        return True

    def change_turn(self):
        """ Changes the current turn
        """
        self.turn = Connect4.P2 if self.turn == Connect4.P1
else Connect4.P1

    def checkwin(self):
        """ Checks whether an outcome has been achieved.
Checks if either all columns are full or if four adjacent
spaces within the grid are not empty and are the same colour
horizotally, vertically, or diagonally.

        Returns:
            boolean: Whether or not an outcome has been
reached
```

```
        """
        for row in self.grid:
            for col in range(Connect4.COLS-3):
                if len(set(row[col:col+4])) == 1 and row[col]
!= 0:
                    self.result = Result.P1WIN if self.turn ==
1 else Result.P2WIN
                    return True
        for row in zip(*self.grid):
            for col in range(3):
                if len(set(row[col:col+4])) == 1 and row[col]
!= 0:
                    self.result = Result.P1WIN if self.turn ==
1 else Result.P2WIN
                    return True
        for row in range(3,Connect4.ROWS):
            for col in range(0,4):
                if self.grid[row][col] == self.turn and
self.grid[row][col] == self.grid[row-1][col+1] and
self.grid[row-1][col+1] == self.grid[row-2][col+2] and
self.grid[row-2][col+2] == self.grid[row-3][col+3]:
                    self.result = Result.P1WIN if self.turn ==
1 else Result.P2WIN
                    return True
        for row in range(3,Connect4.ROWS):
            for col in range(3, Connect4.COLS):
                if self.grid[row][col] == self.turn and
self.grid[row][col] == self.grid[row-1][col-1] and
self.grid[row-1][col-1] == self.grid[row-2][col-2] and
self.grid[row-2][col-2] == self.grid[row-3][col-3]:
                    self.result = Result.P1WIN if self.turn ==
1 else Result.P2WIN
                    return True
        for i in range(Connect4.COLS):
            if self.tot[i] != Connect4.ROWS:
                self.result = Result.NONE
                return False
        self.result = Result.DRAW
        return True

    def make_move(self, move):
        """ Updates an item at a row specified by the value in
the index of a column of tot, and a column specified by a
column to the current turn, before incrementing the value in
the index of a column of tot by +1.

        Args:
            move (int): A column in which a move is to be made
on the grid
```

```
        """
        self.grid[self.tot[move]][move] = self.turn
        self.tot[move] += 1
        self.checkwin()
        self.change_turn()

    def undo_move(self, move):
        """ Increments the value in the index of a column of
tot by -1, before updating an item at a row specified by the
value in the index of a column of tot, and a column specified
by a column to the zero.

        Args:
            move (int): A column in which a move is to be
unmade on the grid
        """
        self.tot[move] -= 1
        self.grid[self.tot[move]][move] = 0
        self.checkwin()
        self.change_turn()

    def game_over(self):
        """ Returns True if an outcome has occurred.

        Returns:
            boolean: Whether the game has ended or not
        """
        return self.result != Result.NONE

    def grid_clear(self):
        """ Creates a new grid and tot, before assigning them
to the class properties grid and tot.
        """
        self.grid = self.create_grid()
        self.tot = self.create_tot()
        self.result = Result.NONE
        self.turn = Connect4.P1

    def __repr__(self):
        """ Represents the grid.
        """
        for i in range(len(self.grid)):
            print(self.grid[-i-1])
```

Implementing the Minimax Class

Creating the Evaluation Function

To evaluate the grid, I decided that rather than altering the 'maximising player' attribute for each turn, it would be easier to develop the evaluation function to always evaluate *positively* from the perspective of whichever turn the Minimax object itself is. For example, if it were player 2's turn when the Minimax object was instantiated, then player two's benefits would be positive and vice versa.

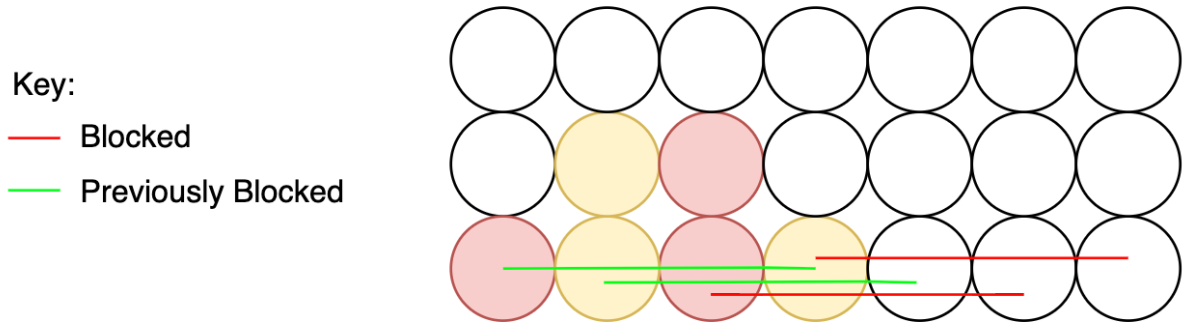
```
    def evaluation(self, grid, maximising_player):
        positions = [
        [ 3, 4, 5, 7, 5, 4, 3],
        [ 4, 6, 8,10, 8, 6, 4],
        [ 5, 8,11,13,11, 8, 5],
        [ 5, 8,11,13,11, 8, 5],
        [ 4, 6, 8,10, 8, 6, 4],
        [ 3, 4, 5, 7, 5, 4, 3]
        ]

        player_score = opponent_score = 0

        positions = [score for row in positions for score in
row]
        grid = [piece for row in grid for piece in row]

        for i in range(len(positions)):
            if grid[i] == maximising_player:
                player_score += positions[i]
            if grid[i] != 0 and grid[i] != maximising_player:
                opponent_score += positions[i]
        return player_score - opponent_score
```

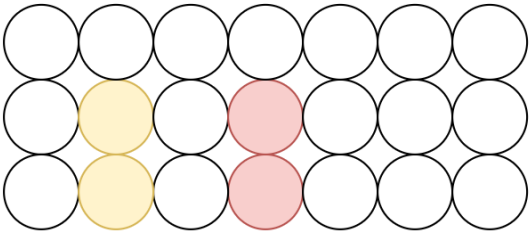
There was an issue with this method, however:



In this grid state, we can see that when the yellow player plays a piece into the centre column, along the bottom row, he blocks the red player from being able to achieve *two* of the *four* possible horizontal victories. This is due to the remaining two having been *already blocked* by another yellow piece. In this state, however, when fed into our evaluation function, the move is still

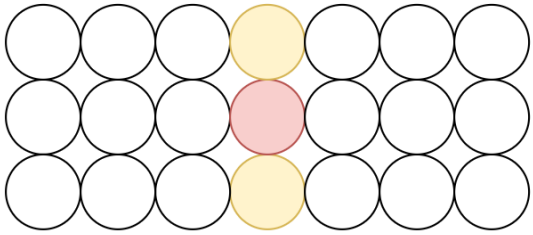
considered as if *all possible victories are being blocked off*. Whilst I initially considered a dynamic, updating grid because of this, I quickly found that due to the only possibilities being player vs Minimax or Minimax vs Minimax, this would occur:

Minimax vs Player/Random



Minimax plays optimal columns as soon as possible, limiting possibility of 'blockages' occurring. Low effect, as outcomes usually reached within few moves.

Minimax vs Minimax



Optimal columns are played first, removing possibility of previous 'blockages' occurring. Minimal effect.

Minimax.py

```
from Connect4 import Connect4, Result
from time import time
import math
import random
from log_config import logging

class Minimax:
    """ Stores the best move, initially a random move, to be
made on a Connect4 game. Has class methods allowing it to
calculate the best move via evaluating the grid in terms of
positive and negative score.
    """

    def __init__(self, game):
        """ Initialises the Minimax object with the current
game and assigns a random column as the best move.

        Args:
            game (object): The current game object
        """
        self.game = game
        self.best_move = random.randint(0,6)

    def minimax(self, depth, max_depth, maximising_player):
        """ Returns a positive infinity, negative infinity, or
0 value depending on whether the game has reached a result, or
the heuristic score of the game state if the maximum depth has
been reached. Otherwise, it calculates all possible moves on
the game state and makes them until these conditions have been
met. Once they have, the movements are evaluated, either by
comparing game state to a table of values as to create a
heuristic 'score' for each move, negative for the opponent,
```

```
positive for the plauer, or by a very high, very low, or
average score for a win, loss and draw respectively. These
scores for each move are compared to find the highest score-
available move, which is then assigned to a class variable.

        Args:
            depth (int): The current 'depth' the Minimax is
operating at, or how many turns have been taken total by the
Minimax
            max_depth (int): The maximum 'depth' at which the
Minimax is allowed to explore, or how many turns the Minimax
is able to take
            maximising_player (int): The player whose point of
view the Minimax is operating from

        Returns:
            int: Either a 0, 500, or -500 if an outcome has
occured, respective of the outcome. Otherwise, an integer
calculated by the evaluation function when given the grid.
Once all nodes have been explored, however, the highest score
achieved by either method is returned
        """
        if self.game.game_over():
            if self.game.result != Result.DRAW:
                res = 1 if self.game.result == Result.P1WIN
else 2
                return 500 if res == maximising_player else -
500
            else:
                return 0

        elif depth == max_depth:
            return self.evaluation(self.game.grid,
maximising_player)

        else:
            best_score = -500 if maximising_player ==
self.game.turn else 500
            for i in range(self.game.COLS):
                if self.game.valid_move(i):
                    self.game.make_move(i)
                    score = self.minimax(depth+1, max_depth,
maximising_player)
                    self.game.undo_move(i)
                    if maximising_player == self.game.turn:
                        if score > best_score:
                            best_score = score
                            if depth == 0:
                                self.best_move = i
```

```
                    else:
                        if score < best_score:
                            best_score = score
                            if depth == 0:
                                self.best_move = i

            return best_score

    def random_move(self):
        """ Returns a random integer between zero and the
number of columns in the grid.

        Returns:
            int: A random integer between zero and the number
of columns in the grid
        """
        best_col = random.randint(0, self.game.COLS)
        return best_col

    def evaluation(self, grid, maximising_player):
        """ Assigns a total positive and negative score to the
current grid depending on where the maximising player and
opponent's pieces are compared to the values stored within a
table.

        Args:
            grid (2D Array): The grid of the game object with
any number of moves made onto it by the Minimax object
            maximising_player (int): The player whose counter
positions will be counted as positive

        Returns:
            int: The evaluation heuristic of the grid fed into
the function from the point of view of the maximising player
        """
        positions = [
        [ 3, 4, 5, 7, 5, 4, 3],
        [ 4, 6, 8,10, 8, 6, 4],
        [ 5, 8,11,13,11, 8, 5],
        [ 5, 8,11,13,11, 8, 5],
        [ 4, 6, 8,10, 8, 6, 4],
        [ 3, 4, 5, 7, 5, 4, 3]
        ]

        player_score = opponent_score = 0

        positions = [score for row in positions for score in
row]
        grid = [piece for row in grid for piece in row]
```

```
        for i in range(len(positions)):
            if grid[i] == maximising_player:
                player_score += positions[i]
            if grid[i] != 0 and grid[i] != maximising_player:
                opponent_score += positions[i]

        return player_score - opponent_score
```

Implementing the GUI Class

Drawing an Interactive Grid

Due to how TkInter works, I found that rather than make a series of canvases or buttons for each 'square' in the grid, it would be much better to have a canvas separated by drawn lines. From there, calculations could be made to find which column the click would be within. This would be more efficient as not only did less objects have to be made but the improved method could reference a known canvas width and height to make the process smoother:

```
class App():
    WINDOW_WIDTH = 700
    WINDOW_HEIGHT = 600
    CANVAS_WIDTH = 500
    CANVAS_HEIGHT = 450
```

These class variables allowed me to define the canvas and window as separate dimensions, rather than having it cling tight to the border of the grid, whilst also being used with the class variables of the Game Class to *dynamically* create a grid of x columns and rows within a canvas.

```
def draw_grid(self):
        for i in range(0, App.CANVAS_WIDTH, App.CANVAS_WIDTH
// Connect4.COLS):
            self.root.canvas.create_line(i, 0, i,
App.CANVAS_HEIGHT)
        for i in range(0, App.CANVAS_HEIGHT, App.CANVAS_HEIGHT
// Connect4.ROWS):
            self.root.canvas.create_line(0, i,
App.CANVAS_WIDTH, i)
```

Creating a New Game/Simulation

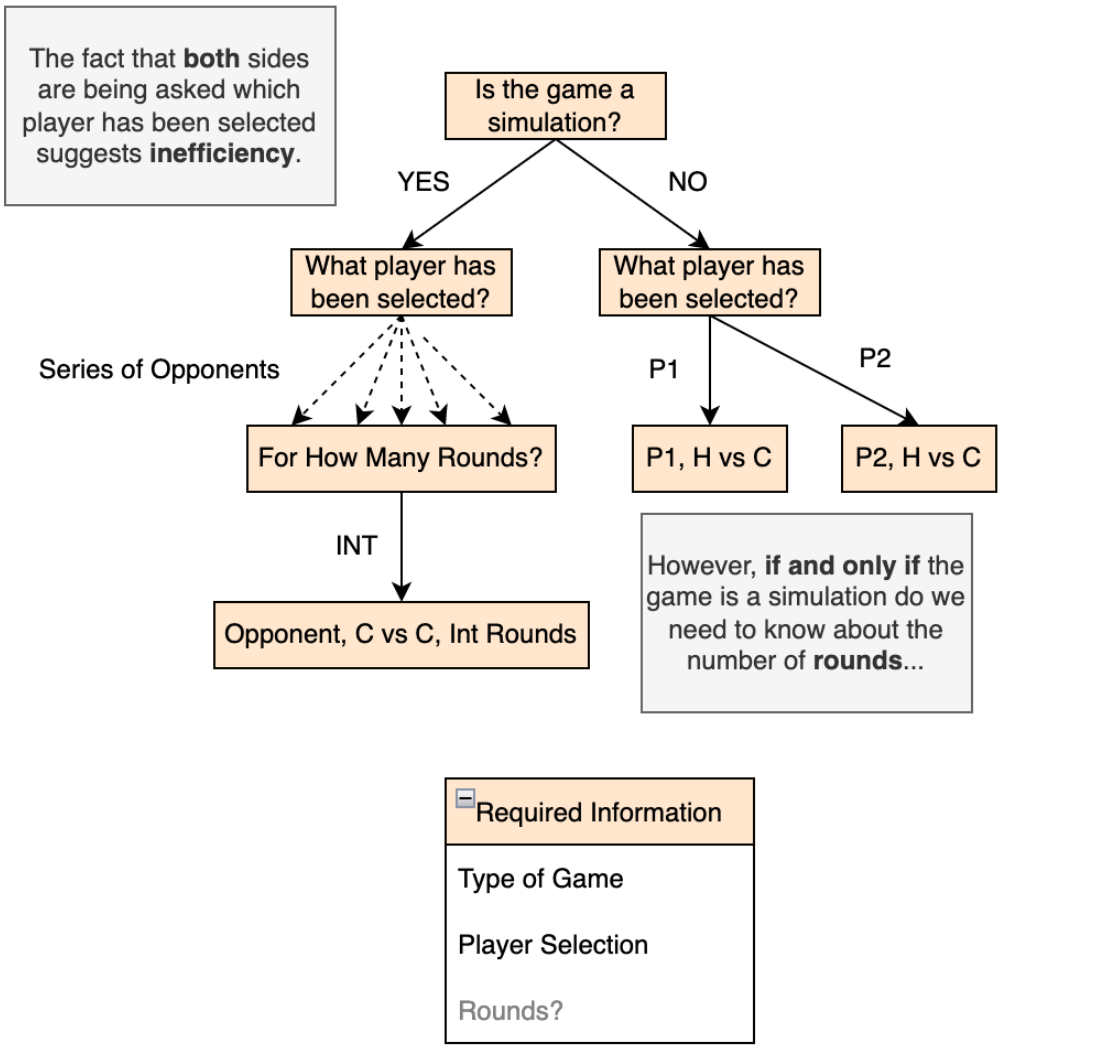
My process for creating the New Game and Simulation screens began by considering what will need to be known by the game once these windows are closed. As shown in the diagram below, whilst I knew I needed to have class variables of GameType (Simulation or Game), PlayerSelection and Rounds. As PlayerSelection would be required for either type of game, I decided to have both windows reference the same IntegerVariable using TkInter:

```
self.p = tkinter.IntVar()
```

Variable is referenced both times *here*:

```
player_select_screen.player_choice1 =
Radiobutton(player_select_screen.frame, text = 'Player 1', bg
= 'grey', variable = self.p, value = 1, command =
self.p.set(1), height = 1, justify = 'center', width = 10)
```

```
s_player_select_screen.player_choice1 =
Radiobutton(s_player_select_screen.frame, text = 'Random
Player', bg = 'grey', variable = self.p, command =
self.p.set(1), value= 1, height = 1)
```



This new system means that despite in both cases the player variable is set to '1', the variable 'self.mode' is used to differentiate between the two. Throughout my code, I used this practise, with all references to 'self.mode' being used to differentiate between the game and a simulation;

Used to disable clicking during a simulation:

```
    def canvas_click(self, event):
        if not self.game.game_over() and self.game.turn ==
self.player and self.mode == True:
```

Used to differentiate between calling for a new game and a simulation:

```
def return_to_game(self, mode):
        self.mode = mode
        self.game.grid_clear()
        self.draw()
        self.player = int(self.p.get())
        if self.mode == True:
            self.root.text.delete('1.0', '100.0')
            self.root.text.insert('1.0', "Welcome to Connect 4
with Minimax.")
            self.root.text.tag_configure("tag_name",
justify='center')
            self.root.text.tag_add("tag_name", "1.0", "end")
            if self.player == self.game.P2:
                self.root.after(1000,
self.min_max_move(self.max_depth.get()))
                self.draw()
        else:
            self.game.turn = self.game.P1
            self.rounds = int(self.rounds.get()) if
self.rounds.get().isdigit() else 10
            if self.rounds < 1 or self.rounds > 50:
                self.rounds = 10
            self.root.after(2000, self.simulation(0,0))
```

In the above lines, if the game is *not* a simulation, then and only then is rounds validated and defined, as otherwise there would be no reason to do so. All of this was designed for peak efficiency as to allow for the Minimax to be the main focus of the project.

Connect4GUI.py

```
import tkinter
import random
from tkinter import ttk, Tk, Canvas, Frame, Button, Text,
Radiobutton, Entry, Label, StringVar, IntVar, OptionMenu
from Connect4 import Connect4, Result
from MinimaxAttempt import Minimax
from log_config import logging


class Piece():
    """ Stores the radius of any piece placed on the grid.
    """
```

```
    R = 30


class App():
    """ Stores and represents the main window of the project
as a TkInter root with class methods which allow it to be
dynamic and interactive with the user. Uses a Connect4 object
in order to display and play the game graphically.
    """
    WINDOW_WIDTH = 700
    WINDOW_HEIGHT = 600
    CANVAS_WIDTH = 500
    CANVAS_HEIGHT = 450
    colour = ['red', 'yellow']

    def __init__(self, game, player):
        """ Initialises the Home Screen as a TkInter root with
a known Mode, Player, and Max_Depth, each of which determine
the type of game the Minimax will play.

        Args:
            game (object): A predefined Connect4 object
            player (int): Either 1 or 2, deciding either which
player the user is or which simulation opponent is being
faced, depending on the 'mode' attribute
        """
        self.player = player
        self.mode = True
        self.root = Tk()
        self.max_depth = tkinter.IntVar()
        max_depth_options = [3, 4, 5, 6, 7]
        self.max_depth.set(max_depth_options[2])
        self.p = tkinter.IntVar()
        self.rounds = tkinter.StringVar()

        self.root.title("Connect 4")
        self.root.frame = ttk.Frame(self.root,
width=App.WINDOW_WIDTH)
        self.root.frame.grid(row=0, column=0)
        self.root.text = Text(self.root.frame, bg = 'black',
fg = 'white', width = 71, selectborderwidth = 1, height = 1.5,
pady = 0, padx = 0)
        self.root.text.insert('1.0', "Welcome to Connect 4
with Minimax.")
        self.root.text.tag_configure("tag_name",
justify='center')
        self.root.text.tag_add("tag_name", "1.0", "end")
        self.root.text.grid(row=1,column=1)
        self.root.canvas = Canvas(self.root.frame,
width=App.CANVAS_WIDTH, height=App.CANVAS_HEIGHT)
```

```
        self.root.canvas.grid(row=2, column=1)
        self.root.canvas.bind('<Button-1>', self.canvas_click)
        self.root.buttonholder = ttk.Frame(self.root,
width=App.WINDOW_WIDTH)
        self.root.buttonholder.grid(row = 3, column = 0)
        self.root.button1 = Button(self.root.buttonholder,
text = 'New Game', activebackground = 'yellow', bg = 'grey',
command = self.player_select_screen, height = 1, justify =
'center', width = 8, padx = 30)
        self.root.button1.pack(side = "left")
        self.root.dropdown =
OptionMenu(self.root.buttonholder, self.max_depth,
*max_depth_options)
        self.root.dropdown.pack(side = "right")
        self.root.description = Label(self.root.buttonholder,
text = 'Minimax Depth:')
        self.root.description.pack(side = "right", padx =
(10,0))
        self.root.button2 = Button(self.root.buttonholder,
text = 'Simulation', activebackground = 'yellow', bg = 'grey',
command = self.s_player_select_screen, height = 1, justify =
'center', width = 8, padx = 30)
        self.root.button2.pack(side = "right")

        self.new_game(game)

        self.draw()
        self.root.mainloop()

    def draw(self):
        """ Clears the canvas in the centre of the window,
before drawing the grid and current pieces on top of it.
        """
        self.root.canvas.delete('all')
        self.draw_grid()
        self.draw_pieces()

    def new_game(self, game):
        """ Creates a new game and sets it to the class
property.

        Args:
            game (object): A predefined Connect4 object
        """
        assert type(game) == Connect4 and game is not None
        self.game = game

    def delay(self, function):
```

```
        """ Runs a function after a 100ms rest without running
any other lines of code before the function has been
completed. This is here due to the after() function in TkInter
continuing to run further lines in the code despite the
initial intended function not having been run yet

        Args:
            function (function): Any function
        """
        self.root.after(100, function)

    def final_result(self):
        """ Updates the text display above the grid to a
message either indicating which player has won the game, or
whether the game was a draw
        """
        self.root.text.delete('1.0', '100.0')
        if self.game.result == Result.P1WIN:
            self.root.text.insert('1.0', f'Player 1 Wins!')
            self.root.text.tag_add("tag_name", "1.0", "end")
        if self.game.result == Result.P2WIN:
            self.root.text.insert('1.0', f'Player 2 Wins!')
            self.root.text.tag_add("tag_name", "1.0", "end")
        if self.game.result == Result.DRAW:
            self.root.text.insert('1.0', f'DRAW!')
            self.root.text.tag_add("tag_name", "1.0", "end")

    def s_final_result(self, win_count_minimax,
win_count_opponent):
        """ Updates the running total scores for the Minimax
algorithm and it's opponent during a simulation, drawing them
each in the text display above the grid, before returning
them.

        Args:
            win_count_minimax (double): The current number of
victories achieved by the Minimax function (draws are counted
as half a win)
            win_count_opponent (double): The current number of
victories achieved by the opponent function (draws are counted
as half a win)

        Returns:
            tuple: both scores updated in accordance to the
outcome achieved
        """
        if self.game.result == Result.P1WIN:
            win_count_minimax += 1
        elif self.game.result == Result.P2WIN:
```

```
            win_count_opponent += 1
        elif self.game.result == Result.DRAW:
            win_count_minimax += 0.5
            win_count_opponent += 0.5
        self.root.text.delete('1.0', '100.0')
        self.root.text.insert('1.0', f'Minimax:
{win_count_minimax}     Opponent: {win_count_opponent}')
        self.root.text.tag_add("tag_name", "1.0", "end")
        self.game.grid_clear()
        self.game.change_turn()
        return win_count_minimax, win_count_opponent

    def canvas_click(self, event):
        """ If the canvas is clicked on, this function checks
whether it is an ongoing game, the player's turn, and a game,
rather than a simulation. If true, then the x-co-ordinates of
the click are calculated to find which column the click took
place in. If valid, a counter is played at the correct height
in the column, otherwise 'That column is full.' is displayed
in a text box and no move is made.

        Args:
            event (object): Stores attributes about a click
that occured on a canvas on a canvas
        """
        if not self.game.game_over() and self.game.turn ==
self.player and self.mode == True:
            if 0 < event.x < App.CANVAS_WIDTH and 0 < event.y
< App.CANVAS_HEIGHT:
                move = self.find_column(event.x)
                if self.game.valid_move(move):
                    self.game.make_move(move)
                    self.draw()
                    self.root.update_idletasks()
                    if self.game.game_over():
                        self.final_result()
                    else:
                        self.root.text.delete('1.0', '100.0')
                        self.root.text.insert('1.0', 'Minimax
thinking...')
                        self.root.text.tag_add("tag_name",
"1.0", "end")
                        self.root.update_idletasks()
                        self.root.after(500,
self.min_max_move(self.max_depth.get()))
                        self.root.text.delete('1.0', '100.0')
                        self.draw()
                        if self.game.game_over():
                            self.final_result()
```

```
                else:
                    self.root.text.delete('1.0', '100.0')
                    self.root.text.insert('1.0', 'That column
is full.')
                    self.root.text.tag_add("tag_name", "1.0",
"end")


    def simulation(self, win_count_minimax,
win_count_opponent):
        """ Decides which opponent the minimax algorithm is
facing, before playing a game of them against one another,
checking whether a win has been achieved every turn. If so,
then final_result is called, and if not, the function is
called recursively. If the maximum number of rounds has been
played, the text box changes from the scores of each player,
being prepended with 'Final score:'.

        Args:
            win_count_minimax (double): The current number of
victories achieved by the Minimax function (draws are counted
as half a win)
            win_count_opponent (double): The current number of
victories achieved by the opponent function (draws are counted
as half a win)
        """
        self.root.text.delete('1.0', '100.0')
        self.root.text.insert('1.0', f'Minimax:
{win_count_minimax}    Opponent: {win_count_opponent}')
        self.root.text.tag_add("tag_name", "1.0", "end")
        if win_count_minimax + win_count_opponent !=
self.rounds:
            if self.player == 1:

self.delay(self.min_max_move(self.max_depth.get())) if
self.game.turn == self.game.P1 else
self.delay(self.random_move())
            else:

self.delay(self.min_max_move(self.max_depth.get())) if
self.game.turn == self.game.P1 else
self.delay(self.min_max_move(self.player+1))
            if self.game.game_over():
                win_count_minimax, win_count_opponent =
self.s_final_result(win_count_minimax, win_count_opponent)
            self.draw()
            self.root.update_idletasks()
```

```
            self.simulation(win_count_minimax,
win_count_opponent)
        else:
            self.root.text.insert('1.0', 'Final Result: ')
            self.root.text.tag_add("tag_name", "1.0", "end")
            self.rounds = tkinter.StringVar()

    def random_move(self):
        """ Makes a random, valid move.
        """
        if not self.game.game_over():
            move = random.randint(0,6)
            while self.game.valid_move(move) == False:
                move = random.randint(0,6)
            self.game.make_move(move)

    def min_max_move(self, max_depth):
        """ Creates a Minimax object and calls minimax on the
current grid with a max depth, making a move in the column
returned.

        Args:
            max_depth (int): The maximum 'depth' that the
Minimax object can explore, or the maximum number of moves
that can be made on the grid by the Minimax object
        """
        if not self.game.game_over():
            minimax = Minimax(self.game)
            minimax.minimax(0, max_depth, self.game.turn)
            if self.game.valid_move(minimax.best_move):
                self.game.make_move(minimax.best_move)
            else:
                self.random_move()


    def draw_grid(self):
        """ Draws lines over the canvas at equal intervals to
create a grid.
        """
        for i in range(0, App.CANVAS_WIDTH, App.CANVAS_WIDTH
// Connect4.COLS):
            self.root.canvas.create_line(i, 0, i,
App.CANVAS_HEIGHT)
        for i in range(0, App.CANVAS_HEIGHT, App.CANVAS_HEIGHT
// Connect4.ROWS):
            self.root.canvas.create_line(0, i,
App.CANVAS_WIDTH, i)

    def draw_pieces(self):
```

```
        """ Draws circles at each point in the grid where
specified by the game object, the colour corresponding to each
player.
        """
        grid = self.game.return_grid()[::-1]
        for row in range(len(grid)):
            for col in range(len(grid[row])):
                if grid[row][col] != 0:
                    self.colour = App.colour[0] if
grid[row][col] == Connect4.P1 else App.colour[1]
                    COL_WIDTH = App.CANVAS_WIDTH //
Connect4.COLS
                    ROW_HEIGHT = App.CANVAS_HEIGHT //
Connect4.ROWS
                    self.circle((COL_WIDTH*col) + (COL_WIDTH
// 2), (ROW_HEIGHT*row) + (ROW_HEIGHT // 2))

    def find_column(self, x):
        """ Calculates the column at which an x co-ordinate is
within.

        Args:
            x (double): The x co-ordinate along the canvas at
which a click occured

        Returns:
            double: The column in which the canvas was clicked
        """
        return x // (App.CANVAS_WIDTH // Connect4.COLS)

    def circle(self, x, y):
        """ Draws a circle at an x and y co-ordinate, with
radius r.

        Args:
            x (double): The x co-ordinate along the canvas at
which a click occured
            y (double): The y co-ordinate along the canvas at
which a click occured
        """
        xl = x - Piece.R
        yl = y - Piece.R
        xr = x + Piece.R
        yr = y + Piece.R
        self.root.canvas.create_oval(xl, yl, xr, yr,
fill=self.colour)

    def return_to_game(self, mode):
```

```
        """ Updates whether the game is a simulation or active
game, clears the grid, sets the player, and begins either a
simulation, or a game, updating the text display to reflect
this.

        Args:
            mode (boolean): False if the new game is a
simulation, True otherwise
        """
        self.mode = mode
        self.game.grid_clear()
        self.draw()
        self.player = int(self.p.get())
        if self.mode == True:
            self.root.text.delete('1.0', '100.0')
            self.root.text.insert('1.0', "Welcome to Connect 4
with Minimax.")
            self.root.text.tag_configure("tag_name",
justify='center')
            self.root.text.tag_add("tag_name", "1.0", "end")
            if self.player == self.game.P2:
                self.root.after(1000,
self.min_max_move(self.max_depth.get()))
                self.draw()
        else:
            self.game.turn = self.game.P1
            self.rounds = int(self.rounds.get()) if
self.rounds.get().isdigit() else 10
            if self.rounds < 1 or self.rounds > 50:
                self.rounds = 10
            self.root.after(2000, self.simulation(0,0))

    def player_select_screen(self):
        """ When the 'New Game' button is pressed, a new
window is created with a text display, two radiobuttons and an
Enter button, where the player can select which player they
want to be in the new game.
        """
        player_select_screen = tkinter.Toplevel(self.root)
        player_select_screen.title("Player Select Screen")
        player_select_screen.frame =
ttk.Frame(player_select_screen, width=App.WINDOW_WIDTH)
        player_select_screen.frame.grid(row=0, column=0)
        player_select_screen.text =
Text(player_select_screen.frame, bg = 'grey', fg = 'black',
width = 21, selectborderwidth = 1, height = 1.5, pady = 0,
padx = 0)
        player_select_screen.text.insert('1.0', "Would you
like to be Player 1 or Player 2?")
```

```
        player_select_screen.text.tag_configure("tag_name",
justify='center')
        player_select_screen.text.tag_add("tag_name", "1.0",
"end")
        player_select_screen.text.grid(row=1,column=1)
        player_select_screen.player_choice1 =
Radiobutton(player_select_screen.frame, text = 'Player 1', bg
= 'grey', variable = self.p, value = 1, command =
self.p.set(1), height = 1, justify = 'center', width = 10)
        player_select_screen.player_choice2 =
Radiobutton(player_select_screen.frame, text = 'Player 2', bg
= 'grey', variable = self.p, value = 2, command =
self.p.set(2), height = 1, justify = 'center', width = 10)

player_select_screen.player_choice1.grid(row=2,column=1)

player_select_screen.player_choice2.grid(row=3,column=1)
        player_select_screen.enter =
Button(player_select_screen.frame, text = 'ENTER', bg =
'grey', command = lambda: [player_select_screen.destroy(),
self.return_to_game(True)], height = 1, justify = 'center',
width = 10)
        player_select_screen.enter.grid(row=4,column=1)
        player_select_screen.mainloop()

    def s_player_select_screen(self):
        """ When the 'Simulation' button is pressed, a new
window is created with a text display, two radiobuttons, an
entry box and an Enter button, where the player can select
which opponent they want their minimax to face, and for how
many rounds.
        """
        s_player_select_screen = tkinter.Toplevel(self.root)
        s_player_select_screen.title("Simulation Player Select
Screen")
        s_player_select_screen.frame =
ttk.Frame(s_player_select_screen, width=App.WINDOW_WIDTH)
        s_player_select_screen.frame.grid(row=0, column=0)
        s_player_select_screen.text =
Text(s_player_select_screen.frame, bg = 'grey', fg = 'black',
width = 50, selectborderwidth = 1, height = 1.5, pady = 0,
padx = 0)
        s_player_select_screen.text.insert('1.0', "Minimax
Player vs Who?")
        s_player_select_screen.text.tag_configure("tag_name",
justify='center')
        s_player_select_screen.text.tag_add("tag_name", "1.0",
"end")
        s_player_select_screen.text.grid(row=1,column=1)
```

```
        s_player_select_screen.player_choice1 =
Radiobutton(s_player_select_screen.frame, text = 'Random
Player', bg = 'grey', variable = self.p, command =
self.p.set(1), value= 1, height = 1)
        s_player_select_screen.player_choice2 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax
Player: Depth 3', bg = 'grey', variable = self.p, command =
self.p.set(2), value= 2, height = 1)
        s_player_select_screen.player_choice3 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax
Player: Depth 4', bg = 'grey', variable = self.p, command =
self.p.set(3), value= 3, height = 1)
        s_player_select_screen.player_choice4 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax
Player: Depth 5', bg = 'grey', variable = self.p, command =
self.p.set(4), value= 4, height = 1)
        s_player_select_screen.player_choice5 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax
Player: Depth 6', bg = 'grey', variable = self.p, command =
self.p.set(5), value= 5, height = 1)
        s_player_select_screen.player_choice6 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax
Player: Depth 7', bg = 'grey', variable = self.p, command =
self.p.set(6), value= 6, height = 1)
        s_player_select_screen.round_entry_label =
Label(s_player_select_screen.frame, anchor = 'w', bg = 'grey',
text = 'Enter the number of rounds you wish to see (1 - 50):
\n (Anything out of range will default to 10)')
        s_player_select_screen.round_entry =
Entry(s_player_select_screen.frame, bg = 'white', textvariable
= self.rounds)

s_player_select_screen.player_choice1.grid(row=2,column=1)

s_player_select_screen.player_choice2.grid(row=3,column=1)

s_player_select_screen.player_choice3.grid(row=4,column=1)

s_player_select_screen.player_choice4.grid(row=5,column=1)

s_player_select_screen.player_choice5.grid(row=6,column=1)

s_player_select_screen.player_choice6.grid(row=7,column=1)

s_player_select_screen.round_entry_label.grid(row=8,column=1)

s_player_select_screen.round_entry.grid(row=9,column=1)
        s_player_select_screen.enter =
Button(s_player_select_screen.frame, text = 'ENTER', bg =
```

```
'grey', command = lambda: [s_player_select_screen.destroy(),
self.return_to_game(False)], height = 1, justify = 'center',
width = 10)
        s_player_select_screen.enter.grid(row=10,column=1)
        s_player_select_screen.mainloop()
```

Connect4Main.py

```
import Connect4
import Connect4GUI
import MinimaxAttempt
from log_config import logging

#Instantiates an App Object, with a Connect4 game with 7 Rows,
6 Columns and Player 1 Selected parsed in.
app = Connect4GUI.App(Connect4.Connect4(7, 6), 1)
```

How to run the program

To run the program, you are required to load all files into the visual studio IDE, or any IDE with an ability to translate and run python, before compiling Connect4Main.py and running it, no other files are required to be created or deleted.

TESTING

In this section I will be testing my code to certify that my project works as intended and solves the problem at hand. Any input of data will be tested for validity in terms of boundary and erroneous nature, where boundary implies the data being outside of a set 'limit', e.g. entering an integer value for Rounds outside 1 - 50, whilst erroneous data implies data being outside an expected value or type, e.g. entering a string value for Rounds. These will be tested alongside Normal data, which will justify that data within these boundaries of the correct type are being accepted, otherwise there is no assuring my code is correct.

As my project is mainly event-based and as a result takes very few inputs, I will have to test things such as button presses and clicks on the canvas for error, as well as data representation such as the text display above the grid.

Testing video: https://youtu.be/MVLX8W8K2Vc
(Home Screen Test 14 and 15, as well as Simulation Screen Test 11 were both omitted, as their testing methods would require a view of the console, and screenshot evidence has already been provided for their existence. Simulation Screen Test 10 and 9 are sped up as to save time.)

Tests

Home Screen					
Test Number	**Test Description**	**Test Input**	**Expected Output**	**Test Output**	**Pass?**
1	When the program is run, does the Home Screen get displayed?	Program ran	Home Screen loaded and displayed	Home Screen is displayed	**Objective 1 Met**
2	When the Home Screen is initially displayed, does the text display a welcome?	Program ran	Text display reads 'Welcome to Connect 4 with Minimax.'	The display reads 'Welcome to Connect 4 with Minimax.'	
3	When the screen is clicked, does the canvas place a piece in the correct area?	Canvas clicked in the first column available	Grid altered and counter drawn in the first column at the lowest row	The counter of the correct colour is drawn in the grid in the first column in the lowest row	
4		**Err:** Screen clicked outside of the grid	No change in the grid	No change in the grid	
5		**Bound:** Canvas	No change in the grid	No change in the grid, while the text	**Objective 2 Met**

		clicked in a full column		display changes to read 'This column is full.'	
6	When the 'New Game' button is pressed, does the new game window open whilst keeping the Home Screen open?	New Game button pressed	New Game screen opens, while the Home Screen remains in the background.	New Game screen opened and focused, whilst the Home Screen became unfocused, yet still running.	
7	When the 'Simulation' button is pressed, does the new game window open whilst keeping the Home Screen open?	Simulation button pressed	Simulation screen opens, while the Home Screen remains in the background.	Simulation screen opened and focused, whilst the Home Screen became unfocused, yet still running.	
8	With either secondary window open in the foreground, is the Home Screen still interactable and running?	Having opened the New Game and Simulation window, the grid of the Home Screen was clicked on	A move is played in the column selected	A counter of the correct colour is played in the column, whilst the New Game window became unfocused.	
9	Does pressing the close button on the Home Screen close it, as well as any other secondary windows created?	Having opened the Simulation Window, the close button of the Home Screen was clicked on	The home window and simulation window closes	Both screens closed.	
10	When a game is completed, does the text display the correct message above the board and does the grid cease all play?	Selecting Player 2, the Minimax algorithm won against the user. The grid was then pressed twice.	The text display reads that 'Player 1 Wins!', and the grid becomes unplayable.	The text displays 'Player 1 Wins!' and no pieces are played to the grid.	**Objective 4 Met**
11	When a simulation occurs, does the text display the	A simulation of 1 rounds was selected between	The text display reads 'Minimax: _ Opponent: _', updating	The text display reads 'Minimax: _ Opponent: _',	**Objective 7 Met**

	running total of victories in the game?	Minimax and Random Player from the Simulation Screen	according to the current score	updating according to the current score until the final round, after which it displays the final scores, prepended by 'Final Score:'.	
12	When a game is completed, are the 'New Game' and 'Simulation' buttons still interactable?	Having reached a Minimax Victory, the New Game button was pressed	New Game Window opens	New Game Window opened, whilst the main grid remained unplayable.	
13	When the New Game and Simulation buttons are pressed multiple times, are multiple windows created?	New Game Button pressed three times	Three New Game windows created	Three New Game Windows were created.	
14	When a minimax depth is selected, does the minimax algorithm run with that depth?	Depth of 3 selected and grid clicked, code edited to print depth in terminal.	'3' printed into terminal, move made at the correct depth.	Move correctly made, with '3' printed into terminal	**Objective 8 and 9 Met**
15		Depth of 3 selected and grid clicked, then depth of 5 selected and grid clicked, code edited to print depth in terminal.	'3' then '5' printed into terminal, move made at the correct depths.	Both moves made at the correct depths, with both numbers printed in terminal.	**Objective 10 Met**
16	If the screen is clicked rapidly, can multiple counters be placed before the Minimax algorithm has its turn?	Screen clicked two times once a counter was placed before Minimax made its move	One counter placed in the clicked column, then no change to the grid.	Only one counter placed in the column.	**Objective 3 Met**

Player Select Screen					
Test Number	**Test Description**	**Test Input**	**Expected Output**	**Test Output**	**Pass?**
1	When the New Game screen is created, is there a pre-selected input?	New Game Button pressed	Player 2 already selected, whilst other buttons are interactable	Player 2 selected, unable to deselect without picking Player 1.	
2	When the New Game screen is created, does the text display a description of the screen?	New Game Button pressed	Text display reads 'Would you like to be Player 1 or Player 2?'	Text display reads 'Would you like to be Player 1 or Player 2?'	
3	When the Enter button is selected, is the current player selection returned to the Home Screen and the screen closed?	Player 1 radiobutton selected and Enter button pressed	New Game screen closes, and new game begins with the user taking the first move.	New Game screen closes, and grid cleared, user can interact with the grid.	**Objective 5 Met**
4		**Err:** Player 1 radiobutton selected and exit button pressed.	No change in the current grid state	No change in the grid.	
5		**Err:** Three New Game windows created and Player 2 selected on the third, then Enter button pressed	New Game screen closes, and new game begins with the user taking the second move. The remaining New Game windows remain open	Both remaining windows stay open, whilst the selected screen closes and user cannot interact with the grid initially.	
6	When one radiobutton has been selected, does selecting another deactivate the original?	Player 1 radiobutton selected, then Player 2 radiobutton selected	Player 2 radiobutton becomes selected, whilst Player 1 radiobutton becomes unselected	Player 2 radiobutton becomes selected, whilst Player 1 radiobutton becomes unselected.	

Simulation Screen					
Test Number	**Test Description**	**Test Input**	**Expected Output**	**Test Output**	**Pass?**
1	When the Simulation Game screen is created, is there a pre-selected input?	Simulation Button pressed	Minimax Depth 7 already selected, whilst rounds input box remains empty	Both selected.	
2	When the Simulation Screen is created, does the text and label display a description of the screen?	Simulation Button pressed	Text display reads 'Minimax vs Who?' and label reads 'Enter the number of rounds you wish to see (1 – 50): (Anything out of range will default to 10)'	Both displayed correctly.	
3	When the Enter Button is pressed, is the current opponent selection returned to the home screen and the screen closed?	Random Player radiobutton selected and Enter button pressed	Simulation screen closes, and simulation begins between Minimax and Random Player	Simulation screen closed and grid cleared, simulation began.	
4		**Err:** Random Player radiobutton selected and exit button pressed.	No change in the current grid state	No change in the grid.	
5		**Err:** Three Simulation Game windows created and Minimax Depth 3 selected on the third, then Enter button pressed	Simulation screen closes, and simulation begins between Minimax Depth 3 and Minimax. The remaining Simulation windows remain open	Both remaining windows stay open, whilst the selected screen closes, and simulation begins between Minimax Depth 3 and Minimax.	
6	When one radiobutton has been selected, does selecting another deactivate the original?	Random Player radiobutton selected, then Minimax Depth 7	Minimax Depth 7 radiobutton becomes selected, whilst Random Player radiobutton	Minimax Depth radiobutton becomes selected, whilst Random Player	

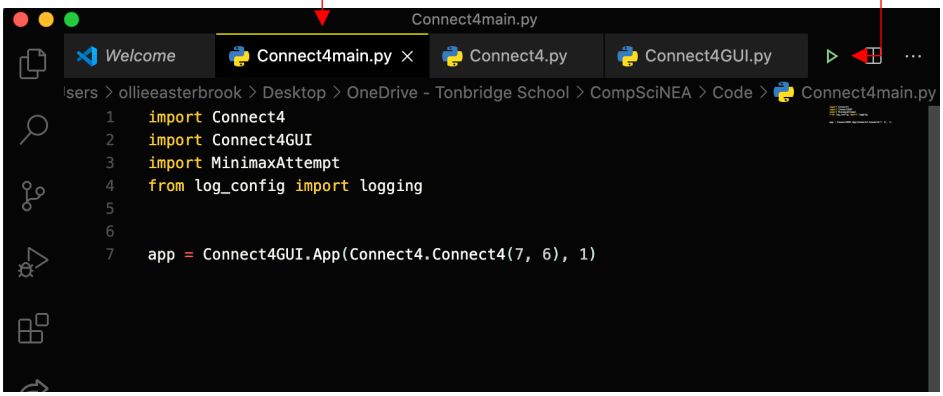
		radiobutton selected	becomes unselected	radiobutton becomes unselected.	
7	When the Enter Button is pressed, is the number of rounds entered returned to the home screen and the screen closed?	'12' entered into the entry box, then Enter button selected.	Simulation begins, playing out for 2 rounds.	12 rounds of the simulation occur, actively displaying the result of the games is displayed in the text display.	
8		**Err:** '12' entered into the entry box, then exit button selected	No change in the current grid state	No change in the grid.	
9		**Err:** 'abc' entered into the entry box, then Enter button selected	Simulation begins, playing for 10 rounds.	10 rounds of the simulation occur, actively displaying the result of the games is displayed in the text display.	
10		**Bound:** '1001' entered into the entry box, then Enter button selected	Simulation begins, playing out for 10 rounds.	10 rounds of the simulation occur, actively displaying the result of the games is displayed in the text display.	
11	When an opponent is selected, does the correct opponent begin to play the simulation?	Minimax Player : Depth 3 radiobutton selected, then Enter button selected, code altered to print the maximum depth of the minimax algorithm before playing,	Simulation begins, printing out '3' repeatedly.	Simulation begins, printing out '5' (the selected depth of Minimax), then '3' (depth of opponent), alternatively.	

Evidence[6]

Evidence 1 – Tests 1 – 5

With Connect4main.py open…

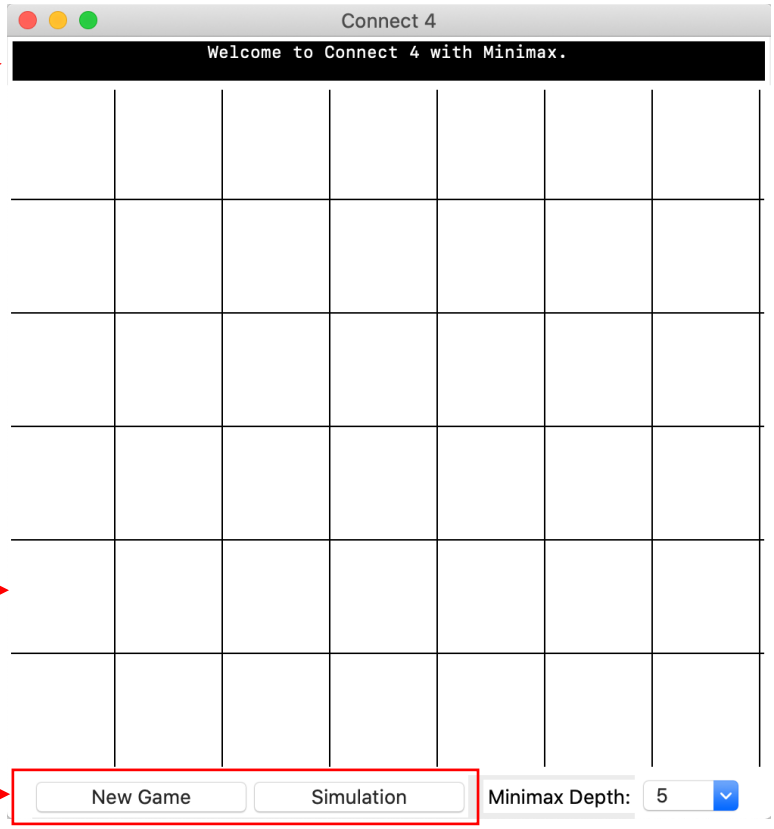
… the run button is pressed



This window is created

2 – Welcome message generated in text display, immediately after running the program.

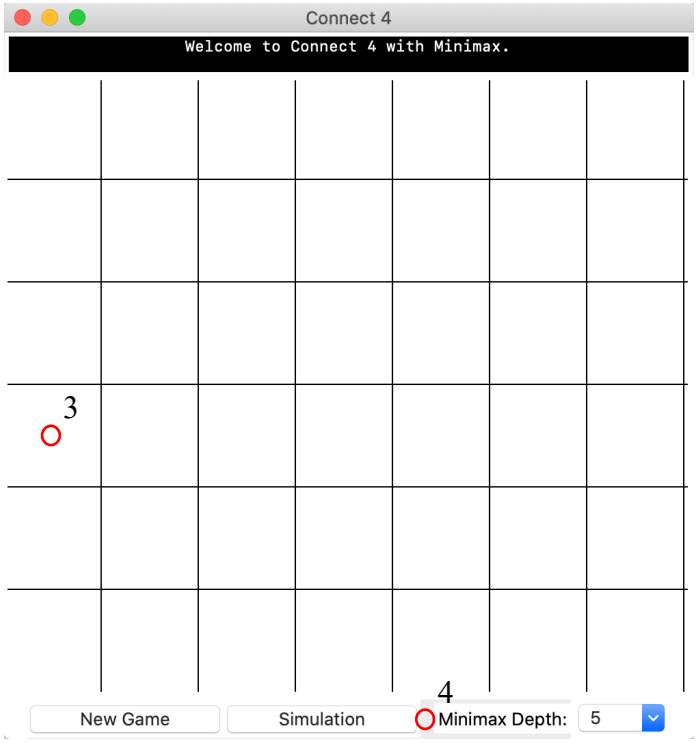
1 – Home screen comprised of canvas, text box and two buttons visible.



[6] As a prerequisite, the MacOS updated to Monterey overnight while I was testing, hence why the UI differs slightly between some images.

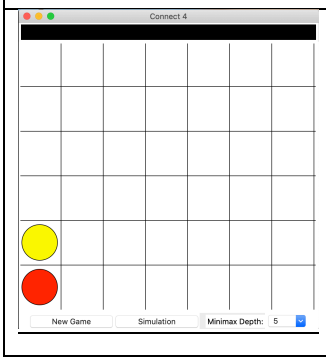
Qualification Code: 7517
Candidate Number: 3025

Centre Number: 61679

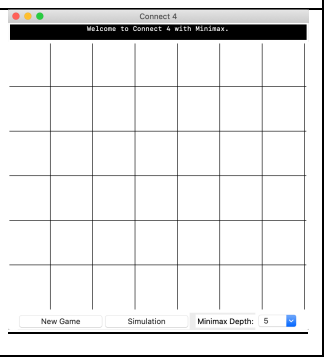


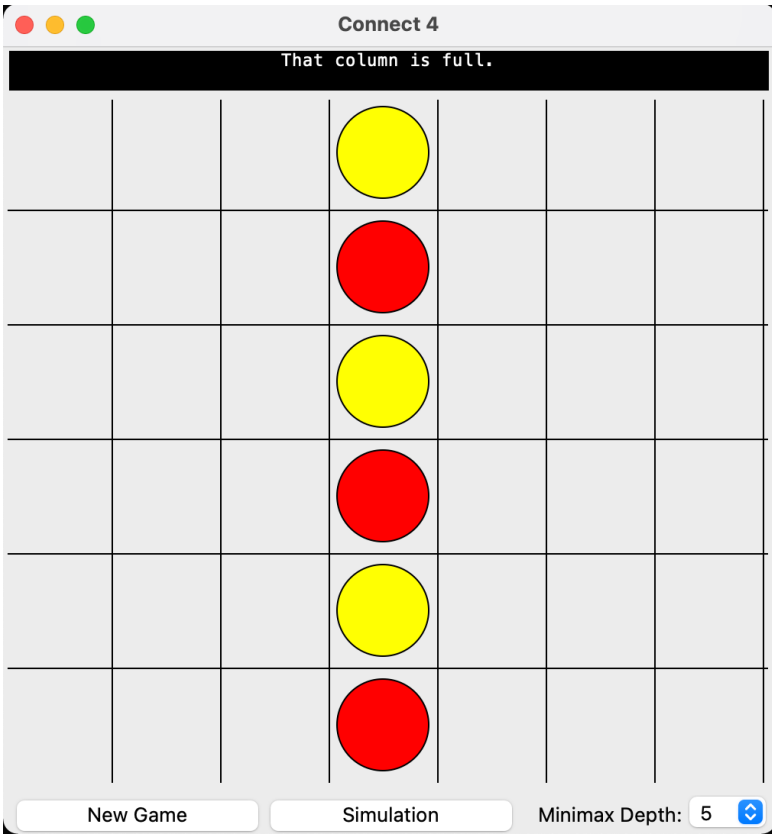
For tests 3 and 4, the screen was clicked in the locations specified, with the test being repeated three times to be sure no anomalies occurred with such a precise click.

Test 3



Test 4



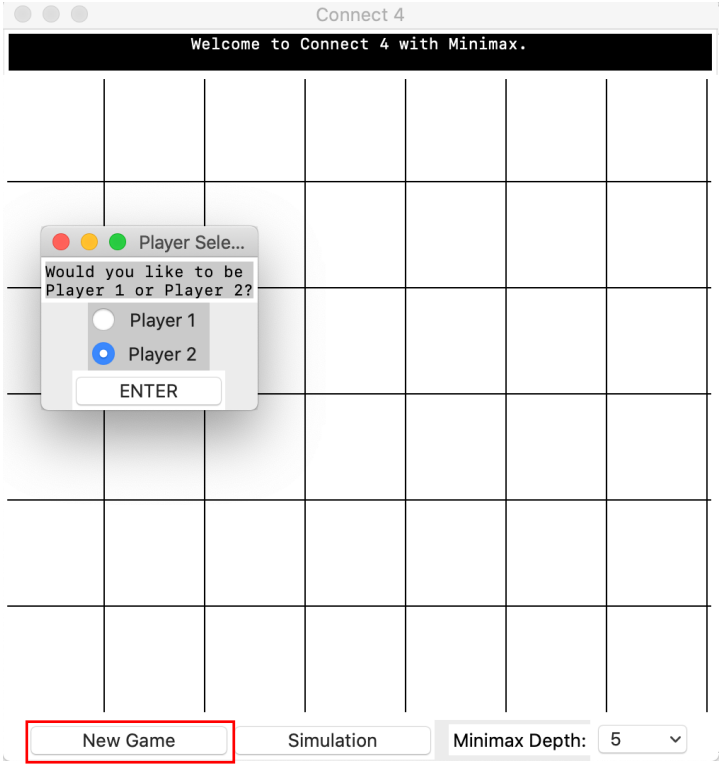


5 – With my regular minimax enabled, I found that the algorithm would play this configuration each time. I clicked in the middle column once it was full and the text was displayed; no change occurred in the grid.

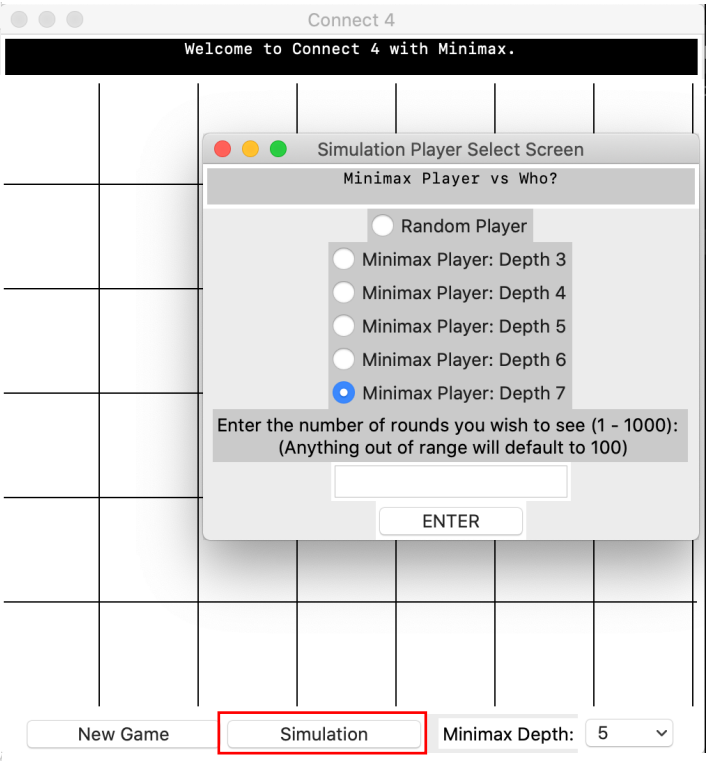
Qualification Code: 7517
Candidate Number: 3025

Centre Number: 61679

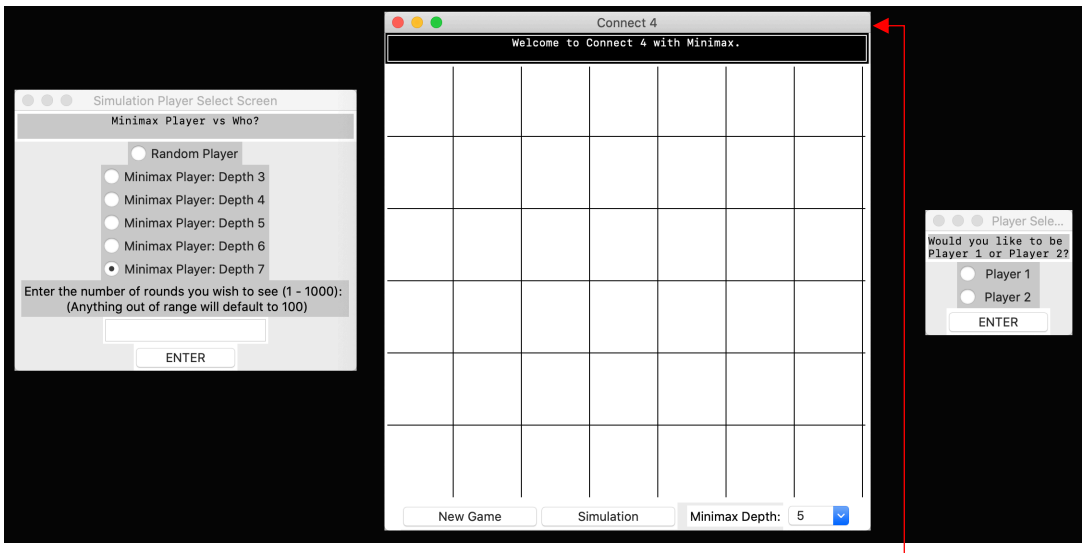
Evidence 2 – Tests 6 – 8



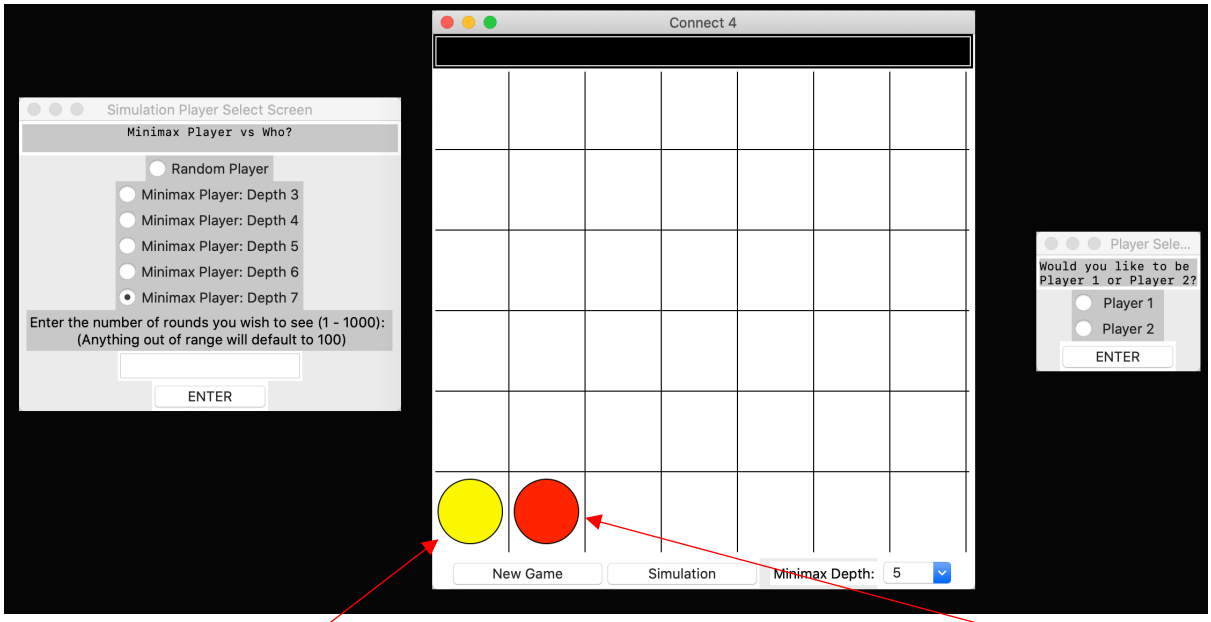
6 – New Game Button selected, with new window generated on the side, deselecting the main window.



7 – Simulation Button selected, with new window generated on the side, deselecting the main window.



Making sure both windows were open and visible, I selected the main window, as you can see by the fact that it is *highlighted*.



I edited the minimax to always play in the left side column for tests not testing its evaluation function. This mode will be adequately dubbed 'Suboptimal mode'.

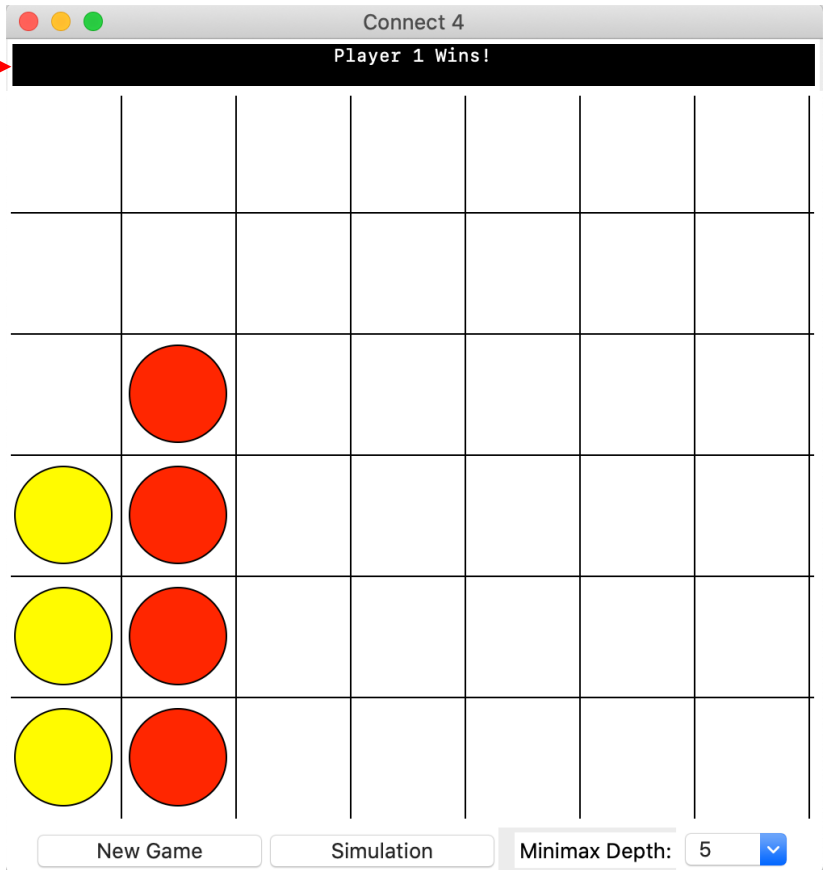
8 - Having clicked in the second column, a counter was placed in its position, while the text box displayed 'Minimax thinking…' before the Minimax move was made and the text disappeared, implying that the game is fully interactable.

Evidence 3 – Tests 10 – 11

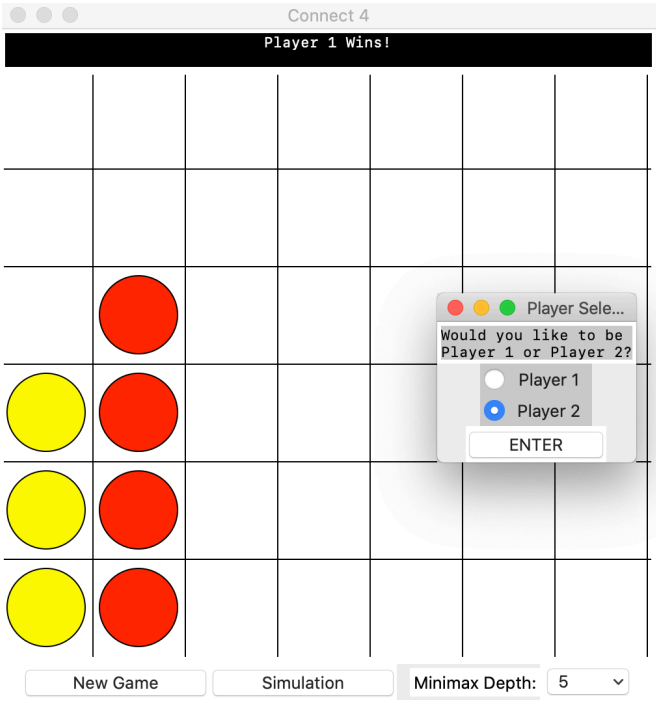
As both tests checked the text display, I decided to perform them both together.

With the Minimax playing sub optimally, I at first played a game as Player 1 to be sure that the result would correctly be displayed:

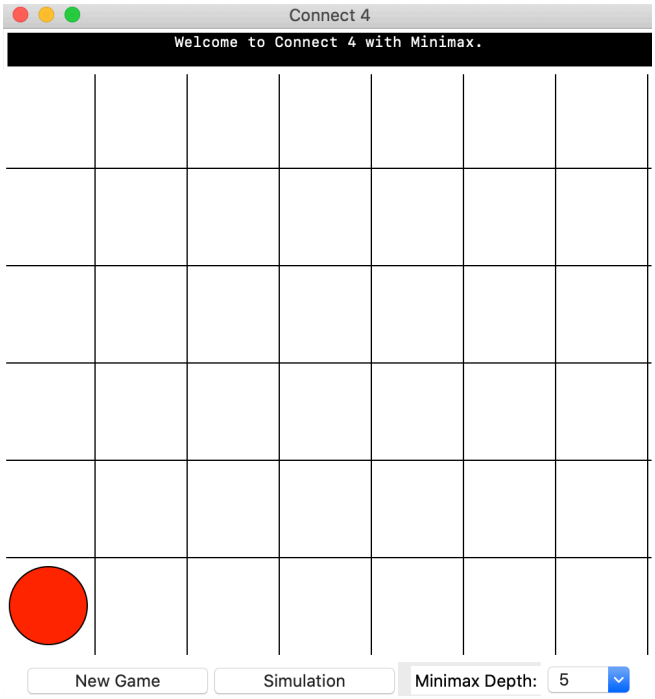
Text Display shows that Player 1 wins, correctly.

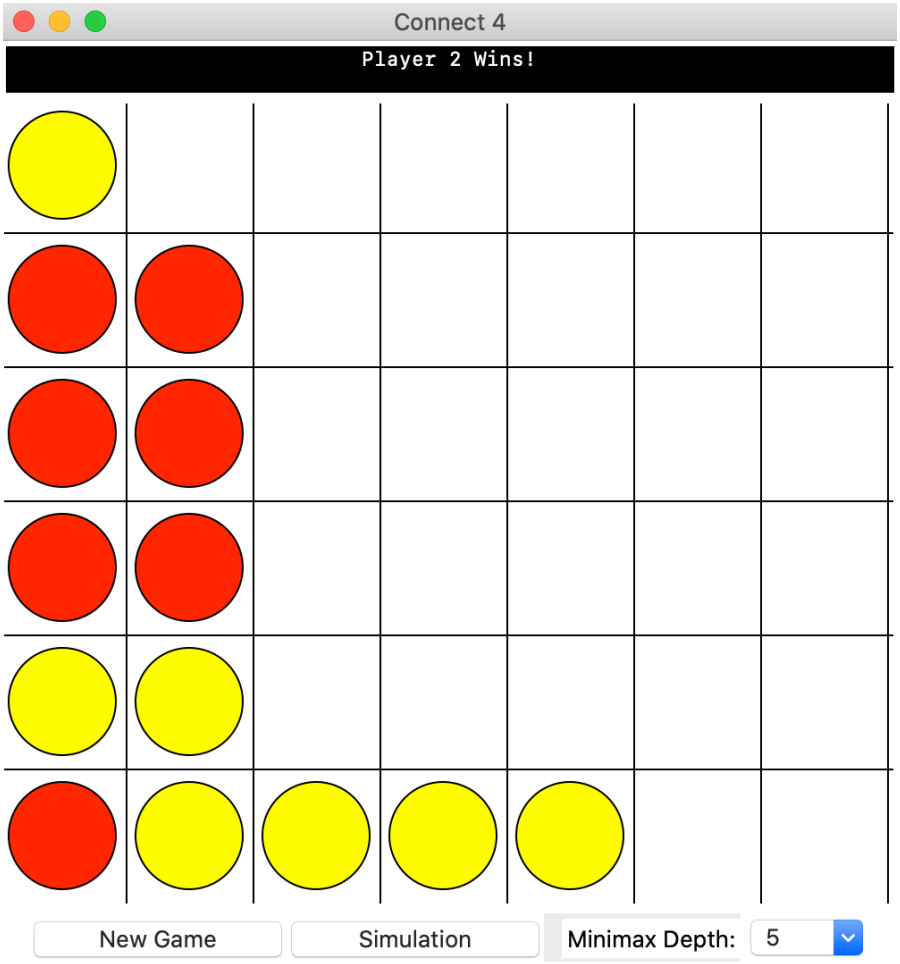


As shown in the image below, I then selected a new game as Player 2 and selected 'Enter'…



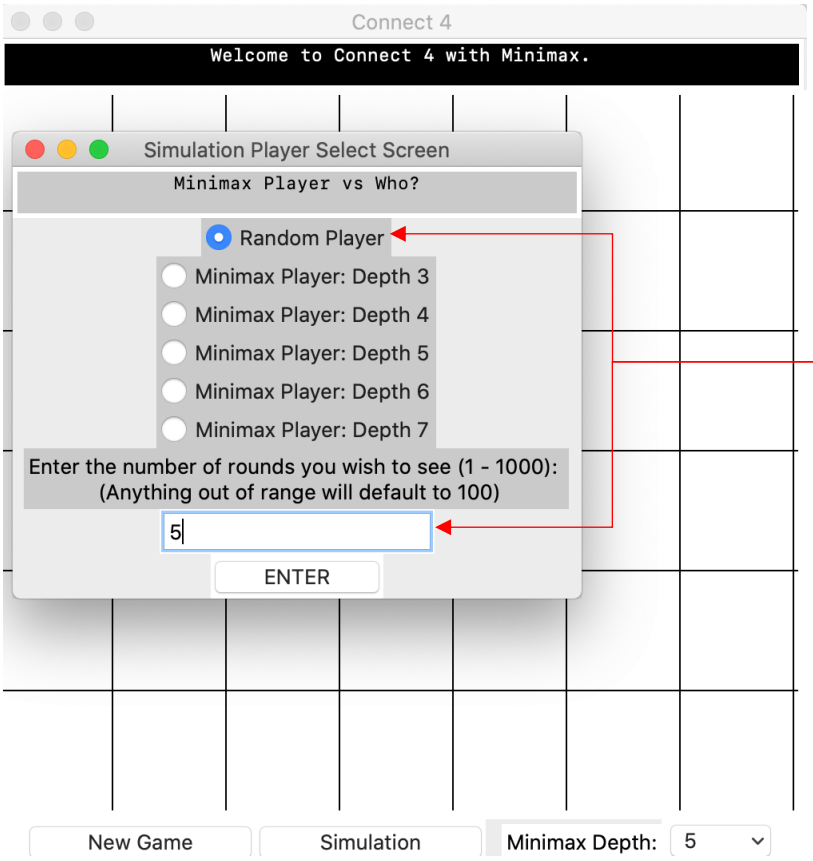
The game begun again, with Minimax taking the first move.





I won again, with some minor misplays due to forgetting about 'Suboptimal mode' being activated.

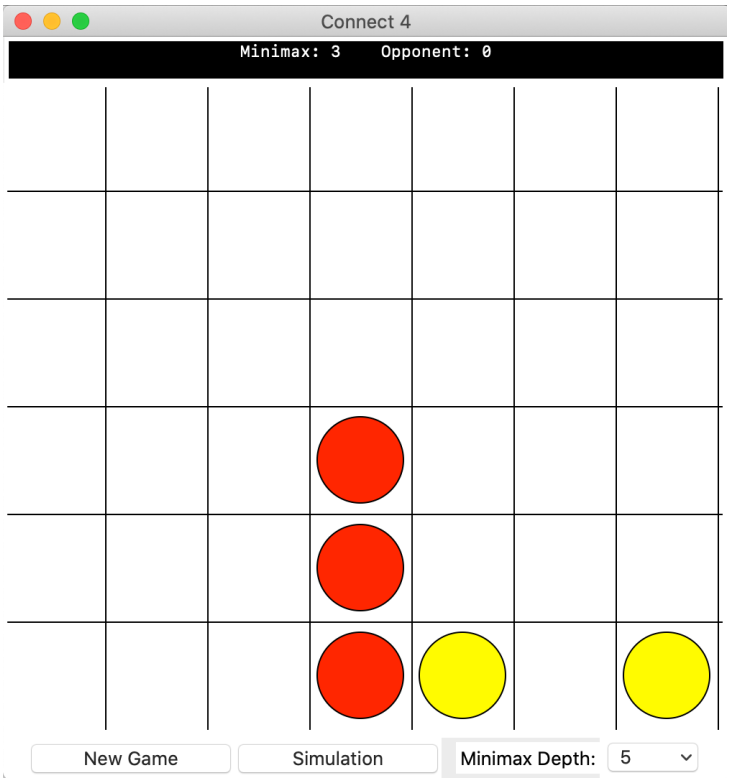
10 – 'Player 2 Wins!' is correctly displayed in the text display. After I click the grid twice in the third column, no move is made to the grid.



'New Game' was selected as Player 1 to create a new game, then 'Simulation' was selected.

To test the text display, I set a maximum round of 5, against a random player, before hitting 'Enter'

A game began, where the Minimax, no longer in suboptimal mode, began to play against a random opponent.



Text Display

Initial Display: 'Minimax: 0 Opponent: 0'

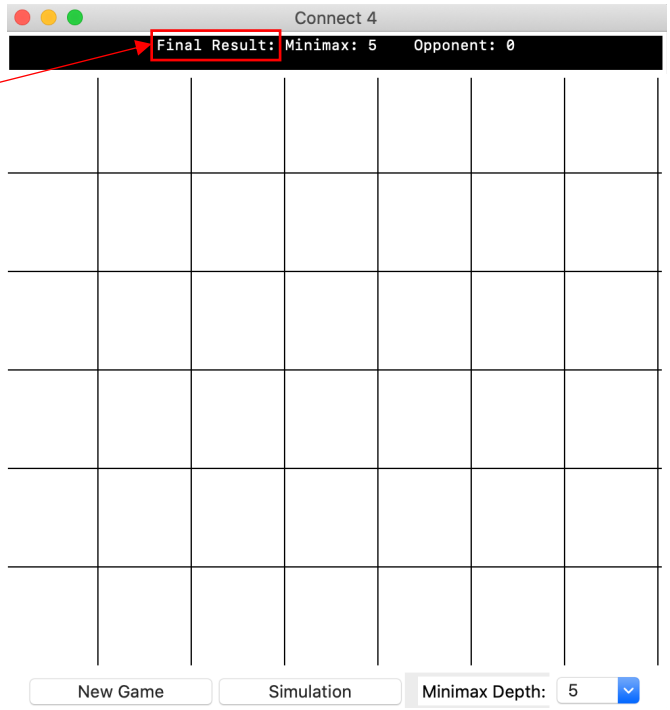
As the game continued, each time an outcome was achieved, the scores on the text display would update, before the grid was cleared and another game begun.

Final Display: Minimax: 5 Opponent: 0'

Qualification Code: 7517
Candidate Number: 3025

Centre Number: 61679

11 – Once all the rounds had been played, the games stopped playing, before 'Final Result:' was displayed before the results of the game, the grid unplayable.



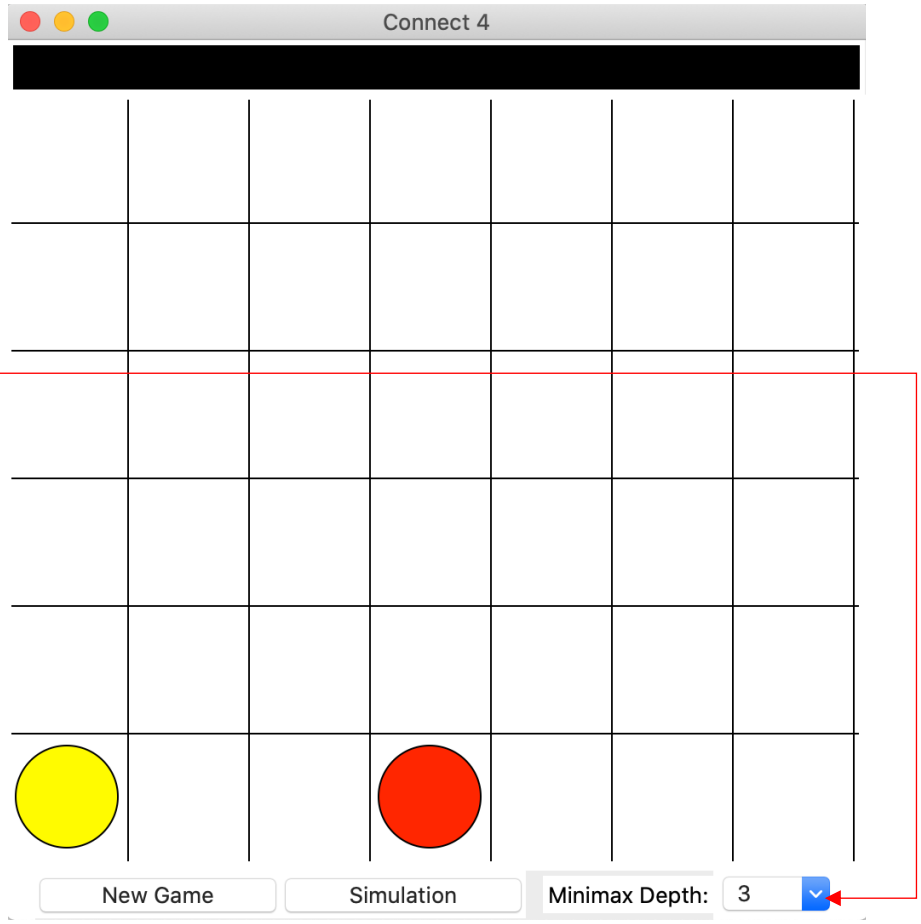
Evidence 4 – Tests 14 & 15

```
print(max_depth)
return best_score
```

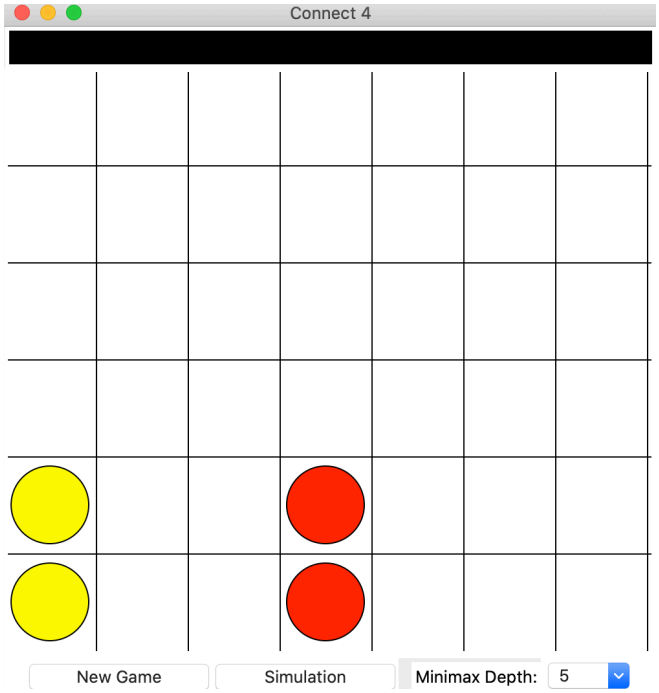
By adding the line above to my Minimax object within the Minimax function, it would always print the maximum depth whenever minimax is called.

I set the Minimax Depth to 3, as seen from the drop-down menu and made a move, then checked the terminal.

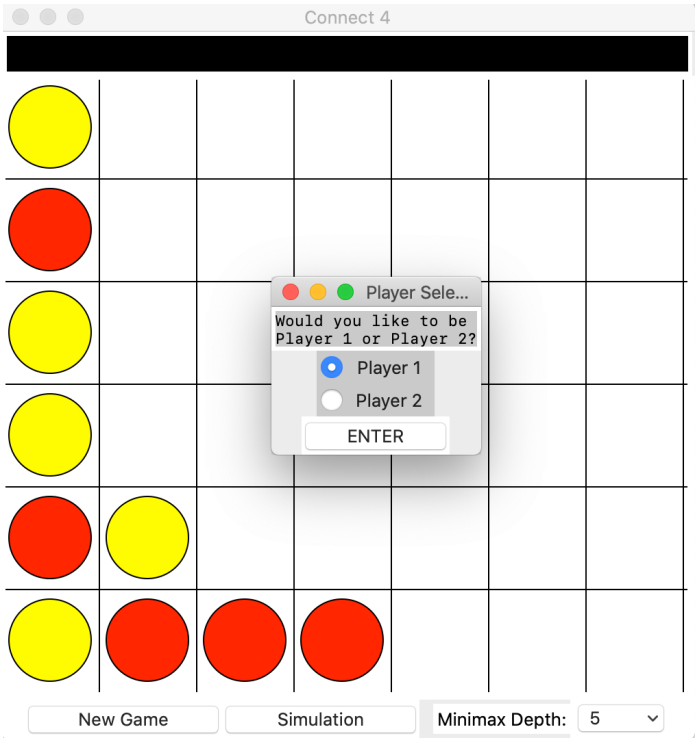
15 – The terminal displayed the depth of '3' repeatedly, and the move was correctly made (with the suboptimal algorithm).



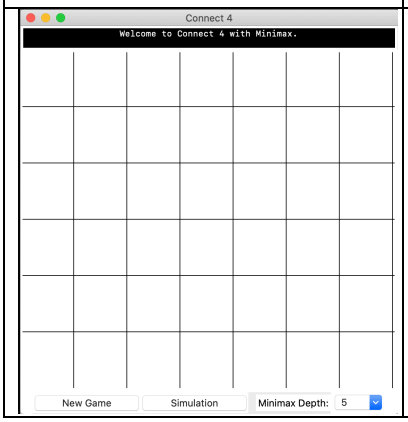
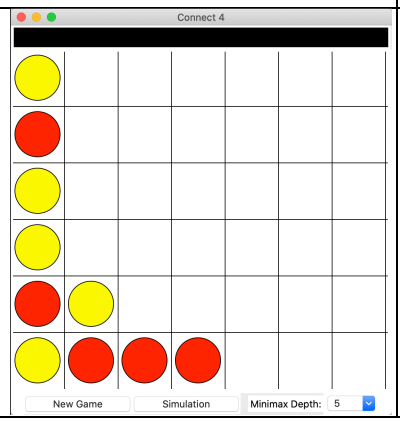
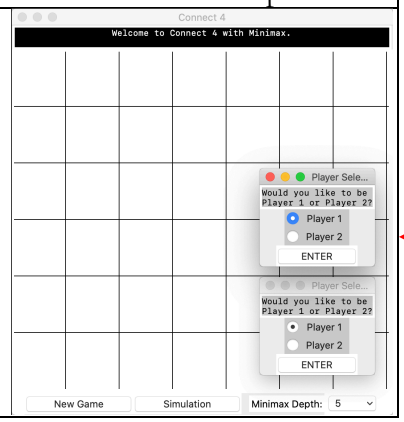
16 – Having changed the depth to 5, I played a secondary move, to find '5' now printed into the terminal and a move made.



Evidence 5 – New Game Screen Tests 3 – 5

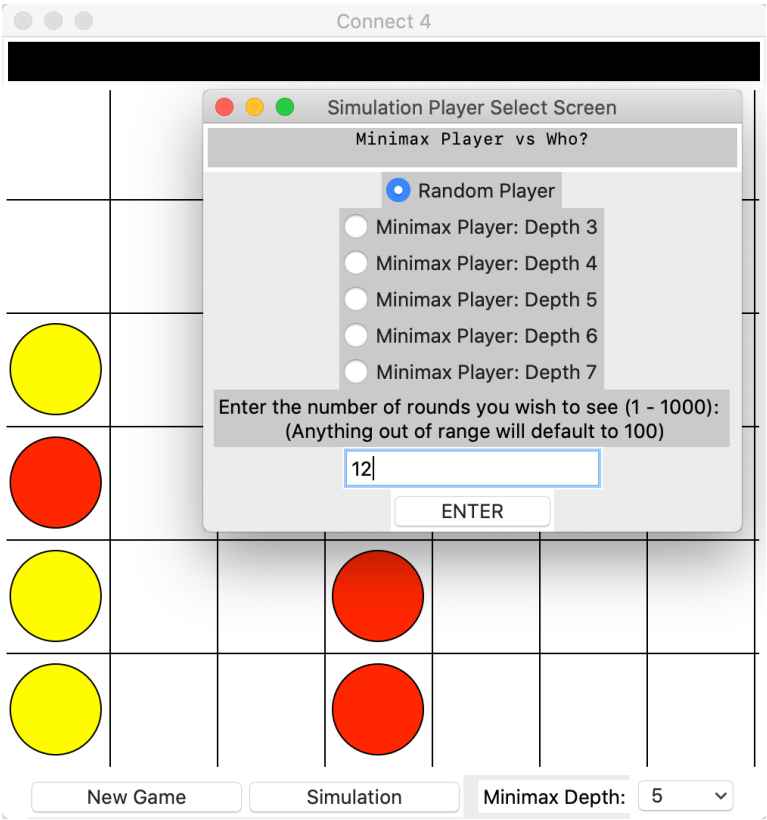


To make the difference more obvious for evidence, I have begun a basic game with 'Suboptimal' Minimax, as otherwise, all results look nearly the same.

3 – Selecting Player 1, then hitting the enter button, the window closed, and the grid was cleared.	4 – Selecting Player 1, then hitting the exit button, the window closed, and the grid remained the same.	5 – Opening two more 'New Game' windows, selecting Player 1 on the first, then hitting the enter button, the window closed, and the grid was cleared, with the other two windows open.
		

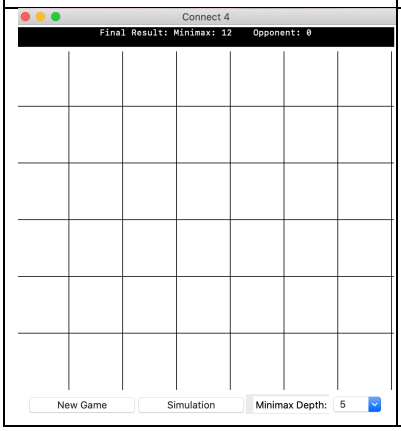
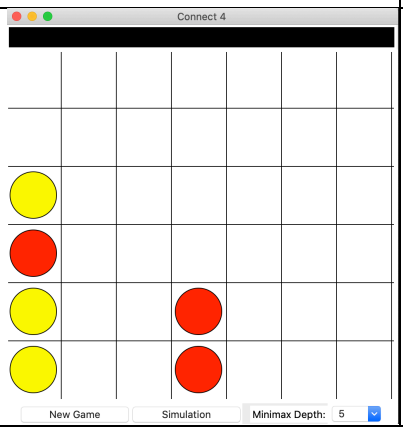
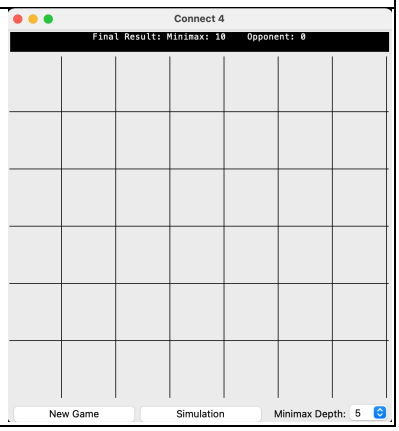
The grid was initially selected and highlighted; I selected the Player Select Screens afterwards to make the screenshot easier to display.

Evidence 6 – Simulation Screen Tests 7 – 10



Similar to the previous evidence, I have played a small game to make the difference obvious in that I am starting a new game.

Different inputs were used in accordance with the test; **12** was simply for the first test.

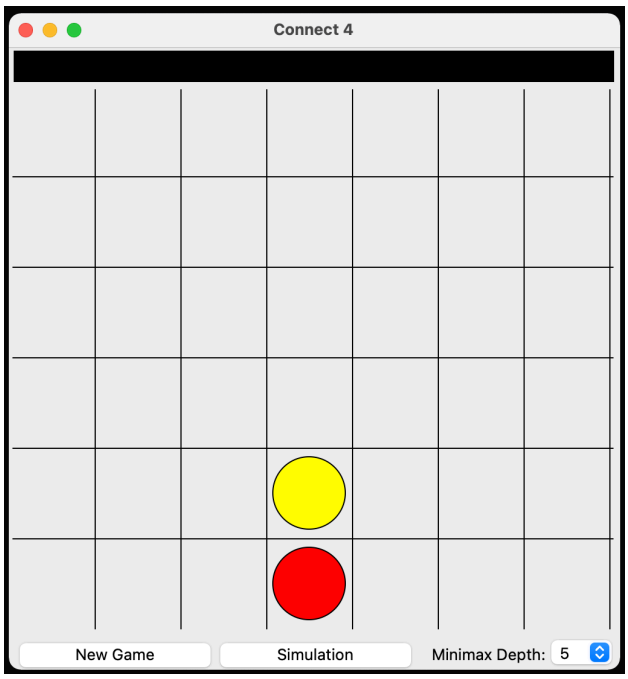
7 – Entering in '12' rounds, the selected the enter button, the screen closed, the grid was cleared and a simulation began of 12 rounds.	8 – Entering in '12' rounds, then selecting the enter button, the screen closed but the grid remained the same.	9 + 10 – The same result occurred for both, where entering in an invalid value ('abc' and 1001) and selecting enter played out a simulation for 10 rounds.
		

System Tests

System Test 1

In this test, I am instead going to be sure the system works together as a whole. To do this, I am going to attempt to open the program, playing a game against the default minimax level. From there, I will start a new game, alter the minimax depth to six, altering it to a depth of three after three moves and continue playing until the game is eventually concluded. The game should not be playable past the outcome of either game until the 'New Game' button is hit and should close fully once I have hit the exit button.

1. I open the program by running the 'connect4main.py' script, generating this screen. I click on the canvas to begin a game, playing a counter in the middle column. The text display quotes 'Minimax thinking…' before a minimax move is played, placing a counter in the middle column.



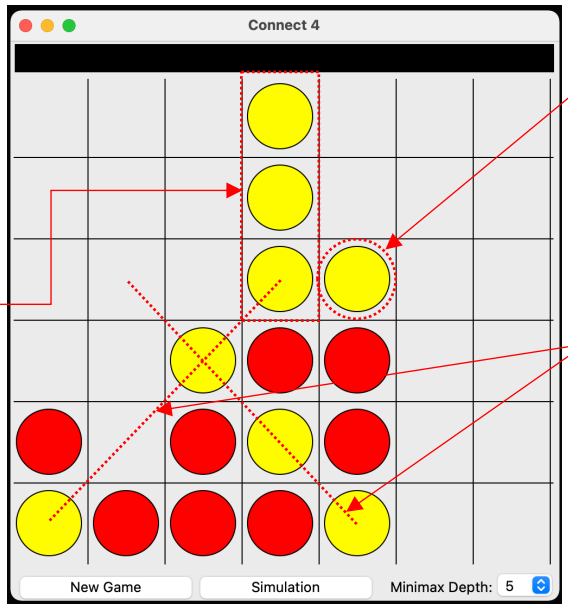
This makes sense according to my evaluation system, as with a depth of five, the total number of counters played on each side is **three**:

0	1	2	3	4	5

Three red, three yellow counters, *maximum*. Therefore, it is impossible for an outcome to have been achieved with four counters in a row.

Therefore, taking from my *table*, the best next move would be the grid square with highest available value. In this case, the *middle* column, with a score of 10.

2. From here, I continue to try and play the 'optimal' move according to the evaluation function, with the minimax continuing to play alongside me. Here, you can see a midpoint of the game, where the minimax has initially played according to the table, before hitting outcomes and acting accordingly to block my moves.

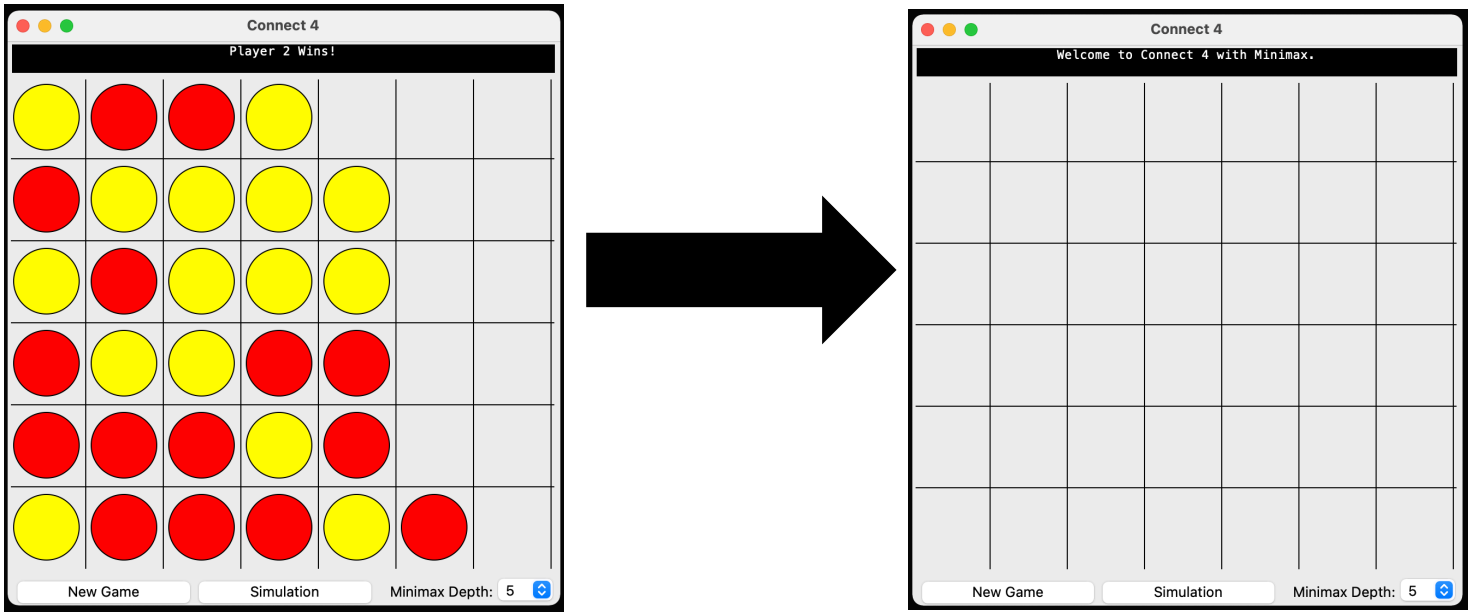


The minimax has successfully identified where I would win from and blocked my movements.

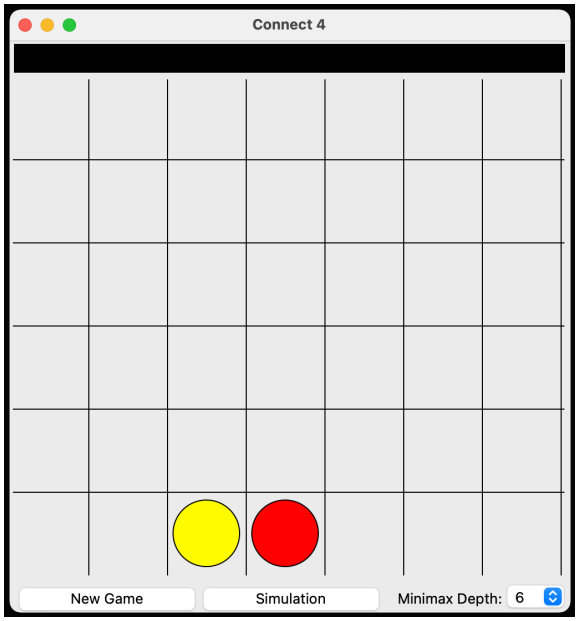
While no outcomes were being met, the minimax continued to play moves in the centre column, severely limiting my possibility of winning diagonally.

We can also see the minimax's own attempts to win, within the two diagonals formed here.

3. The game ends with the Minimax winning, which is shown in the text display above the grid as 'Player 2 Wins!'. I click the 'New Game' button, selecting 'Player 1' and hitting Enter.



4. Once the screen has closed, I alter the value in the 'Minimax Depth:' dropdown menu to 6, rather than 5 and play a move in the middle column. The text display quotes 'Minimax thinking…' before a minimax move is played, placing a counter in the third column.

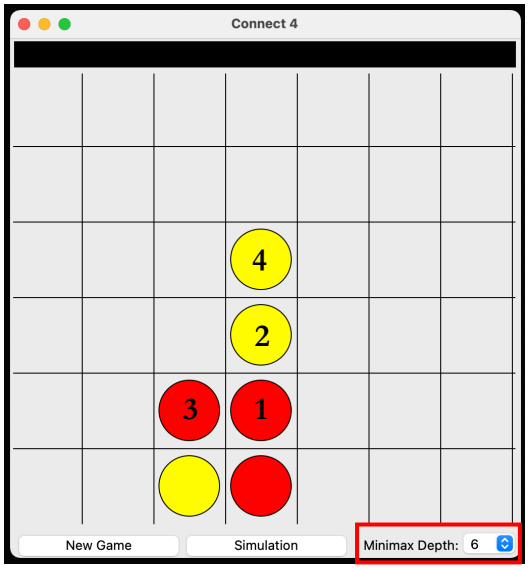


As the minimax is now travelling *further* down the game state tree than a minimax of depth five, it can reach a point at which an *outcome* occurs:

- Three yellow counters
- Four red counters

The outcome could *only* have been a horizontal victory for red on the bottom row, hence why the yellow counter has played at the left of the red, to minimise the options of the other player.

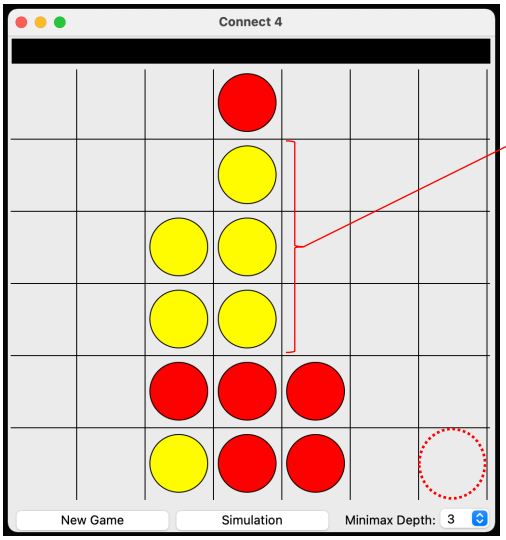
5. I continue to try and play the 'optimal' move two more times, before changing the value of the Minimax depth to 3, continuing to play the game.



Move	Reasoning Observed
1	Continuing to play in the centre column should limit the diagonal movement of my opponent.
2	My opponent has not continued to play along the bottom horizontal. Play in the centre column to limit their diagonal movement.
3	Opening up in the third column should allow me to create more opportunities to win
4	My opponent cannot win in any configuration within the next six moves. Play in the centre column to limit their diagonal movement.

Whilst none of these moves were reliant on the fact that the minimax depth was six, if any move had been made which would allow me to win at this depth, a lower level minimax may have missed it…

6. The difference in the game is noticeable, with this point in the game showing that the minimax has continued to play the move down the centre to limit diagonal movement, missing the moves I made to put myself in a winning scenario.



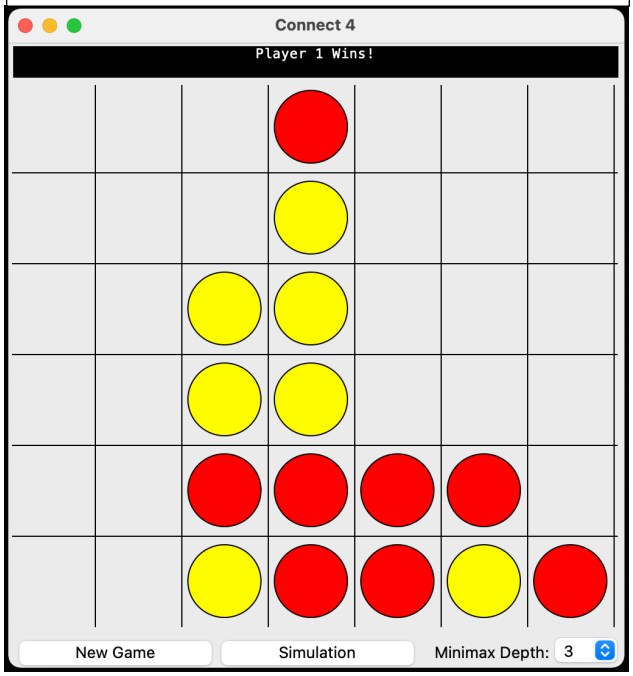
We see that while the minimax mainly played towards the centre of the board…

It has ignored the state where if I am to make a move in the furthest column, I now win.

This is because last turn, it was not able to see the turn past the one where it blocks this move from me.

7. With the outcome of the game being a victory for Player One, I press the close button on the main window, closing down the window and returning me to my VDE.

Final state of the game



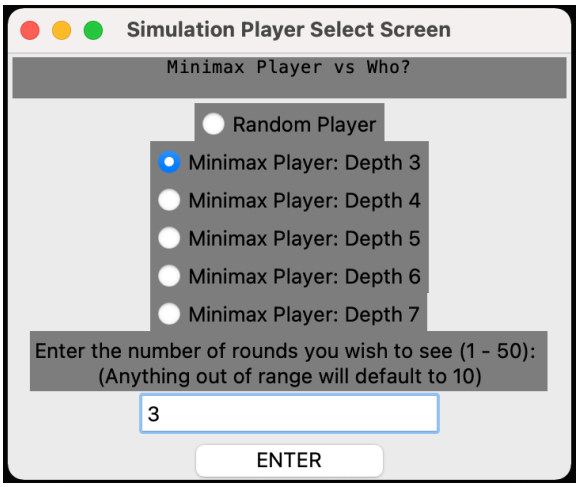
This test allowed me to play a game of Connect 4 against a Minimax algorithm of my choosing. The grid successfully updated to play wherever I clicked, as well as not allowing me to place counters when it wasn't my turn, nor when the game was over. The Minimax algorithm was shown to be competent as well as dynamic in how it played, posing a challenge to the user.

I believe that this system test was successful at achieving a challenging game of Connect 4 against a Minimax opponent.

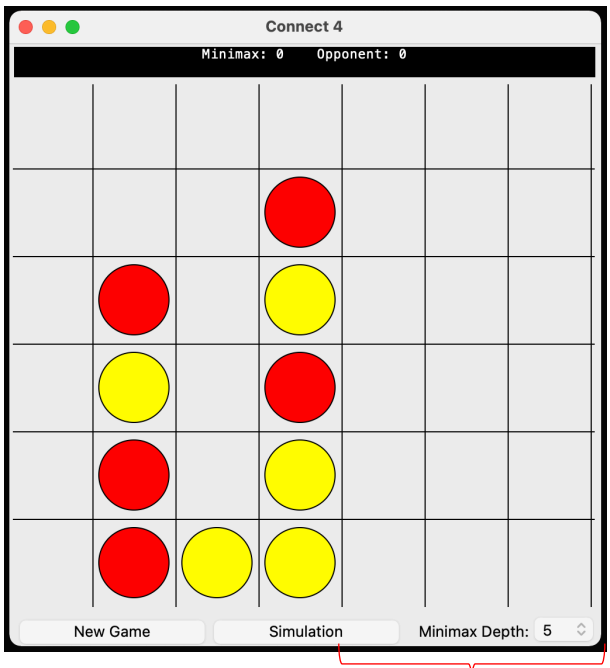
System Test 2

This test will involve running the code again, this time selecting a simulation with two different opponents, at two different minimax levels, running the first for five rounds, and the second for ten rounds, displaying that the rounds, opponents, and minimax levels all function within a simulation run of the game.

1. I open the program again, with the same home screen being displayed to me. I click on the 'Simulation' button, selecting 'Minimax Player: Depth 3', and entering in '5' rounds to be played. From there, I hit enter.



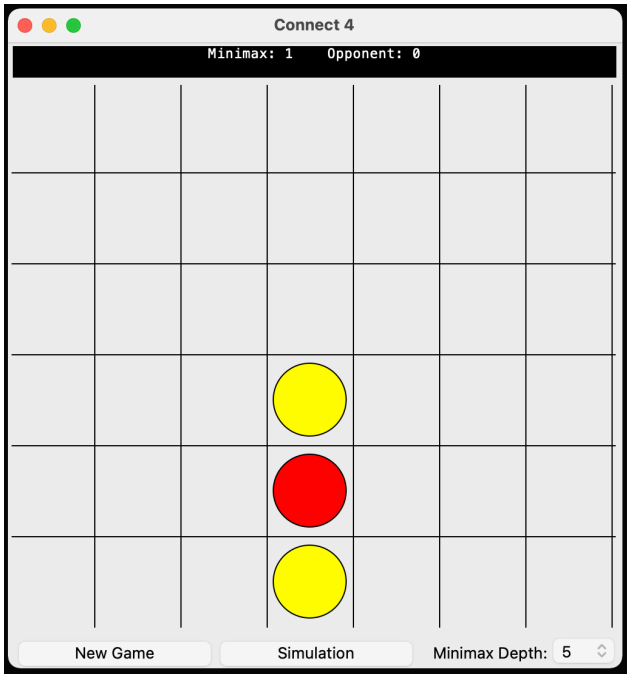
2. Once the screen has closed and the grid is cleared, the game begins playing a faster game within the grid between the two opponents, where even if I click, I cannot play.



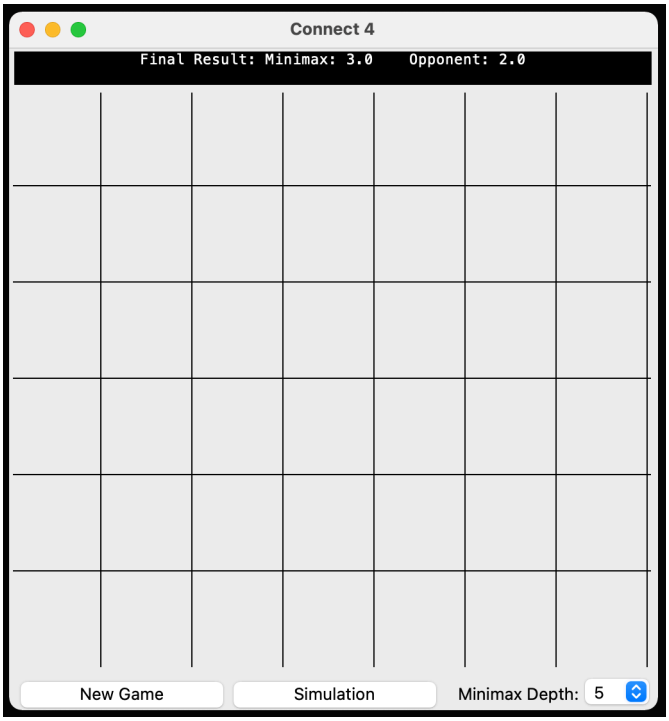
The game *initially* begins slowly, yet as more and more options are eliminated from the game state, it begins to move much faster than the average game.

Throughout this game, I clicked in columns 4, 5 and 6. No moves were made.

3. Once an outcome of the game was reached (in this case, the level 5 Minimax wins), the score for Minimax was incremented once and the grid clears, playing another game between the two from scratch.



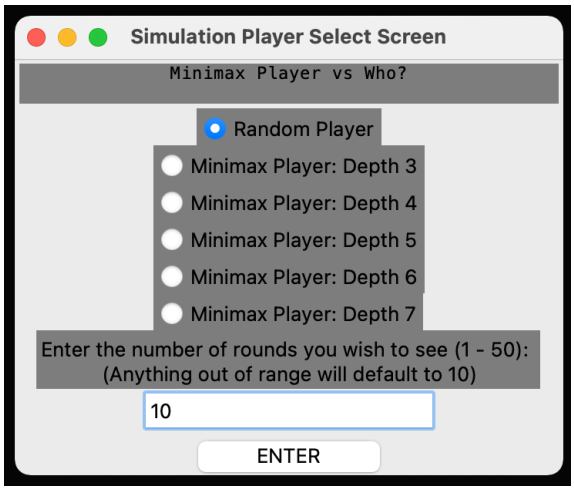
4. The five rounds play out, and the game stops with a cleared grid and the final scores displayed in the text box.



Round	W/D/L?	Minimax	Opponent
1	W	1.0	0.0
2	D	1.5	0.5
3	D	2.0	1.0
4	D	2.5	1.5
5	D	3.0	2.0

As often happens, once the Minimax levels reach similar values, it is very often that they play the same game one after the other, resulting in multiple draws. I noticed that the minimax of depth 3 played marginally faster than 5, as it didn't have to traverse as many game states.

5. From here, I select the 'Simulation' button, selecting 'Random Opponent', and entering in '10' rounds, before hitting enter again.



The radiobuttons functioned as intended, where selecting 'Random Player' *deselected* the current choice, which was at the time, 'Minimax Player: Depth 7'.

As 10 is the default number of rounds, I theoretically could have inputted any non-numerical or numerical value outside of the range, or 10. I decided that to properly simulate an actual experience, I would input 10, as expected.

6. Every round of the game plays out quickly, with the scores being updated in accordance again. The difference between each opponent can be seen in this state, where whilst the random moves have simulated a semi-credible opponent, the Minimax has played as expected, playing closest to the middle-centre of the grid, except where necessary to move elsewhere.

I decided to focus on the **highlighted move** from the diagram.

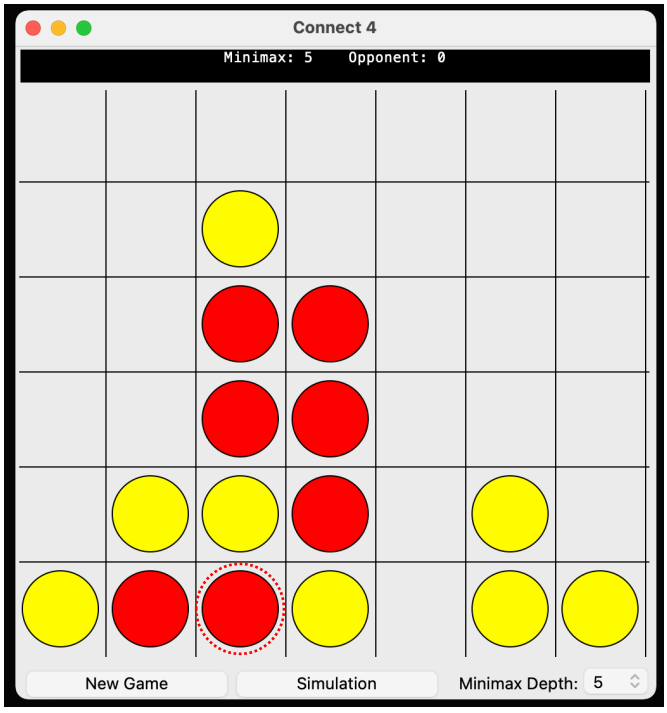
At this point in the game, the Minimax had played multiple moves down the centre column, as shown by the 'tower' visible in the centre column.

Moves made by Random Opponent:
- Fourth Column, First Row
- Sixth Column, First Row

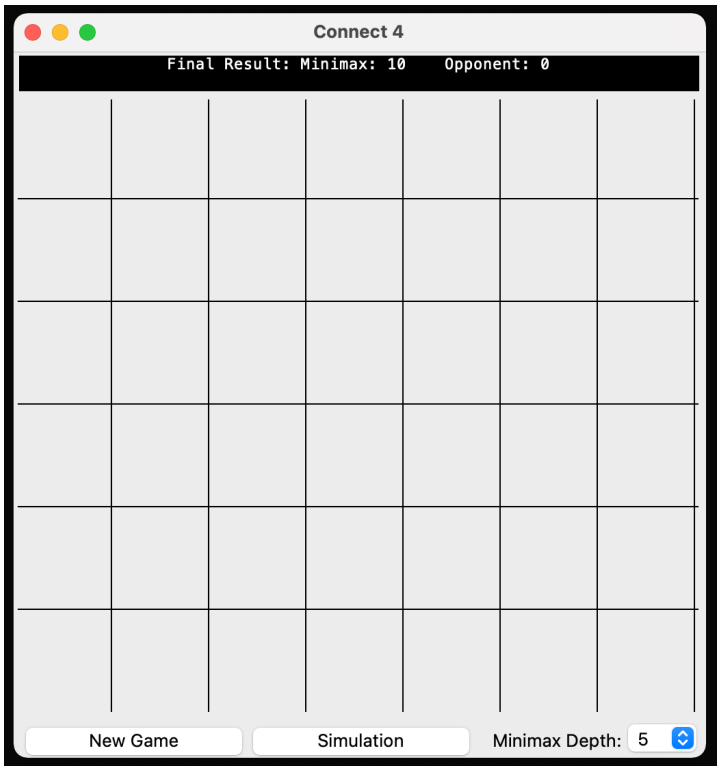
The Minimax made the highlighted move in anticipation of a move into:

- Fifth Column, First Row

This move *would* have allowed the random player a theoretical perfect win in one move, where the Minimax would have been unable to block it, hence why playing in this column limits the movement, whilst remaining as close to the centre as possible.



7. With the final scores being displayed again, I press the close button on the main window, closing down the window and returning me to my VDE.



As expected, my Minimax was able to beat the random opponent 100% of the time. There are *possible* games in which the random opponent only plays actual optimal moves, resulting in a win, although this is simply an implementation of the infinite monkey theorem:

The result is highly unlikely, almost impossibly so.

This test allowed me to observe two different simulations of varying lengths between a minimax algorithm of depth of my choosing and an opponent of my choosing. The text box correctly displayed all scores and information throughout the game, whilst the grid played repeated, fast games between the two selected algorithms. Throughout each simulation, I was unable to interact with the grid, as should happen. There were visible differences in how each opponent played and how my algorithm played in response to these opponents.

I believe that this system test was successful at displaying simulations of Connect 4 between differing opponents and varying rounds.

EVALUATION

General Appraisal

I believe my project has been a success. My coding skills have improved over time, and the experience of working on such a large project has given me a more in-depth view on how to approach large scale programming such as this. As for my analysis, I found it beneficial for understanding Mr Huxley's current issues and addressing them; without it, I may have skewed off into a direction that would have not directly aided him. The design section of my project is what allowed me to successfully plan out my program in an efficient way, whilst also sub sectioning my time for easier management, a skill I hope to carry into other projects such as this. Learning the syntax and understanding Tkinter was a challenge that I was happy to overcome, as I now feel far more confident in addressing larger GUI projects in more streamlined methods, where widgets and grids allow for simpler formatting than previously possible with Python alone. A separate challenge was designing my evaluation function. Whilst not challenging in terms of programming, I found it unique in how it introduced an unexpected problem; there were many ways to calculate the heuristic value, and by approaching it with my own methodology, I was happy to find it successful, as many other, more basic options, such as calculating based off of three in a row two in a row always lead to far more complex and difficult to debug code for an algorithm that behaved no better than my own. Perhaps my biggest issue with this project was the minimax algorithm itself, where, due to the many different classes and alterations made to the code, many corrections and 'trial runs' had to be made to be sure errors weren't being made over time. Despite all of these, however, my object-oriented programming and coding have improved substantially, and I believe I have met all criteria set out to me by Mr Huxley in a substantial way.

Meeting Objectives

No.	Objective	How was it met?	How successfully?	How could it be improved?
1	When loaded, the screen must display a game of Connect Four with buttons to play a new game or simulate a game.	Using Tkinter, I created a window within which a canvas was used to draw an empty grid on top of. Two buttons were then placed within a widget below to act as the 'New Game' and 'Simulation' buttons. A text box was also included to greet and guide the user through the screen and game.	Entirely.	No Improvements.
2	When the Connect Four grid is clicked on, the program must check if it is the player's turn, as well as a valid move3, before placing a counter at the correct row in the column where the player	I created a function which ran upon the event of the canvas being clicked on, containing an if loop, which would only run if the game wasn't over and it was the player's turn. From there, the co-ordinates of the click are found within the canvas and calculated as to which column it was within using the game class' column number. Another if statement checks whether the move is valid, using the 'tot' property of the game object, which if found to be valid,	Entirely.	No Improvements.

	clicked and updating the grid.	is then used to update the grid in the appropriate location. A circular piece is then drawn into the grid accordingly, after which the turn is changed.		
3	If not the correct turn, or the move invalid, then have no move be made.	The if statement mentioned in the above objective which incorporated the 'tot' object checks whether the number of counters in the column is equal to the total rows within the grid using a function within the game class. If it is found that the number of counters in the column is equal to the number of rows, then the text display outputs 'That column is full'. If the canvas is clicked without it being the player's turn, then the first if loop of the function does not run, doing nothing.	Entirely.	No Improvements.
4	The board must be able to check whether the game has been won and display the victor, or a draw with a visual indicator.	I made a function that runs after every move made, taking in the inputs of the current grid. If any vertical, horizontal, or diagonal connect four is found, then the game becomes over and the victor is displayed in the text display above the grid.	Entirely.	No Improvements.
5	When the "New Game" Button is pressed, the program must allow the user to select which player they wish to be, store this value, then clear the grid and start a new game with those values.	For when the button is pressed, I gave it a function which creates a new window with two radiobuttons, an enter button and a text display, asking the player to select which player they want to be. One of the radiobuttons begin selected, and whenever either are selected at any point, a tkinter integer variable (a property of the App class) becomes updated to an integer corresponding to the selected player. Whenever the Enter button is selected, the new window is destroyed and a function is run which sets the mode property to True and clears the grid, where True indicates that the game is not a simulation.	Entirely.	No Improvements.
6	When the "Simulation" Button is pressed, the program must allow the user to select an opponent, as well as the number of rounds of the game to be played,	For when the button is pressed, I gave it a function which creates a new window with two radiobuttons, an enter button, an entry box with a label and a text display, asking the player to select an opponent for the Minimax algorithm to play against. One of the radiobuttons begin selected, and whenever any are selected at any point, a Tkinter integer variable (a property of	Entirely.	No Improvements.

	store these values, then clear the grid start a new game with those values.	the App class) becomes updated to an integer corresponding to the selected player. The entry box for rounds begins with a default of 0, and whenever updated by the user, a Tkinter integer variable (a different property of the App class) becomes updated to the integer entered. Whenever the Enter button is selected, the new window is destroyed and a function is run which sets the mode property to False and clears the grid, indicating that a simulation will begin between the two opponents.		
7	The screen should display a win count of each of the algorithms, as well as their depths / characteristics as the simulation continues.	After each move in the simulation, where the algorithm opponent is decided by the Tkinter integer variable updated in the objective above, a function is run to check whether the game is over. If so, then a new function is run to update the current score, granting one point to the total score of whichever player won; one half going to each player if a draw was achieved. These scores are constantly shown in the text box above the grid throughout the game, updated after an outcome is achieved. Another if loop within the function checks whether the total scores is equivalent to the number of rounds entered, and if so, displays the final scores in the text box.	Partially – I did not find enough time to display information about each algorithm or move, although I did manage to make the movements faster than a game played between human and algorithm.	Include the time taken for each move, or total number of potential game states visited by either of the algorithms after the game is over, displayed alongside the final scores.
	The program should be able to run a minimax algorithm on the current game state to predict an optimal move.	I created a Minimax object which can be instantiated with a game, a player, and an opponent. The class contained three functions, one returning a random valid move, one evaluating the board state, and a third using the other two in a standard minimax format to return an optimal move on the current game object's grid based upon an evaluation function. It takes the current value of the Minimax depth specified by the drop-down menu to use as a maximum depth that the Minimax algorithm could travel to.	Entirely.	A more in depth, accurate evaluation function. Whilst not necessary, as it already returns mostly optimal moves, it would remove the slight chance of a poor move being made.
	The program should be able to make the move on the current grid	Once the Minimax object has been defined, calling the third function known as minimax returns an integer value representing the column in which the next move should be played. I then	Entirely.	No Improvements.

	when the minimax algorithm is called.	call a function belonging to the game class to make a move in the column.		
	The depth of the Minimax should be specifiable by the user and the program should be able to alter it accordingly.	I added a drop-down menu and a label displaying 'Minimax Level:' which, when the user alters the value within it, uses the value's index within a list to find the value, before updating a variable property of the class, which then stores the maximum depth at which the Minimax algorithm operates.	Entirely.	No Improvements.

End-User Feedback

I allowed my end-user to test and use my program, asking for feedback on the result of the project to see if I had strayed too far from his initial idea, or if he could give me any possible critiques. This was the feedback he gave:

Connect Four with Minimax Implementation Feedback Form

1. Do you believe that the final product has successfully encapsulated your initial idea?

Absolutely! The user interface is no more complex than is strictly necessary. I like the theatric 'pause' between each of the computer's moves. The facility to alter the Minimax depth is also very impressive.

2. Are there any sections that exceed/fall below your expectations? If so, which, and why?

Perhaps more sophisticated graphics could have been used, possibly also sound?

3. Is there anything omitted you would have wanted added to the project?

A way of stepping through the moves, after a game, with feedback on where a poor move was made.

4. Is there anything else you would like to mention about the project of note?

I have thoroughly enjoyed the client/developer interaction that has taken place throughout the project.

End User Signature: [signature] Candidate Signature: OEastersbrook

Positives

- The program encapsulates his original idea.
- The simplistic UI is appreciated, as it makes the game simpler.
- The simulation feature is a fantastic way of grasping how the depth affects Minimax, as well as showing how contrasting two AI players can lead to much more interesting games mathematically.
- This program meets all the requirements he gave me to satisfaction, with some section exceeding his expectations.
- The text box is an appreciated addition, as he found it useful to have all information required for him condensed into a singular area.

Negatives

- Some more animations or potentially sound effects would have made the game feel more active as you play, although probably would have not added much.
- The lack of much information other than just the Minimax depth is underwhelming, knowing the time taken, number of potential games visited, etc. would have been a fun addition.
- Other opponents for the Minimax to go against, such as another algorithm altogether, would have been an intriguing addition, although this wasn't part of his initial proposition to me, and therefore, the issue is very minor.

Analysis of Feedback

My end user seemed very happy, suggesting that I had met all criteria he had initially pictured, with many of his critiques ranging into 'extra' detail that he had not specified originally. He stated that the program would be excellent for finding a challenge in a game, as well as locating weaknesses in his own play style. It was by looking through other examples that I decided to use a simplistic UI, as I assumed my own experience as a player would translate well over to Mr Huxley. I was pleased to find not only recognition but praise for this approach, as I feared he would have liked to see the same complexity he expected in the form of sound effects and animations.

I agree with Mr Huxley's critique of my animations, sound effects and opponents making the project feel of little depth, although due to my lack of coding skill and time restraints, I also agree with his comment on these issues being minimal overall, as focusing on other opponents would not have given me the time to focus on learning Tkinter and may have affected the quality of the whole project. As for the lack of information displayed, I find this to be a major flaw of my project. I did not account for where this display would be in my initial design, and by abstaining from creating this, I continued my development with this idea as an afterthought, implying that I failed to reach this section of the criteria for the project. Despite this, the small sections displayed, such as the depth of minimax and total scores after a simulation do display some information, if minimal.

However, overall, Mr Huxley did say that objectives he hoped to be met were met, with the simulation feature and text box receiving particular praise for design, as in the way I had pictured while designing. The desire to display the algorithm in a simulation against others meant that what originally seemed to be an obscene number of rounds was a great way to keep the game running in the background of his Computer Science society and interest any passers-by. Meanwhile, the condensed information being displayed through the text box was equally

beneficial, as it prevented focus being drawn away from the game. Both of these were complimented by Mr Huxley, and I am happy that my thoughts in design have come to fruition.

Suggested Improvements

If I were to make changes, then my main objectives would be:

- New simulation opponents of differing algorithms
- A way to alter the number of 'default' rounds
- A way to display information of the Minimax algorithm

All of these were created by either a lack of coding skill, or the time limit on this project. The Simulation opponents would not only pose a more interesting challenge, yet also hopefully allow the player to decide the 'optimal' AI to play the game and can use this to find the 'optimal game' that Mr Huxley was interested in finding. The 'default' rounds being altered would make the program not only more accessible to any potential user, yet also help avoid a problem I found while testing the game myself; if any minor error is made inputting rounds, the player is then frozen into a game of 10 rounds with no way out other than to close and reopen the program. The information being displayed would allow those interested in the maths and calculation sections of Connect 4 and the algorithm, being more useful for understanding how and why each move is made where it is, as would be useful for Mr Huxley in understanding the game.

To create new simulation opponents, I could either investigate into other kinds of algorithms such as MCTS or create a *deep learning algorithm* to try and achieve the optimal game itself through guided learning. This would prove difficult, given my coding skill, whilst also limiting the mathematics and explanation I can give to Mr Huxley, making this choice possibly harmful to the overall project.

The number of default rounds being altered would potentially call for another window to be created; a 'settings' window. Whilst a TkInter IntVar could be used to implement the change once this window is created, this settings window could house other possibilities, such as the ability to save and load games. As for how useful these additions would be, due to Connect4's short length, it is most likely limited.

Appendix

Connect4.py

```
from enum import Enum

class Result(Enum):
    """ An Enumerator referenced by the Connect4 class to differentiate between
different 'states' of the game.
    """
    NONE = 0
    P1WIN = 1
    P2WIN = 2
    DRAW = 3

class Connect4:
    """ A playable game of Connect4 with functions allowing it to be played with
validation checks to find valid inputs and outcomes to the game. It updates
properties of the object to reflect the current state of the game.
    """
    P1 = 1
    P2 = 2
    COLS = 0
    ROWS = 0

    def __init__(self, COLS, ROWS):
        """ Initialises the game object, creating a grid, a total occupied spaces
per column, the current turn and the result of the game.

        Args:
            COLS (int): The number of columns desired in the grid
            ROWS (int): The number of rows desired in the grid
        """
        Connect4.COLS = COLS
        Connect4.ROWS = ROWS
        self.grid = self.create_grid()
        self.tot = self.create_tot()
        self.turn = Connect4.P1
        self.result = Result.NONE

    def create_grid(self):
        """ Returns a 2D array with ROWS number of rows and COLS number of columns,
filled with zeroes, which represent empty spaces in the grid.

        Returns:
            2D array: An array containing ROWS arrays, each filled with COLS
zeroes.
        """
        grid = []
        for i in range(Connect4.ROWS):  #Creating the blank 'rows'.
            grid.append([])
            for j in range(Connect4.COLS):  #Adding zeroes to each 'row'.
```

```
                grid[i].append(0)
        return grid

    def return_grid(self):
        """ Returns the grid.

        Returns:
            2D array: Represents the grid
        """
        return self.grid

    def create_tot(self):
        """ Returns an array of zeroes, COLS items long, representing the number of
counters in each column of the grid.

        Returns:
            array: An array containing COLS zeroes
        """
        tot = []
        for i in range(Connect4.COLS):
            tot.append(0)
        return tot

    def valid_move(self, move):
        """ Returns False if the column specified by move is full or outside of the
grid.

        Args:
            move (int): The column in which a move is trying to be made

        Returns:
            bool: Whether or not the move is valid
        """
        if self.tot[move] == Connect4.ROWS or move > Connect4.COLS-1 or move < 0:
            return False
        return True

    def change_turn(self):
        """ Changes the current turn
        """
        self.turn = Connect4.P2 if self.turn == Connect4.P1 else Connect4.P1

    def checkwin(self):
        """ Checks whether an outcome has been achieved. Checks if either all
columns are full or if four adjacent spaces within the grid are not empty and are
the same colour horizotally, vertically, or diagonally.

        Returns:
            boolean: Whether or not an outcome has been reached
        """
```

```
        for row in self.grid:  #Remember grid upside down!
            for col in range(Connect4.COLS-3):  #Horizontal victories
                if len(set(row[col:col+4])) == 1 and row[col] != 0:
                    self.result = Result.P1WIN if self.turn == 1 else Result.P2WIN
                    return True
        for row in zip(*self.grid):  #Vertical victories
            for col in range(3):
                if len(set(row[col:col+4])) == 1 and row[col] != 0:
                    self.result = Result.P1WIN if self.turn == 1 else Result.P2WIN
                    return True
        for row in range(3,Connect4.ROWS):  #Diagonal victories - bottom left to
top right
            for col in range(0,4):  #Only possible to 'start' these victories from
the bottom left square
                if self.grid[row][col] == self.turn and self.grid[row][col] ==
self.grid[row-1][col+1] and self.grid[row-1][col+1] == self.grid[row-2][col+2] and
self.grid[row-2][col+2] == self.grid[row-3][col+3]:
                    self.result = Result.P1WIN if self.turn == 1 else Result.P2WIN
                    return True
        for row in range(3,Connect4.ROWS):  #Diagonal victories - bottom right to
top left
            for col in range(3, Connect4.COLS):   #Only possible to 'start' these
victories from the bottom right square
                if self.grid[row][col] == self.turn and self.grid[row][col] ==
self.grid[row-1][col-1] and self.grid[row-1][col-1] == self.grid[row-2][col-2] and
self.grid[row-2][col-2] == self.grid[row-3][col-3]:
                    self.result = Result.P1WIN if self.turn == 1 else Result.P2WIN
                    return True
        for i in range(Connect4.COLS):
            if self.tot[i] != Connect4.ROWS:
                self.result = Result.NONE
                return False
        self.result = Result.DRAW
        return True

    def make_move(self, move):
        """ Updates an item at a row specified by the value in the index of a
column of tot, and a column specified by a column to the current turn, before
incrementing the value in the index of a column of tot by +1.

        Args:
            move (int): A column in which a move is to be made on the grid
        """
        self.grid[self.tot[move]][move] = self.turn
        self.tot[move] += 1
        self.checkwin() #The game will always be unplayable after the winning move
is made; function will only end after grid has been checked.
        self.change_turn()

    def undo_move(self, move):
```

```
        """ Increments the value in the index of a column of tot by -1, before
updating an item at a row specified by the value in the index of a column of tot,
and a column specified by a column to the zero.

        Args:
            move (int): A column in which a move is to be unmade on the grid
        """
        self.tot[move] -= 1
        self.grid[self.tot[move]][move] = 0
        self.checkwin() #The game can be updated from an unplayable state, allowing
for an 'unmade' move to allow the Minimax to keep exploring depths.
        self.change_turn()

    def game_over(self):
        """ Returns True if an outcome has occurred.

        Returns:
            boolean: Whether the game has ended or not
        """
        return self.result != Result.NONE

    def grid_clear(self):
        """ Creates a new grid and tot, before assigning them to the class
properties grid and tot.
        """
        self.grid = self.create_grid()
        self.tot = self.create_tot()
        self.result = Result.NONE
        self.turn = Connect4.P1

    def __repr__(self):
        """ Represents the grid.
        """
        for i in range(len(self.grid)):
            print(self.grid[-i-1])
```

Connect4GUI.py

```
import tkinter
import random
from tkinter import ttk, Tk, Canvas, Frame, Button, Text, Radiobutton, Entry,
Label, StringVar, IntVar, OptionMenu
from Connect4 import Connect4, Result
from MinimaxAttempt import Minimax
from log_config import logging


class Piece():
    """ Stores the radius of any piece placed on the grid.
    """
    R = 30
```

```
class App():
    """ Stores and represents the main window of the project as a TkInter root with
class methods which allow it to be dynamic and interactive with the user. Uses a
Connect4 object in order to display and play the game graphically.
    """
    WINDOW_WIDTH = 700
    WINDOW_HEIGHT = 600
    CANVAS_WIDTH = 500
    CANVAS_HEIGHT = 450
    colour = ['red', 'yellow']

    def __init__(self, game, player):
        """ Initialises the Home Screen as a TkInter root with a known Mode,
Player, and Max_Depth, each of which determine the type of game the Minimax will
play.

        Args:
            game (object): A predefined Connect4 object
            player (int): Either 1 or 2, deciding either which player the user is
or which simulation opponent is being faced, depending on the 'mode' attribute
        """
        self.player = player
        self.mode = True
        self.root = Tk()
        self.max_depth = tkinter.IntVar()
        max_depth_options = [3, 4, 5, 6, 7]
        self.max_depth.set(max_depth_options[2])
        self.p = tkinter.IntVar()
        self.rounds = tkinter.StringVar()   #By defining these as TkInter
Variables, you can easily set them using buttons on separate screens. Use .get(),
.set().

        self.root.title("Connect 4")
        self.root.frame = ttk.Frame(self.root, width=App.WINDOW_WIDTH)
        self.root.frame.grid(row=0, column=0)
        self.root.text = Text(self.root.frame, bg = 'black', fg = 'white', width =
71, selectborderwidth = 1, height = 1.5, pady = 0, padx = 0)
        self.root.text.insert('1.0', "Welcome to Connect 4 with Minimax.")
        self.root.text.tag_configure("tag_name", justify='center')
        self.root.text.tag_add("tag_name", "1.0", "end")
        self.root.text.grid(row=1,column=1)
        self.root.canvas = Canvas(self.root.frame, width=App.CANVAS_WIDTH,
height=App.CANVAS_HEIGHT)
        self.root.canvas.grid(row=2, column=1)
        self.root.canvas.bind('<Button-1>', self.canvas_click)
        self.root.buttonholder = ttk.Frame(self.root, width=App.WINDOW_WIDTH)
        self.root.buttonholder.grid(row = 3, column = 0)
        self.root.button1 = Button(self.root.buttonholder, text = 'New Game',
activebackground = 'yellow', bg = 'grey', command = self.player_select_screen,
height = 1, justify = 'center', width = 8, padx = 30)
        self.root.button1.pack(side = "left")
```

```
        self.root.dropdown = OptionMenu(self.root.buttonholder, self.max_depth,
*max_depth_options)
        self.root.dropdown.pack(side = "right")
        self.root.description = Label(self.root.buttonholder, text = 'Minimax
Depth:')
        self.root.description.pack(side = "right", padx = (10,0))
        self.root.button2 = Button(self.root.buttonholder, text = 'Simulation',
activebackground = 'yellow', bg = 'grey', command = self.s_player_select_screen,
height = 1, justify = 'center', width = 8, padx = 30)
        self.root.button2.pack(side = "right")

        self.new_game(game)

        self.draw()
        self.root.mainloop()

    def draw(self):
        """ Clears the canvas in the centre of the window, before drawing the grid
and current pieces on top of it.
        """
        self.root.canvas.delete('all')
        self.draw_grid()
        self.draw_pieces()

    def new_game(self, game):
        """ Creates a new game and sets it to the class property.

        Args:
            game (object): A predefined Connect4 object
        """
        assert type(game) == Connect4 and game is not None
        self.game = game

    def delay(self, function):
        """ Runs a function after a 100ms rest without running any other lines of
code before the function has been completed. This is here due to the after()
function in TkInter continuing to run further lines in the code despite the initial
intended function not having been run yet

        Args:
            function (function): Any function
        """
        self.root.after(100, function)  #Use whenever 'animating' Minimax.

    def final_result(self):
        """ Updates the text display above the grid to a message either indicating
which player has won the game, or whether the game was a draw
        """
        self.root.text.delete('1.0', '100.0')
        if self.game.result == Result.P1WIN:
            self.root.text.insert('1.0', f'Player 1 Wins!')
```

```
            self.root.text.tag_add("tag_name", "1.0", "end")
        if self.game.result == Result.P2WIN:
            self.root.text.insert('1.0', f'Player 2 Wins!')
            self.root.text.tag_add("tag_name", "1.0", "end")
        if self.game.result == Result.DRAW:
            self.root.text.insert('1.0', f'DRAW!')
            self.root.text.tag_add("tag_name", "1.0", "end")

    def s_final_result(self, win_count_minimax, win_count_opponent):
        """ Updates the running total scores for the Minimax algorithm and it's
opponent during a simulation, drawing them each in the text display above the grid,
before returning them.

        Args:
            win_count_minimax (double): The current number of victories achieved by
the Minimax function (draws are counted as half a win)
            win_count_opponent (double): The current number of victories achieved
by the opponent function (draws are counted as half a win)

        Returns:
            tuple: both scores updated in accordance to the outcome achieved
        """
        if self.game.result == Result.P1WIN:
            win_count_minimax += 1
        elif self.game.result == Result.P2WIN:
            win_count_opponent += 1
        elif self.game.result == Result.DRAW:
            win_count_minimax += 0.5
            win_count_opponent += 0.5
        self.root.text.delete('1.0', '100.0')
        self.root.text.insert('1.0', f'Minimax: {win_count_minimax}     Opponent:
{win_count_opponent}')
        self.root.text.tag_add("tag_name", "1.0", "end")
        self.game.grid_clear()
        self.game.change_turn()
        return win_count_minimax, win_count_opponent    #Due to draws, this is a
'double' data type tuple.

    def canvas_click(self, event):
        """ If the canvas is clicked on, this function checks whether it is an
ongoing game, the player's turn, and a game, rather than a simulation. If true,
then the x-co-ordinates of the click are calculated to find which column the click
took place in. If valid, a counter is played at the correct height in the column,
otherwise 'That column is full.' is displayed in a text box and no move is made.

        Args:
            event (object): Stores attributes about a click that occured on a
canvas on a canvas
        """
        if not self.game.game_over() and self.game.turn == self.player and
self.mode == True:
```

```
            if 0 < event.x < App.CANVAS_WIDTH and 0 < event.y < App.CANVAS_HEIGHT:
                move = self.find_column(event.x)
                if self.game.valid_move(move):
                    self.game.make_move(move)
                    self.draw()
                    self.root.update_idletasks()    #Updates the grid while the
game is still being played, rather than after Minimax plays.
                    if self.game.game_over():
                        self.final_result()
                    else:
                        self.root.text.delete('1.0', '100.0')
                        self.root.text.insert('1.0', 'Minimax thinking...')
                        self.root.text.tag_add("tag_name", "1.0", "end")
                        self.root.update_idletasks()
                        self.root.after(500,
self.min_max_move(self.max_depth.get()))
                        self.root.text.delete('1.0', '100.0')
                        self.draw()
                        if self.game.game_over():
                            self.final_result()
                else:
                    self.root.text.delete('1.0', '100.0')
                    self.root.text.insert('1.0', 'That column is full.')
                    self.root.text.tag_add("tag_name", "1.0", "end")


    def simulation(self, win_count_minimax, win_count_opponent):
        """ Decides which opponent the minimax algorithm is facing, before playing
a game of them against one another, checking whether a win has been achieved every
turn. If so, then final_result is called, and if not, the function is called
recursively. If the maximum number of rounds has been played, the text box changes
from the scores of each player, being prepended with 'Final score:'.

        Args:
            win_count_minimax (double): The current number of victories achieved by
the Minimax function (draws are counted as half a win)
            win_count_opponent (double): The current number of victories achieved
by the opponent function (draws are counted as half a win)
        """
        self.root.text.delete('1.0', '100.0')
        self.root.text.insert('1.0', f'Minimax: {win_count_minimax}    Opponent:
{win_count_opponent}')
        self.root.text.tag_add("tag_name", "1.0", "end")
        if win_count_minimax + win_count_opponent != self.rounds:
            if self.player == 1:    #self.player remains SEPARATE from self.turn,
therefore this can stay
                self.delay(self.min_max_move(self.max_depth.get())) if
self.game.turn == self.game.P1 else self.delay(self.random_move())
            else:
                self.delay(self.min_max_move(self.max_depth.get())) if
self.game.turn == self.game.P1 else self.delay(self.min_max_move(self.player+1))
```

```
            if self.game.game_over():
                win_count_minimax, win_count_opponent =
self.s_final_result(win_count_minimax, win_count_opponent)
            self.draw()
            self.root.update_idletasks()

            self.simulation(win_count_minimax, win_count_opponent)
        else:
            self.root.text.insert('1.0', 'Final Result: ')
            self.root.text.tag_add("tag_name", "1.0", "end")
            self.rounds = tkinter.StringVar()   #Otherwise .get() and .set() do not
work on a second simulation

    def random_move(self):
        """ Makes a random, valid move.
        """
        if not self.game.game_over():
            move = random.randint(0,6)
            while self.game.valid_move(move) == False:
                move = random.randint(0,6)
            self.game.make_move(move)

    def min_max_move(self, max_depth):
        """ Creates a Minimax object and calls minimax on the current grid with a
max depth, making a move in the column returned.

        Args:
            max_depth (int): The maximum 'depth' that the Minimax object can
explore, or the maximum number of moves that can be made on the grid by the Minimax
object
        """
        if not self.game.game_over():
            minimax = Minimax(self.game)
            minimax.minimax(0, max_depth, self.game.turn)
            if self.game.valid_move(minimax.best_move):
                self.game.make_move(minimax.best_move)
            else:
                self.random_move()

    def draw_grid(self):
        """ Draws lines over the canvas at equal intervals to create a grid.
        """
        for i in range(0, App.CANVAS_WIDTH, App.CANVAS_WIDTH // Connect4.COLS):
            self.root.canvas.create_line(i, 0, i, App.CANVAS_HEIGHT)
        for i in range(0, App.CANVAS_HEIGHT, App.CANVAS_HEIGHT // Connect4.ROWS):
            self.root.canvas.create_line(0, i, App.CANVAS_WIDTH, i)

    def draw_pieces(self):
        """ Draws circles at each point in the grid where specified by the game
object, the colour corresponding to each player.
        """
```

```
        grid = self.game.return_grid()[::-1]
        for row in range(len(grid)):
            for col in range(len(grid[row])):
                if grid[row][col] != 0:
                    self.colour = App.colour[0] if grid[row][col] == Connect4.P1
else App.colour[1]
                    COL_WIDTH = App.CANVAS_WIDTH // Connect4.COLS
                    ROW_HEIGHT = App.CANVAS_HEIGHT // Connect4.ROWS
                    self.circle((COL_WIDTH*col) + (COL_WIDTH // 2),
(ROW_HEIGHT*row) + (ROW_HEIGHT // 2))

    def find_column(self, x):
        """ Calculates the column at which an x co-ordinate is within.

        Args:
            x (double): The x co-ordinate along the canvas at which a click occured

        Returns:
            double: The column in which the canvas was clicked
        """
        return x // (App.CANVAS_WIDTH // Connect4.COLS)

    def circle(self, x, y):
        """ Draws a circle at an x and y co-ordinate, with radius r.

        Args:
            x (double): The x co-ordinate along the canvas at which a click occured
            y (double): The y co-ordinate along the canvas at which a click occured
        """
        xl = x - Piece.R
        yl = y - Piece.R
        xr = x + Piece.R
        yr = y + Piece.R
        self.root.canvas.create_oval(xl, yl, xr, yr, fill=self.colour)

    def return_to_game(self, mode):
        """ Updates whether the game is a simulation or active game, clears the
grid, sets the player, and begins either a simulation, or a game, updating the text
display to reflect this.

        Args:
            mode (boolean): False if the new game is a simulation, True otherwise
        """
        self.mode = mode
        self.game.grid_clear()
        self.draw()
        self.player = int(self.p.get())
        if self.mode == True:
            self.root.text.delete('1.0', '100.0')
            self.root.text.insert('1.0', "Welcome to Connect 4 with Minimax.")
            self.root.text.tag_configure("tag_name", justify='center')
```

```
            self.root.text.tag_add("tag_name", "1.0", "end")
            if self.player == self.game.P2:
                self.root.after(1000, self.min_max_move(self.max_depth.get()))
                self.draw()
        else:
            self.game.turn = self.game.P1
            self.rounds = int(self.rounds.get()) if self.rounds.get().isdigit()
else 10     #Rounds validation
            if self.rounds < 1 or self.rounds > 50:
#Also rounds validation
                self.rounds = 10
            self.root.after(2000, self.simulation(0,0))

    def player_select_screen(self):
        """ When the 'New Game' button is pressed, a new window is created with a
text display, two radiobuttons and an Enter button, where the player can select
which player they want to be in the new game.
        """
        player_select_screen = tkinter.Toplevel(self.root)
        player_select_screen.title("Player Select Screen")
        player_select_screen.frame = ttk.Frame(player_select_screen,
width=App.WINDOW_WIDTH)
        player_select_screen.frame.grid(row=0, column=0)
        player_select_screen.text = Text(player_select_screen.frame, bg = 'grey',
fg = 'black', width = 21, selectborderwidth = 1, height = 1.5, pady = 0, padx = 0)
        player_select_screen.text.insert('1.0', "Would you like to be Player 1 or
Player 2?")
        player_select_screen.text.tag_configure("tag_name", justify='center')
        player_select_screen.text.tag_add("tag_name", "1.0", "end")
        player_select_screen.text.grid(row=1,column=1)
        player_select_screen.player_choice1 =
Radiobutton(player_select_screen.frame, text = 'Player 1', bg = 'grey', variable =
self.p, value = 1, command = self.p.set(1), height = 1, justify = 'center', width =
10)
        player_select_screen.player_choice2 =
Radiobutton(player_select_screen.frame, text = 'Player 2', bg = 'grey', variable =
self.p, value = 2, command = self.p.set(2), height = 1, justify = 'center', width =
10)
        player_select_screen.player_choice1.grid(row=2,column=1)
        player_select_screen.player_choice2.grid(row=3,column=1)
        player_select_screen.enter = Button(player_select_screen.frame, text =
'ENTER', bg = 'grey', command = lambda: [player_select_screen.destroy(),
self.return_to_game(True)], height = 1, justify = 'center', width = 10)
        player_select_screen.enter.grid(row=4,column=1)
        player_select_screen.mainloop()

    def s_player_select_screen(self):
        """ When the 'Simulation' button is pressed, a new window is created with a
text display, two radiobuttons, an entry box and an Enter button, where the player
can select which opponent they want their minimax to face, and for how many rounds.
        """
```

```
        s_player_select_screen = tkinter.Toplevel(self.root)
        s_player_select_screen.title("Simulation Player Select Screen")
        s_player_select_screen.frame = ttk.Frame(s_player_select_screen,
width=App.WINDOW_WIDTH)
        s_player_select_screen.frame.grid(row=0, column=0)
        s_player_select_screen.text = Text(s_player_select_screen.frame, bg =
'grey', fg = 'black', width = 50, selectborderwidth = 1, height = 1.5, pady = 0,
padx = 0)
        s_player_select_screen.text.insert('1.0', "Minimax Player vs Who?")
        s_player_select_screen.text.tag_configure("tag_name", justify='center')
        s_player_select_screen.text.tag_add("tag_name", "1.0", "end")
        s_player_select_screen.text.grid(row=1,column=1)
        s_player_select_screen.player_choice1 =
Radiobutton(s_player_select_screen.frame, text = 'Random Player', bg = 'grey',
variable = self.p, command = self.p.set(1), value= 1, height = 1)
        s_player_select_screen.player_choice2 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax Player: Depth 3', bg =
'grey', variable = self.p, command = self.p.set(2), value= 2, height = 1)
        s_player_select_screen.player_choice3 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax Player: Depth 4', bg =
'grey', variable = self.p, command = self.p.set(3), value= 3, height = 1)
        s_player_select_screen.player_choice4 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax Player: Depth 5', bg =
'grey', variable = self.p, command = self.p.set(4), value= 4, height = 1)
        s_player_select_screen.player_choice5 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax Player: Depth 6', bg =
'grey', variable = self.p, command = self.p.set(5), value= 5, height = 1)
        s_player_select_screen.player_choice6 =
Radiobutton(s_player_select_screen.frame, text = 'Minimax Player: Depth 7', bg =
'grey', variable = self.p, command = self.p.set(6), value= 6, height = 1)
        s_player_select_screen.round_entry_label =
Label(s_player_select_screen.frame, anchor = 'w', bg = 'grey', text = 'Enter the
number of rounds you wish to see (1 - 50): \n (Anything out of range will default
to 10)')
        s_player_select_screen.round_entry = Entry(s_player_select_screen.frame, bg
= 'white', textvariable = self.rounds)
        s_player_select_screen.player_choice1.grid(row=2,column=1)
        s_player_select_screen.player_choice2.grid(row=3,column=1)
        s_player_select_screen.player_choice3.grid(row=4,column=1)
        s_player_select_screen.player_choice4.grid(row=5,column=1)
        s_player_select_screen.player_choice5.grid(row=6,column=1)
        s_player_select_screen.player_choice6.grid(row=7,column=1)
        s_player_select_screen.round_entry_label.grid(row=8,column=1)
        s_player_select_screen.round_entry.grid(row=9,column=1)
        s_player_select_screen.enter = Button(s_player_select_screen.frame, text =
'ENTER', bg = 'grey', command = lambda: [s_player_select_screen.destroy(),
self.return_to_game(False)], height = 1, justify = 'center', width = 10)
        s_player_select_screen.enter.grid(row=10,column=1)
        s_player_select_screen.mainloop()
```

Minimax.py

```
from Connect4 import Connect4, Result
from time import time
import math
import random
from log_config import logging

class Minimax:
    """ Stores the best move, initially a random move, to be made on a Connect4
game. Has class methods allowing it to calculate the best move via evaluating the
grid in terms of positive and negative score.
    """

    def __init__(self, game):
        """ Initialises the Minimax object with the current game and assigns a
random column as the best move.

        Args:
            game (object): The current game object
        """
        self.game = game
        self.best_move = random.randint(0,6)

    def minimax(self, depth, max_depth, maximising_player):
        """ Returns a positive infinity, negative infinity, or 0 value depending on
whether the game has reached a result, or the heuristic score of the game state if
the maximum depth has been reached. Otherwise, it calculates all possible moves on
the game state and makes them until these conditions have been met. Once they have,
the movements are evaluated, either by comparing game state to a table of values as
to create a heuristic 'score' for each move, negative for the opponent, positive
for the plauer, or by a very high, very low, or average score for a win, loss and
draw respectively. These scores for each move are compared to find the highest
score-available move, which is then assigned to a class variable.

        Args:
            depth (int): The current 'depth' the Minimax is operating at, or how
many turns have been taken total by the Minimax
            max_depth (int): The maximum 'depth' at which the Minimax is allowed to
explore, or how many turns the Minimax is able to take
            maximising_player (int): The player whose point of view the Minimax is
operating from

        Returns:
            int: Either a 0, 500, or -500 if an outcome has occured, respective of
the outcome. Otherwise, an integer calculated by the evaluation function when given
the grid. Once all nodes have been explored, however, the highest score achieved by
either method is returned
        """
        if self.game.game_over():   #Outcome
            if self.game.result != Result.DRAW:
                res = 1 if self.game.result == Result.P1WIN else 2
```

```
                return 500 if res == maximising_player else -500
            else:
                return 0

        elif depth == max_depth:    #Max depth
            return self.evaluation(self.game.grid, maximising_player)

        else:   #Otherwise
            best_score = -500 if maximising_player == self.game.turn else 500
            for i in range(self.game.COLS):
                if self.game.valid_move(i):
                    self.game.make_move(i)
                    score = self.minimax(depth+1, max_depth, maximising_player)
#Recursive call, consider as changing turn and making another move
                    self.game.undo_move(i)  #End of recursive call, -1 step.
                    if maximising_player == self.game.turn:
                        if score > best_score:
                            best_score = score
                            if depth == 0:
                                self.best_move = i
                    else:
                        if score < best_score:
                            best_score = score
                            if depth == 0:
                                self.best_move = i

            return best_score

    def random_move(self):
        """ Returns a random integer between zero and the number of columns in the
grid.

        Returns:
            int: A random integer between zero and the number of columns in the
grid
        """
        best_col = random.randint(0, self.game.COLS)
        return best_col

    def evaluation(self, grid, maximising_player):
        """ Assigns a total positive and negative score to the current grid
depending on where the maximising player and opponent's pieces are compared to the
values stored within a table.

        Args:
            grid (2D Array): The grid of the game object with any number of moves
made onto it by the Minimax object
            maximising_player (int): The player whose counter positions will be
counted as positive

        Returns:
```

```
            int: The evaluation heuristic of the grid fed into the function from
the point of view of the maximising player
        """
        positions = [
        [ 3, 4, 5, 7, 5, 4, 3],
        [ 4, 6, 8,10, 8, 6, 4],
        [ 5, 8,11,13,11, 8, 5],
        [ 5, 8,11,13,11, 8, 5],
        [ 4, 6, 8,10, 8, 6, 4],
        [ 3, 4, 5, 7, 5, 4, 3]
        ]

        player_score = opponent_score = 0

        positions = [score for row in positions for score in row]
        grid = [piece for row in grid for piece in row]

        for i in range(len(positions)):
            if grid[i] == maximising_player:
                player_score += positions[i]
            if grid[i] != 0 and grid[i] != maximising_player:
                opponent_score += positions[i]

        return player_score - opponent_score
```

Connect4Main.py

```
import Connect4
import Connect4GUI
import MinimaxAttempt
from log_config import logging

app = Connect4GUI.App(Connect4.Connect4(7, 6), 1)
#Creates an app object with the game already instantiated, with Player 1 as the
default.
```